Peering into Advanced Mathematics through Sage-colored Glasses

John Harris,
Karen Kohl, and
John Perry,
all employed at the University of Southern Mississippi
(at least until the Provost's Office reads this debacle)

2000 *Mathematics Subject Classification.* 97U30, 97N80

ISBN 978-1-365-45825-5
Fourth revision, 3rd November 2016

Contents

Acknowledgments

This work was supported in part by a Summer Grant for the Improvement for Instruction, awarded by the Office of the Provost at the University of Southern Mississippi. Without that support, this would have taken a lot longer to complete, if ever.

We would also like to thank:

- William Stein and various Sage developers and users for moral support and occasional financial support to attend Sage Days workshops.
- All the developers who have contributed to the Sage project, whether directly or indirectly, deserve the scientific community's profound gratitude, though we can only assure them of ours.
- The image of Glenda, the Plan 9 Bunny on p. 210 was downloaded from Bell Labs' Plan 9 website and is used according to the terms given. (We *think* we have permission — the wording's a tad vague.)
- The image of Rabbit Island on p. 196 was taken by Kim Bui and is used under a specially-granted license (CC BY-SA 2.0).
- Amber Desrosiers and Candice Bardwell Mitchell suffered through an early draft of this text, and found more typographical errors than we care to admit.
- Valerio de Angelis kindly pointed out a large number of typographical errors, and suggested an improvement to a joke on infinite loops.
- Two of us have spouses and children that deserve apologies more than thanks, two of us have cats that deserve whatever it is that cats deserve (everything?), and one of us has a bunny and a turtle, both of which deserve more attention. Textbooks don't typically have a special section for apologies or whatever cats and bunnies deserve, so we thank them instead.

Preface

> It is said that despite its many glaring (and occasionally fatal) inaccuracies, the Hitchhiker's Guide to the Galaxy itself has outsold the Encyclopedia Galactica because it is slightly cheaper, and because it has the words "DON'T PANIC" in large, friendly letters on the cover. [1]

We have prepared this lab book as a kind of "Hitchhiker's Guide to Higher Mathematics," resisting the urge to write "Don't Panic" on the cover, in part out of sheer terror that Douglas Adams' estate might sue, in part out of a feeble attempt to maintain the last shred of human dignity we will doubtless retain in the reader's mind once he/she/it has put aside this text.

Why did you write this text?

The main goal of this text is to justify a partial summer salary extended us by our employer's Provost, who doubtless will never err in this fashion again.

No, really, why did you write this text?

The ostensible goal is that our institution offers majors a rather unique class, *Mathematical Computation.* We teach it, and we don't find a text that fits its unique character. We actually believe in the course and think it's a really good idea; many of the students who take it end up agreeing. With any luck, the resulting text will spring up like a fungus, here and there, impervious to eradication, until it comes to dominate the landscape of mathematics education.

What instructional value does this text offer?

We can only promise it will contain errors that range from the merely typographical to the outright mendacious. As to the former, we direct the reader to

`www.math.usm.edu/dont_panic/`

to find corrections to known typographical errors, though if the publication record of one of the authors is any kind of indicator, there will be typographical errors among the typographical corrections. As to the latter, any mendacity contained herein has the merit of sounding better than the outright truth, so in the spirit of the times we decided to include it. We leave it to the reader to sort the wheat from the chaff, though we assure the dedicated reader that there is more wheat than chaff, though for all you know that may turn out to be one of our mendacities.[1]

Speaking somewhat less un-seriously, we would like to think this book would be useful for any situation where an individual is moving from "high school" mathematics, in which we include basic calculus, to "university" mathematics, which includes intermediate calculus and a lot of stuff besides, much of which seems to make the average mathematics major regret having chosen the major. It is no secret that students struggle with the transition from the style of mathematics they have encountered in their youth, which is primarily computational, even to the point of

[1] For what it's worth, at least one outright mendacity is disclaimed as such, with cause.

being thoughtless, to upper-level mathematics, which is increasingly theoretical, and necessarily thoughtful. They have so much trouble with this transition that many universities gave up on students' learning, say, "proof techniques," via osmosis,[2] or even by imitation, and now offer a course dedicated to it. At times it goes by a subtle name like *Transition to Advanced Mathematics,* but often the title is much more obvious: at our institution, for example, it masquerades as *Discrete Mathematics*.[3]

The University's course on *Mathematical Computation* was the brainchild of two of our colleagues who, two decades past, won an NSF grant to develop it. As part of the grant, they wrote a textbook published by Springer. The course became a required part of the major, and even after they retired the department continued to use their text. Why are we not content to continue with that book?

- The course has evolved.
  - Originally the course required multivariable Calculus as a prerequisite, and focused almost exclusively on Calculus problems. As we teach it now, the course requires only second-semester calculus, and tries to introduce students to a more expansive view of mathematics.
  - Many students who encounter abstract topics in higher mathematics are unfamiliar with the objects they encounter, and essentially find themselves dropped into a bizarre new land, surrounded by a bewildering menagerie of exotic and frightening beasts.

    EXAMPLE. Many majors indicate that part of mathematics' appeal is its reliance on enduring, unchanging truths: as some would put it, "One plus one is always two." Yet one of these beasts in advanced mathematics insists resolutely that, contrary to everything they've learned before now, $1+1=0$.

    For the average Joe, the only surefire way to accustom oneself to these critters is to familiarize oneself with them, mainly through experimentation on examples. We would like to familiarize the student with these topics in a friendlier, less formal environment than the typical, axiomatic approach, much as Calculus has evolved to familiarize students with an intuitive idea of limits, *without* insisting on the rigor of the ϵ-δ definition of the limit.
  - A course like this turns out to be very useful for purposes of institutional assessment. Our department uses a portfolio as part of its annual assessment, and a number of these labs can serve as evidence that the student has of course worked with technology, but has also studied Calculus and become familiar with the breadth of mathematics (if not yet its depth).
- The old textbook required Maple. We prefer Sage for a number of reasons:
  - Sage is "free as in beer." We live in a poor part of the nation; while student versions of Maple and other computer algebra systems are arguably affordable, they are nevertheless a nontrivial addition to the high cost our students already bear.

[2]We are indebted to our colleague Bill Hornor for this vivid and entirely accurate description of how most of us learned proof techniques in our youth.

[3]No, we did not reverse the adjectives. Yes, we are trying to be funny. Probably, we may well be failing. You could give us an A for effort.

- Sage is "free as in speech." Instructors can show talented students parts of the code, and encourage them to get involved. There are doubtless a number of good undergraduate research projects that could lead to contributions to the Sage codebase. Talented instructors can even contribute code in ways that improves Sage's use in education. For that matter, two of us have shown that talent*less* instructors can contribute code to Sage. Don't be shy: the community is very supportive!
- Sage's interface relies on Python, an industry-standard programming language that remains in high demand among employers. Teaching students Sage per force teaches them quite a bit of Python, as well — and frankly, most students who graduate with a degree in mathematics aren't going to land jobs that require them to use anything beyond the Calculus, and probably not even that.
- Occasional changes to Maple's interface rendered the old textbook obsolete: we found code and examples that no longer worked. *Do not misread this observation:* Occasional changes to Sage's interface will likely render this text obsolete, as well, but at least you aren't paying for Sage.

Aren't there already some good references on Sage?

Yes, and that's the point; they strike us mostly as references for mathematicians and students to learn *Sage*. This text aims to help students learn *mathematics* via Sage. This puts it into a different niche, which we hope will prove not only profitable,[4] but also useful,[5] both to read and to edit. The LaTeX source to this material will be available online at no charge; users can download, edit, and revise it as they find fitting. Editing the labs to taste is especially advisable in light of an age where, speaking frankly, a nontrivial number of students prefers to expend more effort finding pre-made solutions via a search engine to developing their own solution via their in-house intellectual engines. To this end, we have tried to offer a large number of labs, so that even if you don't modify any of them to taste, the course can vary semester by semester, according to research interests, educational tastes, or daily hormone count. If you are feeling particularly hostile to your students, feel free to contact the authors and we can recommend some of the labs as especially difficult in practice — though, frankly, you can probably guess which ones those are on your own.

How do *you* typically run this course?

Actually, each of us teaches it quite differently. As you might expect, this Guide collects neither the intersection nor the full union of our thoughts, but falls somewhere in between. That said, we do tend to alternate between lecture days and lab days (no lab section is attached to this course); we do tend to proceed in some semblance of the order provided here; and we do assign textbook questions, labs, and tests. Some of us give group assignments and find them useful; some of us have abandoned group assignments as not especially helpful.

[4] Stop laughing.

[5] Really. Stop laughing.

Why this approach?

Since the entire point of computers is to compute quickly,[6] programs to compute quickly became increasingly powerful and widespread. It now allows students to visualize and attack more interesting and more difficult problems.

Accordingly, technology has become an indispensable aspect of most mathematics education. At many universities, students engage in long-term research projects that would have been unthinkable when the authors were in college. Honestly, the amount of computing power in your cell phone is more than we had in our home computers as children — assuming our families could even *afford* a home computer. ***We live in a magical world***, and part of our aim is to open their eyes to the wonders of the world we live in. The learning curve is often too steep, and even if they finish the degree, the enchantment of mathematics is gone — assuming they had any to start with. One large problem is their apparent reluctance to *experiment* with mathematics. We can teach them about groups (say) and give them some examples, but they are often too reluctant to move from the axioms and examples to playing with them on their own.

The access to greater computing power makes it easier to acquaint ourselves with unfamiliar mathematical objects and experiment with them. Matrices can serve as an example you've probably met: it is all too easy to get one element of an intermediate matrix wrong, ending up with every element of the final matrix wrong. The computer makes it easy to do the computations quickly and reliably, allowing them to generate and test conjectures. It doesn't *replace* the brain — they still need to understand the rules of matrix arithmetic! — it *frees* the brain.

Any last words?

You can purchase a hardcopy of this text from

http://www.lulu.com/

and searching for this text. You need not purchase a hardcopy; after all, you can see from the CC-BY-SA license that you can print it or even modify it yourself. You may prefer a hardcopy, though; many people do. If you do purchase one, we assure you that two things will happen.

(1) You will contribute to the death of some trees. But! that was happening even with the old book, so we don't recommend worrying too much about that.
(2) We contribute our share of the proceeds, currently $10, to
   ($5) an existing scholarship for students of mathematics at the University of Southern Mississippi,[7] and
   ($5) an organization whose funds are used to support the development of SageMath.[8]

[6] An amazing fact about mathematics is that you can use so much of it in places its inventors never imagined. The early pioneers of computer science, who were actually mathematicians, almost certainly had no idea that their inventions would lead one day to funny cat videos — and yet there is *so much mathematics* in funny cat videos! Someone should write an article about it one day.

[7] Currently the Pye scholarship.

[8] Tentatively the SAGE foundation.

Part 1

A Hitchhiker's Guide to Advanced Mathematics

Background

A thing they had never needed before became a necessity. (Narrator, [12])

To explore the world of mathematics, you can't just read about it; you have to *engage* it. Some people are gifted with enough confidence and/or aptitude that they take to this on their own; some were lucky enough to be brought up well, and from their youth are accustomed to engaging the world of mathematics.

Most are less lucky, and while they may find the wider world of mathematics intriguing, they struggle to make their way through new and unfamiliar territory. This text aims to encourage you to explore the world of higher mathematics with the help of a computer algebra system, relying on a particular system named Sage. We will encourage you to experiment with problems and form conjectures about their solutions. Sometimes we will encourage you to use the experimentation to formulate an explanation as to why the conjecture is, in fact, true.

But to use computers effectively, you need to learn how to program.

Is this just another programming textbook?

No.

Can you be more specific? Yes.

[Grumble.] Please go into some detail. This text is about *mathematics,* which itself is about *solving problems.* That's important enough to highlight, so that even the skimmers will notice it.

Mathematics is a tool for solving problems.

In particular, we want to introduce you to ideas and techniques of higher mathematics through problems that are arguably approached best with a computer, because

- they are long;
- they are repetitive or tedious; and/or
- they require experimentation.

Computers require instructions, and a group of instructions is called a *program.*[9] So we do study *programming,* but in reference to a specific goal: namely, solving problems in mathematics. That makes this text different from "programming" textbooks, which study programming in reference to a different goal: namely, solving problems in computer science. Again, that distinction is important enough to highlight, so that even the skimmers will notice it.

We study programming in order to solve math problems.

[9] If you are reading this as an electronic document, then from time to time you will notice text in a different color. If you click on it, you will find it links through the internet to additional information, very little of which is due to the authors of this text and, as such, is probably much more reliable and useful. We hope you follow these links and familiarize yourself with that information — that's why we included it! — but strictly speaking it isn't necessary.

Why program? If I wanted to write programs I'd major in computer science. To start with, some problems are too tough to do by hand, so you *have* to use a computer. The purpose of this text is to start you down this path *without* turning you into a computer science major. It will not only introduce you to some tools in a computer algebra system that help you compute, it will encourage you to take steps along the path of becoming a mathematical pioneer. To solve the problems we present, you will need to stop at a particular feature just outside your neighborhood and acquaint yourself with it a little more than you might in the absence of our encouragement. In some cases, we will even lead you back over land you traveled already and ask you to re-examine something a little more carefully. In short, and in repetition, this text is about exploring the world of higher mathematics, with a computer's help.

On the one hand, computers don't understand human languages. Humans are intuitive and poetic, resorting to figurative and abstract language. Computers understand none of that; they really understand one thing only: **on** and **off**. Everything your cell phone, laptop, or desktop does involves a clever arrangement of switches by human beings who are well-trained in the art of directing electric current to the right place at the right time.

On the other hand, most humans don't understand the computer's language of **on** and **off**. Reducing every problem to these two symbols has been extremely effective, but it's uncomfortable to humans (if only because it's so *tedious!*).

Learning to program gives you control over the computer's circuitry, and allows you to work at a level that is much more comfortable. Even the experts typically work with more abstract interfaces that are themselves converted to **on** and **off** signals in several stages. Learning to program also gives you a deeper understanding of computer technology, and an appreciation for the amount of work that goes into building this magical world we live in, where you can speak into a little box in your hand and be heard by someone half a world away.

What kinds of programming languages are there? Without getting into too much detail, there are three types of computer programming languages available today:

- In an interpreted programming language, the computer reads a file that contains commands in the language. It translates each symbol, and executes some sequence of commands based on that symbol. (Here, "symbol" can include words as well as numbers and abstract characters.) It then proceeds to the next symbol, eventually forgetting its translation of the previous one. Examples of interpreted languages include BASIC, Python, and the "shell" languages of command-line prompts.
- In a compiled programming language, the computer reads a file that contains commands in the language. It translates each symbol, but instead of executing any commands, it records the translation into **on** and **off** signals and stores them in a new file, usually called an *executable*. (We write "usually" because it sometimes produces a *library* instead, depending on the programmer's request.) Examples of compiled languages include Fortran, C/C++, and Go.

Before describing the third type of language, let's mention some advantages and disadvantages of each type. Interpreted languages are by nature typically much, much slower than compiled languages, because the computer must re-translate each symbol, regardless of how many times it has translated it before. (This is something of an oversimplification, but neither is it that far from the truth.) On the other hand, interpreted languages are typically much more interactive and flexible than compiled languages, to the point that many users never write an actual program for them, but merely issue one command at a time.

Similarly, the nature of compiled languages means that the precise **on** and **off** signals are tied to a particular architecture, one reason programs compiled to run on a Windows device won't run on Macintosh or Linux. C++ developers have to re-compile each program for a different architecture, and for many programs this becomes quite difficult, especially if the program relies heavily on Windows' particular graphical interface. As a result, interpreted languages can be much more "portable," which means you can simply copy them onto another machine. Python programs are especially famous for their portability. It's arguable that Microsoft Corporation's success is due primarily to its brilliant and ubiquitous BASIC interpreter for home computers of the late 70s and early 80s; it lives on today as Visual BASIC.

- Bytecode languages seek to straddle the gap between the two. In this case the computer reads a file that contains commands in the language, and translates each symbol, but not into the **on** and **off** signals native to the computer's architecture. Rather, it translates them into signals designed for an abstract computer called a *virtual machine,* then stores them into an executable or library that will only work on computers that contain programs that understand the virtual machine's signals. Notable bytecode languages include Pascal,[10] Java, and the .NET languages' Common Language Runtime.[11]

Because the executables and libraries are not in the computer's own signals, bytecode languages are technically a special kind of interpreted language, and their reliance on a virtual machine means they are theoretically slower than compiled languages. In practice, the penalty is usually quite small, because the virtual machine's signals are designed to be translated very easily into an actual computer's signals. Modern techniques make them so efficient that some bytecode languages frequently outperform many compiled languages. Reliance on the abstract virtual machine means bytecode languages are highly portable; we wrote above that the executables work "only" on computers that contain programs that understand the virtual machine's signals, but that also means they run on "any" computer with a program that understands the virtual machine's signals. Java's popularity — it once seemed impossible to find a webpage without a Java applet — was due to its "Write Once, Run Anywhere" philosophy, which depended on its virtual machine capabilities.

Which of these language types do we use in this text? All of them, in fact.

The primary programming language in Sage is *Python,* which we listed above as an interpreted language. Sage and Python are not quite the same, though; Sage programs will not work in plain, vanilla Python, and some Python features are different in Sage.

You can sometimes compile Sage programs using a program Cython; we cover that near the end of the text. However, "Cythonized" Sage will not stand on its own; you have to run it inside a Sage environment.

As you will learn below, Sage is actually assembled from a large number of separate parts. Some of them are written with Java, which means you are using a bytecode language, though you won't actually write any Java programs.

What is this Sage thing you keep yapping about?

Sage is a free, open-source computer algebra system.

[10] In its original incarnations, Pascal was translated to a similar idea called P-code, later into bytecode proper.

[11] Many interpreted languages are now compiled "incrementally." They save each command's interpretation, and as they progress during execution, check to see whether each new command has already been interpreted.

What is a "computer algebra system"? Traditionally, there have been three major kinds of large-scale mathematical software packages:

- *Numerical computing systems* aim for fast computation, relying typically on "floating point" numbers. We won't go into the details right now, but you can think of floating point as a kind of "accurate estimation," similar to the use of significant digits in the sciences. The science behind numerical computing is typically the province of *numerical analysis*. Almost everyone in the developed world has used a numerical computing system at some point by turning to a calculator. Numerical software packages include MATLAB and Octave.
- *Statistical software packages* are a special kind of numerical computing system that focus on special tools proper to statistical analysis. Examples include SAS and the R project.
- *Computer algebra systems* aim for *exact* computation, even if this comes at the expense of the speed one associates with numerical systems. Rather than manipulate approximate values, computer algebra systems manipulate symbols that represent exact values: they don't view π as a decimal number but as a symbol with the properties mathematics associates to it, and they allow us to manipulate expressions that involve variables. Because of this, the science behind computer algebra systems is called *symbolic computation, computer algebra*, or *computational algebra*. A few of the more expensive calculators use symbolic computation, but typically one works with a software package like Maxima, Maple, or Sage.

Why would one sacrifice exactness for the approximate values of a numerical system? The main reason is *speed!* By sacrificing a small amount of precision, a numerical system can easily work with both large and small numbers, vectors, and matrices, all without too much trouble; it is not uncommon to work with hundreds or even thousands of variables in a system of equations.

For example, what happens if you add the fractions $1/2$, $1/3$, $1/5$, and $1/7$? Each of them requires only two digits to write (numerator and denominator), but the exact sum, $247/210$, requires six digits — a 300% increase in size! If you turn to floating point numbers with at most 4-digits, the sum becomes instead

$$0.5000 + 0.3333 + 0.2000 + 0.1429 = 1.176.$$

The sum is no larger than the original numbers. True, it's a *little* bit wrong, but the error is less than $1/500$; that's *much* more accurate than most day-to-day tasks need.

The rapid growth in size is one reason people generally dislike fractions; no one likes to work with objects whose complexity grows quickly. In this case, what's true about people is true about computers, too; if you work with a problem that requires division of integers in Sage, you will almost certainly encounter a massive slowdown as the fractions become complicated and, therefore, more difficult for the computer to handle.

In that case, why would anyone bother with symbolic computation and exact values? For many problems, the inaccuracies of floating point computation makes it absolutely unsuitable. This is especially true when *division* is an inescapable part of the problem.

For example, suppose the computer has to divide by the 4-digit floating point number 0.0001. The resulting quotient grows very large. Yet it's entirely possible that 0.0001 is the result of a floating-point error, and the correct value is actually 0. As you surely know by know, division by 0 is ***bad***. Had the computer known the value was 0, it wouldn't have divided at all! Problems where this can occur often are called "ill-conditioned," and numerical analysts spend a lot of time trying to avoid them.

In some cases, however, you can't, so you resort to exact values. This is especially true as one moves into more abstract mathematical fields. Some people think "abstract" mathematics is "useless" mathematics, but they are quite mistaken: this text will introduce you to several abstract mathematical objects whose very exactness makes possible the extremely accurate and extremely secure communication you achieve on the internet whenever you check your bank account or buy from an online vendor.

What's special about Sage? Sage was "started" by William Stein, a number theorist. He was frustrated with several drawbacks of the computer algebra systems available at the time:

Commercial systems, like Maple, don't allow the user to view the code, much less modify it. In the software world, these systems are called "proprietary", "closed", or "unfree" (by some people). *"Open" or "free" systems* also existed, and as the product of cutting-edge research, they were often better at a particular task than the commercial packages. However, these packages *only* excel at one particular field of mathematics:

- for Calculus, you'd likely use Maxima;
- for linear algebra, you'd likely use Linbox;
- for group theory, you'd likely use GAP;
- for number theory, you'd likely use Pari;
- for commutative algebra, you'd likely use CoCoA, Macaulay, or Singular;

and so forth. Worse still, suppose you needed to transfer the result of one package to another: after performing some number theory with Pari, for instance, you might want to analyze the results using group theory, in which case GAP would be the tool of choice. But there was no easy way to copy the results from Pari into GAP.

Sage, then, was organized to bind these brilliant tools together into one, relatively easy package. Additional features were programmed in Sage proper, and in some cases, Sage has been the leader at solving some problems. As a bonus, Sage's developers have made it possible to interact with many proprietary packages through Sage, so that if Maple has the fastest tools to solve some problem, and you own a copy, you can get it done through Maple, then manipulate the result through Sage.

Why the concern with "free" software? In the world of software, the term "free" has two senses:

Free as in beer: You don't have to pay for it.
Free as in speech: The code is open and viewable, rather than "censored."

Software can be "free as in beer" but not "free as in speech;" that is, it costs nothing, but you can't view the source code. Examples include the numerous "free" programs you can download for a computer or mobile phone.

There are important reasons a researcher or even an engineer should be able to view and/or modify the code in a mathematical software package:

- Good scientific practice requires reproducibility and verifiability. But a researcher can't verify the results of a mathematical computation if she can't check the way it was computed.
- Any software package of significant size will have some mistakes, called *bugs*. If two software packages produce a different answer, a researcher can't decide which one is correct if he can't view the code and evaluate the algorithms.

- Almost all mathematical research builds on previous work. The same is true about software packages, and researchers often need to extend a package with new features in order to accomplish some task. This can be much more difficult to do properly if the researcher can't view the code, let alone modify it.

For example, suppose a mathematician claimed to have a proof that there are infinitely many primes. Most mathematicians would want to see the proof; that's one way we learn from each other. (In some cases, the proof is much more interesting than the theorem.) Indeed, you can find this proof in many, many textbooks, because the Hellenic mathematician Euclid of Alexandria recorded what is considered one of the most beautiful proofs of this fact over two thousand years ago [11, Book IX, Proposition 20]:

THEOREM. *There are infinitely many prime numbers.*

PROOF. Consider any finite, nonempty set of primes, $P = \{p_1, \ldots, p_n\}$. Let $q = p_1 \cdots p_n + 1$. Since $q \neq 1$, at least one prime divides it, but the remainder of dividing q by any $p \in P$ is 1, so none of P's primes divides it. So there must be some prime number not listed in P. But P is an *arbitrary*, finite set of primes, which means that *no* finite set of primes can list all of them. In other words, there are infinitely many prime numbers. □

By exposing the proof plainly, Euclid makes it easy to verify the result. He also makes it easy to question the argument: you might wonder, for instance, how Euclid knows that at least one prime divides any positive integer that is not 1. (As a matter of fact, he proves that earlier on.)

Compare this to another famous theorem attributed to Pierre de Fermat, a French jurist who studied mathematics as a hobby [10, p. 61, Observatio Domini Petri de Fermat]:

THEOREM. *If $n > 2$, the equation $a^n + b^n = c^n$ has no solution with integers $a, b, c \geq 1$.*

PROOF. I have found a truly wonderful proof of this fact. This margin's smallness will not contain it. □

These two sentences set off a search for a proof that lasted more than three centuries; Andrew Wiles found a proof in 1994, and to date there are no other proofs. Most people agree that Fermat did not, in fact, have a proof, but we shouldn't think ill of him: he never actually *told* anyone he had a proof; he merely wrote this comment down in the margin of a book. His son published a copy of the book after Fermat's death, and included the notes Fermat wrote in the margin.

To stretch the analogy further, suppose we were to claim the following:

THEOREM. *The number*

$$2^n - 1$$

is prime for $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$.

PROOF. Trade secret. □

You would be right to doubt our claim, for at least two reasons: there is no *easy* to way to verify it, and in fact it is wrong! Yet this claim was made by a well-respected mathematician named Marin Mersenne [9], offered without proof, and was for some time generally accepted. You will meet this claim again in a lab later on.

What are some advantages of Sage? As mentioned before, Sage makes it easy to experiment with mathematical objects that you will use increasingly more in classes after this one. Later

classes will probably not require Sage explicitly, but if you never use Sage again after this class, that would be like going through a Statistics class and doing all the work by hand:[12] *WHY?!?*

Another advantage to Sage is that you interact with it through a Python interface; programming in Sage is, to some extent, indistinguishable from programming in Python. This confers the following benefits:

- Recall that Python is one of the more widespread interpreted languages.
  - Many employers want Python experience. Learning Sage well helps you learn Python, and helps make you more employable.
  - Many packages are available to enhance Python, and work well with Sage. Indeed, many such packages are already packaged with Sage.
  - As mentioned earlier, you can often speed up a program by "Cythonizing" it.
- Python is a modern language, offering many ways to express an elegant solution to a problem. By learning Sage, you are learning *good* programming practices.

Keep in mind that Python and Sage aren't the same thing, nor is either a subset of the other. Sage commands do not work in plain Python, and some Python commands don't work the same way in Sage that they would in Python.

How do I get going with Sage?

Probably the easiest way is to visit the SageMathCloud server at cloud.sagemath.com, register for a free account, start a project, and create a Sage worksheet. We strongly urge the reader to pony up the dough for a membership, which at the time of this writing costs $7/month and gives access to faster, more reliable servers. You can use it for free, but for various reasons, the free servers sometimes reset on you. (If you are part of a class, however, ask the instructor if the department has ponied up for a class package; the discount is substantial.) The drawback to this approach is that you have to pay to get good service. The advantage is that you always have a reasonably up-to-date version of Sage at your fingertips, and you don't have to worry about a hardware crash that wipes out all your data, because it's stored on servers that rely on multiple backups and fallback mechanisms.

Another way to use Sage is via an online server that is not a SageMathCloud server. This requires you to know somebody who knows somebody who… knows somebody who runs a server you can use. Quite a few institutions do this, including the one that employs the authors; in fact, our department maintains at least two such servers at the time of this writing. The drawback to this approach is that you depend on the server's maintainer to keep the software up-to-date and the data backed up. Indeed, one of our department's servers runs a version of Sage that is years out of date.

The last way to use Sage is from a copy on your own machine. You can download it from www.sagemath.org and install it on your computer. (Look for the link to "Downloads".)

- If your machine runs Linux, this is a relatively simple process, though you may have to install some packages through your package manager. (In the past, we have had to install a system called `m4`. We don't remember what it is.) Binaries are available for some Linux distributions; Ubuntu is typically one of these, and Fedora has been from time to time. For the rest, you'll likely need to download the Sage source and compile it on your

[12] At least one of the authors actually tried this when he was in college. In fact, Statistics was the one class that broke his resistance to using a calculator. The calculator's ability to perform exact computation of fractions impressed him; until then, he had only seen calculators perform arithmetic with fractions via floating point.

computer. The good news is that this is usually quite painless, as long as you have already installed the required packages, and these are listed in the directions. The bad news is that installing from source takes quite a long time, so prepare for some hurry-up-and-wait.

- If your machine runs OSX, try downloading a binary for your architecture. You can try compiling from source, as with Linux, but in that case, hope that Apple hasn't recently "upgraded" Xcode, because you need Xcode to install Sage, and every major release of Xcode typically breaks Sage in some way.[13]
- If your machine runs Windows, you are in the unusual position of having to suffer what Linux and OSX users typically suffer: Sage doesn't work natively on Windows, so you have to use it through a virtual machine. This can be a little tedious to set up, but once you get the hang of it, it works fine. The correct way to do this has changed several times over the years, so we are reluctant to give more precise advice than this. The good news is that instructions on installing and running Sage will be available at the website.

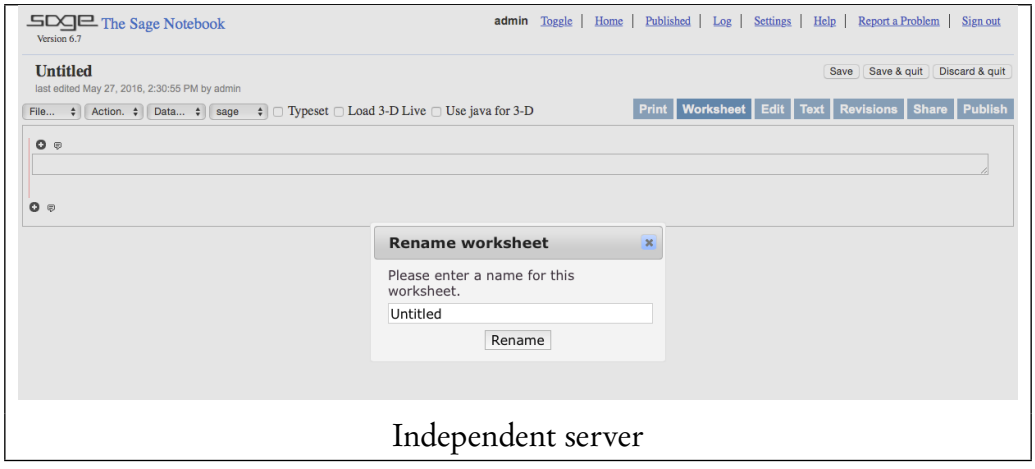
Once you have one of these methods up and running, you start using it!

Worksheet or command-line?

There are two typical ways to use Sage: from a browser, in what's called a *Sage worksheet*, or from the command line. If you have installed Sage to your machine, you can do both; see the section on Command-line Sage to see how to start a Sage worksheet from the command line.

Sage worksheets. If you have access to Sage via a web browser (either SageMathCloud or another online server) you will likely prefer to work with a Sage worksheet. We recommend our students to start with Sage in this fashion, because the worksheet provides a more user-friendly environment: it is easy to review and re-execute commands, to add text for organization and narrative, and further to save your entire session, then later reload the work and the output. You can't do any of that with the command line.

When you start a worksheet, you should see one of these two screens:



Independent server

[13] Sage is by no means the only software package that suffers this consequence.

SageMathCloud

You can (and should) change the title.

- In the independent server, you can do that at the beginning with the "Rename worksheet" dialog you see in the screenshot. You can do it later by clicking on the title (in the upper left, currently "Untitled") and the same dialog will pop up.
- In SageMathCloud, you can do that by clicking on the circled `i` in the upper left and choosing "Rename…". A new screen will appear, prompting you to rename the file. Make sure you keep the `.sagews` added at the end.

There are other options you can monkey with, but for now we'd recommend you move on to the next chapter, since most of those options are of small importance for our purposes. The ones we do need we'll discuss in due course.

Command-line Sage. If you choose to run Sage from the command line, you need to open a *shell*, also called *a command-line prompt*. You will see some sort of prompt, which can vary quite a bit; whenever we mean a shell prompt we'll simply put a blue greater-than symbol: `>`. At the prompt, type `sage`, press `Enter`, then wait for Sage to start up. This might take a few seconds, but eventually you will see something to this effect:

```
> sage
SageMath Version 6.7, Release Date:  2015-05-17
Type "notebook()" for the browser-based notebook interface.
Type "help()" for help.
sage:  _
```

The underscore (_) might actually look like a little box on your system. Once you see that, you're in good shape for the next chapter. If you *don't* see it, or if you see some sort of error, you need to talk with your instructor or tech support and see what went wrong.

If you'd like to run a Sage worksheet in a browser, but don't want to run SageMathCloud and don't have access to another server, type `notebook()` and press `Enter`. You will see a lot of messages, for instance:

```
sage:  notebook()
The notebook files are stored in:  sage_notebook.sagenb
Open your web browser to http://localhost:8080
Executing twistd --pidfile="sage_notebook.sagenb/sagenb.pid" -ny
"sage_notebook.sagenb/twistedconf.tac"
/Applications/sage-6.7-untouched/local/lib/python2.7/site-packages/
Crypto/Util/number.py:57:  PowmInsecureWarning:  Not using
mpz_powm_sec.  You should rebuild using libgmp >= 5 to avoid timing
attack vulnerability.
_warn("Not using mpz_powm_sec.  You should rebuild using libgmp >=
5 to avoid timing attack vulnerability.", PowmInsecureWarning)
2016-05-27 14:30:49+0300 [-] Log opened.
2016-05-27 14:30:49+0300 [-] twistd 14.0.2
(/Applications/sage-6.7-untouched/local/bin/python 2.7.8) starting
up.
2016-05-27 14:30:49+0300 [-] reactor class:
twisted.internet.selectreactor.SelectReactor.
2016-05-27 14:30:49+0300 [-] QuietSite starting on 8080 2016-05-27
14:30:49+0300 [-] Starting factory <__builtin__.QuietSite instance
at 0x1181bb638>
```

For the most part, you *do not* need to worry about those messages.[14] You don't even have to follow the directions to open your web browser to that site; on many machines, the browser will open the webpage automatically. Besides, it will take a few seconds for things to get started, so sit back and relax a few seconds. If your browser doesn't open, Don't Panic! Open it yourself and see if the web address *your* Sage advises works. If it does, you're in good shape for the next chapter.

If it doesn't,

PANIC!

Yes, it really is okay to panic once in a while. Get it out of your system. Once you're done, look carefully at the messages, and see if any are error messages; these would be helpful. Then visit

`https://groups.google.com/forum/#!forum/sage-support`

and search to see if that error message has been discussed. If not, start a new post, inquiring about what's going wrong.

Getting help

If you're reading this as part of a class, then your instructor should be helpful. (Maybe not *very* helpful, but helpful all the same.) We already mentioned the `sage-support` forum last chapter. Aside from these options, Sage will answer many questions on its own.

Docstrings. If you want to know how a command works, type the name of the command, followed by a question mark, then execute the command. Sage will provide you useful information on the command, typically with examples.

[14] Well, maybe the security warnings about `libgmp`, if you see them. I should look into that.

```
sage: simplify?
Signature:  simplify(f)
Docstring:
    Simplify the expression f.

    EXAMPLES: We simplify the expression i + x - x.

        sage:  f = I + x - x; simplify(f)
        I
In fact, printing f yields the same thing - i.e., the simplified
form.
Init docstring:  x.__init__(...)  initializes x; see help(type(x))
for signature
File:  /Applications/sage-6.7-untouched/local/lib/python2.7/site-
packages/sage/calculus/functional.py
Type:  function
```

Finding methods for objects. Another way to get help is to see what commands an object will accept as *methods*. A "method" is a command that is very specific to a particular Sage object; you invoke it by typing the object's name, followed by a dot, then the name of the method.[15] To find the names of all the methods an object accepts, type its name, followed by a dot, then press the `Tab` key.

For example, `simplify()` doesn't work very well on the following expression:

```
sage: rats = 1/(x**2 - 1) + 1/(x - 1)
sage: simplify(rats)
1/(x**2 - 1) + 1/(x - 1)
```

We *really* want to simplify that expression as fully as possible, so let's see if it accepts a method that will perform a more thorough simplification. Type `rats.` and then press `Tab`; you should see over 200 possible methods. Some of them are not really appropriate for `f`; we won't go into the reasons for this, but if you look carefully, you should find at least two useful methods. One of them is `full_simplify()`.

```
sage: rats.full_simplify()
(x + 2)/(x^2 - 1)
```

That's a very nice result.

The thought of hunting through over 200 possible methods may seem intimidating, but many objects accept very few methods. In practice, this isn't such a difficult technique, since there are many ways to search through the list.

[15] Another term for "method" is "message;" both terms are used in computer science, but "method" is the jargon for Sage.

Exercises

True/False. If the statement is false, replace it with a true statement.

1. This is just another programming textbook.
2. Mathematics is about counting numbers.
3. While students don't much care for fractions, computers find it easier to work with fractions than with floating point numbers.
4. "Free" software is written by unemployed slackers living in their parents' basements.
5. All famous mathematical results were appreciated from the start for their clear, elegant proofs.
6. Bytecode is technically not an interpreted language.
7. Interpreted languages are appreciated above all else for their speed.
8. Some software of significant size is bug-free.
9. Abstract mathematics is useless in the real world.
10. It is easy to verify and improve on proprietary mathematics packages.
11. *Bonus:* The answer to all these True/False questions is "False."

Multiple Choice.

1. Which of the following is *not* an example of a computer algebra system?
   A. A package that performs fraction arithmetic without any roundoff error at all.
   B. A package that assigns numbers to musical notes and uses fractals to produce new music.
   C. A package that performs polynomial arithmetic with approximate values for coefficients.
   D. A package that focuses on sets of abstract objects defined by precise properties.
2. Which of the following is *not* a step in the usual process of compiling a program?
   A. The compiler reads the source from a file.
   B. The compiler translates each symbol into a combination of **on** and **off** signals.
   C. The compiler saves the translation to a file, called an executable or a library.
   D. The compiler executes the translated commands immediately before quitting.
3. Sage's primary focus is what kind of computational mathematics?
   A. numerical computation with approximate values
   B. statistical computation with probable values
   C. symbolic computation with exact values
   D. fuzzy computation with uncertain values
4. Which of the following computer algebra systems would you use to compute a derivative?
   A. Maxima
   B. PARI
   C. Schoonship
   D. SINGULAR
5. Which of the following well-known computer languages tries to straddle the gap between interpreted and compiled languages?
   A. C/C++
   B. Fortran
   C. Java
   D. Python
6. Which of the following is a primary motivation of the movement for "free" mathematics software?
   A. Antipathy to censorship
   B. International collaboration

C. Lack of grant funding
D. Verifiability of results

7. Which of the following mathematicians is famous in spite of popularizing a "fact" that was very wrong?
   A. Pierre de Fermat
   B. Marin Mersenne
   C. William Stein
   D. Andrew Wiles
8. Which of the following mathematicians is famous in spite of his non-scientific day job?
   A. Pierre de Fermat
   B. Marin Mersenne
   C. William Stein
   D. Andrew Wiles
9. Which of the following is *not* an advantage to working with Sage?
   A. You acquire practical skills that employers value.
   B. You work with cutting-edge software.
   C. You no longer have to worry about careful computation.
   D. You can use the material learned here in other classes.
10. In which of these ways does Sage work with Python?
    A. Sage's interface is essentially a Python interface.
    B. Sage is a Python library you can load into any Python interpreter.
    C. Python is one of the computer algebra systems in Sage.
    D. Sage uses the Cython compiler to compile all Python programs.

Bonus: Which answer is correct?

A. The next one.
B. The next one.
C. The next one.
D. The first one.

Short answer.

1. Explain how the quote at the beginning of this chapter is related to its main thesis.
2. Describe a real-world analogy for the difference between compiled, interpreted, and bytecode languages.
3. A common problem in mathematics textbooks is to compute the 1001st derivative of $\cos x$. The best way to find the answer is *not* to compute all 1001 derivatives; instead, you compute a few, notice a pattern, and deduce quickly what the 1001st derivative should be. Explain how this compares to what we want you to get out of this text.
4. Not all mathematicians find the arguments in favor of free software convincing, and the use of proprietary software is widespread. Why do you think this is the case?
5. Even if Sage is less popular than a proprietary package like Maple or Mathematica, learning Sage can make it easier to work with those packages, as well. Why?

Basic computations

> What's in a name? that which we call a rose / By any other name would smell as sweet… (Shakespeare)

Create a new worksheet and call it "My First Sage Worksheet."

To describe interaction with Sage, we use the following format:

```
sage: some input
some output
```

The text "`some input`" indicates a command you type into Sage. If you are using Sage from the command line, you will type this at a blue "`sage:`" prompt; hence the blue color and the label. If you are using Sage from a worksheet, you will not see a prompt; instead, you will type the command in a "cell," which in some versions of Sage is outlined by a small box.

To execute the command:

- at the command line, press the `Enter` or `Return` key;
- in a worksheet, hold `Shift` and press the `Enter` or `Return` key.

Sage will then interpret and process your command. The worksheet interface will show a colored or flashing bar while Sage does this; the command-line interface will simply pause.

Once the output is complete, you will see the text indicated by "`some output`." If the command succeeded, you will see an answer that looks more or less sensible. If an error occurred, the text will include a lot of information, some of it in `red` (the last line especially, which is all we will copy) and probably won't make sense unless you already know Python. For instance:

```
sage: some bad input
SomeError:  some message to explain the error
```

We will explain several types of errors as we progress through our explorations of Sage. As a convenience, we are adding errors to the textbook's index. If you are working on some computation, and you run across some error you don't recognize, see if you can find it in the index; if so, the page numbers it references will prove helpful.

Here's an example of a successful computation; you should try this yourself now and make sure you get the same result.

```
sage: 2 + 3
5
```

That looks reassuring; at least Sage can do that! Here's one where the user forgot to let go of the `Shift` key before pressing the number 3.

```
sage: 2 + #
SyntaxError:  invalid syntax
```

If you actually run this in Sage, you'll notice there's a bit more text in the error, as well, as well as a lot of color changes. In our case, this turns out to look like:

```
File "<ipython-input-3-7c2c726856a7>", line 1
  Integer(2) + #
               ^
SyntaxError:  invalid syntax
```

You'll notice we *only* copied the last line, which specifies the type of error! When necessary for the discussion, we'll sometimes include the rest of the information as well, but you can often figure out the problem just from looking at the last line.

It may happen that the line you have to type is too long to fit in one line. It *will* happen to the authors of this text, since we don't have that much horizontal space on the page. In cases like this, you can keep typing, which can make the code harder to read, or you can tell Sage you want to continue on the following line by typing a backslash `\`, which Sage interprets as "line break," then pressing `Enter`. We will do this rather regularly to help make code more readable. There is no need to wait until we bump up to the end of the line for this, and sometimes it may be more readable to add the line break early. For instance:

```
sage: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \
....:   + 11 + 12 + 13 + 14 + 15
120
```

(When using a Sage worksheet). One advantage to using Sage worksheets is that you can use HTML commands to add explanatory narrative to your work. Simply type

```
sage: %html
```

…and everything following that line will be considered HTML text. The toolbar will change to allow for HTML formatting, but if you are familiar with HTML tags then you can add them directly. You can format it the same way as you execute a Sage command: hold `Shift`, press `Enter`. If needed, you can then edit the HTML cell again by double-clicking on it. This is enormously useful when breaking up a long worksheet into sections, with headers that organize related parts of the work.

Yer basic arithmetic

As you might expect, Sage offers the same basic arithmetic functions you can find in any calculator. Here are a few that will prove useful.[16]

[16] You can also type *a*^*b* for exponentiation, but for various reasons we don't recommend it: in some situations, Sage will interpret it as a different operator.

`a+b`	adds a and b
`a-b`	subtracts a and b
`a*b`	multiplies a and b
`a/b`	finds the ratio of dividing a by b
`a // b`	finds the quotient of dividing a by b
`a%b`	finds the remainder of dividing a by b
`a**b`	raises a to the bth power
`sqrt(a)`	the square root of a
`abs(a)`	the absolute value of a

TABLE 1. Sage operators for basic arithmetic

Go ahead and try these with some numbers, both approximate and exact.

Yer basic comparisons. Sage can also compare objects, to a certain extent.

`a>b`	is a strictly greater than b?
`a>=b`	is a greater than or equal to b?
`a==b`	is a equal to b?
`a<=b`	is a less than or equal to b?
`a<b`	is a strictly less than b

TABLE 2. Sage operators for basic comparisons

When using these comparisons, Sage will return `True` or `False`, which obviously correspond to "yes" or "no."

```
sage: 2 < 3
True
sage: 2 > 3
False
```

You have to be careful with comparison of equality, as the sign is doubled. Problems will arise if you forget it.

```
sage: 2 = 3
ValueError:  The name "2" is not a valid Python identifier.
```

If you see that error message, the problem is almost certainly due to the use of only one equals sign when you need two.

Aside from comparing numbers, *many* symbolic objects can be compared. We won't talk about sets for quite some time, for instance, but there is a natural ordering of sets based on subset

properties, and Sage will compare sets based on that fact. Consider, for instance, the sets $\{2,3\}$, $\{2,4\}$, and $\{2,3,4,5\}$:

```
sage: {2,3} < {2,3,4,5}
True
sage: {2,3,4,5} > {2,3}
True
sage: {2,3} < {2,4}
False
```

Exact v. approximate values. Sage offers symbols to represent the exact values of some important numbers; these should be easy to remember.

`pi`	π, the ratio of a circle's circumference to its radius
`e`	$e = \lim_{x\to\infty}\left(1+\frac{1}{x}\right)^x$, *or*, the value of a such that the derivative of a^x is a^x
`I`	i, the "imaginary" number ($i^2 = -1$)
`oo` *or* `+Infinity`	∞, infinity (unbounded above)
`-oo` *or* `-Infinity`	$-\infty$ (unbounded below)

TABLE 3. Sage symbols for important constants

We italicize these identifiers in the text, both to help with readability and to highlight that they should represent fixed values.[17] Sage does not italicize them in output.

We explained already that the strength of a computer algebra system like Sage lies in its ability to manipulate exact values, rather than approximate values. Nevertheless, we sometimes need to compute with approximate values, and Sage allows this as well.

To use approximate arithmetic, type at least one number as a decimal!

To see how it works, consider the following commands and their results.

```
sage: 2/3
2/3
sage: 2./3
0.666666666666667
```

In the first case, you entered exact values, so Sage gives you the *exact* quotient of dividing 2 by 3, the fraction $2/3$. In the second case, you're specifying at least one decimal, so you're getting the *approximate* quotient of dividing 2 by 3. This may look like floating-point, but it's not quite; it's actually an object Sage calls a `RealNumber`. Don't worry about those details right now.

[17] As we note on page 29, though, Sage does not have true constants, so a user or program can change the symbols' meanings.

Constants, variables, and indeterminates

A subtle distinction in usage. The mathematical symbols e, π, and i represent *constants*, by which we mean their values are definite and fixed. If someone writes Euler's equation,

$$e^{i\pi} + 1 = 0,$$

then every educated mathematician will know the values of e, i, and π, even if this is the first time they see the equation. Thanks to the symbols we described above, this holds true in Sage, as well:

```
sage: e**(i*pi) + 1
0
```

Of course, there are times when any of these symbols can mean something else. For instance, x_i refers to the ith number in a list, not to some manipulation of x by the imaginary number.

Other one-letter symbols in mathematics represent one of two kinds of *variables*, by which we mean their values are not necessarily fixed during a problem. The symbol x, for instance, can represent any of an infinite number of values. Mathematicians generally work with two kinds of variables; and in ordinary parlance, we tend to refer to both kinds as variable, but in fact there is an important distinction, which we can see in an expression as simple as the polynomial

$$x^2 - c^2.$$

- In many situations, specific values for x and c are very important. The equation

  $$x^2 - c^2 = 2$$

  is not always true; it depends on the values of x and c, and in practice we often try to find values for which this is true.
- In other situations, however, x and c represent arbitrary values. The equation

  $$x^2 - c^2 = (x + c)(x - c)$$

  is true regardless of the values of x and c.

This matters a great deal for computer algebra systems, because a symbol can likewise refer to a specific or indeterminate value, and we need to pay attention to this distinction from time to time. We will agree on the following convention:

- When a symbol has been assigned a specific value, we usually call it a *variable*, because we can change that value at any time.
- When *we have no intention* of changing a variable's value, we call it a *constant*.
- When a symbol is not to hold any specific value, but is purely symbolic for an arbitrary value from some set, we call it *indeterminate*.

Sage provides only one indeterminate at startup: `x`. If you want to work with another indeterminate, such as `y`, you have to create it.

Sage does not offer a way to define your own constants, the way some programming languages do. In Sage, a constant is just a variable that you try really hard not to change.

Resetting a variable or indeterminate. Since Sage has no way for you to define true constants, you can, if you choose, reassign the value of `I` to something else, and it is very likely that you will do that someday, in some circumstance. If that happens, it is easy to fix with the `reset()` command; include the symbol between quotes inside the parentheses.

```
sage: I^2 + 1
0
sage: I = 2
sage: I^2 + 1
5
sage: reset('I')
sage: I^2 + 1
0
```

Creating variables and indeterminates. To create a variable, use the assignment construct,

identifier = *expression*

where *identifier* is a legitimate name for an identifier of a symbol and *expression* is a legitimate mathematical expression. For instance,

```
sage: sqrt2 = sqrt(2)
```

assigns the value of $\sqrt{2}$ to the symbol `sqrt2`. After making this assignment, you can use the symbol `sqrt2` in any expression, and Sage will see it as $\sqrt{2}$. For example:

```
sage: sqrt2**2
2
```

Unlike programming languages that allow you to assign to only one variable at a time, Sage inherits Python's more flexible assignment, which allows you to assign to many variables in one go. For example,

```
sage: sqrt2, sqrt3 = sqrt(2), sqrt(3)
sage: sqrt2**2
2
sage: sqrt3**4
9
```

The first line in this sequence of statements assigns *both* $\sqrt{2}$ and $\sqrt{3}$ to the variables `sqrt2` and `sqrt3`, *in that order.* The two subsequent statements show that the assignments were indeed correct. Because of this, you can probably see that the statement

```
sage: sqrt2, sqrt3 = sqrt3, sqrt2
```

has the effect of swapping the values of `sqrt2` and `sqrt3`.

To create an indeterminate, Sage provides a command, `var()`. Type the name you'd like the indeterminate to have between the parentheses, *in quotes*. You can create several such indeterminates by listing them between the quotes, as well; just leave a space between each name. If successful, Sage will print the names of the newly created indeterminates between parentheses. For example:

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

You can then manipulate these variables to your heart's content.

```
sage: (y + z) + (z - y)
2*z
```

Valid identifiers. Each variable or indeterminate needs a valid identifier. Sage accepts identifier names using the following sequences of characters:

- The name must start with a letter (uppercase or lowercase) or the underscore (_).
- The name can contain any mix of letters, numbers, or the underscore.
- The name cannot be a *reserved word*, also called a *keyword*. You'll encounter these throughout the course, but it's unlikely you'd choose them.

Unlike mathematical convention, the names of variables and indeterminates can be longer than one symbol; we see this already in `pi`, which Sage offers at the start. In many cases, a name that is longer than one symbol can be understood much more readily than a name that is only one symbol; compare, for instance, `d` to `derivative`. On the other hand, if a name is *too* long and you use it repeatedly, it grows tiresome to type and can even make a program *harder* to understand. A mathematically literate person, for instance, would understand `ddx` just as well as `derivative`, and might well prefer the former to the latter. In general, we'd recommend somewhere around four to six characters in a name.

Expressions, and commands to manipulate them

A mathematical expression consists of any meaningful combination of mathematical symbols. One such expression is an *equation*, and in Sage you can assign equations as the values of variables. You definitely need to do this this from time to time, but the equals sign already has a meaning: it assigns the value of an expression to a symbol to create a variable! To refer to an equation, then, you use two equals signs in a row:

```
sage: eq = x**2 - 3 == 1
```

As before, if you forget to double the equals signs, Sage will give you an error:

```
sage: eq = x**2 - 3 = 1
SyntaxError:  can't assign to operator
```

If you see that error message, the problem is almost certainly due to the use of only one equals sign when you need two.

It is often useful to rewrite expressions in different ways, and a computer algebra system is, essentially, nothing more than a sophisticated tool for rewriting expressions. Here are some useful commands for rewriting expressions.

`factor(`*exp*`)`	factor the expression *exp*
`simplify(`*exp*`)`	simplify the expression *exp* a little bit
`expand(`*exp*`)`	perform multiplication and other operations on the expression *exp*
`round(`*exp*`, `*n*`)`	round the expression *exp* to *n* places after the decimal

TABLE 4. *Some* commands to manipulate expressions (there are a lot more)

Some words to the wise. The traditional mathematical symbol for multiplication is the simple, raised $\times$ or the dot symbol $\cdot$. Alas, computer keyboards have neither symbol by default; you generally have to use a workaround if you want to type the symbol. The traditional workaround is to type the asterisk, and that is what Sage uses.

It is common in mathematics to omit the multiplication symbol: $2x$, for instance, or $abcd$. You can't do this in Sage; that will give you various errors:

```
sage: 2x
SyntaxError:  invalid syntax
sage: var('a b c d')
(a, b, c, d)
sage: abcd
NameError:  name 'abcd' is not defined
```

In both cases, Sage thinks you're trying to type the name of an identifier, which name it doesn't recognize:

- For `2x`, you cannot start an identifier name with a number so the error is in syntax.
- For `abcd`, Sage has no practical way of knowing that you mean the products of `a`, `b`, `c`, and `d`, rather than a different identifier named `abcd`, so the error is in fact one of the name. (Remember that in Sage, unlike most mathematics, variables and indeterminates can have names longer than one symbol.)

Both of these expressions work fine if you type the multiplication symbol where you want it to be.[18]

Transcendental functions. Sage offers everything a scientific calculator offers. Here are some common transcendental functions that you will find useful throughout this text and in your math curriculum:

[18] In some versions of Sage there is a way to make multiplication work without explicitly writing the multiplication symbol in these circumstances, but it is not available by default, so you'd have to run a special command for it. This is a bad idea for beginners, and you'd still have to type a space between the symbols in any case, so we will not describe that technique.

`sin(`a`)`, `cos(`a`)`, `tan(`a`)`, `cot(`a`)`, `sec(`a`)`, `csc(`a`)`	the trigonometric functions, evaluated at a
`arcsin(`a`)`, `arccos(`a`)`, `arctan(`a`)`, `arccot(`a`)`, `arcsec(`a`)`, `arccsc(`a`)`	the inverse trigonometric functions, evaluated at a
`exp(`a`)`	e^a (synonym for `e**a`)
`ln(`a`)`, `log(`a`)`	the natural logarithm of a
`log_b(`a`, `b`)` or `log(`a`, `b`)`	the logarithm base b of a

TABLE 5. Sage commands for common transcendental functions

We can also compute the hyperbolic trigonometric functions, and their inverses, by appending `h` at the end of the usual name, and before the left parenthesis.

You may have noticed that `log(`a`)` is synonymous for `ln(`a`)`. This is not the usual custom in American textbooks, where `log(`a`)` is synonymous for $\log_{10} a$. To compute what Americans call the "common" logarithm, we have to use the `log_b()` command:

```
sage: log(100)
log(100)
sage: log_b(100,10)
2
```

Be careful when trying to use these functions in larger expressions. It is common in mathematics to write $\sin^2 \pi/4$ when you mean $(\sin(\pi/4))^2$. Sage does not make this distinction. There is really only one way to do it right, and that is to write what you mean:

```
sage: sin^2(pi/4)
TypeError:  'sage.rings.integer.Integer' object is not callable
sage: (sin^2)(pi/4)
TypeError:  unsupported operand type(s) for ** or pow():
'Function_sin' and 'int'
sage: (sin(pi/4))^2
1/2
```

When you see *`TypeError`* with these messages, it's a safe bet you've "typed" something wrong.[19]

- The first kind of `TypeError` should be easy to debug: look for a place where a number is followed immediately by the opening of parentheses — in our case, `2(pi)/4`. When this happens, Sage thinks you're trying to use `2` as a function. The way Sage evaluates expressions, it sees $\sin^{2(\pi/4)}$, and while it might not seem sensible to us to view $2(\pi/4)$ as a function 2 evaluated at the point $\pi/4$, that's how Sage views it.

[19] Technically, Sage is referring to the "type" of object, not to what you've "typed," but we're desperate enough to roll with the pun.

- The second kind of `TypeError` can be harder. In general, the problem is that you're trying to perform an operation with two objects for which Sage doesn't know how to perform the operation. Nine times out of ten, you've typed something wrong, so re-examine what you've typed. In this case, you've typed a convenient shorthand, which Sage doesn't understand (with reason).

Mathematical functions and substitution. A useful aspect of indeterminates is that you can substitute values into them. Unlike assigning a value to a variable, indeterminates don't retain that value after the computation. We use three ways to substitute in Sage.

The first way is via the `subs()` method, which is shorthand for the `substitute()` method. Recall that a *method* is a command that is specific to an object. You access this feature in the following way:

- Type the expression's name, then a dot, then `subs(` .
- After the parenthesis, list the assignments. There are two ways to do this:
  - List each assignment as an equation: *indeterminate=value*.[20]
  - List each assignment as a "dictionary". To do this, open a pair of braces, list the assignments in the form `indeterminate:value`, separating each assignment by a comma, then close the braces.
- Now close the parentheses that began after `subs`: `subs(`*assignments*`)`.

Here is an example:

```
sage: f = x**2
sage: f.subs(x=2)
4
sage: f.subs({x:2})
4
```

The first substitution illustrates assignment via equations; the second illustrates assignment via dictionaries. The second approach is the most reliable, error- and warning-free way to substitute into any mathematical expression that contains indeterminates.[21]

You can actually do this without specifying the method's name!

```
sage: f(x=2)
4
sage: f({x:2})
4
```

You *should not* use this approach without specifying the *indeterminate's* name. If you have only one indeterminate, as with `x` above, you may forget that you need to name it. In this case, Sage will issue what's called a warning:

[20] This does not always work. We will return to the topic later.

[21] In particular, the second approach works even inside user-defined functions, when you want to pass an indeterminate as an argument to the function. We talk about this later on.

```
sage: f(2)
DeprecationWarning:  Substitution using function-call syntax
and unnamed arguments is deprecated and will be removed from a
future release of Sage; you can use named arguments instead, like
EXPR(x=..., y=...)
4
```

This warning might not appear on the last line. It's not an error, and Sage manages to guess the correct substitution and outputs the correct answer after the warning. In addition, a nice thing about `DeprecationWarning`s is that they appear only once, even if you keep making the same "mistake." All the same, it's a bit scary to see it there, and it's entirely possible that the threat to remove this guess-and-go feature will come to pass,[22] so try not to make that mistake. This can be a real problem if you provide too many values; Sage won't know what to assign where, and will complain:

```
sage: f(3,2,1)
ValueError:  the number of arguments must be less than or equal to
1
```

This won't happen when you specify assignments, as Sage knows what goes where:

```
sage: f = x^2 - y^2
sage: f(x=3,w=2,z=1)
-y^2 + 9
```

The third way to substitute is useful when you're using a mathematical expression called a *function*. Recall that a function maps elements of one set, called the *domain*, to elements of another set, called the *range*, in such a way that every input has a well-defined output. Defining and using functions is straightforward in Sage:

```
sage: f(x) = x**2
sage: f(2)
4
```

Functions also have the useful property that they define any indeterminate that appears between the parentheses of their definition. You can then use those indeterminates in other contexts without first using the `var()` command.

[22] The warning has appeared for quite a few years now.

`lim(`f`, x=`a`)` *or* `limit(`f`, x=`a`)`	compute the two-sided limit of $f(x)$ at $x = a$
`lim(`f`, x=`a`, dir=`*direction*`)` *or* `limit(`f`, x=`a`, dir=`*direction*`)`	compute the one-sided limit of $f(x)$ at $x = a$, with *direction* one of `'left'` or `'right'`
`diff(`f`, x)` *or* `derivative(`f`, x)`	compute the derivative of $f(x)$ with respect to x
`diff(`f`, x, m)` *or* `derivative(`f`, x, m)`	compute the mth derivative of $f(x)$ with respect to x
`integral(`f`, x)` *or* `integrate(`f`, x)`	compute the indefinite integral (antiderivative) of $f(x)$ with respect to x
`integral(`f`, x, a, b)` *or* `integrate(`f`, x, a, b)`	compute the definite integral over $[a, b]$ of $f(x)$ with respect to x

TABLE 6. Commands for exact computation of limits, derivatives, and integrals

```
sage: f(w,z) = 4*w**2 - 4*z**2
sage: f(3,2)
20
sage: factor(w**2 - z**2)
(w + z)*(w - z)
```

Notice that we were able to work with `w` and `z` directly, even though we didn't define them.

Yer basic Calculus

We now turn to the question of Sage's offerings for the Calculus.

Exact computation. We look at exact computation first. Sage offers three commands that perform the exact computation for calculus. You can see in Table 6 these commands, with both synonyms and different usage options:

- For limits, we have the synonyms `lim()` and `limit()`, which we can use to compute either two-sided limits (default) or one-sided limits (specify a direction). You must specify both the indeterminate (x) and the value (a).
- For differentiation, we have the synonyms `diff()` and `derivative()`, which we can use either to differentiate once or, by specifying an optional argument, to differentiate multiple times.
- For integration, we have the synonyms `integral()` and `integrate()`, which we can use to compute either the indefinite integral, also called the antiderivative, or if we specify the limits of integration, to compute the definite integral, which you probably first learn as the area under a function.

Let's take a quick look at how these work.

Limits. First up is limits. Roughly speaking,

- $\lim_{x \to a^+} f(x)$ is the y-value approached by f as x approaches a from the right (that is, from numbers larger than a);
- $\lim_{x \to a^-} f(x)$ is the y-value approached by f as x approaches a from the left; and
- $\lim_{x \to a} f(x)$ is the y-value approached by f as x approaches a from both sides.

If a function is continuous at a, we can find the limit by substitution, which is boring.

```
sage: f(x) = x^2 + 1
sage: limit(f(x), x=1)
2
sage: f(1)
2
```

Limits are much more interesting when the function is discontinuous at a. Here are some problems you should remember from your Calculus courses:

```
sage: limit((x**2 - 1)/(x - 1), x=1)
2
sage: limit(x/abs(x), x=0)
und
sage: limit(x/abs(x), x=0, dir='left')
-1
sage: limit(1/x, x=0, dir='left')
-Infinity
```

The first example shows how Sage automatically detects and works around division by zero, when possible. The second example shows what can go wrong when there is no way to work around it: *und* is shorthand for, "the limit is undefined." The third and fourth examples show how to compute one-sided limits in the case where two-sided limits do not exist.

Speaking of $1/x$, here is a two-sided result you might not expect:

```
sage: limit(1/x, x=0)
Infinity
```

If you know the correct answer, then upon seeing that you might be tempted to

PANIC!

…but you shouldn't. Do *not* misread that answer. Sage is not claiming the result is ∞; it actually has a separate symbol for that.

```
sage: limit(1/x, x=0, dir='right')
+Infinity
```

Notice that this is a *signed* infinity, whereas the previous result was unsigned. What the unsigned infinity indicates[23] is that "the limit of the absolute value of the expression is positive infinity, but the limit of the expression itself is not positive infinity or negative infinity." [8]

To avoid this, the best thing to do is probably to evaluate your limits from each side manually; in this case, Sage reports `+Infinity` or `-Infinity` as appropriate. You can also check whether the limit produces `unsigned_infinity`:

```
sage: limit(1/x, x=0) == unsigned_infinity
True
```

…but in that case you'll probably want to check the left- and right-hand limits, anyway.

Derivatives. Recall that the derivative of $f(x)$ at $x = a$ is

- the slope of the line tangent to $f(x)$ at $x = a$ (if such a line exists); or, equivalently,
- the limit of the slopes of the secant lines connecting f at $x = a$ and $x = b$, as $b \to a$; or, equivalently,
- the limit of the slopes of the secant lines connecting f at $x = a$ and $x = b$, as the distance between a and b approaches 0 ($\Delta x \to 0$); or, equivalently,
- $\lim_{\Delta x \to 0} \dfrac{f(a+\Delta x) - f(a)}{\Delta x}$.

We can also talk about the derivative as a function; that is, $f'(x)$ is

- the value of $f'(a)$ whenever $x = a$; or, equivalently,
- $\lim_{\Delta x \to 0} \dfrac{f(x+\Delta x) - f(x)}{\Delta x}$, which you probably spent a lot of time manipulating in your calculus class with an h instead of a Δx.

Sage's `diff()` and `derivative()` functions compute the derivative as a function; if you want to compute the derivative at a point, define a function and substitute.

```
sage: diff(x**2, x)
2*x
sage: df(x) = diff(x**2, x)
sage: df(1)
2
sage: diff(cos(x), x, 1042)
-cos(x)
```

That last example gave us the 1042th derivative of $\cos x$. That would take an awful long time to do by hand, unless you happened to notice a pattern.

Integrals. The word "integral" has two different meanings.

The *indefinite* integral of $f(x)$ is its antiderivative; that is, $\int f(x)\,dx = F(x)$ where F is any function such that $F'(x) = f(x)$. There are actually infinitely many such antiderivatives; an

[23] The subsequent quote is taken from the documentation on Maxima, the subsystem Sage uses to evaluate limits. At the time of this writing, Sage's documentation lacks this information, and can confuse the user by simplifying `(Infinity == +Infinity).full_simplify()` to `True`. The trouble is that `Infinity` has one meaning when it appears in the output of `limit()`, and another meaning altogether when you type it at a command line.

important result of calculus is that any two of them differ only by a constant. You usually resolve this in calculus by adding a "constant of integration" to your integral; for example,

$$\int \cos 2x \; dx = \frac{1}{2}\sin 2x + C.$$

As you will see, Sage omits the constant of integration.[24]

The *definite* integral of $f(x)$ over the interval I is the limit of weighted sums over n subintervals of I as n approaches ∞. Speaking precisely,

$$\int_I f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x, \quad \text{where } x_i \text{ is on the } i\text{th subinterval of } I.$$

The interval can be either finite — $[a,b]$ — or infinite — $[a,\infty)$ or $(-\infty,b]$ — so long as the integral converges and is not improper. If it is improper, you have to break it into pieces; for example,

$$\int_{-1}^{1} \frac{1}{x^3}dx = \lim_{t\to 0^-} \int_{-1}^{t} \frac{1}{x^3}dx + \lim_{t\to 0^+} \int_{t}^{1} \frac{1}{x^3}dx.$$

Sage, though, can handle such integrals without your breaking them apart.

You have seen from the table that Sage allows you to compute both indefinite and definite integrals.

```
sage: integrate(x**2, x)
1/3*x^3
sage: integrate(x**2, x, 0, 1)
1/3
sage: integrate(1/x**(1/3), x, -1, 1)
0
sage: integrate(1/x, x, 1, infinity)
ValueError:  Integral is divergent.
```

These seems straightforward:

- the first gives us $\int x^2 dx$;
- the second gives us $\int_0^1 x^2 dx$;
- the third gives us $\int_{-1}^{1} 1/\sqrt[3]{x}\; dx$, which is improper due to an asymptote at $x = 0$, though Sage handles it with ease; and
- the fourth gives us $\int_1^\infty 1/x\; dx$, which does in fact diverge; you need a power larger than 1 in the denominator to converge.

Sage itself is aware of this latter property; we show this over a series of steps.

[24] Your calculus instructor would be appalled, absolutely appalled.

```
sage: var('p')
p
sage: integrate(1/x**p, x, 1, infinity)
ValueError:  Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may*
help (example of legal syntax is 'assume(q>0)', see 'assume?'  for
more details)
Is q positive, negative or zero?
```

Oops! That's a perfectly sensible request. The error message is also helpful: it introduces a new command to us, the `assume()` command. We won't make very much use of this, but this is one case where it comes in handy. Let's assume $p > 1$:

```
sage: assume(p > 1)
sage: integrate(1/x**p, x, 1, infinity)
1/(p - 1)
```

Sage asserts that

$$\int_1^\infty \frac{1}{x^p}\,dx = \frac{1}{p-1}$$

whenever $p > 1$, a fact that you should be able to verify by hand. Now, if we assume that p is smaller than 1, we encounter two problems. The first you might not expect:

```
sage: assume(p <= 1)
ValueError:  Assumption is inconsistent
```

If you really want to change this, you can; use the `forget()` command to forget anything you `assume()`'d.[25] We really do want to change this, so,

```
sage: forget()
sage: assume(p <= 1)
sage: integrate(1/x**p, x, 1, infinity)
ValueError:  Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may*
help (example of legal syntax is 'assume(p>0)', see 'assume?'  for
more details)
Is p positive, negative or zero?
```

That may surprise you, but not for long: if $p \leq 0$ then we're looking at x^q with $q \geq 0$, whereas $p > 0$ is what we had in mind. Since Sage can't read our minds, let's add the assumption:

[25] You can also `forget()` only part of what you assumed. So aside from the use of `forget()` as we have shown it, you could type `forget(p <= 1)`. This would be especially useful if several assumptions were in play, and you wanted to forget only one.

```
sage: assume(p>0)
sage: integrate(1/x**p, x, 1, infinity)
ValueError:  Integral is divergent.
```

Although it reports an error, we should consider this error a success, as it verifies what we already knew.[26]

(Don't forget to `forget()` when you're done `assume()`ing.)

Numerical integration. Integrals feature a twist that derivatives do not: you cannot always compute an "elementary form" of an integral.[27] Consider, for instance, $\int e^{x^2}\,dx$.

```
sage: integrate(e**(x**2), x)
-1/2*I*sqrt(pi)*erf(I*x)
```

You have probably not seen $\operatorname{erf}(x)$ before, and that's fine.[28] The point is that it's not an elementary function. If you try to get help on it in Sage, you will see the following:

```
sage: erf?
...
erf(x) = frac{2}{sqrt{pi}} int_0^x e^{-t^2} dt.
...
```

In other words,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt.$$

So there's no way to simplify this further.

The following example appears when you try to compute the arclength of an ellipse centered at the origin, with horizontal axis of length 2 and vertical axis of length 1.

```
sage: f(x) = sqrt(1 - x^2/4)
sage: df(x) = diff(f, x)
sage: integrate(sqrt(1+(df(x))**2), x, -2, 2)
integrate(sqrt(-1/4*x^2/(x^2 - 4) + 1), x, -2, 2)
```

[26] It may come as a surprise that Sage requests us to assume that $p > 0$ when, in fact, the integral diverges for $p \leq 0$, as well. From the *mathematical* point of view, the only assumption we need is $p \leq 1$. Branching through all the special cases can be too complicated to implement in computer science, however, so the user must frequently work out some of these things on his own.

[27] The precise definition of an "elementary form" is beyond the scope of this text, but you can basically think of it as any algebraic combination of the functions you studied in precalculus, including trigonometric, exponential, logarithmic, and hyperbolic functions.

[28] For those who are interested, it's the Gaussian error function, and you're likely to see it in a probability course.

In this case, Sage's answer to the integral is just another integral. That doesn't seem especially helpful, but there's really not much Sage can do.[29] The integral does not reduce to elementary terms.

In some cases, an integral *does* reduce to elementary terms, but you get something so complicated that you might well prefer not to work with it. Here's an example.

```
sage: integrate(x^10*cos(x), x)
10*(x^9 - 72*x^7 + 3024*x^5 - 60480*x^3 + 362880*x)*cos(x) + (x^10
- 90*x^8 + 5040*x^6 - 151200*x^4 + 1814400*x^2 - 3628800)*sin(x)
```

…and that's not even that bad.

In many cases, you really just want a numerical value for the integral. Perhaps you want the area, or some accumulation of values, or some other application. In this case, you don't have to go through the indefinite integral; you can approximate the definite integral using one of the techniques of numerical integration. Sage's command for this is `numerical_integral()`.

`numerical_integral(`f`, `a`, `b`)`	estimate $\int_a^b f(x)\,dx$
`numerical_integral(`f`, `a`, `b`, max_points=`n`)`	estimate $\int_a^b f(x)\,dx$ using no more than n points

Its usage requires a little explanation, but before we do that, let's consider an example.

```
sage: numerical_integral(sqrt(1+(df(x))**2), -2, 2)
(4.844224058045445, 4.5253950830572916e-06)
```

Right away you should notice several differences.

- We do *not* specify the variable of integration for `numerical_integral()`.
- The result consists of two numbers. The second one is in a kind of scientific notation, which in this case you should recall as being approximately 4.525×10^{-6}.

What we're receiving as our answer here is an ordered pair. The first value is Sage's approximation of the integral; the second is an estimate of the error. In other words, our answer is certainly correct up to the 4th decimal place, though rounding prompted by the 6th decimal place (where the error may occur) creates some ambiguity in the 5th. We can say that the integral lies within the interval $(4.8442195, 4.8442286)$:

```
sage: A, err = numerical_integral(sqrt(1+(df(x))**2), -2, 2)
sage: A - err, A + err
(4.844219532650363, 4.844228583440528)
```

[29] At least one other computer algebra system will answer something to the effect of `EllipticE(...)`, which looks promising until you read the documentation on it. Like `erf(...)`, it just restates the original integral. In this case, the name reflects that it's an *elliptic integral*.

Mathematical structures in Sage

This is an opportune time to introduce you to one of the more powerful features of a computer algebra system: its ability to work in different mathematical contexts. In this course we will focus especially on *rings* and *fields:*

A ring: is a set where addition and multiplication behave according to the properties you'd expect:

Closure: Adding or multiplying two elements of the ring gives another element of the ring.

Associative: The result of adding or multiplying three elements doesn't depend on which two you add or multiply first: $a+(b+c)=(a+b)+c$ and $a(bc)=(ab)c$.

Identities: You can find two elements "0" and "1", for which adding "0" changes no element of the ring, and multiplying by "1" changes no element of the ring: $a+0=a=0+a$ and $a\times 1=a=1\times a$.

Distributive: You can distribute multiplication over addition: $a(b+c)=ab+ac$.

Addition enjoys two additional properties:

Commutative addition: The result of adding two elements of the ring doesn't depend on their order in the sum: $a+b=b+a$.

Additive inverses: You can find the "opposite" of any element, so that adding them returns "0": $d+b=0=b+d$. We usually denote this opposite as a negative, so that $a+(-a)=0=(-a)+a$.[30]

Notice that a ring *might not* have commutative multiplication or multiplicative inverses. Omitting this allows us to organize sets of matrices as rings, as matrices satisfy the properties listed above, but do not satisfy the other two.

A field: is a commutative ring where you can also "divide," by which we mean, you can find a "multiplicative inverse" for every nonzero element, by which we mean, for each nonzero a you can find a b *in the same ring* such that $ab=1$. The analogy to division is only that: an analogy; we *do not* typically use the division symbol outside of the systems you're accustomed to; we write them with an exponent of -1: rather than a/b, we write $a\cdot b^{-1}$.

The usual suspects. You have already spent *a lot* of time working in rings and fields, though you probably were not told this. Sage offers you a number of rings and fields directly:

[30] You can also think of this property as "closure under subtraction," but people don't usually speak of it in such terms.

Set	Traditional mathematical symbol	Structure	Sage symbol
Integers	$\mathbb{Z}$	ring	`ZZ`
Rational numbers (integer fractions)	$\mathbb{Q}$	field	`QQ`
Real numbers	$\mathbb{R}$	field	`RR` (but see below)
Complex numbers	$\mathbb{C}$	field	`CC` (but see below)

TABLE 7. Common rings and fields in Sage

When you type some value, Sage makes an educated guess as to which type of object you want. You can find this using a `type()` statement:

```
sage: a = 1
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: b = 2/3
sage: type(b)
<type 'sage.rings.rational.Rational'>
sage: c = 2./3
sage: type(c)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: d = 1 + I
sage: type(d)
<type 'sage.symbolic.expression.Expression'>
```

That last one is not actually in the complex ring $\mathbb{C}$, as we'd expect! While $1+i$ is in fact a symbolic expression, and that will suffice for most of our purposes, we can compel Sage to view it as an element of $\mathbb{C}$. We do this by typing `CC`, followed by parentheses, in which we type the complex number we want to work with. The drawback is that we lose the precision of *symbolic* computation:

```
sage: d2 = CC(1+I)
sage: type(d2)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: d3 = d2 + (1 - I)
sage: d3
2.00000000000000
```

Do you see what happened? Instead of receiving the *exact* value 2, we received an approximation of 2. In this case, the approximation looks exact, but that's beside the point; work it hard enough, and error will start to creep in. For instance:

```
sage: d3 - 1.999999999999
1.00008890058234e-12
```

Why did this happen? In order to compute effectively in the real and complex fields, Sage resorts to approximations of the numbers involved. You'll see this right away if you look at `d2`:

```
sage: d2
1.00000000000000 + 1.00000000000000*I
```

The very process of converting `d2` into a "complex" number means we have surrendered some precision. The same will happen if you work in `RR`. Sometimes you just have to do that, but it's a fact you have to keep in mind. Forgetting it can lead to puzzling results, such as:

```
sage: 2.99 - 2.9
0.0900000000000003
```

Oops!

It is often useful to separate the real and imaginary parts of a complex number. Sage provides two useful commands to do this. They work whether the complex number is a symbolic expression or the number lies in `CC`.

`real_part(z)`	real part of z
`imag_part(z)`	imaginary part of z

```
sage: real_part(12-3*I)
12
sage: imag_part(CC(12-3*I))
-3.00000000000000
```

The norm of a complex number is also available.

`norm(z)`	norm of z

If you are unfamiliar with the norm of a complex number, it is comparable to the absolute value of a real number, in that it gives an idea of the number's size. Speaking precisely, the norm is

$$||a+bi|| = a^2+b^2.$$

If this reminds you of the Pythagorean Theorem, it should, as it is nearly identical to the Euclidean distance formula.

```
sage: norm(2+3*I)
13
```

You can sometimes convert from reals to integers, but not always.

```
sage: a = ZZ(1.0)
sage: a
1
sage: b = ZZ(1.2)
TypeError:  Attempt to coerce non-integral RealNumber to Integer
```

Another symbol you will encounter from time to time is $\mathbb{N}$, the set of nonnegative integers, also called the **natural numbers**. This is not a ring because, for instance, $1 \in \mathbb{N}$ but $-1 \notin \mathbb{N}$.

The unusual suspects. A *lot* of the applications of mathematics of the last half-century arise in the context of "modular arithmetic." Without going into too much detail, modular arithmetic consists of performing an operation, then taking the remainder after division by a fixed number, called the modulus. A very old example of this arises when dealing with time:

```
sage: current_hour = 8
sage: hours_busy = 20
sage: time_free = current_hour + hours_busy
sage: time_free
28
```

The problem here is obvious: there is no "28th" hour of the day. We can find the correct time by dividing by 12, and taking the remainder:

```
sage: time_free = (current_hour + hours_busy) % 12
sage: time_free
4
```

It is now obvious that the individual in question is not free until 4:00.

Explicitly asking for modular arithmetic in each operation is burdensome. It turns out that modular arithmetic makes for a perfectly acceptable ring, so that we can add one new row to our table of rings and fields:

Set	Traditional mathematical symbol	Structure	Sage symbol
Integers modulo n ($n > 1$)	$\mathbb{Z}_n$	ring	`ZZ.quo(n)`

TABLE 8. Common rings and fields in Sage

Let's see how this works in practice. We'll test our previous problem by working in $\mathbb{Z}_{12}$.

```
sage: Z_12 = ZZ.quo(12)
sage: current_hour = Z_12(8)
sage: hours_busy = Z_12(20)
sage: time_free = current_hour + hours_busy
sage: time_free
4
```

The advantages to this approach will not look as apparent here as they are in practice, but they really are there: exponentiation by large numbers, for instance, is much faster when we go this way. Because modular arithmetic is a powerful tool, we will have you work with it quite a lot from here on.

Exercises

True/False. If the statement is false, replace it with a true statement.

1. Although it works for exponentiation, you should avoid using the `^` symbol in that context.
2. `6 // 4 == 1 and 6 % 4 == 2`.
3. Although i is a fixed mathematical constant and Sage provides the symbol `I` to represent it, you cannot always count on the equation `I**2 == -1` to be true in Sage.
4. You use the `var()` command to create new variables. You create new indeterminates by assigning values to them.
5. If you assign `e = 7` and later want to restore `e` to its original value, so that `ln(e) == 1`, use `reset('e')`.
6. The name `l337` satisfies the requirements for a Sage identifier. (The first symbol is a lower-case L.)
7. The name `1337` satisfies the requirements for a Sage identifier. (The first symbol is the number 1.)
8. In Sage, `log(a) == ln(a)` for any value of a.
9. Before defining a function with the indeterminate `w`, you have to define the indeterminate `w`.
10 If a one-sided `limit()` returns `+Infinity`, then the actual limit is ∞.
11. If a two-sided `limit()` returns `Infinity`, then you need to check the limit from both sides.
12. Sage can simplify all integrals to elementary form using the `integral()` command.
13. An indefinite integral has infinitely many solutions, but Sage provides only one.
14. Sage cannot handle definite integrals when there is an asymptote in the interval.
15. `Z_12(12) == 0` , where `Z_12` is defined as above.
11. *Bonus:* The answer to all these True/False questions is "False."

Multiple Choice.

1. If you encounter an `AttributeError` while working in Sage, which of the following is *not* an appropriate course of action?
   A. Ask your instructor.
   B. Look in the index to see if something like it is covered somewhere in the text.
   C. PANIC!
   D. When all else fails, inquire at the `sage-support` website.
2. For which of the following expressions does Sage *seem* to return a nonzero value?
   A. `e**(I*pi) + 1`

B. `I**2 + 1`
C. `(cos(pi/3)+I*sin(pi/3))**6 - 1`
D. `sin(pi/4)**2 + cos(pi/4)**2 - 1`

3. Which of the following options does Sage *not* offer, even though many programming languages do?
   A. constants
   B. variables
   C. identifiers
   D. indeterminates
4. Which of the following identifier names is probably not the wisest idea for the ith value of a in a sequence?
   A. `a`
   B. `a_curr`
   C. `ai`
   D. `a_i`
5. Which of the following identifier names might not be the wisest idea for the value of a function at a point x_0?
   A. `y0`
   B. `y_0`
   C. `function_value`
   D. `yval`
6. One way to compute an approximate value of $\log_{10} 3$ in Sage by typing `log_b(3.,10.)`. Which of the following would also work?
   A. `log(3.)`
   B. `log(3.)/log(10.)`
   C. `log(10.)/log(3.)`
   D. `log(10.^3)`
7. Which of the following methods of substituting `2` for `x` in a mathematical *function* `f` is guaranteed to work in every circumstance?
   A. `f(2)`
   B. `f(x=2)`
   C. `f.subs(x=2)`
   D. `f({x:2})`
8. Which of the following methods of substituting `2` for `x` in a mathematical *expression* `f` is guaranteed to work in every circumstance?
   A. `f(2)`
   B. `f(x=2)`
   C. `f.subs(x=2)`
   D. `f({x:2})`
9. Which of the following would you expect as the result of the command, `limit(x/abs(x), x=0)`?
   A. `Infinity`
   B. `+Infinity`
   C. `ind`
   D. `und`

10. Which of the following would you expect as the result of the command, `limit(1/(x-1), x=1, dir='right')`?
    A. `Infinity`
    B. `+Infinity`
    C. `ind`
    D. `und`
11. Which of the following would you expect as the result of the command, `limit(sin(1/x), x=0, dir='right')`?
    A. `Infinity`
    B. `+Infinity`
    C. `ind`
    D. `und`
12. Sage will do your Calculus homework for you on the following integrals:
    A. Exact integrals, because Sage can simplify every integral to elementary form.
    B. Approximate integrals, because Sage only provides one answer for an indefinite integral, rather than all possible answers.
    C. Any kind of integral, because Newton figured those out centuries ago, though computers have only now become efficient enough to do them.
    D. None of them, because a good Calculus teacher checks the steps, not just the answer, and Sage doesn't show the steps.[31]
13. Which of the following is not a guaranteed property of a ring?[32]
    A. Closure of multiplication
    B. Existence of a smallest element
    C. Existence of an additive identity
    D. Closure of subtraction
14. Which of the following is not a good reason to perform arithmetic integers modulo n?
    A. Confusing other people
    B. Modeling "real world" problems
    C. Faster solution to some problems
    D. Mathematical curiosity
15. If `R` is a variable that references a ring, which command will compel Sage to view the number referenced by the variable `a` as an element of `R`, rather than as an integer?
    A. `R(a)`
    B. `R a`
    C. `a:  R`
    D. `R: a`

Bonus: Which answer is correct?
    A. All of them.
    B. Some of them.
    C. At least one of them.
    D. Any that is not wrong.

Short answer.

1. Describe another situation where modular arithmetic would come in useful.

[31] . . . and every Calculus teacher is good, *right?*

[32] Be careful on this one; we're being a bit tricky — but only a bit.

2. For Multiple Choice question #2, the full simplification is in fact zero for all four answers, but that Sage doesn't perform that simplification automatically. What command(s) could you issue to make Sage detect this?
3. Let's revisit this Sage interaction that appears earlier in the notes.

```
sage: d2 = CC(1+I)
sage: type(d2)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: d3 = d2 + (1 - I)
sage: d3
2.00000000000000
sage: d3 - 1.999999999999
1.00008890058234e-12
```

   Sage reports that the difference between `d3` and `1.999999999999` is $1.00008890058234 \times 10^{-12}$. This is slightly wrong; the correct answer should be a clean 1×10^{-12}.
   (a) Describe a sequence of commands that use exact arithmetic to give us the correct answer.
   (b) Why might someone not care very much that the approach using the ring `CC` is slightly wrong?
4. In a number of situations, you have to raise a number a to a large exponent b, modulo another value n. Choose a not-too-large number a (double digits is fine) and raise it to larger and larger exponents b, modulo n using the `%` operator, until you see a noticeable slow down. (Don't get too carried away, as once it slows down, it *really* starts to slow down.) Then compare that operation when Sage performs it in the ring $\mathbb{Z}_n$. Write down these values for a, b, and n, and indicate whether it really is faster using the ring than using the `%` operator.
5. We claimed only that $\mathbb{Z}_n$ is a ring, but for some values of n it's actually a field. There is no question of whether multiplication is commutative (it is) but whether every element has a multiplicative inverse is not so clear. Check this for the values $n = 2, 3, 4, 5, 6, 7, 8, 9, 10$ by testing as many products ab as needed to decide the question.
   (a) Do you see a pattern in the values of n for which $\mathbb{Z}_n$ is a field?
   (b) When $\mathbb{Z}_n$ is *not* a field, do you see a pattern in the values of a for which you can find a multiplicative inverse b?

   Hint: We don't require a multiplicative inverse for 0, 1 is its own inverse (after all, $1 \times 1 = 1$), and in $\mathbb{Z}_n$ we conveniently have $n = 0$. Along with the commutative property, this means you need check at most $\frac{(n-2)(n-1)}{2}$ products; with $n = 10$ you need check only 35. So if you're clever about it, it's not as burdensome a problem as you might think.

Bonus: Why were we able to say, with confidence, that in #5 we need at most $\frac{(n-2)(n-1)}{2}$ products to determine whether $\mathbb{Z}_n$ is a field?

Pretty (and not-so-pretty) pictures

Visualization often provides insights we do not obtain by other methods. This doesn't apply merely to mathematics; hence, the saying, "A picture is worth a thousand words." Many of the exercises and labs we assign will ask you to produce some pictures, then draw conclusions from them.

Sage offers a very nice set of tools to draw graphs, along with an intuitive interface for using them. This chapter takes a look at the two-dimensional objects you might have to use. Everything we do here is strictly Cartesian geometry; that is, we assume that we're working on a Cartesian plane.

Starting in this chapter, we won't merely list commands with a short explanation; rather, we'll list the commands *and their options.* Keep in mind that the description we give of the options may not be complete:

- We focus only on a few aspects of the command that we think are most useful for the tasks at hand and for what you'd need in the future.
- Sage is bound to change in the future, and it's unlikely *every* Sage command would long remain the same as what it is now.

Remember that every Sage command will offer you a fuller explanation and list of options if you simply type its name, followed by the question mark. For instance:

```
sage: point?
Signature:      point(points, **kwds)
Docstring:
   Returns either a 2-dimensional or 3-dimensional point or sum of
   points.
```

...and so forth.

2D objects

We start with a look at some fundamental two-dimensional objects.

"Straight" stuff. You can illustrate an awful lot of mathematics just by looking at points and line segments. Sage offers three commands to graph such things: `point()`, `line()`, and `arrow()`. Strictly speaking, `line()` produces a *crooked* line *segment,* not necessarily a *straight line.*

Every description starts with a template for the command, listing required information and optional information (usually named *options*), followed by a bulleted list describing the information. You don't necessarily have to give all the information, so for each piece of information, we add in parentheses the value Sage assumes for that information when you omit it.

Both commands refer to "collections." We cover collections in depth later on, but for now you can use tuples, a sequence of objects enclosed in parentheses. For instance, `(2,3)` is a tuple of

`point(` *position*`,` *options*`)`

- *position* is
  - an ordered pair, *or*
  - a collection of ordered pairs
- *options* include:
  - `pointsize =` *size* (10)

FIGURE 1. `point()`

`line(` *points*`,` *options*`)`

- *points* is a collection of ordered pairs
- *options* include:
  - `thickness =` *thickness* (1)

FIGURE 2. `line()`

`arrow(` *tailpoint*`,` *headpoint*`,` *options* `)` *or* `arrow(` *path*`,` *options* `)`

- *tailpoint* is an ordered pair; the arrow starts here
- *headpoint* is an ordered pair; the arrow ends here
- *options* include:
  - `arrowsize =` *arrowhead's size* (5)
  - `head =` *arrowhead's location* (1)
    - 0 means "at tailpoint"
    - 1 means "at headpoint"
    - 2 means "both endpoints"
  - `width =` *stem width* (2)

FIGURE 3. `arrow()`

The following options are common to all 2D objects:

- `alpha=` *transparency* (1.0)
- `color=` *color, which we discuss in a dedicated section* (`'blue'` or `(0,0,1)`)
- `linestyle=` *drawing style of the line, which can be* `'dashdot'`, `'dashed'`, `'dotted'`, or `'solid'` (`'solid'`)
- `zorder=` *distance to viewer, compared to other objects* (0)

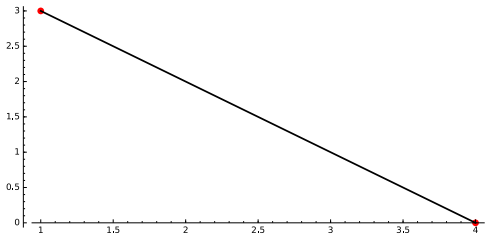
FIGURE 4. Options common to all 2D objects

integers. In this case, there are only two elements in the tuple, so we also call it an ordered pair. Tuples can be much longer: `(x, x**2, x**3, x**4)` is a tuple of symbolic expressions, though *not* an ordered pair.

Figure 4 lists some common options for all two-dimensional objects. In this section, we'll discuss all the options but `color`; we dedicate a special page to that.

In addition to illustrating the objects, our first example will show how intuitive it is to combine several images into one: simply add them with the `+` operator. To aid in readability, we will assign several objects to variables and combine them at the end.

```
sage: p1 = point(((1,3),(4,0)), color='red', \
                 pointsize=60)
sage: p2 = line(((1,3),(4,0)), color='black', \
                 thickness=2)
sage: p1 + p2
```

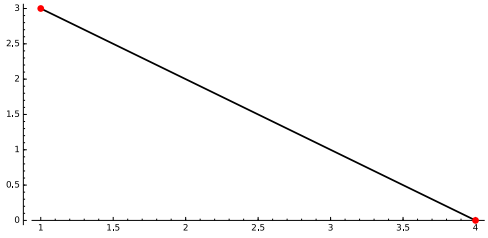


We have a nice, black line that connects two red dots. (Notice the use of the line break to start a new line. Remember that you don't have to do this, especially in the likely circumstance that you have more space than we do.)

If you look closely, you'll notice that red dots lie underneath the black line. Most people don't think that looks very good, and there's a surefire way to fix it; this is where the `zorder` option comes in. We've already noted that an object's `zorder` indicates how close it should seem, relative to other objects in the image. The larger the number, the "closer" to the viewer. You can think of the number `0` as being a "middle location;" positive numbers will appear "in front of" zero, and negative numbers will appear "behind" it.

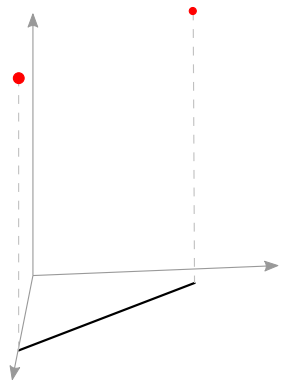
We can use this to "lift" the points over the line.

```
sage: p1 = point(((1,3),(4,0)), color='red', \
                  pointsize=60, zorder=5)
sage: p2 = line(((1,3),(4,0)), color='black', \
                  thickness=2)
sage: p1 + p2
```



Sage now considers the points to be at level 5, closer to the viewer, while the line remains at level 0, farther away. This is why the points now lie above the line.

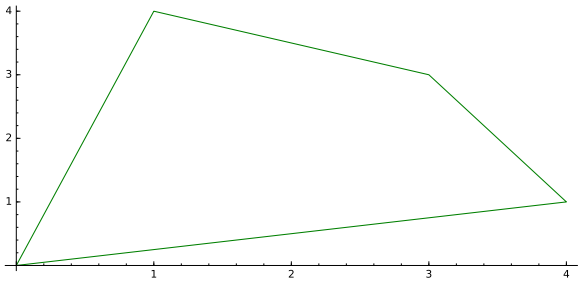
Why this option called `zorder`? This will make sense if you're acquainted with 3-dimensional graphs: it indicates the z-value of the graphics object, when viewed from above the x-y plane. We illustrate this with a 3D image:



Imagine yourself standing above this, looking down: you would see the red dots, at $z = 5$, in front of the line, at $z = 0$. (The dashed lines illustrate that the red points lie above the endpoints of the line.)

We mentioned earlier that the `line()` command really produces a "crooked" line segment. Here's an example of this at work.

```
sage: line(((0,0),(1,4),(3,3),(4,1),(0,0)), color='green')
```



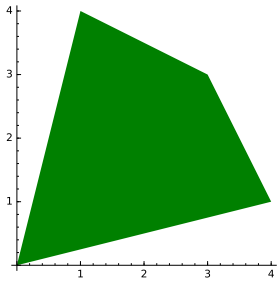
In this case, we've used the line to draw a closed figure. While you can do this, it would be more convenient to use the `polygon()` command, especially if you wanted to fill the object. With a polygon, you don't have to specify the first point again as the last point, because the implicit meaning of a polygon is a closed figure.

```
sage: polygon(((0,0),(1,4),(3,3),(4,1)), color='green')
```

`polygon(` *points*`,` *options*`)`

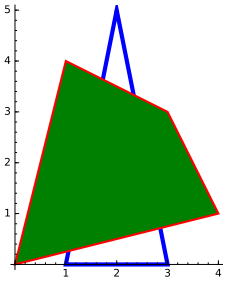
- *points* is a collection of ordered pairs
- *options* include:
  - `color =` *fill color* (`'blue'` or `(0,0,1)`)
    note that this differs from the usual interpretation of color
  - `edgecolor =` *edge color* (`'blue'` or `(0,0,1)`)
  - `fill =` *whether to fill the polygon* (`False`)
  - `thickness =` *edge thickness* (1)

FIGURE 5. `polygon()`



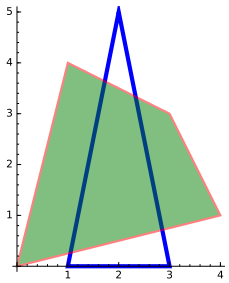
You don't *have* to fill a polygon, though. Let's add another polygon beneath this one. While we're at it, we'll add a red edge to this one, just to make it stand out.

```
sage: p1 = polygon(((0,0),(1,4),(3,3),(4,1)), color='green', \
                   edgecolor='red', thickness=2)
sage: p2 = polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
                   zorder=-5)
sage: p1 + p2
```



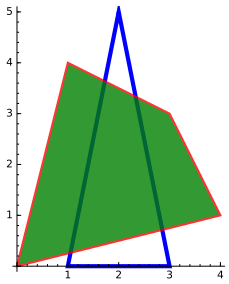
In some situations, we want to "see through" the top figure, to see the one underneath. This is where the `alpha` option comes into play. This option controls the transparency of an object. Its values range from 0 (invisible) to 1 (opaque).

```
sage: p1 = polygon(((0,0),(1,4),(3,3),(4,1)), color='green', \
                   edgecolor='red', thickness=2, alpha=0.5)
sage: p2 = polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
                   zorder=-5)
sage: p1 + p2
```



That's a little *too* transparent. We just want a hint that something lies underneath; we don't want to make the blue triangle seem as if it's actually on top. We modify `alpha` accordingly.

```
sage: p1 = polygon(((0,0),(1,4),(3,3),(4,1)), color='green', \
                   edgecolor='red', thickness=2, alpha=0.8)
sage: p2 = polygon(((1,0),(2,5),(3,0)), thickness=4, fill=False, \
                   zorder=-5)
sage: p1 + p2
```



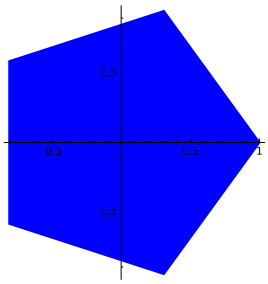
Be careful with the `polygon()` command. It can't read your mind; it follows the points in the precise order you specify. It doesn't take much to change a pentagon to a pentagram.

```
sage: polygon(((1,0), (cos(2*pi/5),sin(2*pi/5)), \
              (cos(4*pi/5),sin(4*pi/5)), \
              (cos(6*pi/5),sin(6*pi/5)), \
              (cos(8*pi/5),sin(8*pi/5))), thickness=2)
```

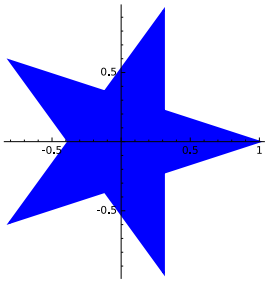
`circle(` *center*`,` *radius*`,` *options*`)`

- *center* is an ordered pair
- *radius* is a real number
- *options* include
  - `fill =` *whether to fill the circle* (`False`)
  - `edgecolor =` *color of the circle's edge* (`'blue'` or `(1,0,0)`)
  - `facecolor =` *color used to fill the circle* (`'blue'` or `(1,0,0)`)
  - if you specify `color`, then Sage ignores `edgecolor` and `facecolor` *even if you specify them explicitly*

FIGURE 6. `circle()`




```
sage: polygon(((1,0), (cos(4*pi/5),sin(4*pi/5)), \
              (cos(8*pi/5),sin(8*pi/5)), \
              (cos(2*pi/5),sin(2*pi/5)), \
              (cos(6*pi/5),sin(6*pi/5))), thickness=2)
```



We won't illustrate arrows as defined in this section; see an example in a following section. Feel free to experiment with them on your own.

"Curvy" stuff. We consider three more objects: circles, ellipses, arcs, and text.[33]

The first example is an arc, primarily to illustrate the relationship between `angle`, *radius*, `r2`, and `sector`. To that end, we'll add some dashed lines to help illustrate, obtaining the added benefit of illustrating the `linestyle` option.

[33] Text is "curvy"!

`ellipse(` *center*`,` *radius*`,` *options*`)`

- *center* is an ordered pair
- *radius* is a real number
- *options* include
  - `fill =` *whether to fill the circle* (`False`)
  - `edgecolor =` *color of the circle's edge* (`'blue'` or `(1,0,0)`)
  - `facecolor =` *color used to fill the circle* (`'blue'` or `(1,0,0)`)
  - if you specify `color`, then Sage ignores `edgecolor` and `facecolor` *even if you specify them explicitly*

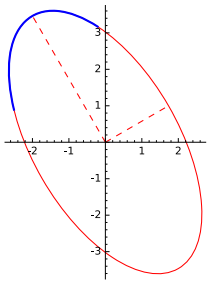
FIGURE 7. `ellipse()`

`arc(` *center*`,` *radius*`,` *options*`)`

- *center* is an ordered pair
- *radius* is a real number
- *options* include
  - `angle =` *angle between arc's axis and x-axis; length of this axis will be* radius (0)
  - `r2 =` *a second radius, for the arc of an ellipse; perpendicular to* `angle` (equal to *radius*, obtaining the arc of a circle)
  - `sector =` an ordered pair, indicating the angles that define the beginning and the end of the sector ($(0, 2\pi)$)

FIGURE 8. `arc()`

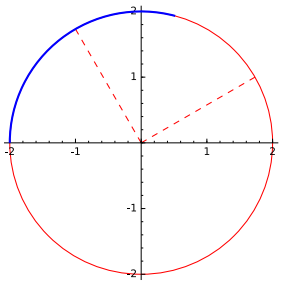
```
sage: p1 = arc((0,0), 2, angle=pi/6, r2=4, sector=(pi/4,5*pi/6), \
              thickness=2)
sage: p2 = line(((0,0), (2*cos(pi/6),2*sin(pi/6))), color='red', \
              linestyle='dashed')
sage: p3 = line(((0,0), (4*cos(pi/6+pi/2),4*sin(pi/6+pi/2))), \
              color='red',linestyle='dashed')
sage: p4 = ellipse((0,0),2,4,pi/6,color='red',zorder=-5)
sage: p1 + p2 + p3 + p4
```



Notice is that the ellipse's "horizontal" axis is tilted at the angle $\pi/6$, as illustrated by the dashed red line in the first quadrant. The "vertical" axis is likewise tilted; so the `angle` option has effectively rotated the entire affair by $\pi/6$. To draw the arc, we specified the sector $(\pi/4, \pi/6)$. This starts the arc at the angle $\pi/4 + \pi/6 = 5\pi/12$.

You may be wondering how that last statement can be true when the blue arc clearly starts in the second quadrant, rather than the first quadrant — after all, $5\pi/12 < 6\pi/12 = \pi/2$, and $\pi/2$ is the vertical axis. The answer lies in the distortion caused be the ellipse's "vertical" axis; it pulls the arc's endpoint "upward," but since the ellipse rotated, "upward" effectively means to the north-by-northwest, obtaining a point in the second quadrant. Don't take our word for it, though; try it with a circle by omitting any specification of `r2`, and see how things make more sense.

```
sage: p1 = arc((0,0), 2, angle=pi/6, sector=(pi/4,5*pi/6), \
              thickness=2)
sage: p2 = line(((0,0), (2*cos(pi/6),2*sin(pi/6))), color='red', \
              linestyle='dashed')
sage: p3 = line(((0,0), (2*cos(pi/6+pi/2),2*sin(pi/6+pi/2))), \
              color='red', linestyle='dashed')
sage: p4 = circle((0,0),2,color='red',zorder=-5)
sage: p1 + p2 + p3 + p4
```



An aside, for those who lack confidence in their trigonometry (as well as for those who don't, but should). If you're wondering where we came up with the ordered pairs for the lines, let's review some trigonometry. Recall first that, in radian notation, *a full turn* is considered to have an angle of 2π, rather than 180°. We traditionally split the full turn from there into a half turn $(2\pi/2 = \pi)$, a quarter turn $(2\pi/4 = \pi/2)$, a sixth of a turn $(2\pi/6 = \pi/3)$, and a twelfth of a turn $(2\pi/12 = \pi/6)$.

Meanwhile, the functions $\cos\alpha$ and $\sin\alpha$ represent the x- and y-values of a point on the unit circle that lies at angle α from the horizontal. For example, the point on the unit circle at an angle of $\pi/3$ from the horizontal would be

$$\left(\cos\left(\frac{\pi}{3}\right), \cos\left(\frac{\pi}{3}\right)\right) = \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right).$$

What if you want a circle of different radius? In this case, the relationship between cosine, sine, and the triangle formed by the radius and the x- and y-values mean you should scale the point by multiplying by the circle's radius. For example, the point on a centered circle of radius 2 at an

`text(` *text string*`,` *position*`,` *options*`)`

- *text string* is the label, enclosed in either single or double quotes
- *position* is an ordered pair
- *options* include
  - `fontsize =` *size of text* (10)
  - `rotation =` *angle to rotate text,* ***in degrees*** (0)

FIGURE 9. `text()`

angle of $\pi/6$ from the horizontal would be

$$\left(2\cos\left(\frac{\pi}{6}\right), 2\sin\left(\frac{\pi}{6}\right)\right) = \left(\sqrt{3}, 1\right).$$

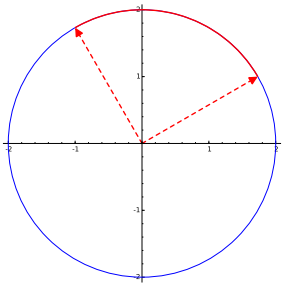
One point of a computer algebra system is to save us the trouble of simplifying. Rather than work out `(sqrt(3),1)` as a point on the line, we fed Sage the expression `(2*cos(pi/6), 2*sin(pi/6))` and let Sage figure out the rest on our behalf.

Likewise, when we wanted to place a point along the angle for the ellipse's "vertical" axis, rather than figure out the simplified value, we simplify asked Sage to do the rotation, then find the corresponding points:

$$\left(4\cos\left(\underbrace{\frac{\pi}{6}}_{\text{start}} + \underbrace{\frac{\pi}{2}}_{\text{rotate}}\right), 4\sin\left(\underbrace{\frac{\pi}{6}}_{\text{start}} + \underbrace{\frac{\pi}{2}}_{\text{rotate}}\right)\right).$$

We can illustrate this in Sage:

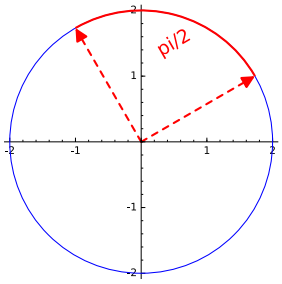
```
sage: p1 = circle((0,0),2)
sage: p2 = arrow((0,0),(2*cos(pi/6),2*sin(pi/6)), \
              linestyle='dashed', color='red')
sage: p3 = arrow((0,0), (2*cos(pi/6+pi/2), 2*sin(pi/6+pi/2)), \
              linestyle='dashed', color='red')
sage: p4 = arc((0,0), 2, angle=pi/6, sector=(0,pi/2), color='red',\
              thickness=2, zorder=5)
sage: p1 + p2 + p3 + p4
```



Notice the use of both `arrow()` and the `linestyle` option.

Text. Wouldn't it be nice to add some labels to that last picture? The `text()` command lets us put labels into an image. Our example reuses the images from the previous example.

```
sage: p5 = p1 + p2 + p3 + p4
sage: p6 = text('pi/2', (.5,1.5), color='red', fontsize=18, \
                rotation=30)
sage: p5 + p6
```



That looks okay, but wouldn't it be nice if it looked a little more "professional"? Wouldn't it be nicer to have the Greek letter π instead of a couple of Latin characters? One way to accomplish this would be to change the keyboard layout on your computer so that you can type Greek letters, but it still wouldn't look like a real fraction.[34] Instead, Sage offers an elegant mathematical solution via LaTeX.

Without getting *too* far into the weeds, LaTeX is a kind of "markup language" developed during the 1970s and 1980s. It is extremely common for mathematicians to type papers and books in LaTeX (you're reading an example) because it offers extremely intuitive and flexible commands to indicate what you mean to write, and automatically arranges the text in a manner consistent with mathematical tradition, with appropriate placement of exponents, stretching of grouping symbols, etc.

Sage doesn't offer the full range of LaTeX markup; it offers only a subset, but that subset suffices for practical work in graphics. To use LaTeX markup, you basically need to perform only two tasks:

- enclose the substring of mathematics in the LaTeX delimiters `$` and `$`; and
- use appropriate LaTeX markup, as described in the chapter beginning on page 1.

An aside. Users of LaTeX will notice that we use double backslashes when ordinarily they'd expect single backslashes. This is because the single backslash is a control command in Sage[35] strings: `\n`, `\r`, `\t` all have special meanings (as do others). You might get away with a single backslash sometimes, but it can also wreak havoc at the most inconvenient of times. Using a double backslash consistently avoids potential ambiguities.

And yet… If you look at Figure 24 on page 253, you'll notice we used single backslashes. The reason is that we later talk about inserting LaTeX into the worksheet, and in that case a double backslash is *in*appropriate. It's a rather unfortunate bit of confusion to the learner, but the basic rule is pretty simple: the markup requires a single backslash, but in a string of text that means you need a double backslash.

In addition… You can use LaTeX in a worksheet's HTML cells, too! (See p. 26 for an explanation on how to create an HTML cell.) Enclose the text in dollar signs, in "backslashed"

[34] Plus, we tried it. It doesn't work.

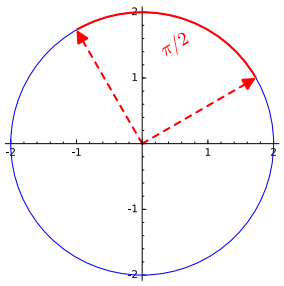
[35] This is because Sage is built on Python.

parentheses, or in backslashed brackets, and you can use LaTeX markup to your heart's content.[36] For instance, the following code will give you a nice layout of the definition of the integral:

```
sage: %html
      The definition of the <b>derivative</b> is <i>the limit of
      the slopes of the secant lines as the distance between the
      points approaches 0;</i> or,\[\frac{d}{dx}f(x) = \lim_{\Delta
      x\rightarrow\infty}\frac{f(x+\Delta x)-f(x)}{\Delta x}\]
```

Back to our regularly scheduled programming. Let's revisit that previous image, using LaTeX this time to make it look good. If you look at Figure 24, you'll find π in the row on Greek letters; to get the Greek symbol, you'd want the markup `\pi`, which in a Sage string we write as `\\pi`.

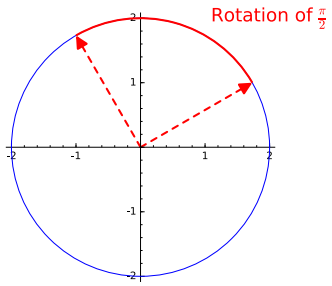
```
sage: p7 = text('$\\pi/2$', (.5,1.5), color='red', \
                fontsize=18, rotation=30)
sage: p5 + p7
```



That looks a *lot* better. But maybe you'd like it to look like a real fraction, instead of a simple division? Look at Figure 24 again, and you'll see a `\frac` markup. This translates to `\\frac` in a Sage string, but it also illustrates something else about LaTeX markup: some commands require additional information. This information is usually passed in pairs of braces, sometimes multiple pairs. For a fraction, this should make sense: a fraction requires both a numerator and a denominator, so it requires two items of information, so two pairs of braces. We illustrate this in the example below, which differs slightly from the ones before.

```
sage: p8 = text('Rotation of $\\frac{\\pi}{2}$', (.5,1.5), \
                color='red', fontsize=18)
sage: p5 + p8
```

[36] Well, maybe not to your heart's content, in more ways than one. Within reason, anyway.



Colors. There are several ways to define colors in Sage. One is by name, and Sage offers an *awful* lot of colors by name, with intriguing options such as `'chartreuse'` and `'lavenderblush'`. You can get a full list of names by typing the following:

```
sage: colors.keys()
['indigo', 'gold', 'firebrick', 'indianred', 'yellow',
'darkolivegreen', ...
```

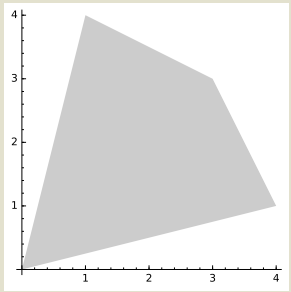
At the present time there are nearly 150 predefined colors, so we've left out an awful lot.

You can actually obtain several million colors (at least 16 million or thereabouts) by manipulating what are called "RGB values." Visible light can be broken into three components — red, green, and blue — and we can ask Sage to assign each of these components, called *channels*, a value from 0 to 1. You can think of "0" as meaning the channel is completely off, and "1" as meaning the channel is completely full. That leads us to the following:

R	G	B	interpretation	result
1	0	0	full red, no green or blue	red
1	1	0	full red and green, no blue	yellow
0.5	0.5	0.5	half-power red, green, blue	gray
0.8	0.8	0.8	eight-tenths power red, green, blue	lighter gray

To use colors this way in Sage, simply pass the three numbers, enclosed in parentheses, as a `color` option.[37] To illustrate this, we revisit our pentagon from before.

```
sage: polygon(((0,0),(1,4),(3,3),(4,1)), color=(0.8,0.8,0.8))
```



[37] It would be more precise to specify it as the `rgbcolor` option, but Sage interprets `color` as rgb, which is fairly standard, anyway. If you are familiar with the Hue-Saturation-Value model of color, you can use the `hsvcolor` option instead, but we don't cover that here.

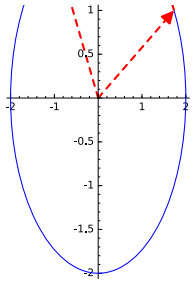
`show(` *graphics object*`,` *options*`)`

- `ymax =` *largest visible y-value* (determined by object)
- `ymin =` *smallest visible y-value* (determined by object)
- `xmax =` *largest visible y-value* (determined by object)
- `xmin =` *smallest visible y-value* (determined by object)
- `aspect_ratio =` *ratio of 1 y-unit to 1 x-unit* (determined by object)
- `axes =` whether to show the axes (`True`)

FIGURE 10. `show()`

Other options for 2D objects. Some options for 2D objects are more appropriately manipulated after adding them together. There are several ways to manipulate them, but the most convenient may be through the `show()` command, which offers several options to fine tune an image. We list the ones of most interest to us in Figure 10. We put these to work on `p5`, the image above of the circle with the rotation.

```
sage: show(p5, ymax=1, aspect_ratio=2)
```



You can see that the largest visible y-value is $y = 1$, and the circle seems to have been stretched vertically, so that a unit of 1 on the y-axis is twice as long as a unit of 1 on the x-axis.

It is often convenient to show graphics without the axes; show offers the `axes` option for this. We illustrate this later, but you might try reworking some of the graphics listed here to see it in action. There are other options to control the axes that we do not review here; most of them can also be controlled via a plot object's methods. You can read about them by typing the name of a plot object, followed by the dot, followed by the tab key; select one of interest, type the question mark, and execute the command to see its help (as explained earlier).

2D plots

By now, you should have learned three basic ways we represent two-dimensional relations we want to plot. They are:

- Cartesian coordinates, either
  - with y as a function of x;
  - as an equation in terms of x and y only, but not necessarily a function; or
  - with x and y in terms of a third variable, t, called a parameter;

  and
- polar coordinates.

`plot(` $f(x)$`, xmin, xmax, options)`

- $f(x)$ is a function or expression in one variable
- *xmin* is the smallest x-value to use for plotting
- *xmax* is the largest x-value to use for plotting
- *options* include
  - `detect_poles =` one of the following (`False`):
    - `False` (draw directly)
    - `True` (detect division by zero and sketch appropriately)
    - `'show'` (same as `True`, but include an asymptote)
  - `fill =` one of the following (`False`)
    - `False` (do not fill)
    - `'axis'` or `True` (fill to the x-axis)
    - $g(x)$ (fill to the function $g(x)$)
    - `'min'` (fill from the curve to its minimum value)
    - `'max'` (fill from the curve to its maximum value)
  - `fillcolor =` *color used for the fill, or* `'automatic'` (`'automatic'`)
  - `fillalpha =` *transparency of the fill, from 0 to 1* (0.5)
  - `plot_points =` *number of points to use when generating an x-y table* (200)

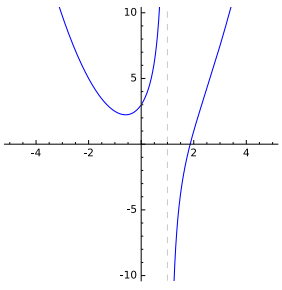
FIGURE 11. `plot()`

Sage offers commands for each of these representations.

Generic plots. The one you'll use most often is the `plot()` command, which accepts a function, or at least an expression in one variable, and produces a plot where the y value is determined by the x-values, as by an x-y table. This description is quite literal; the command works by creating a number of (x,y)-pairs and connecting the dots. Its *options* include those listed in Figure 4 on page 52. One thing to watch for is that *xmin* and *xmax* do *not* mean the same thing here as they do in the `show()` command. Here, they correspond to the smallest and largest x-values for which `plot()` generates an x-y pair. It is quite possible to `show()` more or less than this amount by setting `xmin` and `xmax` to different values in the `show()` command.

We illustrate this command on a function which has poles.

```
sage: p = plot(x^2 - 3/(x-1), -5, 5, detect_poles='show')
sage: show(p, ymin=-10, ymax=10, aspect_ratio=1/2)
```



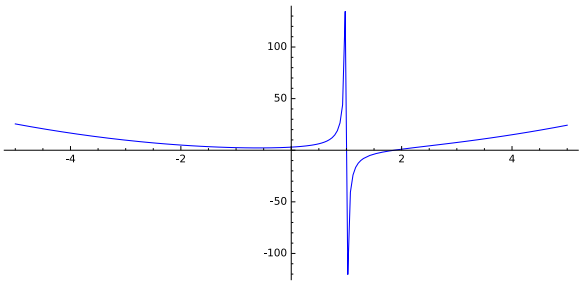
`parametric_plot( ( x(t) , y(t)) , ( tvar, tmin, tmax) , options)`

- $x(t)$ and $y(t)$ are functions or expressions in terms of a parameter t, which define the x- and y-value of a point
- *tvar* is the parameter
- *tmin* is the smallest t-value to use
- *tmax* is the largest t-value to use
- *options* include
  - `fill = True` *(to axis) or* `False` *(none)* (`False`)
  - `fillcolor`, *as in* `plot()`
  - *`fillalpha`*, *as in* `plot()`

FIGURE 12. `parametric_plot()`

Were we not to constrain the y-values, the image would look very different. (You might want to try the `show()` command without `ymin` and `ymax` values, to see this for yourself.) Likewise, if you leave out `detect_poles` Sage will mistakenly connect the two points closest to the pole, which usually looks like an asymptote.

```
sage: p = plot(x^2 - 3/(x-1), -5, 5)
sage: show(p)
```



As we hinted above, this is *technically* wrong, because that apparently vertical line is not; it connects two points. If you saw this happen in real life, you might conclude that this was an asymptote, but you might also be wrong to do so; there could well be some interesting feature of the function taking place there.

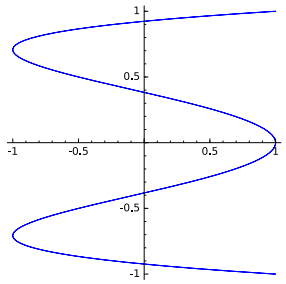
Parametric plots. Closely related to the `plot()` command is the `parametric_plot()` command. Parametric plots are frequently used in situations where the values for x and y depend on the particular time, which happens in physics and engineering. In many cases, $t \in [0,1]$, where "0" indicates "start position" and "1" indicates "end position," but there are perfectly good reasons to use other values. The following uses $t \in [0, 2\pi]$.

```
sage: var('t')
sage: parametric_plot((cos(4*t),sin(t)), (t,0,2*pi))
```

`polar_plot(` $r(\theta)$`, (` θ`,` θ_{min}`,` θ_{max}`) ,` *options*`)`

- $r(\theta)$ is a function or expression defined in terms of another variable θ, for which you can use `x`
- θ is the independent variable representing the angle
- θ_{min} is the smallest angle to use
- θ_{max} is the largest angle to use
- *options* include
  - `fill = True` *(to axis) or* `False` *(none)* (`False`)
  - `fillcolor`, *as in* `plot()`
  - *`fillalpha`, as in* `plot()`

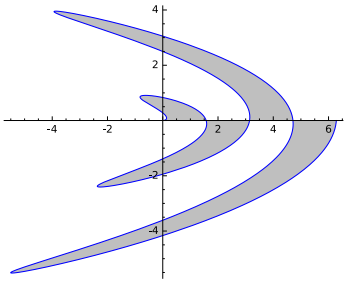
FIGURE 13. `polar_plot()`



This is the *entire* graph of this parametric function. Play around with different values of *tmin* and *tmax* to see why $[0, 2\pi)$ gives us a complete wave.

The following example demonstrates the use of the `fill` option.

```
sage: parametric_plot((t*cos(4*t),t*sin(2*t)),(t,0,2*pi),fill=True)
```



This is already neat, but you can obtain a neater picture yet if you double that curve in different ways. Try it and see what happens.

Polar plots. We turn to polar coordinates. These are commonly used when it's more convenient to describe how an object's position changes as it rotates around the origin. A number of images that are difficult to describe in (x, y)-pairs become quite easy with polar coordinates, such as the abstract *fleur-de-lis* below.

`animate(` *collection*`,` *options*`)`

- *collection* can be a list, set, or tuple of plots
- *options* include
  - `xmin`, `xmax`, `ymin`, `ymax`, and `aspect_ratio`, as in `show()`

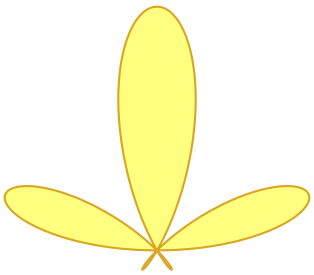
FIGURE 14. `animate()`

`show(` *animation*`,` *options*`)`

- *animation* is an animation generated using `animate()` command
- *options* include
  - `delay =` hundredths of a second before moving to the next frame (`20`)
  - `iterations =` number of times to repeat the animation from the beginning, where `0` indicates "forever" (`0`)
  - `gif =` whether to produce a GIF animation or a WEBM animation (*`False`*)

FIGURE 15. `show()`

```
sage: polar_plot(cos(2*x)*sin(3*x), (x, 0, pi), fill=True, \
                 thickness=2, fillcolor='yellow', \
                 color='goldenrod', axes=False)
```



Animation

Animation both in film and in computer programming consists of swapping several images several times a second. Sage offers an `animate()` command that accepts a collection as input and creates an animation that we can then `show()`. In this case, the options we supply to `show()` are a little different.

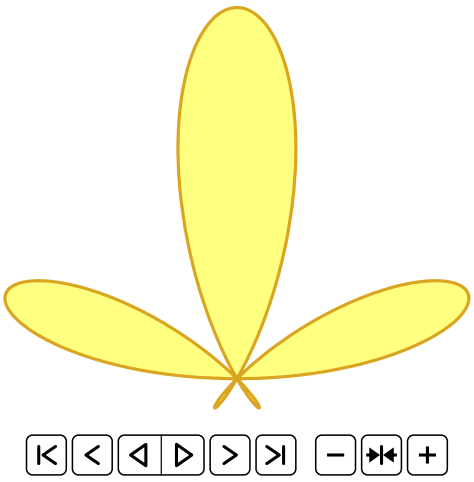
It's important to observe that *we do not* specify the traditional `show()` options `xmin`, `xmax`, etc. when we `show()` an animation; those really are more appropriate in the `animate()` command, which has to put all the frames together, and so needs to know what the largest and smallest x- and y-values should be. Rather, the options show expects are options that apply to *showing* an animation, and which are not proper to the animation itself. That is, it makes sense to show the same animation with a different delay between frames, as the frames themselves don't change. It

doesn't make sense to say that the "same" animation would have a different `xmin` value; the frames themselves have changed in this case.

To illustrate how the `animate()` command works, we'll play around with our *fleur-de-lis* from the previous section.

```
sage: p1 = polar_plot(cos(2*x)*sin(3*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: p2 = polar_plot(cos(3*x)*sin(4*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: p3 = polar_plot(cos(4*x)*sin(5*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: p4 = polar_plot(cos(5*x)*sin(6*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: p5 = polar_plot(cos(6*x)*sin(7*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: p6 = polar_plot(cos(7*x)*sin(8*x), (x, 0, pi), fill=True, \
               thickness=2, fillcolor='yellow', \
               color='goldenrod', axes=False)
sage: panim = animate((p1, p2, p3, p4, p5, p6), xmin=-1, xmax=1, \
               ymin=-0.5, ymax=1.5, aspect_ratio=1)
sage: show(panim, gif=True)
```

You should obtain an animation as a result. *If you view this text in Acrobat Reader*, you should see that animation below; if you are looking at a paper copy of the text, you should instead see the animation's individual frames:



You should take a moment to toy with the options a little bit. For instance, see what happens when you change the final line of that sequence to

```
sage: show(panim, gif=True, delay=10)
```

or to

```
sage: show(panim, gif=True, delay=40)
```

You should notice a definite change in behavior.

Another thing to try is to remove the `gif=True` option. We used it because some common web browsers do not support the WEBM format at the time of this writing. See what happens when you remove it.

Using the approach we've shown here makes it a little burdensome to create plots for all but the most trivial animations. In a later chapter we'll show how you can use loops to speed up the process.

Implicit plots

Some Cartesian plots are easily described as *relations* in terms of x and y, but are difficult to describe as *functions*. Geometrically, they do not satisfy the "vertical line test" that determines a function, and it isn't very easy to rewrite them in parametric or polar form.

Consider the circle. One relation that is *not* a good example of this difficulty is the circle $x^2 + y^2 = a^2$, where a is any constant you like (within reason).[38] We can actually express this curve using a parameter t:

$$(x, y) = (a\cos t, a\sin t)$$

or using polar coordinates:

$$r = a.$$

Nevertheless, the traditional equation of a circle is useful in this respect: it does not admit a function y in terms of x; after all, if we solve for y, we have

$$y = \pm\sqrt{a^2 - x^2},$$

and having a choice of two y values is a bit of a bummer for a function.

Suppose we wanted to graph $x^2 + y^2 = 6$. In some systems, the only way to graph this is to plot the two pieces simultaneously, in this fashion:

[38] "Within reason?" you are likely muttering. "What on earth does that mean?" We mean $a \neq 0$. And finite. And real, definitely real. Complex would be bad, at least for our purposes. So, basically, $a \in \mathbb{R} \setminus \{0\}$.

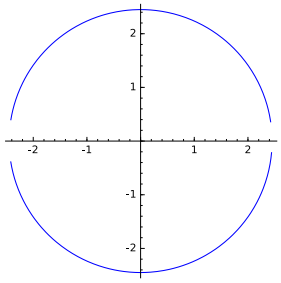
`implicit_plot(` *relation*`, (` *xvar*`,` *xmin*`,` *xmax*`) , (` *yvar*`,` *ymin*`,` *ymax*`) ,` *options*`)`

- *relation* is an equation in terms of two variables
- *xvar* and *yvar* are the indeterminates in the function for which you want to plot along the x- and y-axis, respectively
- *xmin* and *xmax* are the smallest and largest x-values to use to generate points
- *ymin* and *ymax* are the smallest and largest y-values to use to generate points
- *options* include
  - `fill =` *whether to fill the "interior" of the relation; that is, (x,y) pairs where the transformed relation turns negative*
  - `plot_points =` *number of points to use on each axis when generating the plot* (150)

Note that `fillcolor`, `fillalpha`, and `thickness` are *not* valid options for `implicit_plot()`.

FIGURE 16. `implicit_plot()`

```
sage: top = plot(sqrt(5-x**2), -3, 3)
sage: bot = plot(-sqrt(5-x**2), -3, 3)
sage: top + bot
verbose 0 (2733:  plot.py, generate_plot_points) WARNING: When
plotting, failed to evaluate function at 38 points.
verbose 0 (2733:  plot.py, generate_plot_points) Last error
message:  'math domain error'
verbose 0 (2733:  plot.py, generate_plot_points) WARNING: When
plotting, failed to evaluate function at 38 points.
verbose 0 (2733:  plot.py, generate_plot_points) Last error
message:  'math domain error'
```



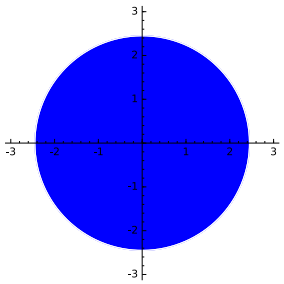
We don't have a full circle here; we have a broken circle. The "verbose" warnings illustrate how Sage wants to make an table of 38 x-y pairs, but fails because the numbers don't quite line up favorably.

One solution is to plot it using parametric or polar plots, but this isn't always available, or not easily. An alternative is to plot the curve *implicitly;* in this case, you specify an equation, its variables, and their range. Sage transforms then transforms the equation

$$f(x,y) = g(x,y) \quad \text{to} \quad f(x,y) - g(x,y) = 0,$$

builds a grid along the window you've specified, checks to see which grid points come *reasonably close to zero*, then connects them in a clever fashion to obtain the correct graph of the function.

```
sage: var('y')
sage: implicit_plot(x**2 + y**2 == 6, (x, -3, 3), (y, -3, 3), \
                    fill=True)
```



Let's clarify how Sage decided which points lie in the interior. We are asking it to graph the equation

$$x^2 + y^2 = 6,$$

so

$$f(x,y) = x^2 + y^2 \quad \text{and} \quad g(x,y) = 6.$$

Sage rewrites this as

$$f(x,y) - g(x,y) = 0,$$

or more precisely

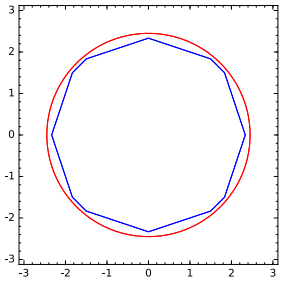
$$x^2 + y^2 - 6 = 0.$$

Sage then divides the x- and y-intervals $(-3,3)$ into 150 subintervals, which sets up a grid on the plane, and substitutes each (x,y) pair on the grid into the left-hand side of the equation. Some of these points may match 0 exactly, but most will not; for instance, $(1.5, 1.92)$ and $(1.5, 1.96)$ lie *close* to the circle, but the former gives a negative value $(-.0636)$ when substituted into x^2+y^2-6, while the second gives a positive value $(.0916)$. This tells Sage three things:

- The circle *should* pass between these two points; after all, its points have value 0 when substituted into $x^2 + y^2 - 6$.
- Of these two points, the point $(1.5, 1.96)$ lies closer to the circle than $(1.5, 1.92)$, so it's best to connect this approximation with other, nearby approximations to the curve.
- Because its value was negative, the point $(1.5, 1.96)$ lies inside the circle, and should be filled if `fill=True`.

To see this in action, we can experiment with the `plot_points` option. We'll plot two unfilled circles, each with a very different number of plot points.

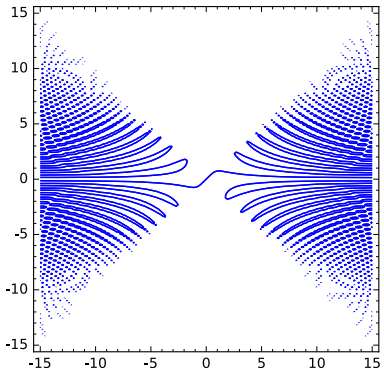
```
sage: p1 = implicit_plot(x^2+y^2-6, (-3,3), (-3,3), \
                         plot_points=5, zorder=5)
sage: p2 = implicit_plot(x^2+y^2-6, (-3,3), (-3,3), \
                         color='red', zorder=-5, \
                         plot_points=300)
sage: p1 + p2
```



The blue graph, which was pointed with only 6 plot points on each axis (the subinterval's endpoints add 1 to the `plot_points` option), looks jagged and uneven, while the red graph looks smooth, despite being generated the same way. This illusion of smoothness is due to the much higher number of points connected to make the graph.

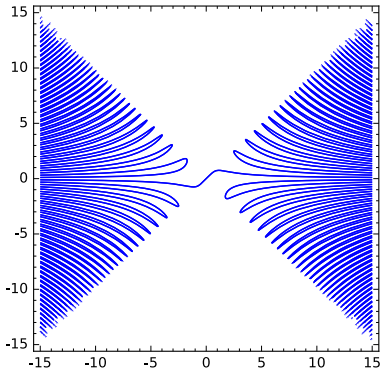
To drive home the importance of the number of plot points in practical situations, let's look at a different example.

```
sage: implicit_plot(x*cos(x*y)-y, (x,-15,15), (y,-15,15))
```



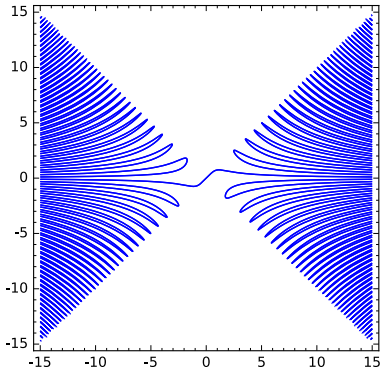
That curve looks just plain… ***awesome.*** But is it one, connected curve? It seems to have a large number of jagged elements. Some parts are also disconnected; this isn't bad in itself, but along with the jaggedness, it suggests a problem. This is one case where you would expect an increase in the number of plot points to be extremely useful, and so it is. If you increase the number of `plot_points` by about 50 or so, over and over until things start to become clear, eventually you come to something like this:

```
sage: implicit_plot(x*cos(x*y)-y, (x,-20,20), (y,-20,20), \
                    plot_points=500)
```



There's quite a difference in this image, isn't there! What's more, there's quite a difference in the time it took to plot them, isn't there? Some things still seem a little unclear, so we can take this a bit further:

```
sage: implicit_plot(x*cos(x*y)-y, (x,-20,20), (y,-20,20), \
                    plot_points=1000)
```



Notice the tradeoff: more plot points resolves more discrepancies, but also leads to longer time. At this point, there's no cause to increase `plot_points` higher, as there's no reason to think we've lost any interesting features.

We conclude with some words of warning. It may happen that you neglect to specify *xvar* and *yvar* when creating your implicit plot; it's a rather natural error to make, especially since `plot()` doesn't require you to specify a variable. In this case, you'll receive a *DeprecationWarning*:

```
sage: implicit_plot(x**2 + y**2 - 6, (-3,3), (-3,3))
DeprecationWarning:  Unnamed ranges for more than one variable is
deprecated and will be removed from a future release of Sage; you
can used named ranges instead, like (x,0,2)
```

The warning here is both explicit and helpful: it tells you what the problem is, and suggests the workaround. The correct usage is the one we described above, and repeat here for good measure:

```
sage: implicit_plot(x**2 + y**2 - 6, (x,-3,3), (y,-3,3))
```

You will also encounter problem if you neglect to define `y` as an indeterminate, or if you neglect to specify the ranges for `x` and `y`; these lead to *`NameError`* (because `y` is unknown) and *`TypeError`* warnings (because Sage expects a different number of arguments).

Exercises

True/False. If the statement is false, replace it with a true statement.

1. We have listed all possible options to the plotting commands.
2. You don't have to give all the information we list for a command.
3. The values for RGB colors range from `0` to 255.
4. No matter what you do, you're stuck with axes in every Sage plot.
5. The `zorder` options helps place objects "on top of" each other.
6. One reason to use the `polygon()` command instead of the `line()` command is that you don't have to re-specify the starting point as the end point.
7. The transparency of a fill can differ from the transparency of the curve being filled.
8. Sage generates plots the same way we learn to do so: connecting the dots between x-y pairs.
9. Like a graphing calculator, Sage is unable to detect places where asymptotes occur.
10. The best way to graph the curve $y^2 = x^3 + x + 1$ is to solve for y by taking the square root, plot the positive and negative branches separately using a `plot()` command, then combine them using addition.
11. *Bonus:* When you don't know the answer to a True/False question, the best strategy is to answer, "Maybe," then pray for partial credit.

Multiple Choice.

1. If an object's x- and y-coordinates are functions of time, the best choice for plotting a graph of its motion is probably:
   A. `plot()`
   B. `implicit_plot()`
   C. `parametric_plot()`
   D. `polar_plot()`
2. If an object's distance from a central point varies as it rotates around the center, the best choice for plotting a graph of its motion is probably:
   A. `plot()`
   B. `implicit_plot()`
   C. `parametric_plot()`
   D. `polar_plot()`
3. If a function is well-defined by one variable, the best choice for plotting its graph is probably:
   A. `plot()`
   B. `implicit_plot()`
   C. `parametric_plot()`
   D. `polar_plot()`
4. To change how transparent a graphics object is, we use this option:
   A. `alpha`
   B. `opacity`
   C. `transparency`
   D. `zorder`
5. To change which object seems "on top of" another, we use this option:

   A. `alpha`
   B. `depth`
   C. `level`
   D. `zorder`
6. What aspect of a plot might suggest you need to increase the value of `plot_points`?
   A. It looks as if it has an asymptote.
   B. Jagged lines or "crooked" curves appear when you expect smoothness.
   C. One equation produces two or more curves.
   D. You buy a better monitor and want to exploit its high definition.
7. If a `plot()` produces an image, part of which resembles a vertical line, which option should you modify to test for an asymptote?
   A. `detect_poles`
   B. `draw_asymptotes`
   C. `plot_points`
   D. `zorder`
8. If you get a *`NameError`* when making an `implicit_plot()`, then in all likelihood you made the following error:
   A. You forgot to specify an equation in `x` and `y`, rather than an expression in `x` alone
   B. You misspelled the name of an option to `implicit_plot()`
   C. You forgot to define the indeterminate `y` using `var()`
   D. You forgot to specify `x` and `y` as *xvar* and *yvar*
9. If you get a *`TypeError`* when making an `implicit_plot()`, then in all likelihood you made the following error:
   A. You misspelled the name, `implicit_plot()`
   B. You misspelled the name of an option to `implicit_plot()`
   C. You forgot to define the indeterminate `y` using `var()`
   D. You forgot to specify ranges for `x` and `y`
10. If we are teaching Calculus, and want to illustrate how to find the area between the curves x^2 and x on $[0,1]$, which command would show the region we want? (The plot should also include both curves that define the region.)
    A. `plot(x**2-x, 0, 1, fill=True)`
    B. `plot(x**2, 0, 1, fill=x)`
    C. `plot(x**2, 0, 1, fill=True) - plot(x, 0, 1, fill=True)`
    D. `plot(x**2, 0, 1, fill=x) + plot(x, 0, 1)`

Bonus: You are stranded on a desert island. Which of these items would you rather have with you?
   A. a towel
   B. a can opener
   C. a bottle of sunscreen
   D. a working laptop with Sage installed

Short answer.

1. Professor Proofsrock avers that we can view the curve $y = x^2$ as a parametric curve $y = t^2$, $x = t$. Plot both curves, using `plot()` for the former and `parametric_plot()` for the latter. Do you agree with his averral? What, besides the plots, makes you agree?

2. Why do you think that the default values for the `show()` options `ymax`, `ymin`, and `aspect_ratio` depend on the object being shown, rather than on specific values? Give a brief explanation for each option.

Programming.

1. Use Sage to plot the following functions over the interval $[1,5]$ on the same graph. Give each a different color, and label it by name with a `text()` object.

$$x, \quad x^2, \quad \log_{10} x, \quad e^x$$

Rank the functions according to which one grows fastest in the long run. Then use Sage's Calculus facilities to confirm that your ranking is correct at $x = 10$ and also at $x = 100$. *Hint:* The derivative tells you the rate of change at a point.

2. Using your knowledge of Calculus, do the following:
   (a) Choose a transcendental function $f(x)$ and a point $x = a$ at which the function is defined.
   (b) Compute the derivative of f at $x = a$. If the derivative is 0, return to step (a) and choose a different point.
   (c) Plot f in a neighborhood of a. Make this curve black, of thickness 2.
   (d) Add to this a plot of the line tangent to f at $x = a$. Make this line blue, and dashed.
   (e) Add to this a plot of the *normal line* at $x = a$. Make this line green, and dashed.
   Hint: The normal line passes through $x = a$ and is perpendicular to the tangent line.
   (f) Add a big red point at $(a, f(a))$.
   (g) Add labels for the point, the curve, and the line. Use LaTeX. Each label's color should correspond to that of the object it labels.
   (h) `show()` the image at an aspect ratio appropriate to a pleasing result.
3. The Mean Value Theorem of Calculus states that if f is a continuous function on the interval $[a, b]$, then we can find some $c \in (a, b)$ such that

$$f(c) = \frac{1}{b-a} \int_a^b f(x)\,dx,$$

and this y-value $(f(c))$ is in fact an "average" value of the function on the interval. In this problem, you will illustrate this result.
   (a) Choose some quadratic function $f(x)$. Choose a reasonable interval $[a, b]$. Plot the graph of f over $[a, b]$, filling to both the minimum and maximum y-values. (This technically requires you to create two plots, then combine them.)
   (b) Solve for the value of c that satisfies the Mean Value Theorem. (You'll have to do this by hand, as we haven't yet discussed how to use Sage to solve problems. It's a quadratic equation, though, so it shouldn't be that hard.)
   (c) Add the line $y = f(c)$ to your filled plot of $f(x)$. Also add a line from $(c, 0)$ to $(c, f(c))$. These lines should have a color different from the plot or its fill.
   (d) Explain how this final image illustrates the Mean Value Theorem.
4. A complex number has the form $a + bi$. The *complex plane* is a representation of the field $\mathbb{C}$ in the Cartesian plane, with the point (a, b) corresponding to the number $a + bi$.
   (a) Choose three complex numbers $a + bi$, with each corresponding point of the complex plane in a different quadrant. Create arrows connecting the origin to each point, each arrow in a different color.

(b) Use Sage to evaluate the product $i(a+bi)$ for each of the points you just chose. Create three new arrows connecting the origin to each point, in a color corresponding to the color you used for the original point.

(c) Plot all six arrows simultaneously. What is the *geometric* effect of multiplying $(a+bi)$ by i?

5. A great deal of higher mathematics is concerned with *elliptic curves*, which are *smooth,* non-self-intersecting curves with the general form

$$y^2 = x^3 + ax + b.$$

Graph several curves of this form, using different values for a and b. Which value(s) produce(s) elliptic curves? Which value(s) do(es) not? Describe the general pattern of what happens as the values of a and b change.

6. We build on the previous problem. A great deal of higher mathematics considers elliptic curves over a finite field, such as $\mathbb{Z}_p$ where p is prime. Choose one of the elliptic curves from the previous problem. Let $p = 5$, and substitute all possible values of x and y into the equation. (There are only 25, so this isn't *too* burdensome, and you can probably do a lot of them in your head. Just don't forget to do them modulo 5.) Plot the points in the plane. Does this curve bear any resemblance to the one you had before?

Bonus: Why were we able to say, with confidence, that in #6 there are 25 possible values of x and y to substitute? What if we had chosen $p = 7$; how would the number of points have changed?

Writing your own functions

Solving more difficult problems requires a bit more than a simple, straightforward list of commands. It is often helpful to organize many, oft-reused tasks into one command that gives us a convenient handle on all of them. Doing this would also allow us to deal with tasks more abstractly, helping us see the forest rather than the trees. It makes it easier to reuse the same code in different places, and likewise for someone else to read and, if desired, modify the code to their own purposes. This idea of **modularity** makes us more "productive," in the sense that we become better problem-solvers.

Technically, you've already been doing this: each Sage command you've been using to simplify expressions and draw pictures is really a package of other commands that was designed (usually carefully) to tackle that particular task. You can actually view some of the code for most Sage commands using a double question mark:

```
sage: simplify??
Signature:  simplify(f)
Source:
def simplify(f):
    r"""
    Simplify the expression 'f'.

    EXAMPLES: We simplify the expression 'i + x - x'.

    ::

        sage:  f = I + x - x; simplify(f)
        I

    In fact, printing 'f' yields the same thing - i.e., the
    simplified form.
    """
    try:
        return f.simplify()
    except AttributeError:
        return f
File:  /Applications/sage-6.7-untouched/local/lib/python2.7/
site-packages/sage/calculus/functional.py
Type:  function
```

This actually gives us a little more than the code. It first lists the command's *signature,* a template for how the command is used. It then lists the "source code," the list of commands that define the `simplify()` command. These commands start after the line labeled `Source:` and continue until the line labeled `File:`; this latter line indicates the file on disk where the command is found. Were you to open that file, you would find the same source listed above therein. Notice that most of the source code in this case consists of the same documentation you see when you type `simplify?` , reflecting that Sage's documentation is embedded in the code itself.

Much of the rest of this text is dedicated to helping you understand what all that source code means, and how to use it to your own ends. We don't expect you to become a full-fledged Sage developer by the time you're done (that would go far beyond its scope) but we do expect you to be able to write new, simple commands that accomplish specific tasks that would be useful for research and education. That requires some dedication on your part, but don't fret: it requires dedication on our part, too, for which we now roll back our figurative sleeves and delve in.

Defining a function

The commands you define in Sage are called *functions,* which agrees with the mathematical use in that you provide *arguments* to a function, and it provides some result.

The basic format. To define a function, we use the following format:

```
sage: def function_name(arguments):
          first_statement
          second_statement
          ...
```

You can think of `def` as shorthand for "define." It is an example of a *keyword* or a reserved word, a sequence of characters which carries special meaning in a programming language, and which the programmer cannot assign some other meaning. What follows next is a valid identifier name (review the rules on p. 31) and is the name of your new function. You must next open and close parentheses. Optionally, you can include additional identifiers within the parentheses; while this is not required, it gives the best, most reliable way to pass information to the your function. We call these identifiers *arguments*,[39] and will look at them more carefully in a moment.

The next thing to notice is that the "statements" are indented. This is an important principle that Sage inherits from Python, which you'll recall is the programming language we use to interact with Sage: whenever we come to a point in a program where we need to organize statements "within" other statements, or where some commands "depend" on others, we indicate which statements depend on the others using indentation.[40] This setup is called a control structure; here, the control structure is the `def` statement, which defines a function, which for our purposes can be considered a shorthand for a list of other statements.

For now, let's look at a fairly simple function. It takes no arguments, but is not entirely useless.

[39] Aptly named, as writing a program so often feels like an argument with the computer. Computer science texts often call arguments "parameters."

[40] Most computer languages do not require indentation, but designate keywords or special characters to indicate where a control structure begins and ends. For instance, many languages follow C in the use of braces `{...}` to open and close a structure; others use variations on `BEGIN...END`.

```
sage: def greetings():
          print 'Greetings!'
```

You can use either single or double quotes around the word "Greetings," but be sure to do it consistently.

If you've typed this correctly, Sage will accept it silently. If you type something wrong and Sage detects it, it will report an error. For instance, if you forget to indent, you will see this:

```
sage: def greetings():
      print "Greetings!"
  File "<ipython-input-33-33dcc1adcd0e>", line 2
    print "Greetings!"
        ^
IndentationError:  expected an indented block

If you want to paste code into IPython, try the %paste and %cpaste
magic functions.
```

It's somewhat unlikely that you'll see an *IndentationError* if you type your program in a worksheet or at the command line, as Sage automatically indents for you at times you should do it.

Here's another error you may see:

```
sage: def greetings()
  File "<ipython-input-34-583f0b909fc6>", line 1
    def greetings()
                   ^
SyntaxError:  invalid syntax
```

Do you see the problem here? If not, spend a few minutes comparing this attempt with the successful one. In particular, look at the end of the line, which is where the carat symbol ^ is pointing. That position is where Sage encounters its problems.

We now return to the assumption that you've defined the function successfully. You can use your new function by typing it, followed by a pair of parentheses, and executing the line.

```
sage: greetings()
Greetings!
```

Sage has done what you requested. It looked up the function named `greetings()`, saw that this function needs to `print 'Greetings!'`, and does what was requested.

An aside. The meaning of the `print` command is hopefully obvious: it prints whatever follows. This is another keyword; you cannot assign it a new value:

```
sage: print = 2
  File "<ipython-input-38-7c34f307b821>", line 1
    print = Integer(2)
          ^
SyntaxError:  invalid syntax
```

In a way, `print` is like a function, but you don't use parentheses to send arguments. You can type parentheses around the text, and Sage won't object, but there is no need. Since it sends text onto the screen, the `print` command can be useful for inspecting a program to see where something went wrong, and we will do this from time to time.

Alas, `print` does not cooperate with LaTeX, so you can't generate pretty output that way. To do that, you have to use LaTeX from an HTML cell (see p. 26), or create a plot.

Another aside. Programmers commonly leave comments in code to explain how a function is supposed to work. The main way of doing this in Sage is with the # symbol; whenever Sage sees this symbol, it ignores everything that follows. For instance, the following implementation of `greetings()` is equivalent to the one above.

```
sage: def greetings():
          # greet the user
          print 'Greetings!'
```

Sage simply ignores the line that begins with the # symbol, and moves immediately to the next one.

It is a *very good idea* to leave comments in functions, especially at the beginning of a sequence of complicated lines. The authors can attest to the fact that they have written code one year that seemed obvious in the moment of programming it, then returned to it weeks, months, or even years later and wondered, "What on earth did I do *that* for?" This then requires several minutes, hours, or even days of puzzling over the code to figure out how it worked. Most of the functions we write in this text are not very long, but we will still include comments in many of them as a way to illustrate the need.

Arguments

An *argument* is a variable that a function uses to receive necessary or merely useful information. We can view an argument as a placeholder for data; its name exists only inside the function, and the information is forgotten once the function terminates. An argument contains only a *copy* of the information sent — not the original information itself. We can illustrate the basic idea by modifying the function of the previous section:

```
sage: def greetings(name):
          print 'Greetings,', name
```

Here, `name` is an argument to `greetings()`. It's also a *mandatory* argument, as we specified no default value. We'll discuss mandatory v. optional arguments presently; for now, it suffices to

understand that you *must* supply a value for mandatory arguments. For example, the following usage of `greetings()` is legitimate in Sage:

```
sage: greetings('Pythagoras')
Greetings, Pythagoras
```

What happened? When Sage reads the command, it sees that you want to call `greetings()` with one argument, which is the string `'Pythagoras'`. It copies that information into the variable `name`, then proceeds to execute all the commands indented beneath the definition of `greetings()`.

On the other hand, now that we have redefined `greetings()`, the following usage results in an error, even though it worked fine earlier:

```
sage: greetings()
TypeError Traceback (most recent call last)
<ipython-input-44-deb3f0bd6096> in <module>()
----> 1 greetings()

TypeError:  greetings() takes exactly 1 argument (0 given)
```

A particular example: the equation of a tangent line. We'll focus on a particular usage for this: suppose you want to write a function that automatically computes the equation at a point $x = a$ of a line tangent to a curve defined by a function $f(x)$. You will recall that the equation of a line with slope m and through a point (a, b) has the form

$$y - b = m(x - a),$$

which we can rewrite as

$$y = m(x - a) + b.$$

In this case, m is the derivative of f at $x = a$; or, $m = f'(a)$. So we can rewrite the equation as

$$y = f'(a) \cdot (x - a) + b.$$

We want a function that produces such an equation, or at least the right-hand side of it.

To define this function, we have to answer several questions:

- What information does the function need?
  We need the function $f(x)$, and the x-value a.
- How do we use that information to obtain the result we want?
  - We first need to compute $f'(a)$. That requires computing $f'(x)$, then substituting $x = a$.
  - We also need to compute b. That is the y-value at $x = a$, so it suffices to compute $f(a)$; that is, substituting $x = a$ into $f(x)$.
- How do we communicate the result to the client?
  For now we'll just `print` it, but eventually we'll discuss a better way.

Putting this information together leads us to the following definition:

```
sage: def tangent_line(f, a):
          # point-slope form of a line
          b = f(a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
          print 'The line tangent to', f, 'at x =', a, 'is',
          print result, '.'
```

Let's put this into practice. A simple example that you can verify by hand is the equation of a line tangent to $y = x^2$ at $x = 1$.

```
sage: tangent_line(x**2, 1)
/Applications/sage-6.7-untouched/src/bin/sage-ipython:1:
DeprecationWarning:  Substitution using function-call syntax
and unnamed arguments is deprecated and will be removed from a
future release of Sage; you can use named arguments instead, like
EXPR(x=..., y=...)
See http://trac.sagemath.org/5930 for details.
  #!/usr/bin/env python
The line tangent to x^2 at x = 1 is 2*x - 1 .
```

Oops! we encountered that `DeprecationWarning` again. This is a one-time thing in each Sage session, and we do get the right answer in the end (`2*x - 1`).

Even so, we ought to change it. Only, where is it occurring? We have to check each statement of the function:

- `df(x) = diff(f,x)`
  This can't be the problem, as it *defines* a function.
- `m = df(a)`
  This uses `df` in function-call syntax, so it's a candidate for the problem. However, it can't actually be the problem, because we defined `df` properly as a function. It *is* appropriate to use function-call syntax with functions.
- `b = f(a)`
  This uses `f` in function-call syntax, so it's a candidate for the problem. As `f` is an argument, into which Sage copied the value supplied when we called `tangent_line()`, we have to look outside the function to see if it is indeed the problem. We called it using the command,

  `tangent_line(x**2, 1)`

  This must be the problem, as `x**2` is an *expression*, not a *function*.

So one way to avoid this warning is to define a function before calling `tangent_line()`, as follows:

```
sage: f(x) = x**2
sage: tangent_line(f, 1)
The line tangent to x |--> x^2 at x = 1 is 2*x - 1 .
```

This isn't very convenient, though, as it requires a lot more typing. It also imposes a condition on the client.

A much better solution is to adjust the program itself, so that it does not assume it receives a function when, in fact, it does not.[41] We can do this by changing the line `b = f(a)` to use an explicit substitution, `b = f(x=a)` . Let's try that.

```
sage: def tangent_line(f, a):
          # point-slope form of a line
          b = f(x=a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
          print 'The line tangent to', f, 'at x =', a, 'is',
          print result, '.'
```

Now we try to execute it:

```
sage: tangent_line(x^2, 1)
The line tangent to x^2 at x = 1 is 2*x - 1 .
```

Since a *DeprecationWarning* appears only once, it may not be obvious that we have actually fixed the problem, as Sage may be suppressing the warning again. You can verify this by restarting Sage: in worksheet mode, click on the Restart button; on the command line, type `exit`, then restart Sage.

Optional arguments. Sage also allows *optional* arguments. In this case, it often makes sense to supply a default value for an argument. This way, you need not supply "obvious" values for the general case; you can supply them only when necessary.

You have already encountered this. Look back at, for instance, Figure 1 on page 52 and Figure 4 on page 52. Both of them speak of options, which are really optional arguments to the functions. For example, you can type

```
sage: point((3,2))
```

or you can type

[41] If you are accustomed to typed languages such as C, Java, or Eiffel, you will notice that this is one case where the use of types — which can often seem overly constrictive — actually helps the programmer. Not only does Sage not *require* types, it really offers no facility for them at all, unless you use Cython. We discuss Cython at the end of the text.

```
sage: point((3,2), pointsize=10, color='blue', alpha=1, zorder=0)
```

as both mean the same thing.

This example also illustrates how optional arguments allow the client to focus on the problem. You often don't care a hoot what color a point is, what size it is, and so on; you just want to see where it's located, typically relative to something else. *If useful*, you might want to specify some of those other options; for instance, you could be looking at several points on a curve, and to help distinguish them you'd color them differently. But this might be completely unnecessary, and even if you wanted to specify the color, the precise size, transparency, and z-order still might not matter. Making these arguments optional means you don't have to specify them.

So how do you name an optional argument when writing your own functions? After the variable's name, place an equals sign, followed by the default value that it should take on. Let's see how we can do this in our function to greet someone: perhaps we'd like the default value to be `'Leonhard Euler'`.

```
sage: def greetings(name='Leonhard Euler'):
          print 'Greetings,', name
```

The argument `name` is now optional. As with the previous version, we can now supply a name:

```
sage: greetings('Pythagoras')
Greetings, Pythagoras
```

…and you can also name the optional argument explicitly (this is useful when there are multiple optional arguments, and you don't want to worry about listing them in the correct order):

```
sage: greetings(name='Pythagoras')
Greetings, Pythagoras
```

…but *unlike* the previous version, we can use the function without a default value:

```
sage: greetings()
Greetings, Leonhard Euler
```

Supplying indeterminates as default arguments. It is often useful to supply an indeterminate as a default argument. To see an example, we return to our example with the tangent lines. It works just fine if the expression is defined in terms of x, but many expressions are more aptly defined in terms of another indeterminate, such as t. Will our function still compute tangent lines for those expressions?

```
sage: var('t')
sage: tangent_line(t**2, 1)
The line tangent to t^2 at x = 1 is t^2 .
```

That's definitely not right. The answer should have been $2t - 1$, not t^2. What went wrong? Several things:

- `df(x) = diff(f, x)` differentiates `f` with respect to `x`, not with respect to `t`. The result is that `df(x)` ends up being `0`, rather than $2t$.
- `m = df(a)` is working with the wrong value of `df(x)`, so we end up with $m = 0$ instead of $m = 2$.
- `b = f(x=a)` asks Sage to substitute `a` in place of `x`, not in place of `t`. The substitution effectively does nothing, with the result that `b` ends up being t^2, rather than 1.
- Put it all together with `m*(x - a) - b` and it becomes clear why Sage returns `t^2` instead of `2*t - 1`.

One way to fix this would be to write a new function that did its work with respect to `t` instead of with respect to `x`. This is a terrible idea, though: the number of possible identifiers is quite large; aside from all 52 upper- and lower-case letters of the Latin alphabet, one can also have combinations of these letters with numbers and the underscore. All of the following names of indeterminates could be used in some perfectly legitimate mathematical context:

- `x`
- `x0`
- `x_0`
- `xi`
- `x_i`

…and that's just indeterminates that start with x. You can play this game with y, t, a, … you get the idea.

So that approach is impractical. The proper way to do it is to specify the indeterminate as an argument to the function itself, something like `tangent_line(f, 1, t)`. *Even better* would be to make the indeterminate an *optional* argument, with default value `x`, so that the client need only specify the indeterminate when necessary. That way, the statements `tangent_line(f, 1)` and `tangent_line(f, 1, x)` mean the same thing. A naïve approach to this would be the following:

```
sage: def tangent_line(f, a, x=x):
          # point-slope form of a line
          b = f(x=a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
          print 'The line tangent to', f, 'at', x, '=', a, 'is',
          print result, '.'
```

The statement `x=x` may look odd, but it makes perfect sense: the first `x` specifies the name of the argument in `tangent_line()`, while the second `x` specifies its default value. Essentially, it states that `tangent_line()` has an optional argument named `x` whose default value is `x`.[42]

So what happens when we try it?

[42]This sort of construction occurs in the Sage source code not infrequently.

```
sage: tangent_line(t**2, 1, t)
The line tangent to t^2 at t = 1 is t^2 + 2*t - 2 .
```

PANIC!

…Well, no, *don't* panic. Let's analyze instead what happened, and try to fix the issue. To do this, we introduce a classical debugging technique of desperation, which goes by the technical name,

print

What does that mean? We print everything that Sage computes in the function, and try from there to see what went wrong. Rewrite the function as follows:

```
sage: def tangent_line(f, a, x=x):
          # point-slope form of a line
          b = f(x=a)
          print b
          df(x) = diff(f,x)
          print df
          m = df(a)
          print m
          result = m*(x - a) + b
          print 'The line tangent to', f, 'at', x, '=', a, 'is',
          print result, '.'
```

You can see that we have a `print` statement after almost every original statement. We left off `result` because that's already printed on the following line. What happens when we execute this version of `tangent_line()`?

```
sage: tangent_line(t**2, 1, t)
t^2
t |--> 2*t
2
The line tangent to t^2 at t = 1 is t^2 + 2*t - 2 .
```

We see that:

- `df` is *correctly* computed as $2t$;
- `m` is *correctly* computed as 2; but
- `b` is *incorrectly* computed as t^2 — as if the substitution didn't take! *Again!*

Notice the subtle difference this time: one substitution actually *did* work. The other one did not. The difference is that one is a mathematical function defined inside our Sage function; the other is an expression defined outside our Sage function.

The apparent solution is to redefine `f` inside our function, as a mathematical function. We then rely exclusively on function notation. There is no danger in this; thanks to the rules of

scope, changing the value of `f` inside `tangent_line()` does not change its original value outside. The successful function adds one line:

```
sage: def tangent_line(f, a, x=x):
          # point-slope form of a line
          f(x) = f
          b = f(a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
          print 'The line tangent to', f, 'at', x, '=', a, 'is',
          print result, '.'
```

That first line redefines the argument `f` as a function. If `f` is an expression, we have avoided the *DeprecationWarning*; if it is a function, Sage will have no trouble assigning it to itself. Try it out:

```
sage: tangent_line(t**2, 1, t)
The line tangent to t |--> t^2 at t = 1 is 2*t - 1 .
```

An end to dysfunctional communication

So far we've been displaying the result of a computation using the `print` command. While this is useful for demonstrating the `print` command, as well as for displaying output, it's an obstacle to efficient automation. You don't want your code to wait for you to see the printed result of some function, then type it into another function. Rather than exhaust yourself this way, it would be much better to have the functions communicate directly. That way, you could call one function inside another and sidestep yourself entirely. You can start a computation, walk away, grab a cup of a caffeinated drink,[43] then come back and find your functions have completed their tasks and are waiting happily for you, tails a-waggin'.

Local v. global scope. Suppose, for instance, you would like to write a Sage function that plots a mathematical function $f(x)$ and the line tangent to f at that point. You could, of course, have it compute the tangent line by copying and pasting the code in `tangent_line()` into the new function, but that both wastes resources and makes the new function harder to read. You could instead employ **modularity** by letting `tangent_line()` compute the tangent line, and letting your function focus purely on plotting the function and the line — but how would the new function use `tangent_line()`'s result? On the one hand, `tangent_line()` has a variable called `result`, so perhaps we could do something like this?

[43] This originally referred to a particular caffeinated drink, while taking potshots at another, more popular one, because one of the authors is a little immature. A second author commented ZOMGPONIES and subsequently convinced the first to change it to the current, generic discourse on caffeinated drinks.

```
sage: def plot_tangent_line(f, a, x=x):
          # use tangent_line() to find the line
          tangent_line(f, a, x)
          plot(result, a-1, b-1)
```

That won't work, and if we were to try it, we would encounter the following error:

```
sage: plot_tangent_line(x^2, 1)
NameError:  global name 'result' is not defined
```

This error tells us that if `result` exists anywhere, it doesn't exist in the "global" Sage environment, so we use its name in vain. We only defined `result` in the `tangent_line()` function, so it remained **local** to that function.

There are two ways around this. One of them is declare `result` as a global variable in the `tangent_line()` function using the `global` keyword; afterwards, result would be accessible outside it, as well:

```
sage: def tangent_line(f, a, x=x):
          # point-slope form of a line
          global result
          f(x) = f
          b = f(a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
```

However, this is a *terrible* idea, for several reasons. First, `result` is about as uninformative a name as you could give. If every function placed its result in a variable named `result` — well, it could work, actually, but it would be rather chaotic. Besides, there's a better way.

Returning from chaos. Sage provides a convenient command for this with another keyword, `return`. Every Sage function returns some value, even if you omit this keyword. You can verify this with the functions we have defined, using a `type()` statement:

```
sage: nothing = tangent_line(f, 1, t)
The line tangent to t |--> t^2 at x = 1 is 2*t - 1 .
sage: type(nothing)
<type 'NoneType'>
```

What Sage returns in these instances is an object called `None`. Programmers often use `None`[44] to indicate that something has not been initialized, but that is a different discussion.

Unlike other computer languages, Sage does not constrain you to return only one result; it is possible to return several results at once. Simply list them after the `return` keyword, separating

[44] Or the analog in other languages: `NULL`, `NIL`, ...

them with commas. For instance, the following program will return both the derivative *and* the antiderivative of a function, assuming both exist:

```
sage: def deriv_and_integ(f, x=x):
          # BOOOO-RING
          df(x) = diff(f, x)
          F(x) = integral(f, x)
          return df, F
```

If this reminds you of how Sage allows you assign to more than one variable in one statement (see p. 30), you are correct! You could use the function above in the following way:

```
sage: f(x) = x**2
sage: df, F = deriv_and_integ(f)
sage: df
x |--> 2*x
sage: F
x |--> 1/3*x^3
```

If you've had experience with languages that don't allow this, you understand how complicated, and somewhat unintuitive, it can be to return more than one value at a time.[45]

What we want to do is modify the `tangent_line()` function to `return` the line, rather than `print` it. This is fairly straightforward:

```
sage: def tangent_line(f, a, x=x):
          # point-slope form of a line
          f(x) = f
          b = f(a)
          df(x) = diff(f,x)
          m = df(a)
          result = m*(x - a) + b
          return result
```

It's that simple. What happens when you use it?

```
sage: tangent_line(t**2, 1, t)
2*t - 1
```

[45] For instance, in C++ one can either pass arguments by reference, resulting in the signature

```
void deriv_and_integ(Function f, Function & df, Function & F, Indeterminate x=x);
```

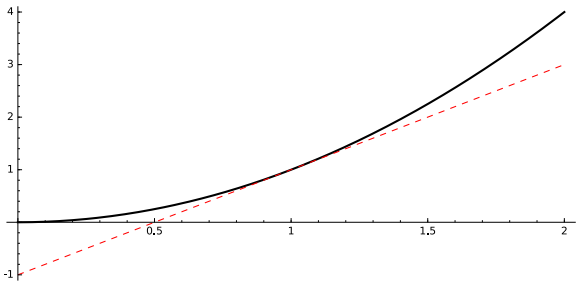
or create a `struct` or `class` that contains fields for both answers:

```
struct DandI_result { Function df; Function F; };
DandI_result deriv_and_integ(Function f, Indeterminate x=x);
```

but the Python approach seems more elegant.

The output may not seem as informative as before, but that's because we've pruned it to its essence. You could change the `return` statement to include the text information we had before, but the bare essence is much more useful. Why? we can now do the following:

```
sage: p1 = plot(t**2, 0, 2, color='black', thickness=2)
sage: par_line = tangent_line(t**2, 1, t)
sage: p2 = plot(par_line, 0, 2, color='red', linestyle='dashed')
sage: p1 + p2
```



Notice what we're doing: we're taking the result of `tangent_line()`, assigning its value to a variable named `par_line`, then passing that as an argument to `plot()`. You can do this *even if you did not envision plotting the result of* `tangent_line()`*!* Indeed, you'll notice we didn't mention the possibility of plotting the tangent line at all before this point. Much of the power of functions lies in the fact that you can use them in situations not imagined when the functions were designed; clients still enjoy the benefits of having the code already written for them.

Pseudocode

People who are serious about working on a computer have to communicate their ideas to each other. Not everyone uses the same programming language, and for the sake of abstraction and communication it's a good idea to avoid reliance on a particular language to describe the solution to a problem. For this reason, mathematicians often use *pseudocode* to describe a computer program — or, to be more precise, the algorithm used in the program. An *algorithm* is a sequence of steps with well-defined input (information used) and output (information returned).

There are different ways of expressing pseudocode; some examples appear in Figure 17. You should notice several properties they all share:

- All of them specify the algorithm's name, required information ("input"), and promised result ("output").
- All of them use some form of indentation to show how some code depends on others.
- All of them rely on "plain" English and mathematical symbols to communicate what is to be done.
- *None* of them resembles computer code in a particular computer language.

This text will provide pseudocode for you to implement, and will ask you to write your own pseudocode on occasion. We have to adopt some standard, and we will do the following:

- Identifiers for variables or names of algorithms appear in italics.
- Keywords for control structures and fundamental information will appear in **bold type**. Right now, we introduce the following keywords:
  - **algorithm** appears at the beginning, and declares the name of an algorithm

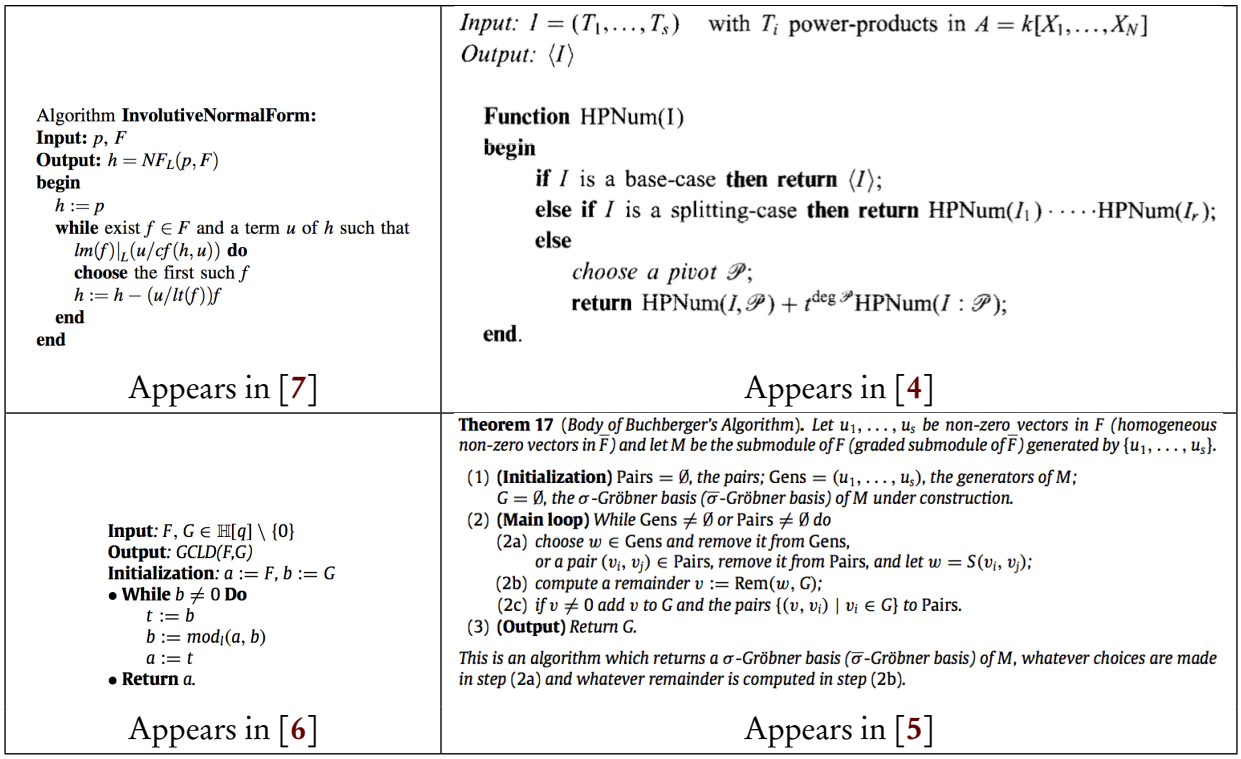


FIGURE 17. Pseudocode from various mathematical publications

- **inputs** appears immediately after **algorithm**, and below it comes an indented list of the required information, *along with the sets they come from*
- **outputs** appears immediately below the list of **inputs**, and below it comes an indented list of promised information, *along with how it relates to the inputs*
- **do** appears immediately below the list of **outputs**, and below it comes an indented list of instructions
- **return** appears as the last statement in the instructions[46]

We can now describe our `tangent_line()` function in pseudocode as follows:

[46]It is common to see **return** placed in other places within the list of instructions, but we will adopt the convention that **return** is always the last statement. While it is sometimes inconvenient to organize the code around this convention, it can help with both debugging and readability, which is important for those who are first learning.

algorithm *tangent_line*
inputs
- $a \in \mathbb{R}$
- x, an indeterminate
- f, a function in x that is differentiable at $x = a$

outputs
- the line tangent to f at $x = a$

do
let $m = f'(a)$
let $b = f(a)$
return $m(x-a)+b$

The following observations of this code are in order.

- We make assignments with the traditional mathematical "let" statement.[47]
- Not only does it not look much like our Sage code, we omit some Sage-specific tasks, such as assigning $f'(x)$ to a variable.[48]
- Both the inputs and some of the commands are listed in a different order.

As we go through the text, we will add additional pseudocode keywords, and discuss how to accomplish additional tasks.

In practice, we almost always formulate pseudocode *before* implementing it in code. We have reversed this practice here, mostly to get your hands wet from the start, but an important aspect of working with a computer is to think about *how* you solve the problem *before* actually writing up the solution — and, really, a program is nothing more than the writeup of a solution to the problem, "How do I do this, that, and the other, given such-and-such and so-and-so?" If you've had another programming class before, you may have been raised in the rather bad tradition of not thinking about what to do before doing it. Try to resist that; we'll help in the exercises, by requiring that you formulate pseudocode *as well as* writing a program.

Scripting

Many mathematicians have a set of tasks that they need to repeat whenever they work on something. Writing Sage functions can help simplify this, but if you type them in a particular session, they apply only to that session, either on the command line or in a worksheet. If you quit the session or open a different one, Sage forgets those functions, and you have to re-define them. It isn't very convenient to copy and paste or (worse) re-type these functions whenever you start a new session, so it's convenient to keep a library of files that record them, and that we can load and use at will.

This is where Sage scripts come in. A "script" is basically a sequence of Sage statements saved to a file. It is convenient to load the file into Sage, thereby avoiding the hassle of retyping everything. The format of a Sage script is identical to whatever we'd write in a Sage cell: it can be a sequence of simple statements, but it can also define one or more functions. Indeed, a large

[47] As you can tell from the samples in Figure 17, there is quite a bit of variety here. Where we have "let $m = f'(a)$," some would write "$m \coloneqq f'(a)$" and some computer languages have adopted this convention so that assignment has the form `m := df(a)` (for example, Pascal, Eiffel, Maple). Some will even simply write "$m = f'(a)$."

[48] Technically, we don't have to do this in Sage, either, but it makes things a lot easier.

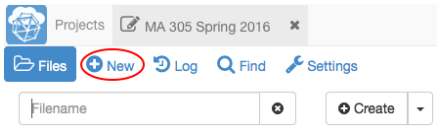
amount of Sage consists of Sage scripts that people have written on account of some need and, due to the task's usefulness, the script subsequently made its way into Sage itself.

To show how to create a Sage script, we recall our program on computing the line tangent to a curve:

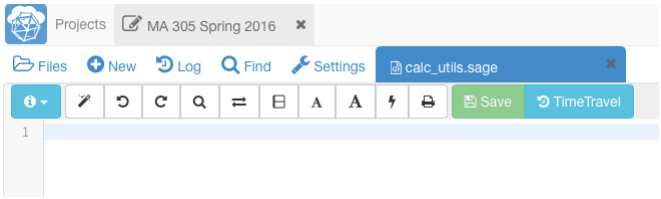
```
def tangent_line(f, a, x=x):
    # point-slope form of a line
    f(x) = f
    b = f(a)
    df(x) = diff(f,x)
    m = df(a)
    result = m*(x - a) + b
    return result
```

To create a script, do one of the following.

- If you run Sage as a worksheet operated by an independent server, you'll have to speak to your instructor or to the server's administrator. This is not difficult to do, but it is a little complicated, so we leave it to them.
- If you run Sage from the command line, bring up a text editor on your computer, type the program above into the editor, then save the file with the name `calc_utils.sage`. We recommend saving it to a special directory in your home directory titled, `SageScripts`, but you should pick someplace that will be both easy to remember and easy to access.
- If you run Sage through the SageMathCloud server, open a project, then select from the top menu `New`.

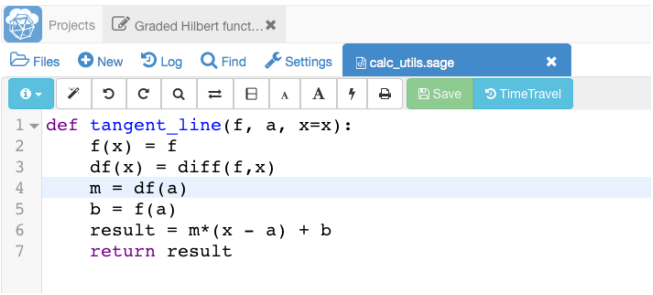


In the textbox beneath the directions to, "Name your file, folder, or paste in a link," type the name `calc_utils.sage`. Then click on "File" (over to the right) and Sage will bring up a new tab with that name.



Go ahead and type the function. If you find the text too small, click on the "larger" `A` beneath `Settings` in the screenshot below. That will make the text larger.[49]

[49] One of the authors routinely clicks this six to eight times, and that's not because he's old. — Well, not *that* old.



Save it. (It should actually save automatically every few moments, but it never hurts to give the button an extra click.)

Once you have written and saved your script, you can turn to a Sage session (either at the command line or in the worksheet) and, at this point, type the following (don't actually type until you read what appears below):

```
sage: %attach /path/to/file/calc_utils.sage
```

Before you actually do this, we want to describe two potential errors, as well as indicate a convention you should be aware of.

First, the convention. We have written `/path/to/file/` as a "template" for you to fill in. It's basically a fill-in-the-blank used commonly as an indication that you have to fill in the correct value:

- If you're using Sage from a worksheet from an independent server, you need to ask your instructor or the server's administrator.
- If you're using Sage from the command line, you should try `'~/SageScripts'`, as that corresponds to the directory we suggested earlier.[50]
- If you're using the SageMathCloud server *and* you followed our directions above, *you need not type* `/path/to/file` *at all!* Otherwise, if you created a special directory to keep your Sage scripts, you need to type the path to that directory.

Once we move past this convention, we move to the possible errors.

If you are in the wrong directory, or you typed the filename wrong, or you didn't follow our advice above to wait to type that in, you will likely see an error along these lines:

```
sage: %attach /path/to/file/calc_util.sage
IOError:  did not find file u'calc_util.sage' to attach
```

This indicates that there is something wrong in what you filled in for `/path/to/file/` or in the filename itself. In this particular example, both are wrong: we didn't actually change `/path/to/file/` to the correct path, *and* we left the `s` off the filename `calc_utils.sage` .

Now for the second error. We sometimes type things wrong. That's alright; it's part of being human; no need to get worked up about it. After all, the computer will get worked up on its own. It is possible to type something wrong, and when that happens, you will get a Syntax error. A common one for beginners will likely be this one:

[50] The tilde character (~) is a standard way to reference a user's home directory. We had suggested creating a directory named `SageScripts` in your home directory, so that should do the trick.

```
sage: %attach /path/to/file/calc_utils.sage
IndentationError:  unindent does not match any outer indentation
level
```

In this case, pay attention to the line where Sage complained, and make sure the indentation lines up with a previous block of code. A similar error will be this one:

```
sage: %attach /path/to/file/calc_utils.sage
    b = f(a)
  ^
SyntaxError:  invalid syntax
```

This should *not* happen to you, but in case it does, and everything looks exactly perfect: the problem is that you copied and pasted text. Appearances in a text editor can be deceiving, and when you copy from one file to another, you can pick up hidden formatting characters, or something similar. The solution in this case is to eliminate the hidden formatting character by deleting those spaces, then retyping them.

With those clarifications out of the way, now try to load the file. If all goes well, Sage will remain silent. You should then be able to execute your function in the usual way:

```
sage: %attach /path/to/file/calc_utils.sage
sage: tangent_line(x**2, 2)
4*x - 4
```

Once you successfully attach `calc_utils.sage` to a Sage session, you can make changes to it and Sage will automatically incorporate the changes, leaving a message along these lines:

```
### reloading attached file calc_utils.sage modified at 08:39:00
###
```

Interactive worksheets

If you're working in a Sage worksheet, you have access to a feature that lets you create an easily-manipulated demonstration. We call this feature *interactive worksheets*. An interactive worksheet has one or more *interactive functions* that allow the user to manipulate their arguments in a "hands-on" way through interface objects like text boxes, sliders, and so forth. Aside from the function itself, there are two key steps to creating an interactive worksheet:

- On the line before the function, but in the same cell, type `@interact` .
- Decide which of the following interface objects you want, and supply the function with "optional" arguments whose default values are initialized to these objects:
  - an input box (to provide a value)
  - a slider (to select a value from a range)
  - a checkbox (to turn some property on or off)
  - a selector (to select some property out of several)
  - a color selector (to… well, hopefully the reason for this is obvious)

`input_box()`	a box for the user to type values, usually numbers; specific options include • `width=`*box's width* (104)
`slider(`*a*`, `*b*`)`	a line with a knob the user can slide left and right, ranging from *a* to *b* with intermediate values
`slider(`*a*`, `*b*`, `*c*`)`	similar to `slider(`*a*`, `*b*`)`, except the user can only slide in increments of *c*
`slider(`*L*`)`	similar to `slider(`*a*`, `*b*`)`, except the user can only select options in the collection *L*. (We are still not telling you what a collection is, but you can still use a tuple, as before.)
`checkbox()`	a box that the user can click on or off; we illustrate its use in a later chapter
`selector()`	a drop-down menu or button bar specific options include • `values=`*collection of values to select from* • `buttons=`whether to use buttons (`False`) • `nrows=`*number of rows if using buttons* (1) • `ncols=`*number of columns if using buttons* (depends on number) • `width=`*button's width, if buttons* (depends on value)
`Color(`*name*`)`	a button that pops up a color chooser; default color is the one supplied by *name*, such as `'red'`
`Color(`*r*`, `*g*`, `*b*`)`	same as `Color(`*name*`)`, except the default color is the red-green-blue combination given by *r*, *g*, *b*

FIGURE 18. Templates for interface objects in an interactive function

`label=`*string*	a label, placed to the left; it may include LaTeX between dollar signs
`default=`*value*	the object's initial value

FIGURE 19. Options common to all interface objects in an interactive function *except* `Color()`

You will find templates for these object in Figure 18, and some common options for all the elements in Figure 19. *This list is not comprehensive,* as we have omitted some options that we think you are less likely to find useful. Other interface objects may also be available that we aren't aware of (they may have been added after we wrote this).

Sliders and selectors are similar in that they can let you choose from a small range of values; for instance, `slider((0, 0.25, 0.5, 0.75, 1))` and `selector((0, 0.25, 0.5, 0.75, 1))` both let you choose from the same list. In this case, however, a slider would be the better choice, as you can order the values from left to right, and a slider communicates that sense better. If the

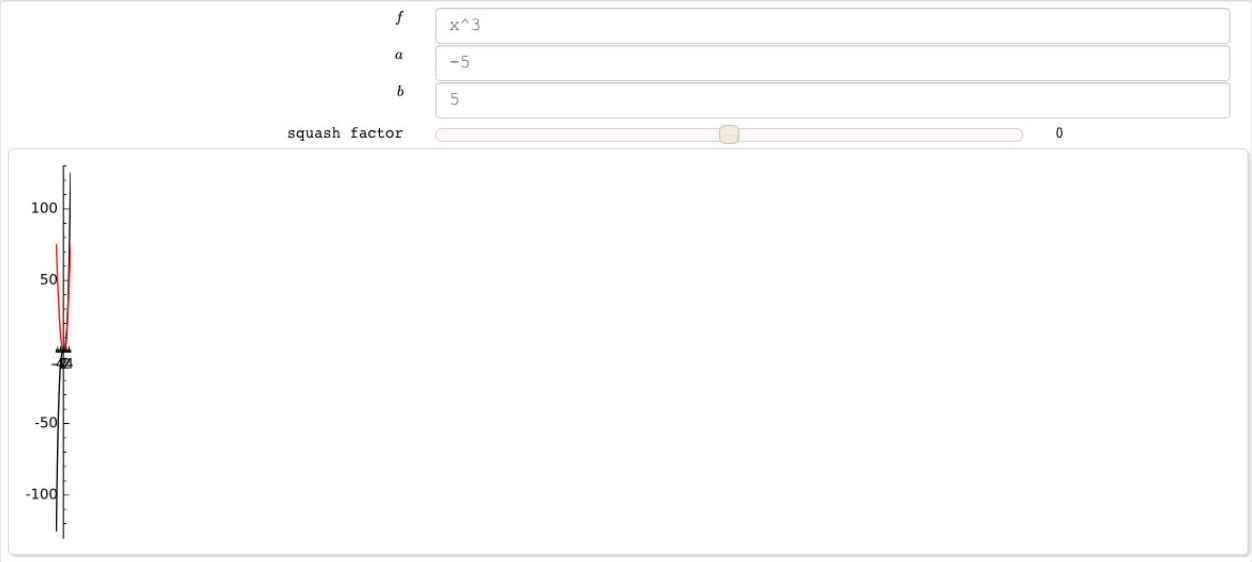


FIGURE 20. An unfortunate squashing

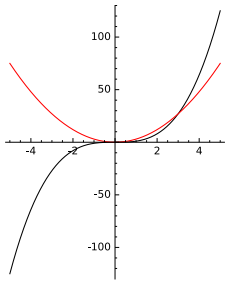
values do not fit well in a left-to-right paradigm, you'd be better off with a selector. Also, sliders don't label the individual values, so if you think it especially important that each value be labeled, it's best to use a selector even if the values *do* fit the left-to-right paradigm.[51]

We illustrate this on two examples. The first is quite simple, merely to illustrate the principle: it allows the user to type a function f, and plots both f and its derivative on an interval $[a, b]$.

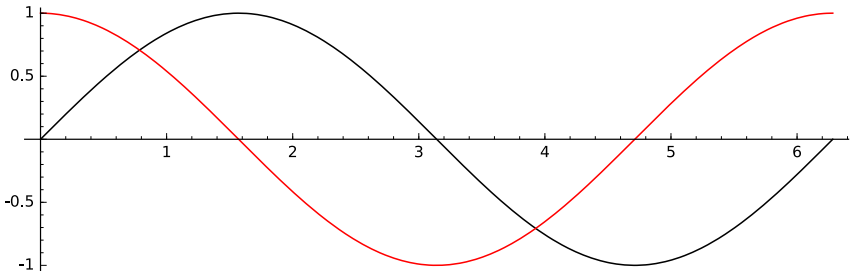
```
sage: @interact
      def plot_f_and_df(f=input_box(default=x**3,label='$f$'),\
                        a=input_box(default=-5,label='$a$'), \
                        b=input_box(default=5,label='$b$'), \
                        w=slider(-19,19,1,default=0, \
                                 label='squash factor')):
          p = plot(f, a, b, color='black')
          df = diff(f)
          p = p + plot(df, a, b, color='red')
          show(p, aspect_ratio=(20-w)/20)
```

If you type this into a Sage worksheet correctly, you should see something akin to Figure 20. You can see that the input boxes have nicely-formatted labels. (As you may have guessed from our use of dollar signs, it's relying on LaTeX.) Unfortunately, that's not an especially clear plot; it's a bit too squashed horizontally. We can fix this by moving the squash factor right; if you slide the knob on the "squash factor" slider all the way to the right, you get a rather nice picture the moment you let go:

[51] Of course someone will find an exception to each of these guidelines; they are merely recommendations.



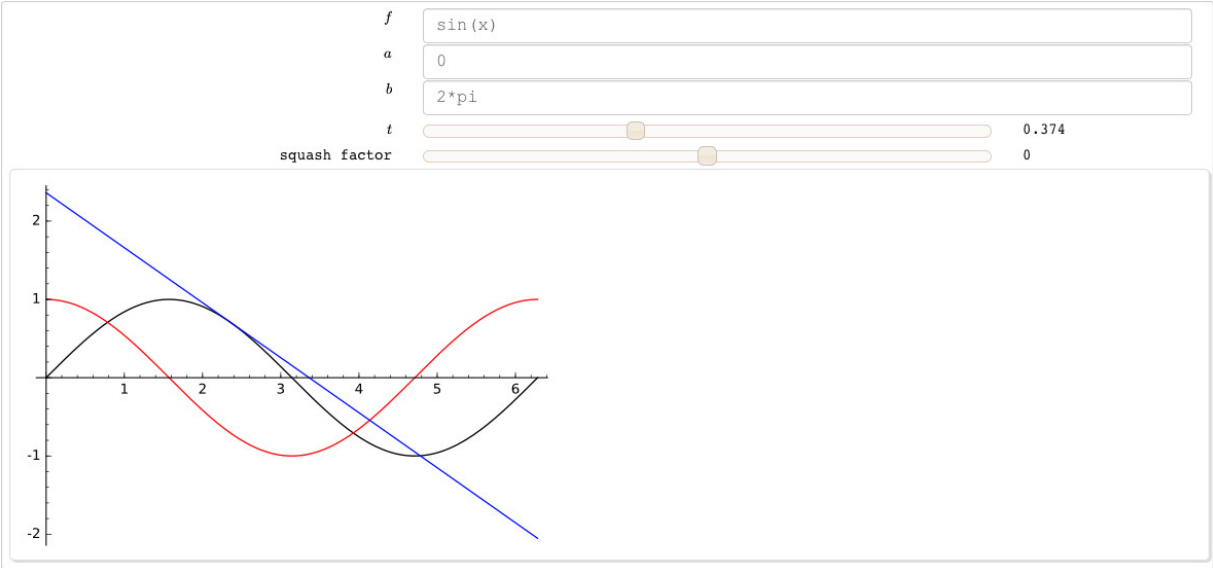
You can also change f, a, and b, but they're input boxes, so rather than sliding anything, you have to change the text. As $a < b$ you should get a reasonable-looking graph the moment something changes. Try another function, such as $\sin x$, with $a = 0$ and $b = 2\pi$, and the graph should change to this (make sure you also change the squash factor back to 0):



You can of course call functions you've defined from inside an interactive function. Let's write an interactive function that plots f and its derivative on $[a,b]$, *as well as* a line tangent to f at a point c somewhere between a and b. We can call the `tangent_line()` function we already wrote to generate the tangent line. While we're at it, we'll switch the defaults to $\sin(x)$ over $[0,2\pi]$.

```
sage: @interact
      def plot_f_and_df(f=input_box(default=sin(x),label='$f$'), \
                        a=input_box(default=0,label='$a$'), \
                        b=input_box(default=2*pi,label='$b$'), \
                        t=slider(0,1,default=0.375,label='$t$'), \
                        w=slider(-19,19,1,default=0,\
                                 label='squash factor')):
          p = plot(f, a, b, color='black')
          df = diff(f)
          p = p + plot(df, a, b, color='red')
          # c tells us how far to move along the interval [a,b]
          # 0 means a; 1 means b; values in between proportional
          c = a + t*(b-a)
          p = p + plot(tangent_line(f, c), a, b)
          show(p, aspect_ratio=(20-w)/20)
```

When you put this into Sage, you should see the following:

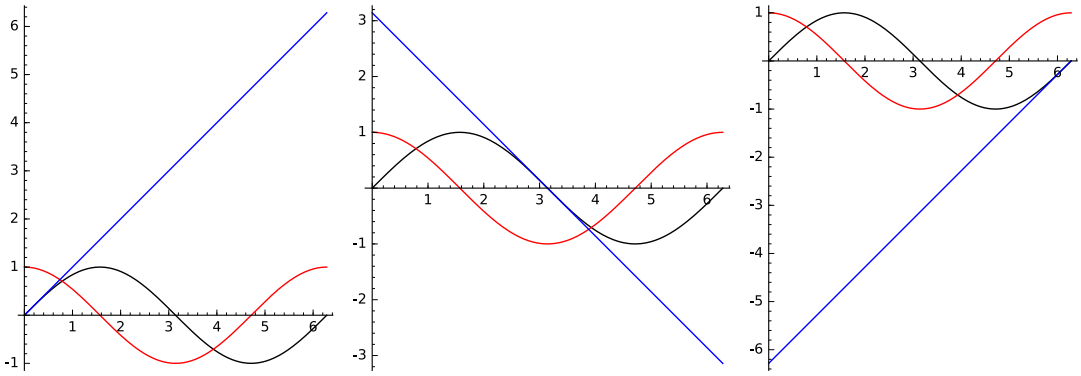


You may have noticed that we didn't have the user specify the point c directly. Instead, she specifies c *indirectly*, by choosing a value t from 0 to 1. The code then calculates a value of c between a and b using the following parametrization:

$$c = a + t\,(b - a).$$

This parametrization is an enormously useful tool, in part because it is relatively easy:

- when $t = 0$, then $c = a + 0 = a$;
- when $t = 1$, then $c = a + (b - a) = b$;
- when $t = .5$, then $c = a + .5(b - a) = .5a + .5b = {}^{a+b}/_{2}$, halfway between a and b;
- and so forth.



Experiment with the slider to see how various values of t naturally let the user select a good point between a and b.

We haven't shown how all the interface objects work, but we have shown the ones we think are most important, as well as an elegant way to have the user select a point in an interval without having to check whether the point actually lies in the interval. We encourage you to experiment with the other objects, especially selecting a color.

Exercises

True/False. If the statement is false, replace it with a true statement.

1. Functions allow us to reuse common groups of commands.
2. If an argument holds the value of some variable outside the function, changing the argument's value also changes that variable's value.
3. Sage allows you to specify optional arguments.
4. If a Sage function does not explicitly `return` a value, then it returns `None`.
5. When we pass indeterminates as arguments to Sage functions, we must redefine any mathematical functions that depend on them.
6. You cannot combine your functions with Sage functions.
7. No one uses pseudocode in real life.
8. Pseudocode communicates ideas using "plain English" and mathematical symbols, rather than a particular language's symbols and formats.
9. The best way to reuse a group of Sage functions in different sessions is to copy-and-paste the code.
10. The `%attach` command permanently attaches a script to Sage, and automatically reloads it when you change it.
11. You can use interactive functions from command-line Sage.
12. If the user has to select from 100 equally-spaced values on a number line, it's best to use a slider.
13. If the user has to select a function from the set $\{\sin x, \cos x, \ln x, e^x\}$, it's best to use a selector.
14. To find the number one-fourth of the distance from 2 to 10, substitute $t = 0.25$ into the expression $2 + 8t$.
15. A disadvantage to interactive functions is that you can't break up their tasks into tasks performed by other functions.

Multiple Choice.

1. We start the definition of a function in Sage using
   A. **algorithm**
   B. its type
   C. its name
   D. `def`
2. The format for an optional argument is
   A. not to list it
   B. to preface its identifier with the keyword `optional`
   C. to preface the function name with the keyword `def`
   D. to place after its identifier the symbol `=` and a default value
3. Which of the following is not an accurate description of an argument?
   A. a stand-in for a variable outside the function
   B. a copy of some data outside the function
   C. a variable whose value you may not know in advance
   D. information the function may or may not need to perform its tasks
4. The best way to report a function's result back to the client is to
   A. list the result in a `return` statement
   B. assign the value to one of the arguments
   C. assign it to a global variable called `result`
   D. assign the function's name to a variable
5. A function that lacks a `return` statement:
   A. raises an error

   B. returns the empty set
   C. returns `None`
   D. does nothing special
6. When a list of commands is subject to a keyword that starts a control structure, such as `def`, you should:
   A. indent the commands in question
   B. add a colon at the end of the control structure's statement
   C. reverse the indentation when the list is finished
   D. all of the above
7. Which of the following properties does our pseudocode standard share with Sage code?
   A. Statements subject to control structures are indented.
   B. We can use `def` in both pseudocode and Sage code.
   C. We specify the inputs' types in Sage, just as we specify which set an input comes from in pseudocode.
   D. all of the above
8. The best way in Sage to substitute for an indeterminate that was passed as an argument is by:
   A. function-call substitution
   B. dictionary substitution
   C. keyword assignment
   D. function-call substitution *after* redefining the function
9. Which of the following identifiers can we *not* assign values to?
   A. a valid identifier
   B. a keyword
   C. a constant
   D. a function name
10. When the user changes a setting in an interactive function, the only guaranteed effect is to change what part of the function?
   A. a global variable
   B. a local variable
   C. an argument of the function
   D. the function's name

Short answer.

1. Explain modularity and why it is a good practice when writing programs.
2. Are modularity and modular arithmetic the same thing?

Programming.

1. Sage does not automatically add arrowheads to a curve. Since the derivative tells us the slope of the curve at a given point, we can determine what directions arrows at the end should point by computing derivatives at the endpoints and putting arrows there. Write a function `arrowplot()` that takes as argument a function f, two endpoints a and b, and an "arrow length" Δx, then plots f on $[a, b]$, adding arrowheads using the principle that $(a, f(a))$ and $(a - \Delta x, f(a) - f'(a)\Delta x)$ would give us two points on a line tangent to f at a (and which would specify a suitable arrow). (We leave it to you to modify this definition for an arrowhead at b.)
2. Write pseudocode for a function that computes and returns the normal line to f at $x = c$. Then:

(a) Implement this in Sage code.
(b) Pick any transcendental function f, and any point c where f is transcendental.
(b) Use this code and the `tangent_line()` function to plot both the normal and tangent lines to f at $x = c$.
(c) Write an interactive function to sketch the graph of a function, a tangent line at a point $x = c$, and the normal line at $x = c$. Aside from any obvious interface elements needed, include at least a slider to control the aspect ratio of the resulting graph.

Your implementation will sometimes fail; for instance, if you use $f = (x-1)^2 + 2$ and $c = 1$, you should encounter a `ZeroDivisionError`.

DON'T PANIC!

You should actually *expect* this error, since computing a perpendicular requires a reciprocal, which means division, which opens the possibility of division by zero. We address this issue in the chapter on Decision-Making. Make sure your code works with functions and points that don't misbehave.

3. Write pseudocode for a function named *avg_value* whose inputs are a function f, and indeterminate x, and the endpoints a and b of an interval I. The function returns the average value of f on the interval. (You may need to review some calculus to solve this problem.)
4. Let's define the operation $a * b = ab + (a + 2b + 1)$.
   (a) Write pseudocode for a function that accepts two integers a and b and returns $a * b$.
   (b) Implement this pseudocode as a Sage function.
   (c) Test your function on several different values of a and b.
   (d) Is there a value of a such that $a * b = b$, regardless of the value of b?
5. Write a Sage function named `dotted_line_segment()` that accepts seven arguments named `x1`, `y1`, `x2`, `y2`, `color`, `pointsize`, and `pointcolor`. The colors' default values should be `'black'`, and the default color of `pointsize` should be `60`. The function returns the sum of:
   - a line segment that connects those points and whose color is `color`, and
   - two points, both of color `pointcolor` and of size `pointsize`, whose locations are (x_1, y_1) and (x_2, y_2).

   (a) Use this function to plot a line segment connecting the points $(0, 0)$ and $(1, 4)$ whose color and point color are both black.
   (b) Use this function to plot a black line segment connecting the red points $(0, 6)$ and $(2, 0)$.

   You need not write pseudocode for this function.
6. Implement the following pseudocode in Sage.

algorithm *Taylor_Truncated4*
inputs
- $a \in \mathbb{R}$
- x, an indeterminate
- f, a function in x that is integrable at $x = a$

outputs
- the truncated Taylor series for $f(x)$ around $x = a$

do
let *result* $= f(a)$
add $f'(a) \cdot (x - a)$ to *result*[a]
add $f''(a) \cdot (x - a)^2 / 2$ to *result*
add $f'''(a) \cdot (x - a)^3 / 6$ to *result*
add $f^{(4)}(a) \cdot (x - a)^4 / 24$ to *result*
return *result*

[a] Here's a hint on this one. To "add … to result" use the construction `result = result + …`

Check it by finding the truncated Taylor series for e^x at $x = 1$, and comparing the truncated series' value at $x = 1.1$ with the value of $e^{1.1}$.

7. Explain how the "squash factor" we used in our interactive functions was used to set the aspect ratio of the plot. Your explanation should address why the leftmost value squashes the graph as far left as possible, why the rightmost value stretches the graph as far right as possible, and why the middle value sits somewhere in between.

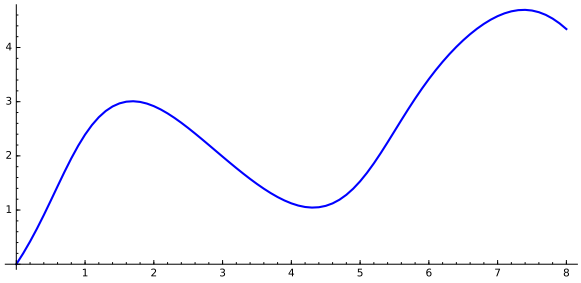
Repeating yourself definitely with collections

There are a number of cases where you want to repeat a task several times. A classic example comes from differential equations; suppose we know

$$dy/dx = \sin y + 2\cos x,$$

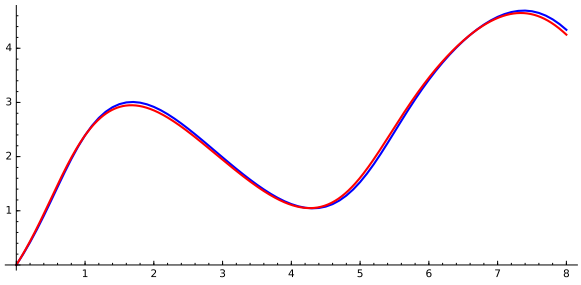
that is, at any point (x, y), the value of y is changing by $\sin y + 2\cos x$. It is difficult, and often impossible, to find the exact formula for y in terms of y, so it is necessary to approximate "future" values of y from a starting point (x, y).[52] In this case you usually decide to make a number of very small "steps forward" with x, and re-evaluate y at each point, tracing out the resulting behavior. In this case, suppose we start at $(0, 0)$; then $y'(0, 0) = 2$, which suggests the function wants to move forward along a line with slope 2. After all, you're computing the derivative, which is the slope of the tangent line, which goes in the same direction as the curve at that point. This technique of moving along the tangent line is known as **Euler's Method**.

However, the curve wants to go in the same direction as its tangent line only for a moment; as soon as we step off the point $(0, 0)$, the derivative changes ever-so-slightly, and the tangent line is no longer valid. To avoid making too much error, we take a tiny step, say $\Delta x = 1/10$, along this line, which takes us to the point $(.1, .2)$. Here, $y'(.1, .2) \approx 2.1$, so we step out along a line of slope 2.1, which takes us to the point $(.2, .41)$. Here, $y'(.2, .41) \approx 2.0$, which takes us to $(.3, .61)$. Here, $y'(.3, .61) \approx 1.9$, which takes us to $(.4, .8)$. Here, $y'(.4, .8) \approx 1.8$, which takes us to $(.5, .98)$. Here, $y'(.5, .98) \approx 1.6$, which takes us to $(.6, 1.14)$. Go far enough, and you'll trace out a picture something like this:



To generate this curve, we used a step size of $\Delta x = 1/10$. Repeat the process from $(0, 0)$ with a smaller step size, say $\Delta x = 1/100$, and you'll get a slightly different result:

[52] For instance, this is how weather prediction works.



This reflects the fact that our approximation incorporates some error. Still, it's not *that* bad; the approximations are pretty close. Then again, we obtained the blue curve using 80 points, and the red curve using 800 points. You don't want to do that by hand, do you?

When a task (or set of tasks) has to be repeated more than once on the result of the previous application, we call this repetition **iteration**. Iteration pops up repeatedly in computational mathematics, so programming languages typically offer a control structure called a **loop**. Unlike Euler's method, we don't always know in advance how many times the loop must repeat, so many loops are **indefinite**. Nevertheless, it is very often the case that we can determine the exact number of times a task must repeat from the outset; we call this a **definite loop**. This chapter introduces definite loops; we postpone indefinite loops for a subsequent chapter.

How to make a computer repeat a fixed number of times?

Pseudocode. We can describe Euler's Method in pseudocode as follows.

algorithm *Eulers_method*
inputs
- df, the derivative of a function
- (x_0, y_0), initial values of x and y
- Δx, step size
- n, number of steps to take

outputs
- approximation to $(x_0 + n\Delta x, f(x_0 + n\Delta x))$

do
  let $a = x_0$, $b = y_0$
  repeat n times
    add $\Delta x \cdot df(a, b)$ to b
    add Δx to a
  return (a, b)

We use **repeat** in pseudocode to indicate that a set of tasks are to be repeated a certain number of times, and indent the tasks to repeat.

Sage code. As with most computer languages, Sage has no keyword named `repeat`.[53] For definite loops, Sage uses a more general-purpose keyword, `for`. When you know you are to repeat a task n times, the construction is fairly simple:

`for` *variable* `in xrange(n):`

For *variable* you choose an identifier that will hold the number of the loop each time you pass through it.

We can now implement Euler's Method in Sage:

```
sage: def eulers_method(df, x0, y0, Delta_x, n):
          # starting point
          a, b = x0, y0
          # compute tangent lines & step forward
          for i in xrange(n):
              b = Delta_x * df(a, b) + b
              a = Delta_x + a
          return a, b
```

This looks pretty straightforward. Let's try it out:

```
sage: df(x,y) = sin(y) + 2*cos(x)
sage: eulers_method(df, 0, 0, 1/10, 80)
```

If you do try this, you'll notice it takes an *awfully long time* to sort itself out, certainly more than a few seconds. To see why this happens, interrupt the calculation (press the "Stop" button in the cloud, click the "Action" menu and click "Interrupt" on an independent server, or hold `Ctrl`and press `C` at the command line) and run the code again with a smaller value of n:

```
sage: eulers_method(df, 0, 0, 1/10, 10)
(1, 1/5*cos(9/10) + 1/5*cos(4/5) + 1/5*cos(7/10) + 1/5*cos(3/5) +
1/5*cos(1/2) ...
```

(The ellipses at the end indicate there's a lot more after that. *Quite* a lot more!)

Do you see what's going on? Sage is computing exact values; and the exact value of this number grows more and more complicated with each iteration. You can modify the code to simplify `b` after each computation, but it doesn't really help. This simply illustrates a drawback of symbolic computation: to gain "exact" values, you sacrifice time. But there's no need to sacrifice that here! After all, we're approximating the value anyway. In that case, let's turn to floating-point values, and see if that speeds things up. Let's replace `1/10` by `.1`, and let's see how it turns out.

[53] Our use of **repeat** is meant to illustrate how pseudocode aims for clarity of communication, rather than mimicry of a particular language, and it is not unheard-of to see **repeat** used in pseudocode. That said, some programming languages do feature a `repeat` structure for loops.

```
sage: eulers_method(df, 0, 0, .1, 80)
(7.99999999999999, 4.340418570291038)
```

This comes to (8, 4.34). Not only did we obtain an answer quickly, it was nearly instantaneous! It's clearly a better idea to rely on floating-point when you know you're approximating anyway.

What just happened? What takes place when we execute a loop? Let's examine what happens, looking closely at the values of a and b each time we pass through the loop.

When we ran the program, both a and b take on the value `0`, because that's what we passed as the arguments for the initial value. We next come to the line

```
for i in xrange(n):
```

Recall that this instructs Sage to repeat the subsequent tasks n times. The value of the loop is stored in the variable i, in case you need it; we don't need i for this problem, but a later example will illustrate how this can be useful.

Now, n is an argument, and in this case we assigned it the value 80. So the indented tasks will repeat 80 times. We now illustrate what occurs the first few times:

When $i = 0$: The first line tells Sage to compute $\Delta x\, df\,(a, b)$ and add it to b, then assign the result to b. After this, $b = .1f\,(0,0) + 0 = .2$.
The second line tells Sage to add Δx to a, then assign the result to a. After this, $a = .1 + 0 = .1$.

When $i = 1$: The first line tells Sage to compute $\Delta x\, df\,(a, b)$ and add it to b, then assign the result to b. After this, $b = .1f\,(.1, .2) + .2 \approx .42$.
The second line tells Sage to add Δx to a, then assign the result to a. After this, $a = .1 + .1 = .2$.

When $i = 2$: The first line tells Sage to compute $\Delta x\, df\,(a, b)$ and add it to b, then assign the result to b. After this, $b = .1f\,(.2, .42) + .42 \approx .66$.
The second line tells Sage to add Δx to a, then assign the result to a. After this, $a = .1 + .2 = .3$.

...and so forth. Repeat this 80 times, and you end up with the value that Sage reported.

How does this work? *or*, an introduction to collections

We discuss in more detail how this process works. Definite loops work by passing over a collection; in general, you can use the `for` keyword in the form

```
for variable in collection:
```

and Sage will perform the following, indented tasks as many times as there are objects in *collection*. On the first pass through the loop, *variable* takes on the value of the "first" element of *collection*; on each subsequent pass, *variable* takes on the value of the element in *collection* that "follows" the current value of *variable*.

We put "first" and "follows" in quotes because in some collections, what Sage considers the "first" element may not be what you expect, and the value it considers to "follow" the current value may not be what you consider to follow it. This is not such a problem as you may think, since it happens only in collections where you should not be expecting a "first," "second," etc. We'll talk about that in a second.

But what is a collection? As you'd expect from its name, a **collection** is an object that "contains" other objects. We can classify collections in two ways.

- The first classification is whether a collection is indexed.
  - **Indexed** collections order their elements that allow you to access any element according to its location.
  - **Unindexed** collections do not order their elements, so you can access *any* element, but not by its location.
- The second classification is whether a collection is mutable.
  - **Mutable** collections allow you to change their values.
  - **Immutable** collections do not allow you to change their values.

We use five kinds of collections.

A **tuple** is an indexed, immutable collection; we refer to a tuple of n elements as an n-tuple. You create a tuple using parentheses or the `tuple()` command, inserting another collection between its parentheses; for instance, the following two commands do the same thing:

```
sage: a_tuple = (3, pi, -I)
sage: b_tuple = tuple([3, pi, -I])
sage: a_tuple == b_tuple
True
```

You can also create an "empty tuple" by placing nothing between the parentheses, either `()` or `tuple()`. As immutable collections, tuples are useful for communicating data that should not be changed; i.e., constants. You have already seen and used tuples; we used them extensively to provide points to the plotting commands.

A **list** is an indexed, mutable collection. You create a list using brackets or the `list()` command, inserting another collection between the brackets or parentheses; for instance,

```
sage: a_list = [3, pi, -I]
```

You can also create an "empty list" by placing nothing between the parentheses or brackets, either `[]` or `list()`. As mutable collections, it is easy to modify both a list and its elements. If you need to store values that might change, you need a list, not a tuple, as you cannot modify a tuple.

A **set** is an unindexed, mutable collection. You create a set using braces or the `set()` command; for instance,

```
sage: a_set = {3, pi, -I}
sage: a_set
{-I, 3, pi}
```

Notice how the elements' "order" changed. You can also create an "empty set" by placing nothing between the parentheses or brackets, either `{}` or `set()`. As they are mutable, you can modify a set. An important property of a set is that it stores only one copy of any element; trying to add additional copies leaves us with only one nevertheless. For example:

```
sage: another_set = {2, 2, 2, 2, 2, 2}
sage: another_set
{2}
```

We do not use sets often in this text; while they have important uses, they can be tricky to use, as they accept only immutable objects, and many Sage objects are mutable. For instance, you can store tuples in a set, but not lists. For that matter, sets themselves are mutable, so you cannot store one set inside another.

```
sage: { a_tuple }
{(3, pi, -I)}
sage: { a_set }
TypeError:  unhashable type:  'set'
```

A **frozen set** is an unindexed, immutable collection. You create a frozen set using the `frozenset()` command; for instance,

```
sage: f_set = frozenset(a_set)
sage: f_set
frozenset({-I, 3, pi})
```

You can also create an "empty frozen set" by placing nothing between the parentheses, `frozenset()`. Frozen sets are especially useful when you need sets of sets: you cannot store a mutable set inside a set, so you store a frozen set inside a set.

A **dictionary** is like a list, in that it is an indexed, mutable collection. It is *unlike* a list in that the indexing is by entry rather than by position. You create a dictionary using braces *and* colons; the braces delimit the dictionary, while the colons indicate a correspondence between **keys** (dictionary entries) and **values** (definitions for the entries). For example:

```
sage: a_dict = {x:2, x^2 + 1:'hello'}
```

In this (rather silly) dictionary, the entry `x` corresponds to the value `2`, while the entry `x^2 + 1` corresponds to the value `'hello'`. Another way to create a dictionary is by using the `dict()` command with a collection of 2-tuples; the first entry in the tuple becomes the key, the second becomes the value.

```
sage: tup_dict = dict(( (x,1), (15,-71), (cos(x),3) ))
sage: tup_dict[cos(x)]
3
```

You can also create an "empty dictionary" by placing nothing between the parentheses of a `dict()` command, `dict()`. We have already used dictionaries when performing dictionary substitution.

How does indexing work? The answer to this questions depends somewhat on the type of collection, but it always involves the brackets, `[]`.

For a dictionary, you type the key between the brackets, and Sage returns the value assigned to that key:

```
sage: a_dict[x^2 + 1]
'hello'
```

We can now explain how dictionary substitution works in an expression: when you type

```
sage: f = x^2 + 2
sage: f({x:1})
```

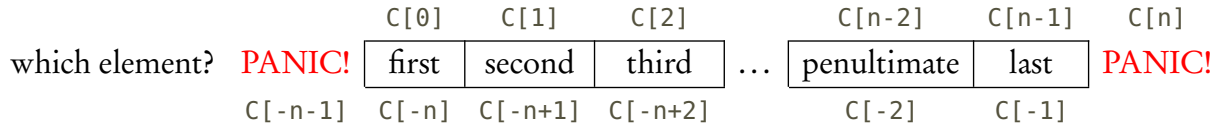
then Sage uses the dictionary `{x:1}` to interpret every `x` in `f` as a `1` instead.

For tuples and lists, indexing has a meaning analogous to that of a subscript in mathematics. Just as a_1, a_2, …, a_i, … indicate the first, second, …, ith, … elements of a sequence, `LorT[`i`]` indicates the element in position i. The slight difference in wording is important; for Sage you need to supply a "legal position," which is not quite the same as you might expect:

```
sage: a_tuple
(3, pi, -I)
sage: a_tuple[1]
pi
sage: a_tuple[2]
-I
```

If you look closely at the numbering, you'll notice that Sage starts numbering its elements at position 0, *not* position 1. To read the first element of `a_tuple`, you would actually type `a_tuple[0]`.

Suppose the collection `C` is either a tuple or a list, and has n elements. The meaning of "legal position" corresponds to the following table:



As usual, PANIC! stands in for, "some sort of error occurs!" We discuss these in the next section, but notice something surprising: negative indices have meaning! They take you backwards through the elements of C. We will not make use of this feature in this text, but there are occasions where it can be put to good use.

Things you can do with collections. We don't address here the applications of collections, so much as the commands use can use on them, and the methods you can send them. We've already seen how you can access elements of indexed collections via the bracket operator.

All five collections. Two commands and one operation are common to all five collections:

`len(`*collection*`)`	the number of elements in *collection*
element `in` *collection*	`True` if and only if *element* is in collection (if *collection* is a dictionary, this means that *element* appears as a key)
`max(`*collection*`)`	the largest element in *collection*
`min(`*collection*`)`	the smallest element in *collection*
`sorted(`*collection*`,` `reverse=`*`True`* or *`False`*`)`	returns a sorted copy of *collection* (reverse order if `reverse=`*`True`*)
`sorted(`*collection*`,` *key*`,` `reverse=`*`True`* or *`False`*`)`	returns copy of *collection*, sorted according to *key* (reverse order if `reverse=`*`True`*)

Most of the time, a program doesn't know beforehand how many elements it has to work with, so having a way to determine that number is enormously useful.

```
sage: len(a_set)
3
sage: len(another_set)
1
sage: sqrt(2) in a_list
False
sage: -sqrt(-1) in a_tuple
True
sage: sorted(a_tuple)
[3, pi, -I]
```

Notice how the `sorted()` command always returns a list, even though we supplied a tuple for its input.

There are times when you may want to sort a collection in a manner different from Sage's default. You can modify the sorting criteria using the `key` option, a function that returns an object that serves as a key for sorting, much as a dictionary. We won't use this often, but the example above illustrates the point: from the point of view of complex analysis, it seems odd to sort $-i$ after 3 and π, when its "norm" is smaller. To sort by the norm, we can write a function that computes the norm of any complex number, and use that as the key:

```
sage: def by_norm(z):
          return real_part(z)**2 + imag_part(z)**2
sage: sorted(a_tuple, key=by_norm)
[-I, 3, pi]
```

Lists and tuples. The next methods and operations apply only to lists and tuples:

LorT`.count(`*element*`)`	the number of time *element* appears in *LorT*
LorT`.index(`*element*`)`	the location of *element* in *LorT*
LorT1`+` *LorT2*	creates a new list/tuple whose elements are those of *LorT1*, followed by those of *LorT2*

It should make sense that you cannot apply these techniques to sets, frozen sets, or dictionaries. No element or key can appear more than once in a set, frozen set, or dictionaries, so `count()` would return at most 1 in each. Sets and frozen sets are unindexable, so `index()` doesn't make sense. For dictionaries, `index()` might make sense, but it isn't directly implemented.[54]

We can only "add" two lists or two tuples; you cannot add a list to a tuple, or vice-versa.

We have exhausted the commands available for tuples, but there's a bit more you can do with a list. Since lists are both indexable and mutable, we can modify a particular element of a list using item assignment:

```
sage: a_list[0] = 1
sage: a_list
[1, pi, -I]
sage: a_list[0] = 3
sage: a_list
[3, pi, -I]
```

As usual, we use `0` because Sage considers the first item to have location `0`.

Besides item assignment, lists feature some methods not available to tuples; see Table 9.[55] A few distinctions are worth making about these commands:

- `.pop()` and `.remove()` differ in that one refers to a *location*, while the other refers to a particular *element*.

```
sage: a_list.pop(1)
pi
sage: a_list.remove(3)
sage: a_list
[-I]
```

There was never an element in location 3, emphasizing that `remove()` looked for an element whose value was 3. This can be fairly sophisticated, as Sage will perform obvious reductions to check whether an element has a given value:

[54] It is actually doable, but somewhat convoluted.

[55] As usual, the list may not be exhaustive, and probably isn't. We're addressing only the commands we think you'll need most often, and the ones available at the time of this writing. To see if the list is exhaustive, remember that you can obtain a list by typing `a_list.` , then pressing the `Tab` key.

L.append(*element*)	add *element* to the end of *L*
L.extend(*collection*)	append the elements of *collection* to *L*
L.insert(*location*, *element*)	add *element* at the given *location* (starting from 0)
L.pop()	removes (and returns) the last element of the list
L.pop(*location*)	removes and returns the element at the indicated location (starting from 0)
L.remove(*element*)	removes the named element
L.reverse()	reverses the list
L.sort()	sorts the list according to Sage's default mechanism
L.sort(*key*)	sorts the list according to the given *key*
L.sort(*key*, reverse=*True* or *False*)	sorts the list in the order *opposite* that given by *key*

TABLE 9. Operations unique to lists

```
sage: a_list = [3, pi, -I, (x+1)*(x-1)]
sage: a_list.remove(x**2 - 1)
sage: a_list
[3, pi, -I]
```

- .sort() differs from sorted() in that it does not copy the list first, and returns *None*. The list is sorted "in-place." You can think of sorted() as leaving the original list intact, and .sort() as changing the list.

Sets and frozen sets. The methods in Table 10 apply only to sets and frozen sets. Operations that modify the set do not apply to frozen sets.

Several of the methods correspond to mathematical operations or relations. An important distinction to make is that methods which end with _update modify the set itself and return *None*, while the corresponding, _update-less methods return a new set and leave the original untouched.

Dictionaries. We will not use most of the features available to a dictionary. The only ones we mention appear in Table 11. We have already shown how to access a dictionary's elements using the bracket operator [], so we merely note that you can add or modify entries in a dictionary the same way that you modify entries in a list.

S.`add(`*element*`)`	add *element* to the set *S*
S.`difference(`*C*`)`	returns a copy of the set or frozen set *S*, minus any elements in the collection *C*
S.`difference_update(`*C*`)`	removes elements in the collection *C* from the set *S* itself
S.`intersection(`*C*`)`	returns the set of elements in both the set or frozen set *S* and the collection *C*
S.`intersection_update(`*C*`)`	removes from the set *S* any elements in *C*
S.`isdisjoint(`*C*`)`	`True` if and only if the set or frozen set *S* has no elements in common with the collection *C*
S.`issubset(`*C*`)`	`True` if and only if all the elements in the set or frozen set *S* are also in the collection *C*
S.`issuperset(`*C*`)`	`True` if and only if all the elements in the collection *C* are also in the set or frozen set *S*
S.`pop()`	removes and returns *an* element of the set *S*
S.`remove(`*element*`)`	removes *element* if it appears in *S*; raises a `KeyError` if it does not
S.`symmetric_difference(`*C*`)`	returns the symmetric difference between *S* and *T*; that is, the elements *not* common to *S* and *T*
S.`symmetric_difference_update(`*C*`)`	remove from *S* any element that appear in *C*, and adds to *S* those elements in *T* that are not in *S*
S.`union(`*C*`)`	returns the union of the set or frozen set *S* with the collection *C*
S.`update(`*C*`)`	adds all the elements of the collection *C* to the set *S*

TABLE 10. Operations unique to sets

```
sage: a_dict
{x^2 + 1:  'hello', x:  2}
sage: a_dict[0] = 'goodbye'
sage: a_dict[0] = -3
sage: a_dict
{0:  -3, x^2 + 1:  'hello', x:  2}
```

D`.clear()`	remove all the entries in D
D`.has_key(`*key*`)`	returns `True` if and only if *key* has a value in D
D`.pop(`*key*`)`	removes the entry for *key* from D and returns its value
D`.popitem()`	remove *some* entry of D and return the tuple `(`*key*`, `*value*`)`
D`.update(`E`)`	adds the definitions in E to D

TABLE 11. Operations unique to dictionaries

A shortcut for creating collections. The two functions `range()` and `xrange()` assist in the creation of collections.

The first, `range()`, creates a list of numbers. There are three ways to use it:

`range(`n`)`	creates the list of numbers $[0,1,\ldots,n-1]$
`range(`a`, `b`)`	creates the list of numbers $[a,a+1,\ldots,b-1]$
`range(`a`, `b`, `d`)`	creates the list of numbers $[a,a+d,\ldots,a+kd]$ where $a+(k+1)d \geq b$

This gives us an easy way of creating and manipulating a set of integers in a sequence, a useful construction for many situations. Keep in mind that this function returns an actual list, which you can manipulate just like any other list.

```
sage: L = range(1, 10, 2)
sage: L
[1, 3, 5, 7, 9]
sage: L.reverse()
sage: L
[9, 7, 5, 3, 1]
```

In many cases, you don't need the actual list of numbers; you just need a way to step through them all. It would actually be wasteful to create the list, as that takes up computer memory and time. If you want to run through a list of numbers without actually creating the list, Sage offers a different command for this: `xrange()`. You invoke it the same way you invoke `range()`, and you can even assign it to a variable and access its elements directly. However, you cannot modify the variable, so in some sense, you can view the result of `xrange()` as being a tuple.

```
sage: L = xrange(1, 10, 2)
sage: L
xrange(1, 11, 2)
sage: L[2]
5
sage: len(L)
5
sage: L.reverse()
AttributeError:  'xrange' object has no attribute 'reverse'
sage: L[2] = 1
TypeError:  'xrange' object does not support item assignment
```

If it remains unclear why these two functions are useful, fret not. One will prove its worth clear in our next major example (Riemann sums), and the other will prove itself useful eventually.

Errors in creating or accessing collections. It would be wise to review what sort of errors crop up when you try to create one or access an element.

- When creating a collection using a command rather than symbols, don't neglect to supply another collection. Sage will not accept a mere, comma-separated sequence of elements.

```
sage: new_tuple = tuple(2, 3, 4)
TypeError:  tuple() takes at most 1 argument (3 given)
```

- Tuples are immutable. Don't try to change values. If you think you might need to change the value of an element, use a list instead.

```
sage: a_tuple[0] = 1
TypeError:  'tuple' object does not support item assignment
```

- While lists and tuples accept negative indices, they don't "wrap around" any more than that. If the collection has length n, don't try to access elements n or $-n-1$.

```
sage: len(a_list)
3
sage: a_list[3]
IndexError:  list index out of range
```

- As mentioned already, don't try to add a list to a tuple, or vice-versa.

```
sage: a_list + a_tuple
TypeError:  can only concatenate list (not "tuple") to list
```

- Set aren't indexed, so you can't access a particular element of a set.

```
sage: len(a_set)
3
sage: a_set[1]
TypeError:  'set' object does not support indexing
```

- Sage raises an error whenever you try to access a non-existent entry of a dictionary.

```
sage: a_dict[12]
KeyError:  12
```

Repeating *over* a collection

We can now return to the main purpose of this chapter, Sage's `for` keyword. We remind you that its general form is

`for` *variable* `in` *collection*`:`

followed by an indented list of tasks; this corresponds to the pseudocode

for *variable* $\in$ collection

followed by an indented list of tasks. On the first pass through the loop, *variable* takes on the value of the "first" element of *collection*; on each subsequent pass, *variable* takes on the value of the element in *collection* that "follows" the current value of *variable*. We call *variable* the **loop variable** and *collection* the **loop domain**, or just "domain".

The effect is that if *element* is `in` the domain C, the `for` loop will at some point assign its value to *variable*, then perform all the indented tasks. This property of a complete treatment of C does not change *even if you modify variable* during the loop; Sage remembers which *element* it selected from C, and selects the element that "follows" it, rather than the value of *variable*. So you can modify the loop variable if needed, without worrying about the behavior of the loop.[56]

On the other hand! if you modify the domain, nasty, *nasty* things can happen. The `for` loop wants to iterate over every element of its domain, but it doesn't make a copy of the domain at the outset, so if the body of the loop modifies the domain, strange things can happen, including an infinite loop.[57] One of the Programming exercises will illustrate this — you should *not* actually try it in Sage unless you're ready to press the Stop button (cloud), select the Action menu, then Interrupt (other server), or hold `Ctrl` and press `C` (command line).

There are many cases where you will want to use the value of the loop variable.

First example: computing a Riemann integral. Recall that we cannot always simplify an indefinite integral to "elementary form," so when we need a definite integral we can approximate its value using one of several methods. The integral is defined as

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\Delta x, \quad \text{where} \quad \Delta x = \frac{b-a}{n} \quad \text{and} \quad x_i^* \text{ lies in the } i\text{th subinterval.}$$

[56] This is very different from many computer languages, such as C, C++, and Java. In those languages, modifying the loop variable during a `for` loop can have catastrophic results.

[57] If you need a definition of "infinite loop," see the index, where we stole a joke from the glossary of the AmigaDOS 3.1 manual.

In "plain English," the integral's value is the limit of the "Riemann sums." These sums approximate the integral using n rectangles of width Δx and height $f\left(x_i^*\right)$. There are three customary ways to select x_i^*:

- by left endpoint, where $x_i^* = a + (i-1)\Delta x$;
- by right endpoint, where $x_i^* = a + i\Delta x$; and
- by midpoint, where $x_i^* = a + (i - 1/2)\Delta x$.

As n increases, the error decreases, so it is possible to approximate the integral by evaluating

$$\sum_{i=1}^{n} f\left(x_i^*\right)\Delta x$$

for a large value of n. The summation symbol Σ instructs us to let the summation variable i grow from 1 to n, and for each value to evaluate the expression on its right, and add that to the growing sum.

Notice the language we are using here: "each" and "every." When a problem's solution involves these sorts of words, that's a telltale sign that you need a definite loop over the collection whose elements are in question.

Pseudocode. In this case, we will create a loop for left-endpoint approximation. It is relatively easy to turn this into pseudocode:

algorithm *Left_Riemann_approximation*
inputs
- $a, b \in \mathbb{R}$
- f, an integrable function on $[a, b]$
- n, the number of rectangles to use in approximating $\int_a^b f(x)\,dx$

outputs
- S, the left-endpoint Riemann sum of $\int_a^b f(x)\,dx$, using n rectangles to approximate the area

do
  let $\Delta x = (b-a)/n$
  let $S = 0$
  for $i \in (1, \ldots, n)$
    let $x_i^* = a + (i-1)\Delta x$
    add $f\left(x_i^*\right)\Delta x$ to S
  return S

Let's look at what this code does. Initially it assigns Δx, which we very much need to know, as it is not part of the input. It then initializes the result, S, to 0, a good idea whenever you want to create a sum. With this complete, it passes into the loop, assigning to i each value from 1 to n. With that value, it performs the two indented tasks underneath: choose a value of x_i^* according to the formula for left endpoints, and add the area of a rectangle to S. Once the loop has passed through every value $1, \ldots, n$, it returns S.

Notice how this code uses the value of the loop variable i to construct x_i^*.

Sage implementation. Implementing this pseudocode in Sage requires a few minor changes. The main problems is that we can't use Greek symbols, subscripts, or decorators, so the names of variables have to change somewhat.

However, we have to make another change that can be easy to miss. The formula we are using expects i to assume the values $1, \ldots, n$. The natural way to have Sage pass over such numbers is with the `xrange()` command, but by default `xrange(n)` starts with `0` and ends with $n-1$! There are several intelligent ways you can make sure that you have the right numbers; we opt for letting j stand for the loop variable and setting $i = j + 1$. That lets us keep the formula the same.[58]

```
sage: def Left_Riemann_approximation(f, a, b, n, x=x):
          f(x) = f
          Delta_x = (b - a) / n
          S = 0
          for j in xrange(n):
              i = j + 1
              # interval's left endpoint
              xi = a + (i - 1) * Delta_x
              # add area of rectangle over interval
              S = S + f(xi) * Delta_x
          return S
```

This works quite well:

```
sage: Left_Riemann_approximation(t^2 + 1, 0, 1, 100, t)
```

Second example: checking whether $\mathbb{Z}_n$ is a field. For another example, recall the exercise on p. 50, where we have to check whether $\mathbb{Z}_n$ is a field. We already know it is a ring, so we needed merely check that every nonzero element has a multiplicative inverse. Checking all the elements by hand is rather burdensome, especially as n grows large. A loop would thus be a desirable tool here. All we have to do is check whether each element has an inverse, and we can do that by checking the product of each element with every other element.

Notice again that we are using the words "each" and "every," which tells us we need to use a definite loop.

Pseudocode. We describe a fairly straightforward implementation of our solution in pseudocode:[59]

[58] Another way is to use the construction `xrange(1,n+1)`, which is really equivalent to our approach, and makes for a somewhat simpler program, but we pass over that for the sake of a simpler discussion.

[59] This is not a great solution. We will improve on it when we discuss decision-making.

algorithm *produce_all_products*
inputs
- $n \in \mathbb{N}$ with $n \geq 2$

outputs
- a list L such that L_i is the set of products of $i \in \mathbb{Z}_n$ with all other elements of $\mathbb{Z}_n$

do
```
Let L = [ ]
for i ∈ Z_n
    Let M = ∅
    for j ∈ Z_n
        add ij to M
    append M to L
return L
```

Let's look at what this code does. It creates a list L, initially empty. (Our pseudocode uses brackets to denote a list, and brackets with nothing between them to denote an empty list. Some authors use parentheses instead, but this is not a hard-and-fast rule, and we don't want to risk confusion with tuples.) The code then loops through all the elements of $\mathbb{Z}_n$, calling the element in each pass i. For each of these elements, it creates a new set M, initially empty. It now loops through $\mathbb{Z}_n$ again, calling the element in each pass j. It's important to notice that i *remains fixed* while this inner loop changes j on each pass; because of this, we can say with confidence that M contains the product of this fixed i with every other element of $\mathbb{Z}_n$ once the inner loop concludes. The code then appends this set to L, concluding the tasks in the outer loop. Once the outer loop has passed through every element of $\mathbb{Z}_n$, the code can return L.

Notice that the code uses the values of the loop variables i and j, which are themselves entries of $\mathbb{Z}_n$.

By keeping L as a list, we guarantee that its elements correspond to the order in which i passes through the elements of $\mathbb{Z}_n$. This is not so important for M; all we want is the products of i and every other element, not necessarily their order, though that could be useful in some contexts.

This pseudocode features an important property of loops: **nesting**. This occurs whenever we include one control structure inside another. This is often useful, but it can also be quite confusing, so you should avoid doing it too much. If your nesting grows to more than 3 or 4, it's a good idea to separate the inner loops into another function. This helps make code easier to understand, and since tasks are often usable in more than one place, it can also save you time down the road, as well.

Sage implementation. It is relatively easy to turn this into Sage code. The main difference with the pseudocode is that we have to create a ring for $\mathbb{Z}_n$ and make sure Sage views i and j as elements of that ring. We first show the "obvious" way to do this, then a smarter way.

```
sage: def produce_all_products(n):
          R = ZZ.quo(n)
          # eventual result:  L[i] is products of i
          L = list()
          for i in xrange(n):
              M = set() # set of multiples of i
              for j in xrange(n):
                  M.add(R(i)*R(j))
              L.append(M)
          return L
```

To see how this works, try it with a few values of n:

```
sage: produce_all_products(4)
[{0}, {0, 1, 2, 3}, {0, 2}, {0, 1, 2, 3}]
sage: produce_all_products(5)
[{0}, {0, 1, 2, 3, 4}, {0, 1, 2, 3, 4}, {0, 1, 2, 3, 4}, {0, 1, 2,
3, 4}]
```

We see that:

- For $\mathbb{Z}_4$, 1 does not appear in the products of `0` and 2. We don't mind `0`, since we care only about nonzero elements, but 2 is fatal: $\mathbb{Z}_4$ is not a field.
- For $\mathbb{Z}_5$, 1 appears in all the products except `0`, so it is in fact a field.

Unfortunately, this code is not a great solution, because as n grows larger, the lists of products get long, and fast, making it difficult to check whether 1 is an element of some set. This may prompt us to ask: *Why are we checking this?* It's easy to have Sage check whether an element appears in a collection. Let's change our pseudocode from this:

> append M to L

to this:[60]

> append `True` to L if $1 \in M$; `False` otherwise

For the Sage code, we use the fact that the expression `1 in M` simplifies automatically (the only change appears in red):

[60] This is still not a great solution. We will improve on it when we discuss decision-making.

```
sage: def produce_all_products(n):
          R = ZZ.quo(n)
          # eventual result:  L[i] is products of i
          L = list()
          for i in xrange(n):
              M = set() # set of multiples of i
              for j in xrange(n):
                  M.add(R(i)*R(j))
              L.append(1 in M)
          return L
```

Look at how this behaves much more conveniently than before:

```
sage: produce_all_products(4)
[False, True, False, True]
sage: produce_all_products(5)
[False, True, True, True, True]
sage: produce_all_products(10)
[False, True, False, True, False, False, False, True, False, True]
```

In this case it's extremely easy to determine whether $\mathbb{Z}_n$ is a field: see if `False` appears anywhere besides the first position (which corresponds to `0`, for which no inverse is needed).

You might wonder if we can't simplify this even further. Indeed we can. One way would require quite a bit of Boolean algebra, so we postpone it for later. But another way is to observe that Sage considers `ZZ.quo(`*n*`)` to be a collection and we can run over its elements, too: (changes appear in red)

```
sage: def produce_all_products(n):
          R = ZZ.quo(n)
          # eventual result:  L[i] is products of i
          L = list()
          for i in R:
              M = set() # set of multiples of i
              for j in R:
                  M.add(i*j)
              L.append(1 in M)
          return L
```

If you test this, you will see how it works just as before, though the code is simpler. Keep in mind how easy Sage often makes it to work with mathematical objects in a natural way.

Comprehensions: repeating *in* a collection

Sage offers a special way to create collections that abbreviates the **for** loop structure and makes it a bit easier to use and read. These are called **comprehensions**, and imitates the set-builder

notation of mathematics. In set-builder notation, we define a set by specifying first a domain, then a criterion for selecting elements from that domain. For instance, the expression

$$S = \left\{ n \in \mathbb{N} : 2 \leq n \leq 2^{10} \right\}$$

uses set-builder notation to define S as the set of all natural numbers that lie between 2 and 2^{10}, inclusive. Comprehensions give Sage a natural way to mimic this in places where it would be useful and feasible.

To define a comprehension, use the following template:

collection`(`*expression* for *variable* `in` *collection_or_range*`)` .

This is effectively equivalent to one of the the following sequence of commands (`D` is a collection; a, b, and n are integers):

```
sage: C = collection()
sage: for d in D:
          C = C.append/insert/update(expression)
```

(Here, you can use *"collection"* for any list or set, choosing append, insert, or update appropriately.) Or:

```
sage: C = collection()
sage: for d in xrange(a, b, n):
          C = C.append/insert/update(expression)
```

In each case, the result is that C contains the values taken on by *expression* for each value of either D (first case) or the specified range of integers (second case).

To see this in practice, we return to our example of the Riemann sums. One way we could use a comprehension is in generating the x-values. Left endpoints have the form

$$a + (i-1)\Delta x \quad \text{where} \quad i = 1, \ldots, n \,,$$

and we can assign this to a list `X` of x-values using a list comprehension as

`X = [a + (i - 1)*Delta_x for i in xrange(1, n+1)]` .

(Remember that we need `n+1` because the `range()` and `xrange()` commands proceed up to, *but not including*, the second number mentioned.) We could then loop over `X`, like so:

```
sage: def Left_Riemann_approximation(f, a, b, n, x=x):
          f(x) = f
          Delta_x = (b - a) / n
          # create all left endpoints in one pass
          X = [a + (i - 1)*Delta_x for i in xrange(1, n+1)]
          S = 0
          for xi in X:
              # add area w/height at f(xi)
              S = S + f(xi) * Delta_x
          return S
```

This is somewhat simpler than before, and it has the advantage of defining the x-values in a way that looks mathematical.

We can also describe a simpler way that gets around the penalty of creating the list `X`. Sage has a `sum()` command that is comprehension-friendly. Rather than initialize a sum variable S to zero, and add partial sums to it on each pass through the loop, we could use the `sum()` command with a list comprehension to simplify the program and make it look more like the mathematical idea. Since we are using left-hand sums, we can rewrite

$$\sum_{i=1}^{n} f(x_i^*)\Delta x$$

as

$$\sum_{i=1}^{n} f(a+i\Delta x)\Delta x$$

which becomes

```
sum(f(a + (i*Delta_x))*Delta_x for i in xrange(1, n+1)).
```

All we've done here is "translate" the mathematical idea into a corresponding Sage command. This can seem harder to read at first, but once you get used to it it's quite natural. The resulting Sage code is

```
sage: def Left_Riemann_approximation(f, a, b, n, x=x):
          f(x) = f
          Delta_x = (b - a) / n
          return sum(f(a + (i*Delta_x))*Delta_x \
                     for i in xrange(1, n+1))
```

That's a *lot* shorter than what we had before.

Animation again

Recall from p. 69 that we created a short animation of a modified *fleur-de-lis* by creating several frames, then joining them with the `animate()` command. Most interesting or instructive animations require quite a number of frames; creating them by hand can be a tiresome task, if not an unfeasible one. On the other hand, most animations also depend crucially on patterns; for instance, the pattern in our animation on p. 69 can be boiled down to

$$\cos(nx)\sin((n+1)x),$$

where n ranges from 2 to 7, inclusive. Of course, we might want more images than that. List comprehensions allow us to create a list of frames, such as the following:

```
sage: frames = [polar_plot(cos(n*x)*sin((n+1)*x), (x, 0, pi),
      fill=True, \
                     thickness=2, fillcolor='yellow', \
                     color='goldenrod', axes=False) \
                 for n in xrange(2,20)]
sage: fdl_anim = animate(frames, xmin=-1, xmax=1, \
                     ymin=-0.5, ymax=1.5, aspect_ratio=1)
sage: show(fdl_anim, gif=True, delay=8)
```

The result should be an animation of reasonable speed and smoothness. *If you view this text in Acrobat Reader,* you should see the same animation below, though the speed might differ slightly; if you are looking at a paper copy of the text, you should instead see the animation's individual frames:

Exercises

True/False. If the statement is false, replace it with a true statement.

1. A **for** statement implements a definite loop.
2. The concept we express in pseudocode as "**repeat** *n* times" has no corresponding keyword in most computer languages.
3. You shouldn't change the value of a loop variable in a `for` loop, as that can affect the next pass through the loop.
4. You shouldn't change the entries of a `for` loop's domain, as that can affect the next pass through the loop.
5. Definite loops are not a useful model for iteration.
6. The only immutable structure Sage offers you is a tuple, so if you want immutability, you're stuck with indexing.
7. You can only construct a set using braces; for example, `{3, x^2, -pi}`.
8. You can only create an empty set on a computer whose keyboard has a ∅ symbol on it.
9. The `key` for sorting returns `True` if and only if the first argument is smaller than the second.

10. List comprehensions allow you to create a list without using the conventional **for** loop structure.

Multiple Choice.

1. Indexing corresponds to which mathematical notation?
   A. functions
   B. expressions
   C. subscripts
   D. apostrophes
2. We cannot access an element of a set *at numerical location i* because:
   A. Sage's programmers were lazy.
   B. Sets aren't ordered, and thus aren't indexable.
   C. That might break the `for` loop.
   D. The premise is incorrect; brackets allow us to access an element of a set at numerical location i.
3. We cannot access an element of a dictionary *at numerical location i* because:
   A. Sage's programmers were lazy.
   B. Dictionaries aren't ordered, and thus aren't indexable.
   C. Dictionaries are ordered by key rather than numerical location.
   D. The premise is incorrect; brackets allow us to access an element of a dictionary at numerical location i.
4. We cannot access an element of a tuple *at numerical location i* because:
   A. Sage's programmers were lazy.
   B. Tuples aren't ordered, and thus aren't indexable.
   C. Tuples are supposedly immutable, and accessing a particular element would violate that rule.
   D. The premise is incorrect; brackets allow us to access an element of a tuple at numerical location i.
5. Which of the following mathematical techniques might motivate the use of a `for` loop?
   A. iteration
   B. checking all elements of a set
   C. repeating a set of tasks n times
   D. all of the above
6. Which of the following collections is immutable?
   A. a tuple
   B. a list
   C. a set
   D. an `xrange()`
7. Which of the following collections is not indexable?
   A. a tuple
   B. a list
   C. a frozen set
   D. an `xrange()`
8. Which of the following symbols or identifiers do we use in our pseudocode standard to test for membership in a sequence or set?
   A. $\in$
   B. ε

C. `in`
D. **in**

9. Comprehensions model which mathematical notation?
   A. function
   B. set
   C. set-builder
   D. subscripts
10. Nesting occurs when:
   A. one or more **for** loops are placed inside another
   B. we create a collection using set-builder notation
   C. we create a set to contain a number of elements of the same type
   D. two birds engage in the Rite of Spring[61]

Short answer.

1. Summarize how the `range()` and `xrange()` commands behave when supplied 1, 2, or 3 inputs.
2. The associative property of multiplication holds for a set S whenever any triplet of element $s, t, u \in S$ satisfies $s(tu) = (st)u$. Suppose a particular set S has 9 elements. How many products would an examination of all possible products require? *Hint:* The answer is large enough that you don't want to do it by hand.
3. Consider the following Sage code.

```
sage: for i in xrange(10):
          for j in xrange(i,10):
              for k in xrange(j,10):
                  print i, j, k
```

   (a) What does it print? Don't give all of it, just enough to demonstrate the pattern. Use words to explain the order the lines are printed.
   (b) Find a quadruple inequality[62] involving `i`, `j`, and `k` that holds for every single line.
4. This problem considers the following Sage code.

```
sage: def ec(k):
          var('y')
          p = Graphics()
          for i in xrange(2, k+1):
              p = p + implicit_plot(x**2 + y**2/(1-1/sqrt(i))==1,
                      (x,-2,2), (y,-2,2), color=(0,i/k,.8-i/k))
          return p
```

   (a) Describe what the call `ec(10)` returns.
   (b) Explain what happens if we use larger and larger values of k in the call `ec(`k`)`.
5. This exercise considers the question of adding consecutive integers.

[61] We'd appreciate it if Stravinksy's heirs would not sue.

[62] A quadruple inequality has the form $a \le b \le c \le d \le e$. Aside from `i`, `j`, and `k`, you'll need to use two constants.

(a) What is the formula for adding the first n positive integers? (You should have seen this before, perhaps in Calculus II in the section on Riemann sums.)
(b) Suzy writes the following function to add the first n positive integers.

```
sage: def sum_through(N):
          total = 0
          for i in xrange(N):
              total = total + i
          return total
```

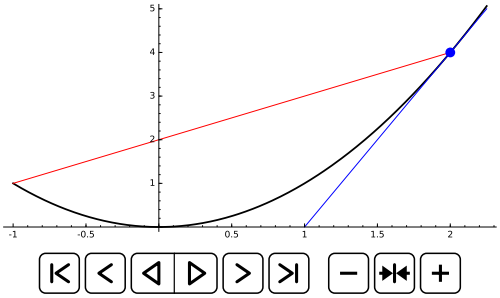
What is the result of her invocation, `sum_through(5)`? Indicate the value of `total` after each pass through the loop.
(c) Use summation notation to describe the value Suzy's program actually calculates. What would the corresponding formula be?
(d) How should Suzy correct her formula?

6. Suppose you have an infant that demands[63] to be fed every 3 hours.
(a) If you start feeding her at 7 am, at what times will feeding occur?
(b) Write a comprehension that generates a list with each of those times for the next 24 hours.
(c) Repeat part (b), only with the assumption that the infant is a little older and only demands to be fed every 5 hours.

Programming.

1. Write a function to compute the mean value of the elements of a collection. (You should remember that "mean value" is a fancy name for "average.") It will be easier with a comprehension inside a `sum()`, but you can also use a `for` loop.
2. Write a function to compute the median value of the elements of a collection. (You should remember that "median value" is a fancy name for "middle value;" that is, half the values are larger, and half are lower.) Probably the best way to do this is to convert the collection to a list, sort it, then return the number in the middle.
3. Write a function that takes a list of points (x_i, y_i) and returns two lists: the list of x-coordinates and the list of y-coordinates.
4. Create an animation that illustrates how a secant line approaches a tangent. Use the function $f(x) = x^2$, with the tangent line through the point $x = 2$, with secant lines that join $x = 2$ with 30 points between $x = -1$ and $x = 2$. Make the plot of f black, with a thickness of 2; color the tangent line blue, and color the secant lines red. Include a blue point at $(2, 4)$ to highlight where the curve and its tangent and secant lines all meet. The result should be comparable to, or better than, the animation you will see below *if you view this text in Acrobat Reader:*

[63] Why, yes, some of us *do* have intimate experience with this. What makes you ask?



5. On p. 104 we talked about a truncated Taylor series with 4 terms. We didn't have **for** loops available, so that approach was both tiresome and inflexible.
   (a) Adapt the pseudocode so that the user can input an arbitrary number of terms.
   (b) Implement the pseudocode as a Sage program.
6. (This problem is for students who have taken multivariable calculus.) The double integral

$$\iint_D f(x,y)\,dx\,dy$$

pops up often in three-dimensional calculus. Here, D is a subset of the x-y plane $\mathbb{R}^2$, called the **domain** of the integral. When D is a rectangular region defined by the x-interval (a,b) and the y-interval (c,d), we can approximate this integral by dividing the domain into $m \times n$ sub-rectangles and evaluating the z-value of f on each sub-rectangle:

$$\iint_D f(x,y)\,dx\,dy \approx \sum_{i=1}^{m}\sum_{j=1}^{n} f\left(x_i^*, y_j^*\right)\Delta x \Delta y$$

where

- m is the number of subintervals we want along (a,b);
- n is the number of subintervals we want along (c,d);
- $\Delta x = \frac{(b-a)}{m}$ is the width of each subinterval of (a,b);
- $\Delta y = \frac{(d-c)}{n}$ is the width of each subinterval of (c,d); and
- $\left(x_i^*, y_j^*\right)$ is a point in the sub-rectangle defined by the ith subinterval of (a,b) and the jth subinterval of (c,d).

We can select $\left(x_i^*, y_j^*\right)$ using a midpoint rule as

$$x_i^* = a + \left(i + \frac{1}{2}\right)\Delta x \quad \text{and} \quad y_j^* = b + \left(j + \frac{1}{2}\right)\Delta y.$$

Write a function `double_integral_midpoint()` that takes as arguments f, a, b, c, d, m, and n, and returns an approximation of the double integral of f over $(a,b) \times (c,d)$ using m subintervals along (a,b) and n intervals along (c,d).
Hint: This will be very similar to the code we used to approximate a single integral, but you will want to nest a **for** loop for y inside a **for** loop for x. You can also do this with a comprehension, but it's a bit more complicated.
7. Let n be a positive integer. The **factorial** of n, written $n!$, is the product

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1.$$

Sage already offers a `.factorial()` function, but in this exercise you'll use loops to do it, two different ways.

(a) Sage's `product()` command works similarly to the `sum()` command: it computes the product of whatever lies between the parentheses, and you can use a comprehension to specify a range, rather than build one explicitly. This corresponds to the mathematical expression

$$\prod_{i=1}^{n} f(i)$$

which computes the product of all the $f(i)$ where $i = 1, 2, \ldots, n$. In this notation,

$$n! = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1.$$

Write a function called `factorial_comprehension()`s that accepts one argument, n, then builds the product using a comprehension.

(b) A more traditional way to compute the factorial is to initialize a product variable P to 1 (the "empty product"), then perform a definite loop from 1 to n, multiplying P by each number along the way. Write a function called `factorial_loop()` that accepts one argument, n, then builds the product using a definite loop.

8. Write a program to compute the **rising factorial** of a number n over k steps. The rising factorial $\mathrm{rf}(n,k)$ is

$$\mathrm{rf}(n,k) = n \times (n+1) \times \cdots \times (n+k-1).$$

While k is always a nonnegative integer, your program should work even if n is not. For instance, $\mathrm{rf}(1/2, 5) = (1/2)(3/2)(5/2)(7/2)(9/2) = 945/32$.

9. Write a program to compute the **falling factorial** of a number n over k steps. The falling factorial $\mathrm{ff}(n,k)$ is

$$\mathrm{ff}(n,k) = n \times (n-1) \times \cdots \times (n-(k-1)).$$

While k is always a nonnegative integer, your program should work even if n is not. For instance, $\mathrm{ff}(9/2, 5) = (9/2)(7/2)(5/2)(3/2)(1/2) = 945/32$.

10. A **permutation** is a way of ordering distinct objects. For example, in the list $(1,2,3)$ there are six permutations: $(1,2,3)$ itself, then $(2,1,3)$, $(3,2,1)$, $(3,1,2)$, $(2,3,1)$, and $(3,2,1)$. Sometimes you want to permute only c of the objects in a set of n. The formula for this is

$${}_nP_c = \frac{n!}{c!}.$$

(See the previous exercise for factorials.) There are two ways to do this.

(a) The obvious, "brute force" way to do this is to compute $n!$ and $c!$, then divide. Write a Sage function named `n_P_c_brute()`, that accepts the two arguments n and c, then calls either `factorial_loop()` or `factorial_comprehension()` to determine $n!$ and $c!$.

(b) A smarter way to do this comes from usin' yer noggin. Expand by hand the factorials in ${}_nP_c$ to see the pattern. Turn the resulting mathematical formula into a Sage function named `n_P_c_smarter()`, that accepts the two arguments n and c, then determines ${}_nP_c$ *without* computing $n!$ or $c!$.

11. Suppose $b > 1$ is an integer. To write a positive number n in base b, we can repeat the following steps:
    - Let $d = \log_b n + 1$; this tells us how many digits the result will have.
    - Let L be an empty list.
    - Repeat d times:

  - Let r be the remainder of n when divided by b.
  - Subtract r from n.
  - Replace n with the value of n when divided by b.
  - Append r to L.
- Reverse L, and return the result.

Convert this casual list of instructions into formal pseudocode, and translate the pseudocode to Sage code.

12. Write a Sage function named `by_degree_then_lc()` that, when given a polynomial f as input, returns a 2-tuple consisting of the polynomial's degree, followed by its leading coefficient. We haven't told you the Sage commands for that, but they work as methods to a polynomial; you can find it in the following way:

```
sage: f = 3*x**2 + x + 1
sage: f.<Tab>
```

…where `<Tab>` indicates that you should press the `Tab` key on the keyboard. Both methods expect the indeterminate as an argument, which is another method to a polynomial that you can find in the same fashion. Make sure you can use the methods successfully before writing the program.

Test this function thoroughly. It should produce the following results:

```
sage: f = 3*x**2 + x + 1
sage: by_degree_then_lc(f)
(2, 3)
sage: g = 2*t**4 - t**2
sage: by_degree_then_lc(g)
(4, 2)
```

Once you have it working, make sure that it sorts the following list of polynomials in `t` into the correct order. Notice that, since we use `t` in this list, your function cannot assume what the indeterminate is, so that should also be an argument to the function, though it can default to `x`.

```
[ t^2 + t + 1, 2*t - 1, 3*t^2 + 4, -3*t^10 + 1]
```

Solving equations

Up to this point, we've had you solve equations by hand, though it makes sense we'd want Sage to solve them for us. This chapter looks at the `solve()` command and some of its relatives, both on single equations and on systems of equations. That leads us into matrices, so we look at some of them, as well.

It is not generally possible to describe the exact solution to every equation in "purely algebraic" terms, by which we mean the use of arithmetic and radicals with rational numbers. This is generally true even when the equation consists merely of polynomials! So even though we focus primarily on methods that find exact solutions, Sage also offers methods to compute approximate solutions to equations, and we take a brief look at them.

The basics

To find the exact solution to an equation or a system of equations, the main tool you want is the `solve()` command.

`solve(`*eq_or_ineq*`, `*indet*`)`	solves the single equation or inequality *eq_or_ineq* for *indet*
`solve(`*eq_or_ineq_list*`, `*indet_list*`)`	solves the collection of equations *eq_or_ineq_list* for the indeterminates listed in the collection *indet_list*

While *eq_or_ineq* should imply an equation or inequality, from Sage's point of view we can use an expression, a function, an equation, or an inequality.

One equation or inequality, for *one* indeterminate

The `solve()` command will of course solve basic high-school algebra problems.

Equations.

```
sage: solve(x**2 - 1 == 0, x)
[x == -1, x == 1]
```

It is very common to encounter equations with `0` on one side; the solution to this kind of equation is called a **root**. We don't actually have to specify that an expression equals `0` when we want to solve for a root; we can simply supply the expression, and the `solve()` command infers that we want its roots.

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

Sage can also solve equations where all the coefficients are symbolic.

```
sage: var('a b c')
(a, b, c)
sage: solve(a*x**2 + b*x + c == 0, x)
[x == -1/2*(b + sqrt(b^2 - 4*a*c))/a, x == -1/2*(b - sqrt(b^2 -
4*a*c))/a]
```

The polynomial $ax^2 + bc + x$ is called *quadratic* because its degree is 2.[64] You can make out the quadratic formula in the two answers:

$$x = -\frac{1}{2} \times \frac{b + \sqrt{b^2 - 4ac}}{a} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

and

$$x = -\frac{1}{2} \times \frac{b - \sqrt{b^2 - 4ac}}{a} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad .$$

This answer matches the one you learned in school,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

In both cases, the result was a list of equations. Each equation indicates a value of the indeterminate that will solve the equation. Because the result is in list form, you can access the solutions using brackets, `[]`. If you want to manipulate the solutions somehow, it's probably best to assign the list to a variable, which we'll usually call by a name like `sols`, then use `sols[i]` to access the *i*th solution.

```
sage: sols = solve(x**5 + x**3 + x, x)
sage: len(sols)
5
sage: sols[0]
x == -sqrt(1/2*I*sqrt(3) - 1/2)
```

So $x = -\sqrt{i\sqrt{3}/2 - 1/2}$ is one of the solutions.

You will sometimes want just the value of the solution. Sage equations offer two useful methods that extract the left- and right-hand sides:

eq_or_ineq`.rhs()`	right-hand side of *eq_or_ineq*
eq_or_ineq`.lhs()`	left-hand side of *eq_or_ineq*

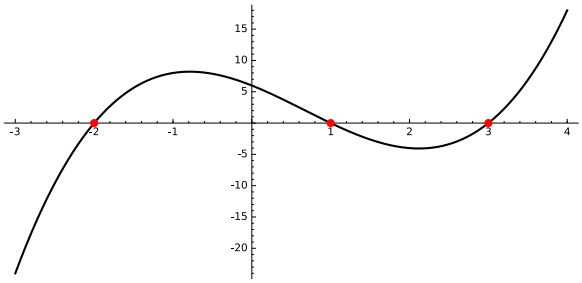
Continuing our previous example,

[64] The degree of a polynomial in a single indeterminate is the largest exponent that appears on that indeterminate. The `.degree()` method will report this to you.

```
sage: sols[0].rhs()
-sqrt(1/2*I*sqrt(3) - 1/2)
```

Again because the result of `solve()` is in list form, you can also iterate over the list of solutions using a `for` loop. Here's just one way you might find this useful. Let's plot both a polynomial and its roots.

```
sage: f(x) = x^3 - 2*x^2 - 5*x + 6
sage: sols = solve(f, x)
sage: X = [sol.rhs() for sol in sols]
sage: p = plot(f, min(X) - 1, max(X) + 1, color='black', \
               thickness=2)
sage: for xi in X:
          p = p + point((xi,0), pointsize=60, color='red', \
                        zorder=5)
sage: p
```



While you should have been able to do this on your own, it involves pulling a lot of different ideas together, so let's break down each step.

- First we defined the function `f`, a polynomial of degree 3.
- We used `solve()` to find its roots, and stored them in the variable `sols`. Were you to peek at `sols`, you would see the following:

```
sage: sols
[x == 3, x == -2, x == 1]
```

- Sage returns solutions in the form of a list of equations, and we want just the value of the root. To do this, we assigned to a new variable, `X`, a list formed using a comprehension. The expression `for sol in sols` loops over the list stored in `sols` and stores each value in the variable `sol`. The expression `sol.rhs()` gives us the right-hand side of that value. The upshot is that `X` is a list that contains the right-hand side of each solution in `sols`. Were you to peek at `X`, you would see the following:

```
sage: X
[3, -2, 1]
```

- We defined a graphics object `p` as the plot of `f` over the interval `min(X) - 1` and `max(X) + 1`. This gives the graph a little room to breathe beyond the roots.
- We performed a definite loop over the solutions stored in `X`. The loop stores entry of `X` in the loop variable `xi`, which Sage uses to add a point to `p`. For example, on the first pass through the loop, it assigns `xi=3`, so it adds the point at $(3, 0)$.

Multiplicities. Sage expressions also offer a method that *sometimes* provides us with both roots and multiplicities.

f`.roots()`	returns the roots of *f*, along with their multiplicities

Multiplicities, you should recall, tell us how many times a root shows up in the factorization of a polynomial. For instance, the polynomial $(x-1)(x+2)^2(x-4)^3$ has three roots, 1, −2, and 4, with respective multiplicities 1, 2, and 3.

```
sage: f = (x - 1)*(x + 2)**2*(x - 4)**3
sage: f.roots()
[(-2, 2), (1, 1), (4, 3)]
```

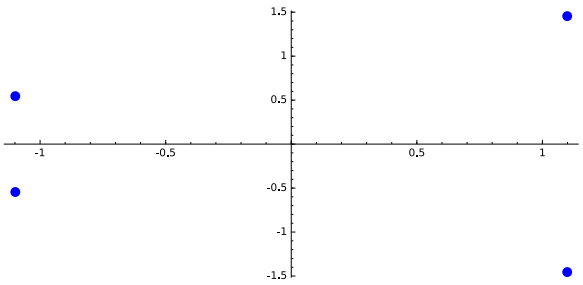
You see that the result is a list of tuples; each tuple lists the root first, and its multiplicity second. Multiplicities are important for many reasons; in one of the exercises for this chapter, you can examine the area around a polynomial's roots and see how multiplicity affects the geometry.

The complex plane. Many polynomials have roots that include imaginary parts. We obviously cannot graph such roots with their functions on the real plane, as there is no place on the real plane to include, for instance, the number i.

Nevertheless, it can be quite instructive to visualize the roots *without* their functions. Any complex number has the form $a + bi$, where a is the real part and b is the imaginary part. We plot this as the point (a, b) on the real plane. This mapping assigns a unique point on the real plane $\mathbb{R}^2$ to every complex number $z \in \mathbb{C}$, which motivates the name of this model, the **complex plane**.

The following function maps a complex number to a point on the real plane.

```
sage: def complex_point(z, *args, **kwds):
          return point((real_part(z), imag_part(z)), *args, **kwds)
sage: sum(complex_point(sol.rhs(), pointsize=90) \
          for sol in solve(x**4 + 4*x + 5, x))
```



What does this code do?

- The first two lines define a function, `complex_point()`, whose only required argument is `z`, a complex number.
  - It creates a point whose x-value is the real part of z and whose y-value is the imaginary part of z.
  - To avoid restating the required and optional arguments of a point, `complex_point()` uses a special trick to accept the required and optional arguments for a regular `point()`, then ignores them except to pass them on to `point()`.
- The last lines create a graphic using the `sum()` command with a comprehension.
  - The comprehension loops over the solutions over x^4+4x+5, which it finds with the command `solve(x**4 +4*x + 5, x)`. Each solution is stored in the loop variable `sol`.
  - The loop sends each value of `sol` the method `.rhs()`, obtaining in return the right-hand side of the equation Sage uses to describe the solution. The loop passes that complex number to `complex_point()`, along with the optional `point()` argument `pointsize=90`. The result is a point in the complex plane, which the `sum()` command finally combines into the image you see.

Solving inequalities. You may recall that inequalities bring complications. To start with, there are usually infinitely many solutions which typically lie on one or more intervals of the real line. These intervals are sometimes bounded, sometimes unbounded. For instance,

- the solution to $x^2-1\geq 0$ is $(-\infty,-1]\cup[1,\infty)$, while
- the solution to $x^2-1<0$ is $(-1,1)$.

This complication is reflected in how Sage describes the solutions of an inequality. The solutions to an inequality are described in *a list of lists*. Each inner list describes an interval that contains solutions. This interval itself contains either one linear inequality, which represents an interval unbounded in one direction, or two linear inequalities, which represent an interval bounded in both directions.

For instance, it is not hard to verify by hand that the inequality $(x-3)(x-1)(x+1)(x+3)\geq 0$ has the following solution:

$$(-\infty,-3]\cup[-1,1]\cup[3,\infty),$$

so that $x=-5$, $x=0$, and $x=12$ are solutions. This is easy to diagram on a number line:[65]

Based on the description we gave in the previous paragraph, how should you expect the solution to look? You know it's a list of lists, so there should be several pairs of brackets within one pair of brackets. There are three intervals, so there should be three inner lists. The outermost intervals are unbounded, so Sage will describe it using only one linear inequality; the middle interval is bounded, so Sage will use two linear inequalities to describe it.

```
sage: solve((x-3)*(x-1)*(x+1)*(x+3) >= 0, x)
[[x <= -3], [x >= -1, x <= 1], [x >= 3]]
```

[65] It should not hurt to think about how you might diagram this in Sage. If it does hurt, that probably means your brain is growing, so keep at it all the same.

Sure enough, Sage describes the leftmost interval using $x \leq -3$; the rightmost using $x \geq 3$, and the innermost using two intervals, $x \geq -1$ and $x \leq 1$. That is also how you should read it, incidentally: "$x \geq -1$ and $x \leq 1$." This is equivalent to the inequality $-1 \leq x \leq 1$.

As the results are lists, you can consider each interval by bracket access or iteration. The `.rhs()` and `.lhs()` methods will separate the left- and right-hand sides of an inequality just as they will for an equation.

Mistakes or surprises that arise when solving

Some equations will give a strange result. Consider this generic 5th-degree equation:

```
sage: solve(a*x**5 + b*x**4 + c*x**3 + d*x**2 + e*x + f, x)
[0 == a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f]
```

Sage seems to be returning the same equation you asked it to solve. If this reminds you of the example on p. 41 where Sage "refused" to compute an indefinite integral, congratulations! It is more or less the same phenomenon: Sage cannot find a way to express the solution by an algebraic expression on the coefficients, except by returning the equation itself to you. It is well-known that we cannot solve *generic* polynomial equations of degree 5 or higher in a manner as "simple" as the quadratic formula. This touches on an topic called *solvability by radicals*.

Remember that, in Sage, an equation uses two equals signs. If you forget to use two equals signs when using the `solve()` command, unpleasant things will occur. You Have Been Warned.™

```
sage: solve(2*x + 1 = 3*x - 2, x)
SyntaxError:  keyword can't be an expression
```

Approximate solutions to an equation

We saw a moment ago that Sage cannot describe roots for the generic fifth-degree polynomial in exact terms. This remains true for many specific fifth-degree polynomials, as well.

```
sage: solve(x**5 - 6*x + 4, x)
[0 == x^5 - 6*x + 4]
```

So sometimes we have to settle for approximate solutions. Another time to opt for an approximate solution is when the exact solution is simply too unwieldy to bother dealing with.

```
sage: solve(x**3 + x + 1, x)[0].rhs()
-1/2*(1/18*sqrt(31)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1) +
1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(31)*sqrt(3) - 1/2)^(1/3)
```

Ouch.

To find an approximate solution to an equation, we use something different from the `solve()` command.

`find_root()`	find a solution of *eq* on the interval (a, b)

To use this, we need some idea where the root is located, so that you can specify a and b. Although the term "root" implies that the equation should have 0 on one side, this isn't strictly necessary; we can provide an equation with non-zero expressions on both sides, and Sage will solve it, all the same.

```
sage: find_root(x**5 - 6*x == -4, -5, 5)
-1.7000399860584985
```

Notice that it returns only one root, even when multiple roots exist on the same interval. When we know multiple roots exist, we can modify the interval accordingly.

```
sage: find_root(x**5 - 6*x == -4, 0, 5)
1.3102349335013999
sage: find_root(x**5 - 6*x == -4, 0, 1.3)
0.693378264514721
```

Should we specify an interval where the equation has no roots, Sage raises an error.

```
sage: find_root(x**5 - 6*x == -4, 1.3, 5)
RuntimeError:  f appears to have no zero on the interval
```

Another approach to try when we're not sure what interval to use is a second form of the `.roots()` method, which allows us to specify a ring in which to look for solutions. In particular, we can specify which ring to look for.

f`.roots(ring=`R`)`	finds the roots of f in the ring R, along with their multiplicities

The default ring is $\mathbb{Q}$, the ring of rationals, but we can ask Sage to solve for approximations to roots outside the rationals by specifying the real or complex rings; in particular, `RR` and `CC`.

```
sage: f = x**5 - 6*x + 4
sage: f.roots()
RuntimeError:  no explicit roots found
sage: f.roots(ring=RR)
[(-1.70003998605850, 1), (0.693378264514721, 1), (1.31023493350140,
1)]
sage: f.roots(ring=CC)
[(-1.70003998605850, 1),
 (0.693378264514721, 1),
 (1.31023493350140, 1),
 (-0.151786605978811 - 1.60213970994664*I, 1),
 (-0.151786605978811 + 1.60213970994664*I, 1)]
```

The `.roots()` approach is not always successful, even when `find_root()` is. Unless we're particularly attached to finding multiplicities, `find_root()` is the method of choice.

```
sage: f = sin(x) + x - 1
sage: f.roots()
RuntimeError:  no explicit roots found
sage: f.roots(ring=RR)
TypeError:  Cannot evaluate symbolic expression to a numeric value.
sage: find_root(f, 0, 1)
0.510973429388569
```

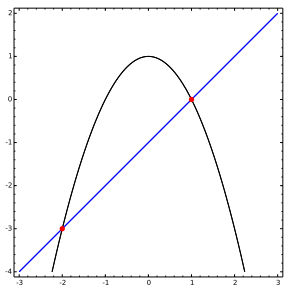
Systems of equations

Many real-world problems involve more than one variable, and more than one relationship between these variables. These give rise to equations in several variables; when we attempt to solve several at a time, we refer to it as a system of equations. We can use `solve()` to solve a system of equations in Sage; just supply the system in a list or tuple, followed by a list or tuple of the polynomials' indeterminates. The solution to a system of equations is similar to that of an inequality: Sage returns a list of lists. Each inner list corresponds to one distinct solution to the equation; it contains equations that indicate the solution to each variable in the system.

```
sage: solve((x**2 + y == 1, x - y == 1), (x,y))
[[y == -3, x == -2], [y == 0, x == 1]]
```

This equation has two solutions, corresponding to the points $(-2,-3)$ and $(1,0)$. We can illustrate the geometric relationship between the curves and these points with a graph:

```
sage: p = implicit_plot(x**2 + y == 1, (x, -3, 3), (y, -4, 2), \
              color='black')
sage: p = p + implicit_plot(x - y == 1, (x, -3, 3), (y, -4, 2), \
              color='blue')
sage: p = p + point((1,0), color='red', pointsize=60, zorder=5)
sage: p = p + point((-2,-3), color='red', pointsize=60, zorder=5)
sage: p
```



Matrices

If the degree of each equation is at most 1, we call them linear equations. So a system of linear equations consists of a set of equations that we try to solve simultaneously. These are just a special case of systems of equations.

Intimately related to systems of linear equations are matrices. Every system of linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

corresponds to a "matrix equation"

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

The matrices **x** and **b** are special matrices in that they have only one column; we call such matrices **vectors**.

We assume you are familiar with the rules of matrix arithmetic, so we do not review them here, but if you are still acquiring your "sea legs" when it comes to matrix arithmetic, we suggest you to think about how the rules of matrix multiplication turn $\mathbf{Ax} = \mathbf{b}$ into the system of equations above. Matrix analysis is essential to understanding systems of linear equations, and since many approaches to non-linear equations involve first transforming them to a system of linear equations, matrices hold a fundamental position in mathematics.

Creating matrices, accessing elements, and modifying fundamental properties. You can create a matrix in Sage using the `matrix()` command. There are several ways to do this; we focus on three.

`matrix(`*row_list*`)`	creates a matrix from *list_of_rows*
`matrix(`*ring*`, `*row_list*`)`	creates a matrix with entries from *ring* using *list_of_rows*

The reason for the two forms is that Sage considers a matrix's ring when deciding how to perform most matrix computations, so it has to know this. You use the first form if you're content to let Sage make an educated guess at the ring. You may find that Sage's choice doesn't work very well for you; in this case, you can correct it gently using the convenient `.change_ring()` method:

M`.change_ring(`*ring*`)`	converts the entries of *M* to reside in *ring* and returns the result
M`.base_ring()`	reports the ring Sage thinks *M*'s elements reside in

Keep in mind that Sage will not change the matrix itself; it produces a *new* matrix and returns it. The original matrix remains in the old ring. We make use of this function a bit further down.

First we illustrate creation of a matrix. To specify the matrix

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

we will list each of its rows as a list inside another list:

```
sage: M = matrix([ \
          [1, 1], \
          [0, 1] \
      ])
sage: M
[1 1]
[0 1]
```

You do not *have* to put each row on its line, but we think that makes it easier to read. You can put the entire matrix on one line if you wish.

It is of course possible to use comprehensions in the definition of a matrix:

```
sage: matrix([[i, i+1, i+2] for i in xrange(3)])
[0 1 2]
[1 2 3]
[2 3 4]
```

Three special matrices you can create are an identity matrix and a diagonal matrix.

`identity_matrix(`*n*`)`	returns the identity matrix of dimension *n*
`diagonal_matrix(`*entries*`)`	returns a matrix with `0` everywhere except the main diagonal, and the values of entries on the main diagonal
`zero_matrix(`*m*`, `*n*`)`	returns an $m \times n$ matrix of zeroes

Again, you can use a list comprehension to create lists of entries for matrices. We demonstrate this for a diagonal matrix:

```
sage: diagonal_matrix([i**2 for i in xrange(3)])
[0 0 0]
[0 1 0]
[0 0 4]
```

You access elements of a matrix using the bracket operator; to access element $M_{i,j}$ use `M[i,j]`. Here, "access" does not mean merely "read;" it also means "write," so you can modify the entries of a matrix in convenient fashion. Keep in mind that, as with lists, the first row starts at position `0`, not position 1, so you'll have to take that into account.

```
sage: M[0,1] = 3
sage: M
[1 3]
[0 1]
```

This ability to modify a matrix's elements means that Sage considers a matrix mutable, just like a list. There are occasions where you will want to work with an immutable matrix. For instance, you can only store immutable objects in a set, so if you want a set of matrices, you have to indicate to Sage that you have no intention of changing the matrices in question. You can do this using the `.set_immutable()` method.

M`.set_immutable()`	makes an object immutable
M`.is_mutable()`	*True* if and only if the object is mutable
M`.is_immutable()`	*True* if and only if the object is immutable

The command works one way only; to make a matrix mutable again, make a copy of it with the `copy()` command.

`copy(`*obj*`)`	returns a copy of the object *obj*

```
sage: { M }
TypeError:  mutable matrices are unhashable
sage: M.set_immutable()
sage: M.is_mutable()
False
sage: { M }
{[1 3]
 [0 1]}
sage: M = copy(M)
sage: M.is_mutable()
True
```

Matrices are not the only objects Sage can copy or make immutable, so these commands have a wider applicability.

Matrix arithmetic and manipulation. You can perform matrix arithmetic in Sage using the customary mathematical symbols.

```
sage: N = matrix([ \
          [1, x], \
          [0, 1]])
sage: M * N
[    1 x + 3]
[    0     1]
```

Sage also offers an enormous number of methods you can send a matrix. We list only a few of them in Table 12. To see them all, remember that you can always discover the methods an object

M`.add_multiple_of_row(`j`, `i`, `c`)`	adds c times each entry of row i to the corresponding entry of row j
M`.characteristic_polynomial()`	returns M's characteristic polynomial
M`.determinant()`	returns M's determinant
M`.dimensions()`	returns a tuple (m,n) where m is the number of rows and n the number of columns
M`.echelon_form()`	returns the echelon form of M while leaving the matrix itself in its original form
M`.echelonize()`	transforms M into echelon form; returns `None`
M`.eigenvalues()`	returns M's eigenvalues
M`.eigenvectors_right()`	returns M's eigenvectors
M`.inverse()`	returns M's inverse
M`.kernel()`	returns the kernel of M (to access the basis vectors of a kernel K use K`.basis()`)
M`.ncols()`	returns the number of columns in M
M`.nrows()`	returns the number of rows in M
M`.nullity()`	returns the kernel's dimension
M`.rank()`	returns M's rank
M`.set_row_to_multiple_of_row(`j`, `i`, `c`)`	sets each element of row j to the product of c and the corresponding entry of row i
M`.swap_rows(`i`, `j`)`	swaps rows i and j of M
M`.submatrix(`i`, `j`, `k`, `ℓ`)`	returns the $k \times \ell$ submatrix of M whose top left corner is in row i, column j of M
M`.transpose()`	returns the transpose of M

TABLE 12. methods understood by a Sage matrix

will understand by typing its identifier, followed by the period, then pressing the `Tab` key. As for the methods we list here, don't worry if you don't understand the purpose of each one, but each of them should at some point prove useful in undergraduate study.

Let's look at how some of these commands might work, as well as highlight some of the errors that may occur. For an example, let's suppose we're interested in transforming a matrix to upper-triangular form, and also in seeing some of the computations that occur along the way. So we'll perform this computation step-by-step rather than looking for a particular Sage command

that does it all at once. Our example matrix will be

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 8 & 3 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix}.$$

We leave it to you to set up a matrix for A in Sage. To transform it to upper-triangular form, we observe that $A_{0,0}$ is already 1, while both $A_{1,0}$ and $A_{3,0}$ are 0, so we need do nothing to them. As $A_{2,0} = 8 \neq 0$, we want to add a multiple of row 1 that eliminates the 8; this is fairly straightforward:

```
sage: A.add_multiple_of_row(2, 0, -8)
[   1   2   3   4]
[   0   2   2   3]
[   0 -13 -23 -30]
[   0   1   2   3]
```

We proceed to the second column. We observe that $A_{1,1} = 2 \neq 1$, so we must divide the row by 2.

```
sage: A.set_row_to_multiple_of_row(1, 1, 1/2)
TypeError:  Multiplying row by Rational Field element
cannot be done over Integer Ring, use change_ring or
with_row_set_to_multiple_of_row instead.
```

PANIC!

…well, no, don't. Sure, we received an error, but this one is quite helpful. It makes it perfectly clear that the problem is that Sage sees A as lying over the ring $\mathbb{Z}$. Integers can't be fractions, and if we multiply the second row by $1/2$, we get $3/2$ in the last spot. We need A to lie over the rational *field* $\mathbb{Q}$, not over the integer *ring* $\mathbb{Z}$. Very well; we've already discussed how to change the ring of a matrix, so let's do that. There's no need to keep the old matrix A, so we'll assign the result of `.change_ring()` to A. Recall that Sage's symbol for the ring of rational numbers is `QQ`.

```
sage: A = A.change_ring(QQ)
sage: A.set_row_to_multiple_of_row(1, 1, 1/2)
sage: A
[   1   2   3   4]
[   0   1   1 3/2]
[   0 -13 -23 -30]
[   0   1   2   3]
```

Excellent! We can now clear out $A_{2,1}$ and $A_{3,1}$.

```
sage: A.add_multiple_of_row(2, 1, 13)
sage: A.add_multiple_of_row(3, 1, -1)
[     1     2     3     4]
[     0     1     1   3/2]
[     0     0   -10 -21/2]
[     0     0     1   3/2]
```

From here you should be able to complete the process on your own, ending up with the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ & 1 & 1 & \frac{1}{2} \\ & & 1 & \frac{21}{20} \\ & & & \frac{9}{20} \end{pmatrix}.$$

(Empty entries represent 0.)

We transform ourselves. One application of matrices involves the use of **transformation matrices** in computer graphics. We discuss three kinds of transformation matrices:

- A scaling matrix has the form

$$\sigma = \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix}.$$

  It has the effect of rescaling a point (x, y) to the point (sx, sy).
- A rotation matrix has the form

$$\rho = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

  It has the effect of rotating a point (x, y) about the origin through an angle α.
- A reflection matrix has the form

$$\varphi = \begin{pmatrix} \cos\beta & \sin\beta \\ \sin\beta & -\cos\beta \end{pmatrix}.$$

  It has the effect of reflecting a point (x, y) across the line through the origin whose slope is $^{1-\cos\beta}/_{\sin\beta}$.

We'll illustrate each of these, introducing as well the `vector` object in Sage.

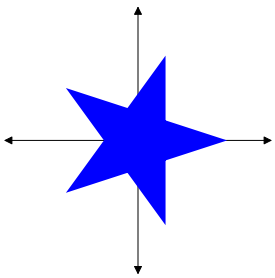
We already mentioned vectors earlier; it is basically a matrix with only one column. A vector is similar to a Sage tuple or list in that it contains an ordered sequence of numbers. It differs from a Sage tuple in that we associate it with certain mathematical operations. What we care about most here is that you can multiply an $m \times n$ matrix to an $n \times 1$ vector; the result is an $m \times 1$ vector. Typically we use square matrices, and the result of multiplying an $n \times n$ matrix to an $n \times 1$ vector is another $n \times 1$ vector, which is convenient for repeated application.

Sage conveniently allows us to use vectors as points in plotting commands, such as `point()`, `line()`, and `polygon()`. This makes it easy to illustrate how the transformation matrices work.

To do this, we'll work on a polygon defined by the following points, *in this order:*

$$(1,0),\quad \left(\cos\frac{4\pi}{5},\sin\frac{4\pi}{5}\right),\quad \left(\cos\frac{8\pi}{5},\sin\frac{8\pi}{5}\right),\quad \left(\cos\frac{2\pi}{5},\sin\frac{2\pi}{5}\right),\quad \left(\cos\frac{6\pi}{5},\sin\frac{6\pi}{5}\right).$$

Define a list `V` whose elements are the vectors with these values. Plot the polygon defined by these points and you will see an almost-nice, five-pointed star:



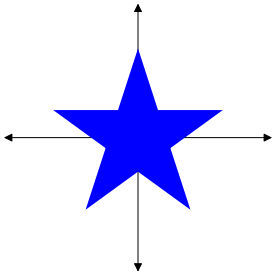
(If you're getting a pentagon instead of a star, make sure the points are in the order above. If not, `.pop()` and `.insert()` until they are.) We say "almost-nice" because it's a little off-kilter. Wouldn't it be nice[66] to have it point upwards?

We can fix this. We have the technology. How shall we go about it? We could rotate it by $1/20$ of a full rotation, or by $2\pi/20$, or by $\pi/10$. Up above, we claimed that the rotation matrix would accomplish this, if only we set $\alpha = \pi/10$. Let's try that. We'll create a corresponding rotation matrix `M`, then use that to create a new list, `U`, whose points are the *rotations* of the points of `V` by $\pi/10$. Creating `U` is easy with a list comprehension:

```
sage: M = matrix([[cos(pi/10), -sin(pi/10)], \
                  [sin(pi/10), cos(pi/10)]])
sage: U = [M*v for v in V]
```

Now plot the polygon formed by `U` to verify that we have, in fact, righted our star:[67]

```
sage: polygon(U)
```



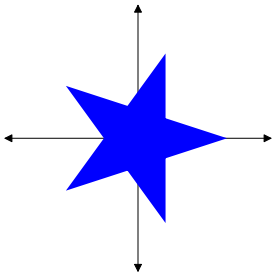
How about reflection? If we look at the original star, we could reflect it about the y-axis, which has slope…

[66] If you're one of those wackos who thinks this star is just fine the way it is, and it would better to rotate the axes instead, well! You have a future in mathematics. If you're not one of those wackos, and think it's better to rotate the star… yeah, you might have a future in mathematics, too. But we're keeping an extra eye on you, just to make sure you don't get out of line. Sort of like this star here.

[67] Or rather, if you believe the previous footnote, we have *wronged* our polygon.

Hmm. Well, now, *that's* annoying. The y-axis has undefined slope. We generally get undefined slope by dividing by 0. The denominator of the slope for the reflection matrix is $\sin\beta$, so if we have $\sin\beta = 0$, we should end up reflecting about the y-axis. That works for $\beta = 0$ and $\beta = \pi$; we'll try the first and hope for the best.

```
sage: N = matrix([[cos(0), sin(0)], \
                  [sin(0), -cos(0)]])
sage: W = [N*v for v in V]
sage: polygon(W)
```



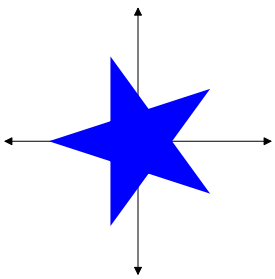
Well, *that's* annoying. It didn't flip at all!

Actually, it did flip; we just can't see it. First off, the slope actually become $^{1-\cos 0}/_{\sin 0} = {}^{1-1}/_{0} = {}^{0}/_{0}$; for a vertical slope, we need *non*-zero divided by zero. A glance at N reinforces this:

```
sage: N
[ 1 0]
[ 0 -1]
```

If you think about how matrix multiplication works, the effect of N on any vector will be to preserve the x value and reverse the y value. So we ended up with a vertical flip, rather than a horizontal flip. Looks as if we want to use $\beta = \pi$.

```
sage: N = matrix([[cos(pi), sin(pi)], \
                  [sin(pi), -cos(pi)]])
sage: W = [N*v for v in V]
sage: polygon(W)
```



Great!

For a reflection matrix, points along the axis of reflection remain fixed: if we were to apply the *corrected* `N` to the vector $(0,2)$, it would return the vector $(0,2)$. This is a special case of what we call an *eigenvector*, a vector that remains on the same line after multiplication by a matrix. Every matrix has its own eigenvectors, and we can "discover" them using the method `.eigenvectors_right()`. In this case:

```
sage: N.eigenvectors_right()
[(-1, [(1, 0)], 1), (1, [(0, 1)], 1)]
```

That's a lot of information, so let's see what it's telling us.

The list contains one tuple per eigenvector. Each tuple contains three items:

- First, it gives us what's called an *eigenvalue*. Eigenvalues tell you how the *size* of the vector changes; it will still lie on the same line through the origin, but the size will change. The basic relationship between a matrix M, an eigenvector $\mathbf{e}$, and the corresponding eigenvalue λ is that $M\mathbf{e} = \lambda\mathbf{e}$. You can also extract a matrix's eigenvalues using the `.eigenvalues()` method.
- Second, it gives a list of eigenvectors that correspond to that eigenvalue. In most cases, you would expect to see only one eigenvector per eigenvalue, but it can happen that you get more than one.
- Finally, it gives the *multiplicity* of the eigenvalue. This relates to the following idea: eigenvalues are the roots $\lambda_1, \lambda_2, \ldots, \lambda_m$ of a polynomial called the matrix's *minimal polynomial*. This polynomial factors linearly as $(x-\lambda_1)^{a_1}(x-\lambda_2)^{a_2}\cdots(x-\lambda_m)^{a_m}$. Each power of a distinct linear factor is the corresponding eigenvalue's multiplicity. We are not interested in this for the current application.

You typically hope for the eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_m$ to be **linearly independent**; that is, the only way to write

$$a_1\mathbf{e}_1 + \cdots + a_m\mathbf{e}_m = 0$$

is if $a_1 = \cdots = a_m = 0$. We can see this in `N`'s eigenvectors, since

$$a_1\begin{pmatrix}1\\0\end{pmatrix} + a_2\begin{pmatrix}0\\1\end{pmatrix} = 0 \quad\Rightarrow\quad \begin{pmatrix}a_1\\a_2\end{pmatrix} = 0 \quad\Rightarrow\quad a_1 = 0,\ a_2 = 0.$$

Unfortunately, not every matrix of dimension n has n distinct eigenvalues; if you look at the scaling matrix

$$S = \begin{pmatrix}3 & 0\\0 & 3\end{pmatrix},$$

you will find that it has only one eigenvalue of multiplicity 2, because it treats every point exactly the same way.

As we mentioned, each eigenvector remains on the same line through the origin when we apply the matrix to it. In terms of points, this means the eigenvectors define a line of points that may move around on that line, either by reversing direction or by changing in size, but the points remain on the same line through the origin. In the case of the reflection matrix `N`, the eigenvectors are $(1,0)^T$, which lies on the x-axis, and $(0,1)^T$, which lies on the y-axis. The eigenvalue for $(1,0)^T$ is -1; this means that $(1,0)$ moves to $(-1,0)$, which still lies on the x-axis. The eigenvalue for $(0,1)^T$ is 1; this means that $(0,1)$ moves to $(0,1)$, which still lies on the y-axis. This isn't true for

points on other lines through the origin; the point $(3,5)$, for instance, moves to the point $(-3,5)$, which lies on a completely different line through the origin. Of course, *we expect that* because `N` is a reflection matrix.

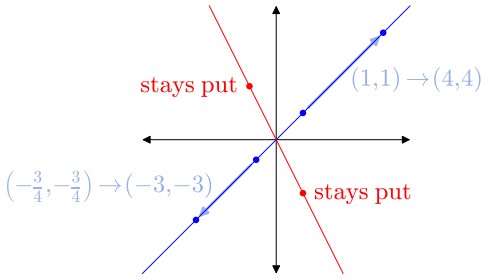
To conclude this investigation, let's consider a different matrix,

$$M = \begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix}.$$

Use Sage to verify that its eigenvectors are

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 \\ -2 \end{pmatrix},$$

with corresponding eigenvalues 4 and 1. This means that we expect points on the same line as $(1,1)^T$ to remain on that line, but to be rescaled by a factor of 4, while points on the same line as $(1,-2)^T$ will remain in the same place:



As you might imagine, things get a little strange with a rotation matrix, because it moves every point in the plane *except* the origin to a different line through the origin. In this case, you would find its eigenvectors to have complex values. Try it!

Exercises

True/False. If the statement is false, replace it with a true statement.

1. Sage has features to find both exact and approximate solutions to equations.
2. Sage can find the exact solution to any equation, so the main reason to approximate solutions is that they're easier to look at.
3. The `solve()` command finds both exact and approximate solutions to an equation.
4. The `.roots()` method is not as successful as the `find_root()` command, even when we specify a ring of approximations to numbers, like `RR` or `CC`.
5. A root's multiplicity is unrelated to the geometry of the curve.
6. The results of the `solve()` command are in a similar format for inequalities and systems of equations.
7. Sage sometimes decides how to solve a problem based on the ring in which its values lie.
8. You must always specify the ring of a matrix when creating it.
9. You can extract eigenvalues from the results of the `eigenvectors()` command.
10. Eigenvalues are of merely theoretical importance.

Multiple choice.

1. Which of the following commands and methods does *not* produce exact solutions *by default?*
   A. `solve()`

   B. `.roots()`
   C. `.eigenvalues()`
   D. `find_root()`
2. The result of `solve()` when given one equation in one variable is:
   A. the value of an approximate solution
   B. a list of lists of solutions and multiplicities
   C. a list of linear equations that describe the solutions
   D. a list of lists of linear equations, each of which describes a solution
3. The result of `solve()` when given several equations in more than one variable is:
   A. the value of an approximate solution
   B. a list of lists of solutions and multiplicities
   C. a list of linear equations that describe the solutions
   D. a list of lists of linear equations, each of which describes a solution
4. The result of `find_root()` when given an equation in one variable is:
   A. the value of the approximate solution
   B. a list of lists of solutions and multiplicities
   C. a list of linear equations that describe the solutions
   D. a list of lists of linear equations, each of which describes a solution
5. The result of `find_root()` when given an equation in one variable is:
   A. the value of the approximate solution
   B. a list of lists of solutions and multiplicities
   C. a list of linear equations that describe the solutions
   D. a list of lists of linear equations, each of which describes a solution
6. Which of the following commands and methods requires you to specify the endpoints of an interval that contains a solution?
   A. `solve()`
   B. `find_root()`
   C. `.roots()`
   D. none of the above
7. Which of the following methods to a matrix returns its eigenvalues?
   A. `eigenvalues()`
   B. `eigenvectors_right()`
   C. all of the above
   D. none of the above
8. The vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ are linearly independent when:
   A. We can find a matrix M whose eigenvectors are $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$.
   B. The set $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ defines a vector space.
   C. The only way $a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = 0$ is if $a_1 = a_2 = \cdots = a_n = 0$.
   D. Vectors declare their independence from a vector space and create their own subspace.
9. Why should we expect a reflection matrix to have two linearly independent eigenvectors?
   A. Every two-dimensional matrix has two linearly independent eigenvectors.
   B. A reflection always moves *one* point to a *second* point, and that's how we count the number of linearly independent eigenvectors.
   C. A reflection is like a mirror, so we ought to expect the second eigenvector to mirror the first.

D. The points of two lines through the origin remain on those lines: the axis of symmetry, and the line perpendicular to it.

10. Why do we expect the eigenvalues of a rotation matrix to have complex values?
    A. A rotation matrix moves points on the real plane to points on the complex plane, and vice-versa.
    B. The only point on the real plane that remains on the same line through the origin is the origin itself.
    C. Eigenvalues are always complex; we just don't notice that when their values are real.
    D. Complex problems require complex solutions.

Programming.

1. Use `solve()` to find the solutions to generic degree-three and degree-four polynomial equations. How many are there? (Don't count by hand! Use Sage to count the solutions!)
2. The `solve()` command does not return the solutions to a single equation in one variable by order of the complex norm. Write a function that accepts an equation as an argument, calls `solve()` to solve it, then sorts the solutions using the norm. *Hint:* You may want to use or adapt the key function `by_norm()` that we defined on p. 113.
4. Write a function that accepts an inequality as its argument, solves it, then produces a graph of its solutions on the number line, similar to the one on p. 138.
5. (a) Write pseudocode for a function that accepts two functions f and g, solves for their intersections, then computes the total area between f and g. (We find the total area by adding the absolute value of the integrals for each interval defined by intersections of f and g.)
   (b) Implement your pseudocode as a Sage function that uses the `roots()` command to find exact solutions in $\mathbb{R}$.
   (c) Write an interactive function that allows the user to specify the functions f and g, as well as a color. It then calls the Sage code you wrote in part (b) to determine the total area between f and g, graphs f and g in black with thickness 2, fills the area between them using the user-specified color, and finally writes the area above the curves.
6. We used the `.roots()` method to find the roots *and* multiplicities of

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

   Create a graph of f on an interval that includes all three roots, but is not so large as to lose the details of how f meanders around them. Make this plot black, of thickness 1. Add to it three other plots of f, each of whose minimum and maximum x values are in the neighborhood of a different root. Make these plots red, of thickness 3. Now that you have this plot, describe a geometric similarity between a root of multiplicity m and the graph of x^m.
7. While Sage has an `implicit_plot()` command, it lacks an `implicit_diff()` command to perform implicit differentiation. (You may want to review the definition of implicit differentiation in your calculus book.) Write a function to do this, implementing the following steps:
   (a) Move everything in `f` to one side of the equation. Make sure `f` is a function in terms of `x` and `y`.
   (b) Define `yf` as an implicit function of x using the command, `yf=function('yf')(x)`.
   (c) Replace `y` in `f` by `yf`. *Hint:* If you defined `f` as a function in `x` and `y`, then it's a matter of redefining `f` as a function of `x` and `yf`.
   (d) Let `df` be the derivative of `f`.
   (e) Solve `df` for `diff(yf)`.

(f) Return the solution — not the list of equations, mind, but the right-hand side. For instance, the result of `implicit_diff(y==cos(x*y))` should be `-sin(x*yf(x))*yf(x)/(x*sin(x*yf(x)) + 1)`.

Decision-making

As mentioned earlier, one of the fundamental questions of mathematics is,

How can we find a root[68] *of a polynomial?*

As that is more or less impractical in many cases, we turn to the related question,

How can we approximate a root of a polynomial?

There are a number of ways to do this, but we will describe a way to "approximate" the root "exactly." That is not really the contradiction it may seem, as our approximation consists of a precise interval (a, b) of rational numbers in which the polynomial has at least one root. So the value will be an approximation, insofar as we're not sure *which* $c \in (a, b)$ is the root, but it's an *exact* approximation, in that the solution is free from all error.[69] As a bonus, the technique we'll look at will:

- work for any continuous function, not just polynomials; and
- on rare occasions,[70] give us the exact value!

This technique is the *method of bisection*, and it's based on a big-time fact from Calculus:

THE INTERMEDIATE VALUE THEOREM. *If a function f is continuous on the interval $[a, b]$, then for any y-value between $f(a)$ and $f(b)$, we can find a $c \in (a, b)$ such that $f(c)$ is that y-value.*

For instance, $f(x) = \cos x$ is continuous everywhere, so it's certainly continuous on the interval $[\pi/6, \pi/3]$. Now, $f(\pi/6) = \sqrt{3}/2$, while $f(\pi/3) = 1/2$, and $\sqrt{2}/2$ lies between $1/2$ and $\sqrt{3}/2$, so there must be some $c \in [\pi/6, \pi/3]$ such that $f(c) = \sqrt{2}/2$.

The method of bisection

Our example with $\cos x$ won't seem impressive; after all, you already knew that $f(\pi/4) = \sqrt{2}/2$. Fair enough, but consider this scenario:

if
- $f(a)$ is positive and
- $f(b)$ is negative

then
- 0 lies between $f(a)$ and $f(b)$, so
- a root c must lie between a and b.

For example, suppose we want to identify the root of

$$f(x) = \cos x - x.$$

[68] In case you've forgotten, a "root" is a value of an expression that, when substituted into the expression, yields 0.

[69] By contrast, floating point answers are an accurate approximation, rather than an exact approximation, in that floating point gives us *one* number which is slightly wrong.

[70] Very rare, but they do exist.

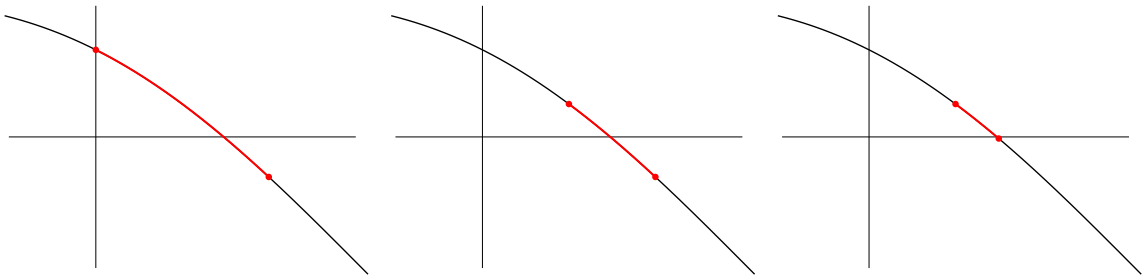


FIGURE 21. The Method of Bisection employs the Intermediate Value Theorem to narrow the interval containing a root from $(0, 1)$ down to $(1/2, 3/4)\ldots$ and further!

If you plot it in Sage, the graph suggests a root lies between $x = 0$ and $x = 1$. But can we trust the graph? Indeed we can; as you learned in Calculus,

- $\cos x$ is continuous;
- x is a polynomial, and all polynomials are continuous; and
- sums, differences, and products of continuous functions are continuous.

So f is continuous, and the Intermediate Value Theorem confirms the visual inspection.

Of course, knowing there's a root between 0 and 1 doesn't exactly build much confidence. We'd like to locate this root more accurately than that. *How?*

In the Method of Bisection, we cut the interval $[a, b]$ into two parts. In this case, we cut it into $[0, 1/2]$ and $[1/2, 1]$.We then test the new endpoint in the function: $f(1/2) > 0$. We already knew $f(0) > 0$, so the Intermediate Value Theorem cannot guarantee that a root lies between 0 and $1/2$. On the other hand, $f(1) < 0$; since 0 lies between $f(1/2)$ and $f(1)$, the Intermediate Value Theorem does guarantee that a root lies between $1/2$ and 1. So we replace $[0, 1]$ by $[1/2, 1]$.

An interval of size $1/2$ is still not terribly impressive, but there's no reason we can't keep going. Cut the interval into two parts again, $[1/2, 3/4]$ and $[3/4, 1]$. The new endpoint satisfies $f(3/4) < 0$, so the root must lie between $1/2$ and $3/4$. Figure 21 illustrates each of these first three steps in the process.

We can persist in this as long as we like. Each time, the uncertainty decreases by $1/2$, so repeating this 20 times gives us an interval of width $1/2^{20} \approx 1.0 \times 10^{-6}$, or 0.000001. That's a little rough to do by hand, but it's straightforward for a computer to repeat; we've already seen how to do that.

But how should we tell the computer to decide which endpoint to replace by the midpoint? Enter the **if/else if/else** control structure:

```
if condition1
    what to do if condition1 is true
else if condition2
    what to do if condition2 is true
...
else
    what to do if neither condition is true
```

This directs the execution based on the listed conditions, allowing the computation to proceed in a manner appropriate to the situation.

Not all parts of the structure are necessary. Only an **if** *must* appear; after all, **else** makes no sense without an **if**. But you can make sense of an **if** that lacks an **else**; this is often useful if a particular value needs some extra "massaging" before the algorithm can proceed.

In our case, we want the algorithm to replace the endpoint with the same sign as the midpoint. That implies the following pseudocode:

algorithm *Method_of_Bisection*
inputs
- $a, b \in \mathbb{R}$
- f, a function such that:
  - f is continuous on $[a, b]$
  - $f(a)$ and $f(b)$ have opposite signs
- n, the number of bisections to perform

outputs
- $[c, d] \subseteq [a, b]$ such that
  - $d - c = 1/2^n (b - a)$, and
  - a root of f lies in $[c, d]$

do
  let $c = a$ and $d = b$
  repeat n times
    let $e = 1/2(c + d)$
    if $f(c)$ **and** $f(e)$ have the same sign
      replace c by e
    else
      replace d by e
  return $[c, d]$

Unfortunately, this is not yet enough to implement in Sage, as one question remains:

How do we decide whether $f(c)$ and $f(e)$ have the same sign?

We consider two ways.

A clever way that you may have thought of is based on the pre-algebra observation that two real numbers have the same sign if and only if their product is positive:

$$(-2) \times (-2) > 0 \quad \text{and} \quad 2 \times 2 > 0 \qquad \text{but} \qquad (-2) \times 2 < 0 \quad \text{and} \quad 2 \times (-2) < 0.$$

So we can replace the test

if $f(c)$ **and** $f(e)$ have the same sign

with

if $f(c) f(e) > 0$

and be done with it. This is fairly easy to implement as Sage code, as Sage offers keywords that match our pseudocode almost identically: `if`, `elif`, and `else`. As you might expect, the commands you want Sage to execute in each case should appear beneath them, indented. So our pseudocode translates as follows.

```
sage: def method_of_bisection(a, b, n, f, x=x):
          c, d = a, b
          f(x) = f
          for i in xrange(n):
              # compute midpoint, then compare
              e = (c + d)/2
              if f(c)*f(e) > 0:
                  c = e
              else:
                  d = e
          return (c, d)
```

Notice the colons! If you accidentally forget one of the colons, Python will raise an error.

```
sage: ...
              if f(c)*f(e) > 0
                  c = e
      ...
SyntaxError:  invalid syntax
```

Once you've typed it in correctly, let's check that this works.

```
sage: method_of_bisection(0, 1, 20, cos(x) - x)
(387493/524288, 774987/1048576)
```

But there is still a more excellent way.

Boolean logic

Until now we've used the Sage identifiers `True` and `False` without much comment; after all, their meanings are fairly obvious unless you have reassigned them. Yet these two ideas lie at the foundation of most practical computation. A field called Boolean logic considers the logical ways we can perform from the two basic concepts of *True* and *False*.[71]

[71] Notice the distinction between Sage identifiers of terms with fixed meaning (`True` and `False`) and the fundamental constants of Boolean logic (*True* and *False*).

Four "fundamental" operations. The four fundamental operations of Boolean logic are the operators **or**, **and**, **xor**,[72] and **not**.[73] They are governed by the following "truth" tables, which indicate how variables with the values *True* or *False* behave under the given operations.

x	**not** x
True	*False*
False	*True*

x	y	x **or** y
True	*True*	*True*
True	*False*	*True*
False	*True*	*True*
False	*False*	*False*

x	y	x **and** y
True	*True*	*True*
True	*False*	*False*
False	*True*	*False*
False	*False*	*False*

x	y	x **xor** y
True	*True*	*False*
True	*False*	*True*
False	*True*	*True*
False	*False*	*False*

Having made the meanings of these terms precise, we hasten to point out that they really ought to be intuitive to you; the only real difference between the precise meanings and your intuitive understandings appears in **or** and **xor**. We generally will not find **xor** necessary in this text, but it does have its uses.

Boolean operators in Sage. These concepts all translate immediately into Sage: `True`, `False`, `not`, `or`, `and`, and `xor`.

With their help, we can now rewrite our Sage code with Boolean logic instead of a mathematical trick. Checking that two real numbers c and e have the same sign is a matter of testing that

$$(f(c) > 0 \textbf{ and } f(e) > 0) \textbf{ or } (f(c) < 0 \textbf{ and } f(e) < 0)$$

and this translates directly into the Sage code

```
(f(c) > 0 and f(e) > 0) or (f(c) < 0 and f(e) < 0)
```

[72] Shorthand for **exclusive or**, which name derives from the fact that, in general, you can *either* cut the cake *or* have it, *but not both.* Phrased slightly different, you can have *exclusively* one option *or* the other.

How does this differ from regular old **or**? The rule does not apply on birthdays, as you can cut your cake **or** have it **or** both. (Complaining that your parents didn't buy enough gifts is also an option, but let's not get carried away.) Since you *can* have both, regular old **or** is considered an "inclusive" or.

You might wonder why we shorten **exclusive or** to **xor**. This is because computer scientists, especially computer engineers, are violently allergic to writing any word that is more than three or four letters long. (Really. You think I'm making this up? — Well, okay, we're exaggerating a little, but only a little.)

[73] This is an example of the outright mendacity we promised in the preface. As promised, it sounds a lot better than the truth. The lie consists in calling the operations "fundamental," for the list contains at least one redundant operation, making at least one of them *not* "fundamental." In particular, we can rewrite a **xor** b as [a **and** (**not** b)] **or** [(**not** a) **and** b]. So we can already reduce the list to **or**, **and**, and **not**.

But that's not all! We can also rewrite a **and** b as **not** [(**not** a) **or** (**not** b)]. (This latter fact is known as DeMorgan's Laws.) So we can actually reduce the list of logical operations to **or** and **not**. While it is fitting that the number of fundamental operations on two objects (*True* and *False*) should number two (**not** and **or**), no one in his right mind does this. It is much better just to lie to you and tell you all four operations are "fundamental."

You may be wondering if the parentheses are actually needed here. *In this case, no*, because Sage always tests `and` before it tests `or`. In general, however, it is a good practice to indicate explicitly the order of operations, as it matters a great deal whether you mean

```
sage: False and False or True
True
sage: False and (False or True)
False
sage: (False and False) or True
True
```

Be sure you understand why we get those results.

We can now state a different Sage code for the Method of Bisection.

```
sage: def method_of_bisection(a, b, n, f, x=x):
          c, d = a, b
          f(x) = f
          for i in xrange(n):
              # compute midpoint, then compare
              e = (c + d)/2
              if (f(c) > 0 and f(e) > 0) or (f(c) < 0 and f(e) < 0)
                  c = e
              else
                  d = e
          return (c, d)
```

Another Boolean relation, and a caution on using it. Until now, we've relied on the Boolean relations `==`, `<`, `>`, `<=`, and `>=`. You may have noticed that we did not list an operator for inequality. Technically, we can translate the pseudocode

$$a \neq b$$

as

```
sage: not (a == b)
```

but that isn't very elegant. Sage offers us a different symbol in these circumstances, `!=`, which *sort of* looks like a sloppy way of slashing through an equals sign, then dropping some ink.

Both equality and inequality can be dangerous when performing comparisons. Exact numbers are no problem, but if your value has floating point, then the computer might mistakenly think two numbers are different when they are not. For example:

```
sage: 14035706228479900.  - 14035706228479899.99 != 0
False
```

That's a very odd conclusion to make, for which we can thank the rounding of floating point numbers. On the other hand, recall from p. 109 the result of Euler's Method, which we concluded was 8.

```
sage: 8 - 7.99999999999999 != 0
True
```

Imagine an algorithm that terminates only if it has actually computed `0`. We probably *do* have equality here, but the computer doesn't detect it yet (if at all).

The solution in cases like this is to test that the difference between the values is sufficiently small, rather than whether they are equal. For instance, if lying within ten decimal digits suffices, the following test would fix the previous one.

```
sage: 8 - 7.99999999999999 > 10**(-10)
False
```

Breaking a loop

There will be occasions when there's no point to continuing a loop. One example would occur if we applied the Method of Bisection to a function whose root is a rational number with a power of 2 in the denominator. For example, $f(x) = 4x + 3$ has a root at $x = -3/4$. Even if you ask the Method of Bisection to repeat 20 times, in all likelihood it will finish far, far earlier. Suppose, for instance, that we start on the interval $(-1, 1)$. The first bisection gives us the interval $(-1, -1/2)$, while the second gives us the interval $(-1, -3/4)$. At this point we've already found a root $f(-3/4) = 0$, so we really ought to quit, but the algorithm doesn't include a way to test this. Instead, it continues on for 18 more iterations, concluding with the unwieldy-looking $(-393217/524288, -3/4)$.

A better approach would be to re-formulate the loop so that it stops the moment it finds a root. Testing for this is straightforward in Sage, but how do we tell it to stop the loop? One way would be to place a `return` statement in the loop, but that's not a good idea if an algorithm is looking for a value so that it can then perform some computation with it *before* returning.

This is where the `break` keyword comes in. A `break` statement tells Sage to stop the current loop, and proceed from the first statement outside the loop's indented block. A `break` applies only once, so if the loop is nested inside another loop, Sage will not break out of it.

We can now reformulate our `method_of_bisection()` function as follows.

```
sage: def method_of_bisection(a, b, n, f, x=x):
          c, d = a, b
          f(x) = f
          for i in xrange(n):
              # compute midpoint, then compare
              e = (c + d)/2
              if (f(c) > 0 and f(e) > 0) or (f(c) < 0 and f(e) < 0)
                  c = e
              elif f(e) == 0:
                  c = d = e
                  break
              else:
                  d = e
          return (c, d)
```

This time, when the code computes `e == -3/4`, it determines that

- the condition for the first `if` statement is false, as neither `f(e) > 0` nor `f(e) < 0`; but
- the condition for the second `if` statement is true, as `f(e) == 0`.

So the code should assign $-3/4$ to both `c` and `d`, then execute the `break` statement. The first statement outside the loop's indented block is `return (c, d)`, so the code will return the tuple $(-3/4, -3/4)$.

```
sage: method_of_bisection(-1, 1, 20, 4*x + 3)
(-3/4, -3/4)
```

Exceptions

Exceptions are one of the newer concepts in computer programming, and they are related to decision-making, if only because they can replace `if`/`else` statements in many situations. We discuss this here briefly.

Suppose you have an algorithm that needs to divide:

let $c = n/d$

For instance, you have a division algorithm for some mathematical objects. This innocuous statement masks a mathematical danger: what if $d = 0$?

We could of course guard the statement with an **if**/**else** sentinel:

if $d \neq 0$
  let $c = n/d$
else
  do something smart about the case $d = 0$ *— scold the user, say*

Of course, the code may contain other variables with their own potentials for dangerous values, leading to the following ugly-looking code:

if *condition1*
  if *condition2*
    if *condition3*
      if *condition4*
        ...
      else if *condition5*
        ...
      else
        ...
    else if *condition6*
      ...
  else
    ...

This is not merely ugly, it is hard to read. Computer scientists have a technical name for it: the Pyramid of Doom.[74]

In pseudocode, we try to list the problems before they occur, by specifying in the **inputs** section that, for instance, $d \neq 0$. For instance, pseudocode that expects to take a derivative at an arbitrary point would probably state something like this:

inputs
  - f, a function that is differentiable everywhere on the real line

...but we don't do it all the time, especially when it should be obvious from the algorithm's purpose. After all, it's pseudocode, which is supposed to be readable. If we specified every constraint, the **inputs** section would become a tedious recitation of pedantries. Mathematicians often rely on a reader's intuition.

We don't have this outlet in actual program code, so we adopt a different approach, called exception handling. Exception handling relies on what we call a `try`/`except` block. As the name suggests, it *tries* a block of code which is indented beneath the `try` statement, and if an error occurs, it applies a different block of code that appears beneath an `except` statement. Both `try` and `except` are keywords, so you cannot use them as identifiers.

The precise usage is as follows:

[74] Honest, I am not making this up.

```
sage: try:
          first_try_statement
          second_try_statement
          ...
      except ExceptionList:
          first_exception_statement
          second_exception_statement
```

When Sage encounters such a block of code, it tries `first_try_statement`, `second_try_statement`, ... If it can perform all the tasks listed under the `try` statement, it skips over the `except` block entirely, and continues on.

However, if Sage encounters some error, it compares it to the exceptions listed in the `ExceptionList`. If one of them matches, it performs `first_exception_statement`, `second_exception_statement`, and so forth, to completion — unless it encounters another error. In that case, Sage gives up and passes the error back to the client.[75]

Here's an example you can test without even writing a function:

```
sage: try:
          1/0
      except ZeroDivisionError:
          Infinity
```

If you type this in a cell or on the command line, then execute it, Sage first tries to divide 1 by 0. That obviously won't work, and Sage raises a *`ZeroDivisionError`*. Our `except` clause catches this error, so Sage passes into that block, finding the statement *`Infinity`*. You should not be surprised, then, that the result is:[76]

---

[75] If you don't know the type of an error, or you want to process all possible errors with the same code, or if you're feeling particularly lazy, it is actually possible to catch *all* errors by omitting the `ExceptionType`. The authors of this text feel particularly lazy most of the time, but for pedagogical reasons we think it better to avoid this.

You may sometimes wish to process information that comes back from the exception. In this case, you can use the construct

```
      except ExceptionType as e:
```

and then look at the details of `e`. We do not go into these possibilities.

[76] It is also possible to "nest" `try...except` clauses; that is, place a `try...except` clause inside another. You can do this in either a `try` or an `except` clause. For instance:

```
sage: try:
          1/0
      except ZeroDivisionError:
          try:
              0**Infinity
          except NotImplementedError:
              print 'No dice'
```

```
+Infinity
```

Rather than process all exceptions in one `except` block, it is possible to process exceptions to one `try` block in several `except` blocks that follow. Line up each `except` clauses with the `try` clause, list the precise error for each except, and supply indented code accordingly. This helps keep `try`/`except` structures from turning into a modern Pyramid of Doom.

In addition to catching errors, a function can invoke another keyword, `raise`, to "raise" errors of its own. The usage is as follows:

```
sage: raise ErrorType, message
```

For `message`, you can use any string. For `ErrorType`, one can raise any exception of the appropriate type; the two usual suspects are:

- *`TypeError`*, where the input is of the wrong type: for instance, the algorithm expects a matrix, but received a number; and
- *`ValueError`*, where the input has the right type, but the wrong value: for instance, the algorithm expects a nonzero number, but received zero.

We will not use the `raise` keyword in general, but you should be aware that this mechanism is how Sage's authors communicate errors to you.

Let's try this approach on something more substantial.

A "normal" example. An exercise on p. 103 asked you to write a Sage function to compute the normal line to a mathematical function f at $x = a$. The function you wrote should work in *most* situations.

```
sage: var('t')
t
sage: normal_line(t**2, 1, t)
1/2*t + 1/2
```

In all likelihood, however, your code would encounter the following error:

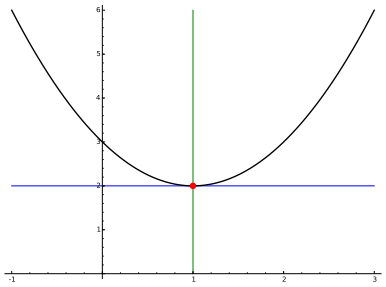
```
sage: normal_line((t - 1)**2 + 2, 1, t)
ZeroDivisionError:  Symbolic division by zero
```

Well, of *course* this would happen:

- the normal line is perpendicular to the tangent line;
- a perpendicular line is a negative reciprocal; and
- the slope of the line tangent to $(t-1)^2+2$ at $t = 1$ is 0.

"Reciprocal" means "division," and division by zero means *`ZeroDivisionError`*.

This problem is unavoidable. What we need is a way to deal with it, and as it happens, we have at least two at our disposal. But first we need to think about the underlying problem; namely, the derivative is 0. In this case, the normal line is actually the vertical line $x = a$. We illustrate this with the function $f(t) = (t-1)^2+2$ at $t = 1$:



A vertical line is not a function, so one option is to raise an exception of our own. Something like this, perhaps?

```
        raise ValueError, 'The normal line is vertical.'
```

This works, and *ValueError* seems the appropriate exception, since the type is just fine (a function, and a differentiable function, to boot) but its value is inappropriate.

The trouble with this approach is that it reports an error when the situation really is salvageable! After all, a normal line *exists;* it just isn't a *function*. A more appropriate approach might be to return an equation for the normal line.

```
        return x==a
```

Now, this might make your original algorithm inconsistent if you returned a function (which, most likely, you did). We can still make the algorithm consistent by modifying the well-behaved case to return the equation `y == m(x - a) + f(a)` instead of the right-hand side alone.

So, how to implement this in Sage? Again, we have two options. The first is to use an `if`/`else` structure as a "guard" around the computation of `m`. This way, no exception occurs at all.

```
sage: def normal_line_with_guard(f, a, x=x):
          f(x) = f
          df(x) = diff(f, x)
          if df(a) == 0:
              # horizontal perpendicular to vertical
              result = x == a
          else:
              m = 1/df(a)
              result = y == m*(x - a) + f(a)
          return result
```

The other approach is to use a `try`/`except` structure to catch a *ZeroDivisionError*.

```
sage: def normal_line_with_catch(f, a, x=x):
          f(x) = f
          df(x) = diff(f, x)
          try:
              m = 1/df(a)
              result = y == m*(x - a) + f(a)
          except ZeroDivisionError:
              # horizontal perpendicular to vertical
              result = x == a
          return result
```

Try them both to see that both functions work with both vertical and skew normal lines:

```
sage: normal_line_with_guard((t - 1)**2 + 2, 0, t)
y == -1/2*t + 3
sage: normal_line_with_guard((t - 1)**2 + 2, 1, t)
t == 1
sage: normal_line_with_catch((t - 1)**2 + 2, 0, t)
y == -1/2*t + 3
sage: normal_line_with_catch((t - 1)**2 + 2, 1, t)
t == 1
```

Which approach is "better?" The second version is generally recommended for two reasons:

- The logic flows better, so it is more readable. The `if`/`else` construct does not make it obvious *why* we test `df(a)==0`, just *that* we test it. It requires the reader to think more about what we're doing. By contrast, the `try`/`except` construct makes it clear that an error might occur (the only reason to use a `try`); if so, what that error is (*`ZeroDivisionError`*); and moreover, what to do in case it does occur (the `except` block).
- Recall that Sage's interface uses Python. As computer languages go, Python's exceptions are efficient. There is a small penalty to using exceptions; in our tests, the `if/else` guard is about 20% faster than the `try`/`except` catch when division by zero actually occurs.

  Don't jump to that conclusion yet! Division by zero happens *rarely,* so we should ask ourselves how the approaches compare in the usual circumstances. It turns out that the `try`/`except` catch is *more than 30% faster* than the `if`/`else` guard!

A more involved example. On p. 277 we describe Dodgson's method to compute a determinant. Dodgson's method is an algorithm you can implement using a **for** loop, so your instructor may have assigned it already. It's a cute algorithm; if you haven't tried it by hand yet, try it now real quick.

Unfortunately, it has problems. Those problems are hard to fix. Another algorithm to compute determinants is based on Gaussian elimination. It also has problems, but these are easy to fix. We start with the pseudocode below, which starts counting rows and matrices from 1.

algorithm *Gaussian_determinant*
inputs
- M, an $n \times n$ matrix

outputs
- $\det M$

do
```
let N be a copy of M
let d = 1
for i ∈ {1,...,n−1}
   for j ∈ {i+1,...,n}
      if N_{j,i} ≠ 0
         let a = N_{j,i}
         multiply row j by N_{i,i}
         subtract a times row i from row j
         divide d by N_{i,i}
return d × (N_{1,1} × N_{2,2} × ··· × N_{n,n})
```

Before implementing it, let's see it in action on

$$M = \begin{pmatrix} 3 & 2 & 1 \\ 5 & 4 & 3 \\ 6 & 3 & 4 \end{pmatrix}.$$

The algorithm begins by copying M to N and setting $d = 1$. It then loops through all but the last row of N, storing the row value in i.

With $i = 1$, the outer loops finds an inner loop with $j = 2$. As $N_{2,1} = 5 \neq 0$, the algorithm stores 5 in a, multiplies row 2 by $N_{i,i} = 3$, subtracts $a = 5$ times row 1 from row 2, then divides d by $N_{i,i} = 3$, obtaining $d = 1 \div 3 = 1/3$. The resulting matrix is

$$N = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 2 & 4 \\ 6 & 3 & 4 \end{pmatrix}.$$

The inner loop next sets $j = 3$. As $N_{3,1} = 6 \neq 0$, the algorithm performs the innermost block again on row 3, obtaining

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

It also modified d, so that $d = 1/9$. So far, so good. The algorithm has completed the inner loop on j when $i = 1$.

The outer loop proceeds to $i = 2$, and finds an inner loop with $j = 3$. As $N_{3,2} = -3 \neq 0$, the algorithm stores -3 in a, multiplies row 3 by $N_{i,i} = 2$, subtracts $a = -3$ times row 2 from row 3,

then divides d by $N_{i,i} = 2$, obtaining $d = (1/9) \div 2 = 1/18$. The resulting matrix is

$$N = \begin{pmatrix} 3 & 2 & 1 \\ 0 & 2 & 4 \\ 0 & 0 & 24 \end{pmatrix}.$$

The algorithm has completed the inner loop on j when $i = 2$. That was the last value of i, so the loops are complete; the algorithm returns

$$d \times \left(N_{1,1} \times N_{2,2} \times N_{3,3}\right) = 1/18 \times (3 \times 2 \times 24) = 8.$$

Either Sage or a more traditional method will confirm that $\det M = 8$.

We can implement this in Sage in the following code:

```
sage: def gaussian_determinant(M):
          N = copy(M)
          d = 1
          n = N.nrows()
          for i in xrange(n-1):
              for j in xrange(i+1,n):
                  if not (N[j,i] == 0):
                      # clear column beneath this row
                      d = d/N[i,i]
                      a = N[j,i]
                      N.set_row_to_multiple_of_row(j,j,N[i,i])
                      N.add_multiple_of_row(j,i,-a)
          return d*prod(N[i,i] for i in xrange(n))
```

If we try it on the remaining matrices given in the exercise on p. 277, we see that it works for the third matrix, but not the second. What goes wrong?

```
ZeroDivisionError:  Rational division by zero
```

Division occurs in only one place in the algorithm: `d` by `N[i,i]`. We must have encountered a `0` on the main diagonal.

How did this happen? We started with

$$M = \begin{pmatrix} 1 & -4 & 1 & 2 \\ -1 & 4 & 4 & 1 \\ 3 & 3 & 3 & 4 \\ 2 & 5 & 2 & -1 \end{pmatrix},$$

and the first pass through the outer loop leaves us with

$$N = \begin{pmatrix} 1 & -4 & 1 & 2 \\ 0 & 0 & 5 & 3 \\ 0 & 15 & 0 & -2 \\ 0 & 13 & 0 & -5 \end{pmatrix}.$$

There we go! During the second pass through the loop, the algorithm divides d by 0, a disaster.

Unlike Dodgson's Method, an easy fix is available. Remember that Gaussian elimination allows us to swap rows, as that merely re-orders the equations that correspond to each row. If we can find another row below $i = 2$ with a nonzero element in column 2, we can swap that row with row 2 and proceed as before. In the example above, row 3 has a non-zero element in column 2, so we swap it and row 2, obtaining

$$N = \begin{pmatrix} 1 & -4 & 1 & 2 \\ 0 & 15 & 0 & -2 \\ 0 & 0 & 5 & 3 \\ 0 & 13 & 0 & -5 \end{pmatrix}.$$

With a non-zero element in row 1, column 1, we can now resume the algorithm. When $j = 3$ there is nothing to do, so the only inner loop that does anything is when $j = 4$, obtaining

$$N = \begin{pmatrix} 1 & -4 & 1 & 2 \\ 0 & 15 & 0 & -2 \\ 0 & 0 & 5 & 3 \\ 0 & 0 & 0 & -49 \end{pmatrix}$$

and $d = 1/15$. Nothing happens on the next value of i, either, so the algorithm returns

$$\frac{1}{15} \times [1 \times 15 \times 5 \times (-49)] = -245.$$

Again, we can verify using Sage or a traditional algorithm for computing determinants that $\det M$ is. . . uhm, 245.

What? We got −245!

PANIC!

Once we're done panicking, let's think about what might have changed, in particular, what might have made the sign go wrong.

The only thing that changed is that we swapped rows. You should remember from past experience with matrices that swapping rows changes the sign of the determinant. *There!* Our method of "saving" the algorithm involved swapping rows, so somehow we have to ensure we multiply the determinant by −1. This isn't too hard; simply multiply `d` by −1.

Before we implement this improved algorithm, we should ask ourselves: what if we hadn't been able to find a non-zero element in column 2 among the rows below row 2? In that case, the structure of the matrix tells us the determinant is 0. This covers all our bases.

We could use an **if/else** control structure to implement the modified algorithm, and this would work just fine (see the exercises). We use a `try/except` this time, partly to illustrate how

it works. However, rather than stuff all this material into the same function, let's split separate tasks into their separate functions, each of which handles a "bite-sized" subproblem.

A function titled `clear_column()` could handle the ordinary task of row reduction. Everything involving saving a zero division could go into a function titled, `unzero()`.

```
sage: def unzero(M, d, i, j):
sage:     # use to swap rows when M[i,i] = 0
          n = M.nrows()
          # look for row k w/nonzero in col j
          for k in xrange(i+1,n):
              if not (M[k,j] == 0):
                  M.swap_rows(k,i)
                  d = -d
                  break
          # M[i,i] == 0?  column clear ("failure")
          if M[i,i] == 0:
              success = False
          else:
              d = d/M[i,i]
              success = True
          return success, M, d
```

The `clear_column()` function could then handle the simplified task of clearing the remaining elements of the column.

```
sage: def clear_column(M, d, i):
          n = M.nrows()
          # clear column beneath this row
          for j in xrange(i+1, n):
              if not (M[j,i] == 0):
                  try:
                      d = d/M[i,i]
                  except ZeroDivisionError:
                      # look for pivot in lower row
                      success, M, d = unzero(M, d, i, j)
                      if not success:
                          raise ZeroDivisionError, 'Result is 0'
                  a = M[j,i]
                  M.set_row_to_multiple_of_row(j, j, M[i,i])
                  M.add_multiple_of_row(j, i, -a)
          return M, d
```

That allows us to simplify the `gaussian_determinant()` function significantly, making it far more understandable.

```
sage: def gaussian_determinant(M):
          N = copy(M)
          d = 1
          n = N.nrows()
          # clear columns left to right
          for i in xrange(n-1):
              try:
                  N, d = clear_column(N, d, i)
              except ZeroDivisionError:
                  return 0
          return d*prod(N[i,i] for i in xrange(n))
```

The function now returns the correct determinant for any matrix you throw at it.

```
sage: M = matrix([[1,-4,1,2],[-1,4,4,1],[3,3,3,4],[2,5,2,-1]])
sage: gaussian_determinant(M)
```

Exercises

True/False. If the statement is false, replace it with a true statement.

1. Sage offers the `if`/`else if`/`else` control structure to make decisions.
2. Sage raises a *`SyntaxError`* if you forget to include the colon at the end of an `if` or `else` statement.
3. Boolean logic is based on the three values *True*, *False*, and *Sometimes*.
4. *True* **and** (**not** (*False* **or** *True*))
5. *True* **or** (**not** (*False* **and** *True*))
6. *True* **and** (**not** (*False* **xor** *True*))
7. *True* **or** (**not** (*False* **xor** *True*))
8. You should always use the inclusive **or** because discrimination is always and everywhere bad.
9. If an algorithm can take three possible courses of action, the only way to implement it is with the following structure:

```
if condition1:
    block1
else:
    if condition2:
        block2
    else:
        block3
```

10. A `try`/`except` catch is less efficient than an `if`/`else` guard, but this penalty is worth it for the sake of readability.

Multiple choice.

1. The best control structure for a *general decision* uses which keyword(s)?

   A. `try`/`except`
   B. `def`
   C. `if`/`elif`/`else`
   D. `for`/`break`
2. A Boolean statement in Sage can have which values?
   A. *`On`*, *`Off`*
   B. *`true`*, *`false`*
   C. *`True`*, *`False`*
   D. `1`, `-1`
3. What does it mean to "nest" control structures?
   A. To place one control structure (like an `if` statement) inside a different control structure (like a `for` statement).
   B. To place one control structure (like an `if` statement) inside the same control structure (another `if` statement).
   C. To program in such a way that control structures aren't necessary.
   D. To build two control structures a nice, comfortable home where they can raise a family of control structures.
4. What indicates the commands to be executed when the condition of an `if` statement is true?
   A. a colon at the end of the line
   B. indentation of the commands
   C. braces `{` and `}`
   D. both a colon and indentation
5. Which keyword indicates that a loop should terminate early because it is effectively finished?
   A. `abort`
   B. `break`
   C. `exit`
   D. `return`
6. Which symbol indicates to Sage that two expressions are *not* equal?
   A. `=/=`
   B. `!=`
   C. `#`
   D. None; you have to use `not (a == b)`
7. Taking the square root of a negative number produces what kind of exceptions?
   A. *`ZeroDivisionError`*
   B. *`TypeError`*
   C. *`ValueError`*
   D. Why should it produce an exception?
8. The best control structure to watch for *a specific error* uses which keyword(s)?
   A. `try`/`except`
   B. `def`
   C. `if`/`elif`/`else`
   D. `for`/`break`
9. Which of the following is not performed in a `try`…`except` statement?
   A. Sage executes every statement indented under the `try` statement that does not raise an error.

B. If no error occurs in the statements listed under the `try` block, Sage skips the statements indented under the `except` block.
C. If an error occurs in the `try` block and the next `except` statement lists that error, or none at all, Sage executes every statement indented under the `except` statement that does not raise an error.
D. If an error occurs in the statements listed under the `except` block and the next `except` statement lists that error, Sage executes every statement indented under the next `except` statement that does not raise an error.

10. Traditionally, what is meant by the "pyramid of doom?"
A. a situation where we have to nest **try** statements very deeply
B. a situation where an **if** has many, many **else if**'s
C. a situation where we have to nest **if** statements very deeply
D. yet another sequel in the Indiana Jones series

11. A good reason to use parentheses when a condition has more than one Boolean operator is that:
A. …it clarifies the condition's logic to anyone who reads it.
B. …it avoids subtle bugs due to the order Sage evaluates `and`, `or`, and `not`.
C. all of the above
D. none of the above

Short answer.

1. Compare and contrast the appropriate times to use `if`/`elif`/`else` as opposed to the appropriate times to use `try`/`except`/`raise`.
2. How would you modify the *Method_of_Bisection* pseudocode to test that $f(a)$ and $f(b)$ have different signs before starting the loop, and returning (a, b) if they do not?
3. If someone were to modify the implementation of `method_of_bisection()` to `raise` an exception in case $f(a)$ and $f(b)$ were the same sign:
   (a) What kind of exception should they use, *`TypeError`*, *`ValueError`*, or something else altogether?
   (b) What kind of structure should they use, `if`/`else` or `try`/`except`?
4. Write a function `abmatrix()` that accepts as input a positive integer n and two objects a and b, then returns the $n \times n$ matrix C where C is the matrix below.

$$\overbrace{\begin{pmatrix} a & b & a & b & \\ b & a & b & a & \cdots \\ a & b & a & b & \\ b & a & b & a & \\ & \vdots & & & \ddots \end{pmatrix}}^{n \text{ columns}} \left.\vphantom{\begin{pmatrix} a & b & a & b & \\ b & a & b & a & \cdots \\ a & b & a & b & \\ b & a & b & a & \\ & \vdots & & & \ddots \end{pmatrix}}\right\} n \text{ rows}$$

Since a and b might be symbolic, you should create this matrix over the symbolic ring.
   (a) Compute $\det C$.
   (b) Why does that result make sense? *Hint:* Think about linear independence, or rather, the lack thereof.

Programming.

1. Write a function that computes the largest element in a list. Sage has a built-in `max` function, but use a loop here.
2. Write Sage functions that:
   (a) returns the largest element of a matrix;
   (b) returns the smallest element of a matrix;
   (c) returns the sum of all the elements of a matrix.
3. Write pseudocode for a Sage function that accepts as input a mathematical function f and a real number a, then returns a tuple with two values: whether f is increasing at $x = a$ (*True* if so) and whether f is concave up at $x = a$ (*True* if so).
4. Write a function `lp()` which accepts as arguments two lists `L` and `M`, verifies that they are the same length, and if so, returns a list `N` with the same number of elements, where `N[i]` is the product of `L[i]` and `M[i]`. If the two lists are not the same length, the function shoud `raise` an `AttributeError`. For instance, the result of `LP([1,3,5],[2,4,6])` would be `[2,12,30]`, while the invocation `LP([1,3],[2,4,6])` would raise the error.
5. When you ask Sage to solve a trigonometric function, it returns only one solution. For instance, the solution to $\sin 2x = 1/2$ is actually $\{\pi/12 + \pi k\} \cup \{5\pi/12 + \pi k\}$. Write a function to add periodic solutions to a trigonometric equation of the form $\sin(ax + b) = 0$.
   Hint: Part of your program will need to solve $ax + b = 0$. You can obtain this information using the `.operands()` method. To see how it works, assign `f(x) = sin(2*x + 3)` and then type `f.operands()`.
6. In this problem, you'll write an interactive function that approximates definite integrals in one of three ways.
   (a) Adapt the *Left_Riemann_approximation* pseudocode to write pseudocode for Riemann approximation using right endpoints. Translate that into Sage code *without* using comprehensions. Verify that your code produces accurate approximations.
   (b) Adapt the *Left_Riemann_approximation* pseudocode to write pseudocode for Riemann approximation using midpoints. Translate that into Sage code *without* using comprehensions. Verify that your code produces accurate approximations.
   (c) Write an interactive function with the following interface objects:
      - an input box for a function $f(x)$;
      - an input box for a left endpoint a;
      - an input box for a right endpoint b;
      - a slider for a number n of approximations, with minimum value 10, maximum value 200, and step size 10; and finally,
      - a selector with options "left approximation," "right approximation," and "midpoint approximation."

   The body of the function will call the Sage function you wrote for the requested approximation and return its value.
7. Typically, a user of the Method of Bisection would not know in advance how many steps she wants the algorithm's main loop to perform. What she has in mind is the decimal precision needed between the endpoints: for example, they should be the same up to the thousandths place. One way to implement this is by replacing the input n with a positive integer d which specifies the number of digits. We can then compute n from d in the algorithm's body.
   (a) Modify the *Method_of_Bisection* pseudocode so that it accepts d instead of n and computes n as its very first step.

Hint: In general, we round $\log_{10} n$ to the next-highest integer find the number of digits in n. ($\log_{10} 2 \approx 0 \mapsto 1$, $\log_{10} 9 \approx 1 \mapsto 1$, $\log_{10} 10 = 1 \mapsto 1$, …) So if the algorithm divided by 10, you could repeat it d times to obtain a precision of d digits. Alas! the algorithm divides by 2, so you have to use a $\log_2$. *How precisely* you have to use it, we leave as a task to you, but you will also have to consider $b - a$, not just d.

(b) Implement the pseudocode in Sage code.

8. Write Sage functions that accept as input a matrix M and return `True` if M satisfies one of these conditions, and `False` otherwise:
   - `is_square()`: M is square
   - `is_zero_one_negative()`: every entry of M is 0, 1, or -1
   - `sums_to_one()`: every row and every column of M sums to 1

   Then write a Sage function that accepts as input a matrix M and returns `True` if all three conditions are satisfied, and `False` otherwise. Make this function modular, so that it invokes the previous three functions rather than repeating their code!

Repeating yourself indefinitely

Earlier we spoke of repeating oneself "definitely," which occurs when we know exactly how many times we have to repeat a task. Our examples for this involved Euler's Method to approximate the solution to a differential equation, Riemann sums to approximate an integral, and checking whether the elements of a finite ring actually form a field. Later we used the same ideas to find the roots of a continuous function via the Method of Bisection and to modify matrices.

Not all loops are definite; sometimes you can't practically know at the outset how many times you have to repeat a task. One such example is **Newton's Method**. This can be faster than the Method of Bisection, but it uses tangent lines, so it requires the function to be differentiable, not just continuous.

(What is the difference between a continuous function and a differentiable function, you ask? We can think of the difference in an geometric way: a continuous function's graph is "unbroken," while a differentiable function's graph is "smooth." A smooth function is necessarily unbroken, while an unbroken function might not be smooth. In fact, differentiable functions are always continuous, while continuous functions are not always smooth. So the Method of Bisection works everywhere that Newton's Method works, though not, perhaps, as quickly; while Newton's Method does not work everywhere the Method of Bisection works. For that matter, Newton's Method might not work everywhere it works,[77] depending on how careless you are with the setup.)

The idea of Newton's Method is based on two facts.

- The line tangent to a curve travels in the same direction as a curve.
- It is easy to find the root of a line: $mx + b = 0$ implies $x = -b/m$.

Put them together, and the main insight is that if you start close to the root of a function, then follow the tangent line down to *its* root, you shouldn't end up too far from the root you started from — and, hopefully, you'll be closer than when you started. You can repeat this process as long as necessary until you achieve the desired accuracy in digits. See Figure 22.

How does we decide that we have arrived at the desired accuracy? The usual trick is round each approximation to the desired number of digits, and stop only when we twice have the same value. As you can imagine, it is not obvious how we might determine in advance the number of steps necessary to do this. It is easier simply to test, after each iteration, whether the previous approximation and the current approximation agree to the specified number of digits. Up to this point, however, we have no way of doing this.

[77] That may seem like a curious combination of words, but it's the unvarnished truth, as we will show shortly. If you think it doesn't make sense, give it a few paragraphs and you'll see. (Modulo some artistic license in the meanings of the words, of course.)

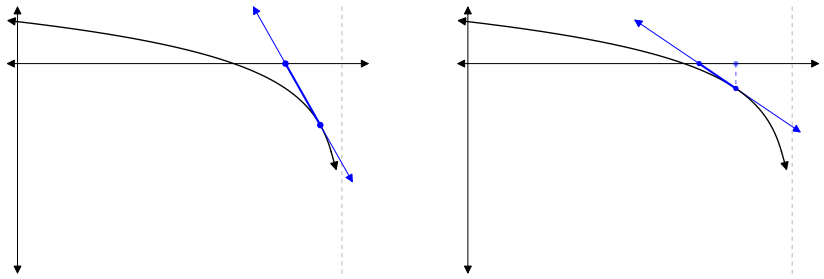


FIGURE 22. The basic idea behind Newton's Method: start near a root, follow the tangent line, and end up at a point (hopefully) closer to the root. Repeat as long as desired.

Implementing an indefinite loop

When you don't know how many times to repeat, but can apply a condition like the one above that simplifies to *True* or *False*, we opt for what is called a **while** loop. Pseudocode for a **while** loop has this form:

while *condition*

and is followed by an indented list of commands. The idea is that the algorithm executes *each* command whenever *condition* is true at the beginning of the loop; the algorithm does not test *condition* after each command, then quit in the middle of the loop if *condition* suddenly becomes true. This translates directly into Sage code: use the `while` keyword, followed by a Boolean condition and a colon,

`while` *condition*`:`

followed by an indented list of commands. If you choose, you can use a `break` command in a `while` loop, just as with a `for` loop, though we don't recommend it.

To implement Newton's Method, we need to do is track two values: the current approximation of the root, say a, and the next approximation to the root, say b. The user specifies a, and we compute b. As long as a and b disagree in at least one of the specified number d of digits, we continue the loop.

A small complication arises here. A **while** loop tests for equality at the beginning of the loop, rather than at the end. However, we compute b *inside* the loop, so we can't know its value at the beginning unless we perform an extra computation before the loop! That would waste space in the program, and make it a little harder to read.

Pseudocode. To make sure the algorithm approximates at least once, we take advantage of the problem's nature and create an **obviously wrong value** for b. In this case, it would be extremely odd indeed if someone were to ask us to round to the ones digit, so we use $b = a + 2$ instead.[78] In this case, it would be extremely odd indeed if a and b agree up to d decimal digits.

To describe the idea in pseudocode, we make use of a third variable, c, as a temporary placeholder for the most recent approximation to the root. The use of a placeholder allows us to

[78] Another, surer way to do this is to set $b = a + 10^{-(d-1)}$, but we don't feel like explaining this beyond saying that if someone is odd enough to specify $d = 0$, then this would give us $b = a + 10^1$, and if someone were to say something more reasonable like $d = 3$, then $b = a + 10^{-2}$, so that $b - a = 10^{-2} > 10^{-3} = d$. And so forth. Beyond that, we leave it to the reader to figure out why $10^{-(d-1)}$ is such a good idea — unless the instructor should assign the corresponding exercise…

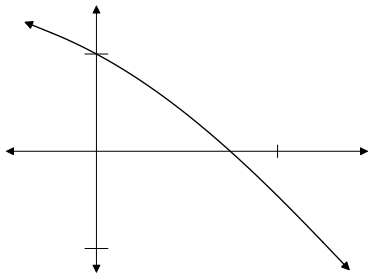
organize the computation in such a way that we can "slide" from older to newer values on each pass through the loop.

algorithm *Newtons_method*
inputs
- f, a function that is differentiable around a
- a, an approximation to a root of f
- d, the number of digits to approximate a root of f near a

outputs
- a root of f, correct to d decimal digits

do
  let $b = a + 2$, $c = a$
  while the first d decimal digits of a and b differ
    let $a = c$
    let b be the root of the line tangent to f at $x = a$
    let $c = b$
  return a

Let's see how this works on the same example we used with the Method of Bisection, $f(x) = \cos x - x$. Both x and $\cos x$ are differentiable, so their difference is, too; we can proceed. Inspection of the graph shows us that a root should occur somewhere between $x = 0$ and $x = 1$; since $x = 1$ looks closer, we'll start the algorithm with $a = 1$ and $d = 3$.



The algorithm initializes b to 3 and c to 1 before entering the loop.

- At the beginning of the first pass through the loop, the algorithm verifies that the first three a and b differ. It assigns $a = 1$, then b the root of the line tangent to $\cos x - x$. To find this, we need the equation of the tangent line; that requires a point, which we have at[79] $(1, \cos 1 - 1) \approx (1, -0.459698)$, and a slope, which we don't. The slope of the tangent line is the value of the derivative, so we compute

$$f'(x) = -\sin x - 1$$

  and evaluate $f'(1) \approx -1.84147$. Our tangent line is thus $y + 0.459698 = -1.84147(x - 1)$. To find the root, set $y = 0$ and solve for x, obtaining the new approximation $b = c = 0.7503638678$.
- At the beginning of the second pass through the loop, the algorithm verifies that the first three $a = 1$ and $b = 0.75038678$ differ. It assigns $a = 0.75038678$, then b the

[79] Well, *approximately* at. We have to resort to floating-point values because in this case the exact values get ugly fast, and slow down computation immensely.

root of the line tangent to $\cos x - x$. We already know the derivative $f'(x)$, and evaluate $f'(0.75038678) \approx -1.6736325442$. Our tangent line is thus $y + 0.0000464559 = -1.6736325442(x - 0.75038678)$. (Notice that the y-value 4.6×10^{-5} is already very, very close to 0.) To find the root, set $y = 0$ and solve for x, obtaining the new approximation $b = c = 0.7391128909$.

- At the beginning of the third pass, the algorithm verifies that $a = 0.75038678$ and $b = 0.7391128909$ differ in their first 3 decimal digits. It assigns $a = 0.7391128909$, then b the root of the line tangent to $\cos x - x$. Passing over the details, we have the new approximation $b = c = 0.7390851334$.
- At the beginning of the fourth pass, the algorithm notices that the first three decimal digits of $a = 0.7391128909$ and $b = 0.7390851334$. The **while** loop terminates, and there is nothing for the algorithm to do but return $a \approx 0.7391128909$.

Notice how quickly Newton's Method reached its goal! How does it compare with the Method of Bisection? Recall that the Method of Bisection halves the interval on each step, so depending on the choice of a and b, we would have needed to run n steps such that $2^{-n}(b-a) < 10^{-3}$, or $n > -\log_2 {}^{10^{-3}}/_{b-a} = 3\log_2 10 + \log_2(b-a) \approx 10 + \log_2(b-a)$. The best guess we can make at (a, b) by eyeballing the graph is probably $({}^1/_2, {}^3/_4)$, so $\log_2(b-a) = \log_2 {}^1/_2 = -1$. Hence the Method of Bisection needs $n > 9$ to approximate the root out to the thousandths place. Using Newton's Method required less than one-third as much time.

Sage code. Sage doesn't have a command that tests immediately whether two numbers differ in their first d decimal digits, so we have to figure out how to do this on our own. This isn't too hard, however; recall that the `round()` command will round a number to the specified number of digits. So, all we need to do is round a and b to three digits, then check whether they are equal.

We also have to tell Sage to solve the equation of the line tangent to the curve. Back on p. 1 we discussed how to construct the equation of a tangent line, so we don't repeat that here. The `solve()` command will give us the root, but remember that `solve()` returns a list of solutions in the form of equations, so we have to extract the solution using `[0]` to obtain the first element of the list, and the `.rhs()` method to obtain the right-hand side of the equation.

These considerations suggest the following Sage code:

```
sage: def newtons_method(f, a, d, x=x):
          f(x) = f
          df(x) = diff(f, x)
          b, c = a + 2, a
          # loop until desired precision found
          while round(a, d) != round(b, d):
              a = c
              m = df(a)
              # find root of tangent line
              sol = solve(m*(x - a) + f(a), x)
              b = sol[0].rhs()
              c = b
          return a
```

How does this work on the example we specified earlier?

```
sage: newtons_method(cos(x) - x, 1, 3)
((cos(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)) +
sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)))*sin(1)
+ cos(1)*sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1)
+ 1)) + cos(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) +
1)))/((sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)) + 1)*sin(1) +
sin(cos(1)/(sin(1) + 1) + sin(1)/(sin(1) + 1)) + 1)
```

Oops! That illustrates what we meant by needing to approximate our answers. let's modify the code so that the assignment of `b` inside the loop changes to

```
sage:         b = round(sol[0].rhs(), d+2)
```

Rounding to $d+2$ places should guarantee that we don't lose any accuracy in the first d places, and it also ensures that we have an approximation, instead of the complicated symbolic expression we came up with earlier. We'll also using a floating-point first approximation, `1.`.

```
sage: newtons_method(cos(x) - x, 1., 3)
0.73911
```

The relationship between definite and indefinite loops. It turns out that definite loops are really just a special kind of indefinite loop, and any **for** loop can be implemented as a **while** loop. In Sage, for instance, we can pass through a list `L` using a `while` loop as follows:

```
sage: i = 0
sage: while i < len(L):
          # do something with L[i]
          i = i + 1
```

Unindexed collections like sets are only a little harder. Suppose `S` is a set; we can `copy()` it, `pop()` elements out of it, and work with them. When we're done, `S` is still there.

```
sage: T = copy(S)
sage: while not len(T) == 0:
          t = T.pop()
          # do something with t
```

Nevertheless, it is useful from the perspective of someone reading a program to distinguish when a loop is definite from when it is indefinite. In the second case, for instance, it is quite possible that the lines indented beneath `while not len(T) == 0:` modify `T`. In this case, the loop is *not* definite; even a `while` loop of this sort can be definite only if the collection is not modified.[80]

[80] Technically, one could modify `T` even in a `for` loop, but the meaning of the word makes it less likely a person would do that. It is routine, however, to perform a `while` loop over a collection that is modified during the loop.

What could possibly go wrong?

We said that **for** loops were "definite" because they iterate over a well-specified finite collection. This guarantees their termination. We have seen that **while** loops work differently: they continue as long as a condition is true, so in fact they might never end. For a very easy example (don't try this at home, kids!):

```
sage: while True:
          a = 1
```

In this loop, the condition is specified to be `True`. It will never change to anything else. Nothing in the loop changes the fact that `True` is `True`, and the **while** loop will never end. This is a good way to "hang" the computer, and if you were naughty enough to ignore our caution (really, don't try this at home, kids!) then you will need to stop Sage by either clicking the Stop button (cloud), selecting the Action menu, then clicking Interrupt (other server), or holding `Ctrl` and press `C` (command line).

This is not the only place it can show up; carefully prepared conditions can also go awry. Newton's Method can lead to an infinite loop when we do the following (one last time: don't try this at home, kids!):

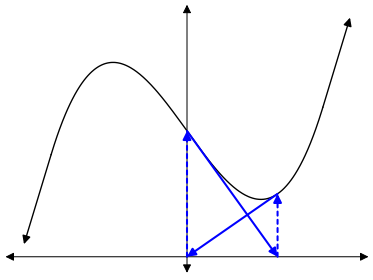
```
sage: f(x) = x**3 - 2*x + 2
sage: newtons_method(f(x), 0, 3)
```

In this case, the derivative is $f'(x) = 3x^2 - 2$. The line tangent to f at $x = 0$ is $y = -2x + 2$; its root is $x = 1$. The line tangent to f at $x = 1$ is $y = -(x-1) + 1$; its root is $x = 0$. We are back where we began… oops!

The reader who looks at this example more carefully may object that it is unrealistic, in that the choice of $x = 0$ seems obviously unwise, being somewhat distant from a root at

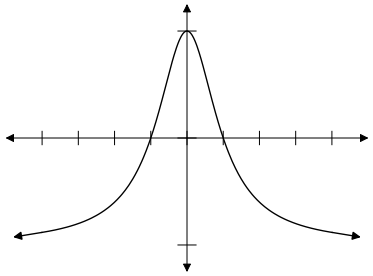
$$-\frac{\sqrt[3]{81}\left(2\cdot\sqrt[3]{81}+3\sqrt[3]{9-\sqrt{57}}\right)}{27\sqrt[3]{9-\sqrt{57}}} \approx -1.7693.$$

It is even more obvious from the graph, which illustrate the oscillation of approximations between $x = 0$ and $x = 1$:



While this is true, it is also the case that starting at $x = 0.5$ will also lead you into this very oscillation; it just takes more steps before the algorithm starts whiplashing between 0 and 1.

Going astray. An example of a *theoretical* infinite loop occurs with $f(x) = 1/1+x^2$.



Suppose we start Newton's method at $x = 3$, which is not too far from the root at $x = 1$. In this case, Sage suddenly quits with the error,

```
sage: newtons_method(f(x), 3., 3)
IndexError:  list index out of range
```

If you look at the additional information provided by Sage, you will see it complaining about the line `b = round(sol[0].rhs(), d+2)`. Basically, *IndexError* means that an entry doesn't exist in the list at the specified index. The index appears between brackets: `[0]` in this case, so Sage can't find an entry in `sol`. This suggests there was no solution.

What happened? If you trace the computations by hand, or else place a `print(a)` at the beginning of the `while` loop, you will see the approximations thrash wildly between negative and positive values of increasingly large size:

$$3,\ -3.66667,\ 8.58926,\ -149.80063,\ 840240.29866,\ -1.48303 \times 10^{17},\ 8.15439 \times 10^{50}.$$

This error is due to the fact that we're working with approximations, and the numbers get so large that the computer decides there is no solution and `solve()` gives up, returning an empty list. Try tracing the tangent lines on the plot to see how this happens.

Again, the reader might object that $x = 3$ does not seem like an especially compelling point to start Newton's Method. True, but $x = 0.09$ is less worrisome, and that will do something similar.

A similar phenomenon can occur in Newton's Method when you aim for one root, but end up at another. This isn't a text on Numerical Analysis, so we won't go into those details, or some other difficulties with Newton's Method; we encourage the interested reader to pursue a text on that.

Division of Gaussian integers

We turn to an interesting ring called the **Gaussian integers**, written $\mathbb{Z}[i]$ for short. Gaussian integers have the form $a+bi$, where a and b are integers. This differs from the complex numbers, which have the superficially similar form $a + bi$, with the difference that for complex numbers a and b are real numbers. In short,

- $2+3i$ is both complex and a Gaussian integer, while
- $2+i/3$ and $2+i\sqrt{3}$ are complex, but not Gaussian integers.

Addition, subtraction, and multiplication of Gaussian integers is identical to complex numbers:

$$\begin{aligned}(a+bi)+(c+di)&=(a+c)+(b+d)i\\(a+bi)-(c+di)&=(a-c)+(b-d)i\\(a+bi)(c+di)&=(ac-bd)+(ad+bc)i\,.\end{aligned}$$

For division, however, we do not want to go from Gaussian integers to complex numbers, as Sage might give us by default:

```
sage: (2 + 3*I) / (1 - I)
5/2*I - 1/2
```

Rather, we would like a way to compute a quotient and remainder, much as Sage offers the `//` and `%` operators for integers. In particular, we'd like an algorithm that behaves much like division of integers:

THE DIVISION THEOREM FOR INTEGERS. *Given an integer n, called the **dividend**, and a nonzero integer d, called the **divisor**, we can find an integer q, called the **quotient**, and a nonnegative integer r, called the **remainder**, such that*

$$n=qd+r \qquad \textit{and} \qquad r<|d|\,.$$

Remember that the absolute value of a real number gives us an idea of its size; the analogue for complex numbers is the norm $||z||$. Translated to Gaussian integers, the theorem would become

THE DIVISION THEOREM FOR GAUSSIAN INTEGERS. *Given a Gaussian integer z, called the **dividend**, and a nonzero Gaussian integer d, called the **divisor**, we can find a Gaussian integer q, called the **quotient**, and another Gaussian integer r, called the **remainder**, such that*

$$z=qd+r \qquad \textit{and} \qquad ||r||<||d||.$$

This theorem says nothing about the remainder being nonnegative because "negative" doesn't really make sense for Gaussian integers.

The Division Theorem for Gaussian Integers is in fact true, and we will prove it by constructing an algorithm that does what it claims. But how should we construct such an algorithm? Division of integers is repeated subtraction, and you can create a perfectly good division algorithm in the following way:

algorithm *divide_integers*
inputs
- $n, d \in \mathbb{Z}$ such that $d \neq 0$

outputs
- $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $r < |d|$

do
  let $r = |n|$, $q = 0$
  if n and d have the same sign
    let $s = 1$
  else
    let $s = -1$
  while $r < 0$ **or** $r \geq |d|$
    add s to q
    replace r by $r - sd$
  return q, r

This algorithm uses s to "step" the quotient in the correct direction towards n. For instance, if we divide -17 by 5, then r and q have the following values at the beginning of each loop (q decreases because we have $s = -1$):

r	-17	-12	-7	-2	3
q	0	-1	-2	-3	-4

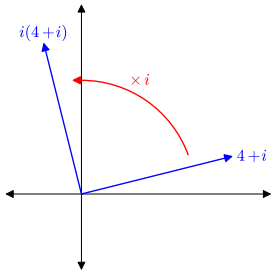
Once $r = 3$ and $q = -4$, the algorithm notices that $r < |d|$, so the **while** loop ends, giving us the correct expression

$$17 = qd + r = (-4) \times 5 + 3.$$

It makes sense that division of Gaussian integers should work the same way. Our general strategy, then, will take this approach:

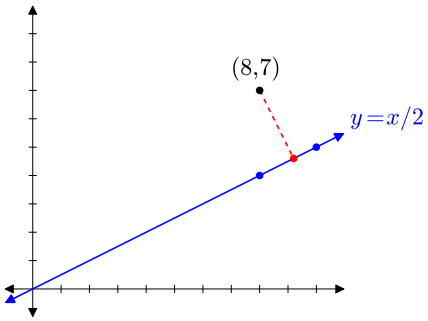
let $r = z$, $q = 0$
while $||r|| \geq ||d||$
  "step q up" using an appropriate value s
  replace r by $r - sd$
return q, r

Now we have to figure out what it means to "step q up." The difficulty lies in the fact that we can "increase" q in two different directions: the real direction and the imaginary direction. As you saw in the introduction to the complex plane on p. 77, multiplying a complex number by i rotates it 90° counterclockwise:



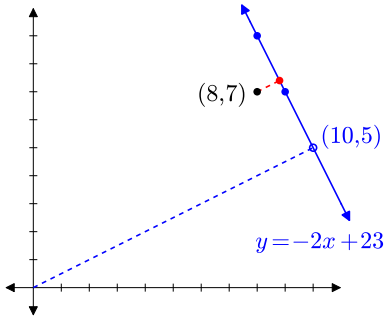
It is, therefore, possible to "step q up" two different ways.

We'll solve this using a two-stage approach: first we minimize the distance when multiplying by integers; then we minimize the distance when multiplying by integer multiples of i. For instance, suppose we divide $8+7i$ by $2+i$. The divisor lies along the line of slope $1/2$, so we "step" along that line in the appropriate direction until any further step would increase the distance. From geometry, we know this means getting as close to the altitude dropped from $(8,7)$ to the line $y = x/2$:



Unfortunately, the *closest* point is $(9\frac{1}{5}, 4\frac{3}{5})$ (in red), which corresponds to a complex number, but not a Gaussian integer. The closest Gaussian integer is the blue point that lies beyond the red point; the only way to check this is to compare the norm of each point as we move along the line.

The next stage is to step from our minimal Gaussian integer along a line perpendicular to the blue one, corresponding to repeated multiplication of d by i. Again, we step along the line until any further step would increase the norm.



The closest point on the line *with integer values* is $(9,7)$, which corresponds to $9+7i$. We obtained this value after multiplying first by 5, then by i:

$$(2+i)(5+i) = 10+2i+5i-1 = 9+7i.$$

The remainder is -1, whose norm is smaller than the divisor's.

Writing this as pseudocode gives us the following.

algorithm *divide_gaussian_integers*
inputs
- $z, d \in \mathbb{Z}[i]$ such that $d \neq 0$

outputs
- $q, r \in \mathbb{Z}[i]$ such that $z = qd + r$ and $||r|| < ||d||$

do
  let $r = z$, $q = 0$
  if $||r - d|| < ||r||$
    let $s = 1$
  else
    let $s = -1$
  while $||r - sd|| < ||r||$
    add s to q
    replace r by $r - sd$
  if $||r - id|| < ||r||$
    let $s = i$
  else
    let $s = -i$
  while $||r - sd|| < ||r||$
    add s to q
    replace r by $r - sd$
  return q, r

As we planned, this algorithm uses an **if** statement to check first whether it is better to step forward or backward by a real-valued factor. It then selects s to be 1 or -1 depending on which is more appropriate. The **while** loop then steps in the appropriate direction, decreasing the norm until further steps would increase it. We cannot rely on the test $||r|| \geq ||d||$ as with integers, because we may need further minimization after the first stage because we still have $||r|| > ||d||$. This is why the loop checks $||r - sd|| < ||r||$ instead; roughly speaking, this translates as "stepping further reduces the norm of r."

The algorithm then uses a second **if** statement to check whether it is better to step forward or backward by an imaginary-valued factor. It selects s to be i or $-i$ depending on which is more appropriate. The **while** loop then steps in the appropriate direction, decreasing the norm until further steps would increase it. We could in fact test $||r|| \geq ||d||$ here, but for the sake of consistency we used the same basic structure.

This translates to the Sage code in Figure 23. How does it perform on some examples?

```
sage: divide_gaussian_integers(4 + I, 1 - I)
(2*I + 1, 1)
sage: divide_gaussian_integers(8 + 7*I, 2 + I)
(5 + I, -1)
```

```
sage: def divide_gaussian_integers(z, d):
          r, q = z, 0
          # which real way to step?
          if norm(r - d) < norm(r):
              s = 1
          else:
              s = -1
          # loop to step
          while norm(r - s*d) < norm(r):
              q = q + s
              r = r - s*d
          # which imaginary way to step?
          if norm(r - I*d) < norm(r):
              s = I
          else:
              s = -I
          # loop to step
          while norm(r - s*d) < norm(r):
              q = q + s
              r = r - s*d
          return q, r
```

FIGURE 23. Sage code for division of Gaussian integers

We already verified the second computation by stepped through it earlier; the first is not hard to do yourself.

An well-orderly retreat. With Newton's method, we encountered an infinite loop with unfortunate combinations of a function f and initial guess a. Is it possible to find two Gaussian integers such that the `while` loops of `divide_gaussian_integers()` to become infinite?

Not in this case, no. The difference is that the loops are built in such a way that we can describe the condition using a nonnegative integer that decreases after every successful pass through the loop. In both loops, that value is $||z - qd||$. Why? The algorithm initializes $r = z$, then "steps" q closer to z if and only if that would decrease the distance between z and qd; in other words, $||z - qd||$. The norm is always an integer, and positive integers enjoy a property called the **well-ordering property**:

Every nonempty subset of $\mathbb{N}$ *has a least element.*

If the loop were infinite, we could consider all the values of $||z - qd||$, having a subset of $\mathbb{N}$. The algorithm performs the loop if this distance decreases, and the well-ordering property guarantees that this can happen only finitely many times, contradicting the supposition that the loop is infinite.

If possible, it is a very good idea when working with **while** loops to formulate them in such a way that you exploit the well-ordering property somehow, thus guaranteeing termination. You can do this in pretty much every circumstance, though the consequence may be undesirable. One way to do this with Newton's Method, for instance, is to set a variable `max_iterations` to

some large number, decrease it by one on every pass through the loop, then rephrase the `while` loop's condition so that it quits the moment `max_iterations` hits `0`. You might even choose to `raise` an exception. We conclude this section with a modified `terminating_newtons_method()` that illustrates this technique.

```
sage: def terminating_newtons_method(f, a, d, x=x):
          f(x) = f
          df(x) = diff(f, x)
          # a + 2 should be too far for premature termination
          b, c = a + 2, a
          max_iterations = 100
          # loop for desired precision, up to threshhold
          while max_iterations > 0 and round(a, d) != round(b, d):
              a = c
              m = df(a)
              sol = solve(m*(x - a) + f(a), x)
              b = sol[0].rhs()
              c = b
              max_iterations = max_iterations - 1
          # error?  report if so
          if max_iterations == 0:
              raise ValueError, 'Too many iterations:  '
                  + 'try a different initial value'
          return a
```

Try it with the nonterminating example from before ($f = x^3 - 2x + 2$, $a = 0$) and see how this behaves much better.

Exercises

True/False. If the statement is false, replace it with a true statement.

1. In Sage, the `while` keyword implements an indefinite loop.
2. A `while` loop stops as soon as the condition becomes false, even if it hasn't completed all the statements indented beneath it.
3. The smartest way to loop through a collection is using a `while` loop.
4. An indefinite loop is a special kind of definite loop.
5. It is impossible to create an infinite loop when using an indefinite loop.
6. It is impossible to prevent *every* indefinite loop from becoming an infinite loop.
7. Newton's Method finds every root the Method of Bisection will find.
8. Newton's Method finds some roots the Method of Bisection will not find.
9. Gaussian division is more complicated than complex division because it requires a quotient *and* a remainder.
10. The well-ordering property states that every subset of the natural numbers $\mathbb{N}$ has a least element.

Multiple choice.

1. A loop that iterates over a specific collection is a:

   A. definite loop
   B. indefinite loop
   C. infinite loop
   D. specific loop
2. A loop that repeats a specific number of times is a:
   A. definite loop
   B. indefinite loop
   C. infinite loop
   D. specific loop
3. A loop that repeats based on whether a condition remains true or false is a:
   A. definite loop
   B. indefinite loop
   C. infinite loop
   D. specific loop
4. The main reason to use an indefinite loop instead of a definite loop is:
   A. An indefinite loop is easier to guarantee termination.
   B. An indefinite loop is harder to guarantee termination.
   C. To decide whether to terminate, the loop must test a Boolean condition.
   D. Definite loops are just a special kind of indefinite loop, so it's more consistent to use indefinite loops exclusively.
5. Which of the following best characterizes the idea behind Newton's Method?
   A. Use the tangent line to find a new approximation, hopefully closer to the correct root.
   B. Use the tangent line to find a new approximation, definitely closer to the correct root.
   C. Bisect the interval repeatedly until the endpoints agree on the first d decimal digits.
   D. Trace along the curve with extreme precision, zooming and moving back and forth until you have the correct result.
6. For which of the following functions would you be better advised to use the Method of Bisection to find a root, rather than Newton's Method?
   A. $|x+3|$
   B. $\sin(x+\pi)$
   C. $(x-3)^2$
   D. $\ln(x-3)$
7. Under what conditions can Newton's Method fail to find a root?
   A. the function is discontinuous
   B. the function is non-differentiable
   C. the starting point leads away from the root
   D. all of the above
8. Which of the following is a correct geometric interpretation of what happens when you multiply a complex number by $-i$? (*Note that it is negative!*)
   A. The new number corresponds to a 90° clockwise rotation in the complex plane.
   B. The new number corresponds to a 90° counter-clockwise rotation in the complex plane.
   C. The new number corresponds to a 180° rotation in the complex plane.
   D. The new number is in the same direction as the original number, only longer.
9. For which of the following pairs of numbers should Gaussian division fail? (dividend comes first, divisor second)
   A. $4+i$, $1+i$

B. $0, 1+i$
C. $1+i, 0$
D. none of the above

10. The reason we raised a `ValueError` in the last version of Newton's Method is that:
A. We consider the function was a bad value, because it is not continuous at a.
B. We consider the starting point a a bad value, because the function is not differentiable there.
C. The values have grown too large, and Sage was unable to find a solution.
D. We consider the starting point a bad value, because it seems caught in an infinite loop.

Short answer.

1. Consider the following Sage function:

```
sage: def aw(n):
          t = 0
          i = 1
          while i <= n:
              t = t + i**2
              i = i + 1
          return t
```

(a) Compute `aw(4)` by hand, showing your work.
(b) Rewrite `aw(n)` as a one-line mathematical expression, using summation notation.
(c) Translate `aw(n)` into a new function `af(n)`, which does the same thing, but uses a `for` loop instead.
(d) Both the `while` loop and the `for` loop are inefficient. How would it be smarter to implement this function?

2. The Syracuse sequence is defined in the following manner. Let $S_1 \in \mathbb{N}$, and

$$S_{i+1} = \begin{cases} S_i/2, & S_i \text{ is even;} \\ 3S_i + 1, & S_i \text{ is odd.} \end{cases}$$

The sequence terminates once $S_i = 1$. It is believed that the sequence terminates with $S_i = 1$ for every value of S_1, but no one actually knows this. Compute by hand the sequence for several different values of S_1, and notice how it always seems to reach $S_i = 1$.

3. Division of univariate polynomials f by g works in the following way (here, $\deg(r)$ means "degree of r" and $\mathrm{lc}(r)$ means "leading coefficient of r"):
- let $r = f, q = 0$
- **while** $\deg(r) > \deg(g)$
  - let $t = x^{\deg(r)-\deg(g)}$, $c = \mathrm{lc}(r)/\mathrm{lc}(g)$
  - replace r by $r - ctg$
  - add ct to q

(a) Choose 5 pairs of polynomials, and perform the division algorithm on them. You can use Sage to help speed the subtractions, but if you use Sage's Division command you won't be able to answer at least one of the next few problems.
(b) Do you think this algorithm exploits the well-ordering property? Why or why not?
(c) Is there a pattern to how many times the algorithm passes through the loop?

(d) Could the loop be written as a **for** loop? If so, how? If not, why not?

4. Show how these `for` loops can be rewritten using a `while` loop in each case:
   (a)
   ```
   for k in range(num):
       <body> # possibly depending on k
   ```
   (b)
   ```
   for elem in L: # L is a collection (list/tuple/set)
       <body>     # depending on elem
   ```
5. Can every `while` loop be rewritten into a `for` loop? Explain!

Programming.

1. In Short Answer #1, you worked with the Syracuse sequence. Write pseudocode for how you might implement this as a program. It should accept an argument for S_1, sets $s = S_1$, apply the formula to s to find its next value, and terminate when $s = 1$. Implement your pseudocode. (You can test whether s is even using Sage's `%` operator, as a number is even when the remainder after division by 2 is `0`.)
2. When we implemented the Method of Bisection, we used a logarithm to determine the required number of steps. This was the only way we were able to use a definite loop. It is possible to rephrase the algorithm with an indefinite loop, similar to Newton's Method: terminate once a and b agree to the specified number of digits.
   (a) Do this. Rewrite both the pseudocode and then the Sage code to see this in action.
   (b) Which method do you prefer, and why?
   (c) Which method do you think is easier for someone else to read, understand, and/or modify? Again, why?
3. In Short Answer #2, you worked on division of univariate polynomials. Write full pseudocode for it — that is, give the algorithm a name, specify the inputs precisely, and the outputs precisely. Then implement it as a Sage program, and test it on the same examples you did in that problem.
4. A positive integer n is **prime** if no integer from 2 to $\sqrt{n}$ divides n.
   (a) Write a Sage function that accepts a value of n as input and finds the smallest nontrivial positive divisor of n.[81] If the number is prime, the function should `raise` a `ValueError`.
   (b) Write another Sage function that accepts a positive integer k as input and returns a list of every prime number less than or equal to k. Have the function invoke the answer to part (a).
   (c) The Goldbach conjecture asserts that every even number is the sum of two prime numbers. Despite having been verified up to unimaginably ginormous values, no one has yet proved the Goldbach conjectire is true for all positive integers. Write a function that accepts a positive integer m as input, raises a `ValueError` if m is not even; otherwise, it finds two prime numbers that sum to m. Have the function invoke the answer to part (b).
5. The greatest common divisor of two integers is the largest integer that divides both of them. An algorithm to compute the greatest common divisor was already known to Euclid of Alexandria more than 2000 years ago:

[81] A positive divisor is nontrivial if it is not 1.

algorithm *gcd*
inputs
- $a, b \in \mathbb{Z}$

outputs
- the greatest common divisor of a and b

do
  let $m = |a|$, $n = |b|$
  while $n \neq 0$
    let $d = n$
    replace n by the remainder of division of m by n
    replace m by d
  return m

(a) Implement this as Sage code, and check that it gives correct greatest common divisors for several large numbers.
(b) Choose a small pair of nonzero integers and trace through the algorithm, step-by-step, showing how it arrives at the gcd.

6. The Extended Euclidean Algorithm not only finds the gcd of two nonzero integers a and b, it finds numbers c and d such that

$$\gcd(a, b) = ac + bd.$$

For instance, $\gcd(4, 6) = -1 \times 4 + 1 \times 6$. We can describe it as follows:

algorithm *extended_euclidean_algorithm*
inputs
- $a, b \in \mathbb{Z}$

outputs
- $c, d \in \mathbb{Z}$ such that $\gcd(a, b) = ac + bd$

do
  let $m = |a|$, $n = |b|$
  let $s = 0$, $c = 1$, $t = 1$, $d = 0$
  while $n \neq 0$
    let q be the quotient of dividing m by n
    let r be the remainder of dividing m by n
    let $m = n$ and $n = r$
    let $w = c$, then $c = s$ and $s = w - qs$
    let $w = d$, then $d = t$ and $t = w - qt$
  return c, d

Implement this as Sage code, and verify that it gives the correct values of c and d for several large values of a and b; that is, verify that the values c and d it returns satisfy $ac + bd = \gcd(a, b)$.

Repeating yourself inductively

Another form of repetition is called *recursion*. The term literally means, "running again," because the idea is that an algorithm solves a problem by breaking it into simpler cases, invoking itself on those smaller cases, then putting the results back together. Those smaller cases typically break their cases into even smaller cases. This may sound a bit dodgy: What's to stop us from hitting an infinite chain?

When organizing a recursion, we try to set it up in such a way that this strategy creates a chain of problems that lead to a smallest possible element. If we can relate them to the natural numbers, then even better, as the natural numbers enjoy a property called the **well-ordering property**:

Every nonempty subset of $\mathbb{N}$ *has a least element.*

(If this sounds familiar, it's because we referred to it in a previous chapter.)

Recursion

A classical example. A number of problems translate naturally to recursion. One example is Pascal's triangle, which looks like so:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
\iddots & & & & & \vdots & & & & & \ddots
\end{array}
$$

The rows of this triangle appear in many places, such as the expansion of binomials:

$$
\begin{array}{lcccccccccccc}
(x+1)^0 & = & & & & & & & 1 & & & & \\
(x+1)^1 & = & & & & & & 1x & + & 1 & & & \\
(x+1)^2 & = & & & & 1x^2 & + & 2x & + & 1 & & & \\
(x+1)^3 & = & & & 1x^3 & + & 3x^2 & + & 3x & + & 1 & & \\
(x+1)^4 & = & & 1x^4 & + & 4x^3 & + & 6x^2 & + & 4x & + & 1 & \\
\vdots & & \iddots & & & & & & & & & & \ddots
\end{array}
$$

Take a moment to try and deduce the triangle's pattern.

Do you see it? if not, take another moment to try and work it out.

Hopefully you saw that each row is defined in the following way. Let the top row be row 1, the next row down row 2, and so forth.

- The first and last elements on *each* row are both 1.

- The remaining elements satisfy $q_j = p_{j-1} + p_j$, where
  - q_j is the element in entry j on row i, and
  - p_{j-1} and p_j are the elements in entries $j-1$ and j on row $i-1$.

To know the entries in row i, then, we first need to know the entries on row $i-1$. We can loop through those entries to give us the subsequent entries. We know how many entries there are in that row: $i-1$, though we can also inquire from the row itself.[82] so we can apply a definite loop. This gives us the following, recursive pseudocode to compute row i:

algorithm *pascals_row*
inputs
- $i \in \mathbb{N}$, the desired row of Pascal's triangle

outputs
- the sequence of numbers in row i of Pascal's triangle

do
```
if i = 1
    result = [1]
else if i = 2
    result = [1,1]
else
    prev = pascals_row(i − 1)
    result = [1]
    for j ∈ (2, 3, . . . , i − 1)
        append prev_{j−1} + prev_j to result
    append 1 to result
return result
```

This translates to the following Sage code:

[82] In the pseudocode, we could have written "$|prev|$" to indicate this. In the Sage code, we could simply write `prev.len()`.

```
sage: def pascals_row(i):
          if i == 1:
              result = [1]
          elif i == 2:
              result = [1, 1]
          else:
              # compute previous row first
              prev = pascals_row(i - 1)
              # this row starts with 1...
              result = [1]
              # ...adds two above next in this row...
              for j in xrange(1, i - 1):
                  result.append(prev[j-1] + prev[j])
              # ...  and ends with 1
              result.append(1)
          return result
```

We can verify it on the lines we saw above, and generate more that you can verify by hand:

```
sage: pascals_row(1)
[1]
sage: pascals_row(5)
[1, 4, 6, 4, 1]
sage: pascals_row(10)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

An example so classical it's medieval. The next example is one of the fundamental problems of mathematics,

How quickly will a population of bunnies grow?

You have to ask?!?

You think we're joking, but we're not.[83] As you might guess, these bunnies are no ordinary bunnies; they obey some special laws of reproduction:

- In the beginning, there was one pair — but they were immature.
- Every pair of bunnies takes one season to mature.
- Once mature, every pair of bunnies produces a new pair of bunnies every season.
- The bunnies are immortal. They have no predators, and they obtain their energy from the sun, so they never run out of food. Amazingly, they never tire of the new children waking them up at night, crawling all over them, and generally getting in the way of nobler pursuits, such as the contemplation of the form of the ideal bunny.[84]

The formula for the number of pairs of bunnies in one season is defined by the number in the previous two seasons:

$$b_n = b_{n-1} + b_{n-2}.$$

This is a perfect example of recursion. It is natural to try and implement this recursively with a function that accepts an integer for the number of seasons that have passed. If it is 1 or 2, the answer is easy: 1. Otherwise, we compute the number of bunnies in the previous two seasons.

algorithm *funny_bunnies*
inputs
- $n \in \mathbb{N}$, the number of seasons

outputs
- the number of bunny couples after n seasons

do
  if $n = 1$ **or** $n = 2$
    let *result* $= 1$
  else
    let *result* $=$ *funny_bunnies*$(n-1)$ + *funny_bunnies*$(n-2)$
  return *result*

To translate this to Sage code, keep in mind that `xrange()` does *not* iterate on the last value, so we need its limit to be `n-1`:

```
sage: def funny_bunnies(n):
          if n == 1 or n == 2:
              result = 1
          else:
              result = funny_bunnies(n-1) + funny_bunnies(n-2)
          return result
sage: [funny_bunnies(i) for i in xrange(2,10)]
[2, 3, 5, 8, 13, 21, 34]
```

[83] At least one journal is dedicated to these bunnies.
[84] Gratuitous (and irrelevant) reference to Platonic philosophy included at no extra charge.

That's a very interesting list of numbers. It is so interesting that it's named after a mathematician who described it during the Middle Ages, Leonardo da Pisa, better known as Fibonacci,[85] so people call it the **Fibonacci sequence**. We can describe it as follows:[86]

$$F_{\text{next}} = F_{\text{total}} + F_{\text{mature}}.$$

That means, "the number of bunnies next season is the sum of the total number of bunnies and the number of mature bunnies." We could also say that the number of total bunnies is the "current" number of bunnies, while the number of mature bunnies is the number of bunnies we had last season. That means "the number of bunnies next season is the sum of the number of bunnies this season and the number of bunnies last season." So we could write:

$$F_{\text{next}} = F_{\text{curr}} + F_{\text{prev}}.$$

An example you can't actually program. Another example is the proof technique called *induction*. The principle of induction is that:

- if you can prove:
  - a property applies to 1, and
  - whenever it applies to n, it also applies to $n+1$,
- then the property applies to all the positive integers.

The idea is that if we can show the first bullet is true, then the second bullet is true for any positive integer m because

- if it's true for $m-1$, then it's true for m; and
- if it's true for $m-2$, then it's true for $m-1$, and

  …
- if it's true for 1, then it's true for 2.

[85] Fibonacci lived in a period known as the High Middle Ages, and didn't care about bunnies so much as the fact that people were still using I, V, X, L, C, D, and M to write numbers and do arithmetic — Roman numerals, that is. He realized this was very inefficient, and it would be better to use the Hindu-Arabic way of writing numbers: 0, 1, 2, …, 9 and all that. Trouble was, most people were accustomed to working with a old-timey calculator called an *abacus*, and had been trained how to use an abacus with Roman numerals. Then, as now, people were reluctant to give up a computational crutch.

To get around this, he wrote a book called *Liber Abaci* ("Book of the Abacus"). The book's main point was to show how you could solve all the same problems with Hindu-Arabic numbers *without* using an abacus, and work far more efficiently, to boot. Getting work done faster allowed one to do more business. The Platonic bunnies were one of the examples he used in the book. It was quite possibly the first time someone used a funny bunny to make money.

Oh, we're not done making you groan yet, not by a long shot. You may think *Liber Abaci* kind of a dull title — certainly not as catchy as the slogan, "Think Different," though Fibonacci would have had stronger grounds to use that slogan than anyone who popularized it today — but people back then were interested more in substance than in style, so Fibonacci succeeded both in weaning people off Roman numerals and in getting his name attached to a sequence of numbers that, ironically, had been studied by Indian (Hindu) mathematicians long before Leo was himself a Fibonacci number.

All this goes to show that the High Middle Ages suffered a critical shortage of 21st century "brights" who could inform them that they were, in fact, supposed to be ignorant poo-flingers disinterested in science and mathematics.

(And, yes, we do recognize the irony in referring to an abacus as a "computational crutch" in a text dedicated to doing math problems with the Sage computer algebra system, but this footnote is designed to be rich in irony.)

[86] F stands for "funny," not "Fibonacci." Why do you ask?

We know it's true for 1 (that's in the first bullet), so the chain of implications tells us it's true for m, which was an arbitrary positive integer.

Alternatives to recursion

Recursion is fairly common, and reasonably easy to implement, but it may not always the best approach. It's not hard at all to compute Pascal's triangle even to the 900th row, and you might be able to take it further if Sage did not place a limit on the number of times you can apply recursion:

```
sage: P = pascals_row(900)
sage: len(P)
900
sage: P[-3]
403651
sage: P = pascals_row(1000)
RuntimeError:  maximum recursion depth exceeded while calling a
Python object
```

This limit is there for good reason, by the way; recursion can use up quite a bit of a special sort of memory,[87] and too deep a recursion can also indicate an infinite loop, or at least that something has gone terribly awry.

With the Fibonacci sequence, things go awry rather quickly, but in a different way. Try computing the nth Fibonacci number for really-not-very-large-at-all values of n, and you'll see the length the problems take a sharp turn upwards quickly. On the authors' machines, for instance, there is already a noticeable lag at $n = 30$; $n = 35$ takes several seconds; and for $n = 40$ you might just be able to make yourself a cup of coffee[88] before it finished. You might think you could get around this delay with a faster computer, and you'd be wrong, very wrong. The problem is that there are too many branches at each step. To see this clearly, let's expand a computation of the 7th Fibonacci number down to one of the two base cases:

$$\begin{aligned}
F_7 &= F_6 + F_5 \\
&= (F_5 + F_4) + (F_4 + F_3) \\
&= [(F_4 + F_3) + (F_3 + F_2)] + [(F_3 + F_2) + (F_2 + F_1)] \\
&= [[(F_3 + F_2) + (F_2 + F_1)] + (F_2 + F_1) + F_2] + [[(F_2 + F_1) + F_2] + (F_2 + F_1)] \\
&= [[[(F_2 + F_1) + F_2] + (F_2 + F_1)] + (F_2 + F_1) + F_2] \\
&\qquad + [[(F_2 + F_1) + F_2] + (F_2 + F_1)]
\end{aligned}$$

Twelve F's are *not* the base cases F_1 and F_2; each of those requires a recursion. That's an awful lot, but it doesn't sound that bad. On the other hand, consider the next number,

$$F_8 = F_7 + F_6.$$

We've already counted 12 recursions for F_7, and you can look up to see that F_6 requires 7. So F_8 will want $12 + 7 + 1 = 20$ recursions. (We need 1 extra for the expansion of F_7 itself.) You can see

[87] For those who know some computer science, we're talking about the stack.

[88] Whether you'd want to drink it is another story, but before the authors get into another argument on caffeinated drinks, we'll stop there.

immediately that the number of recursions of the Fibonacci sequence grows in a Fibonacci-like fashion:

$$0, 0, 1, 2, 4, 7, 12, 20, 33, 54, \ldots$$

Already F_{10} requires $54 \approx 5 \times 10$ recursions; notice the jump in size from F_7, which required "only" $12 \approx 2 \times 7$. With the Fibonacci sequence, the number of recursions is growing too fast!

Computing the 100th Fibonacci number might, therefore, seem out of reach, but there are several ways around this conundrum. The first two approaches to solving this problem rely on the fact that when computing F_7, we computed F_5 twice:

$$F_7 = F_6 + F_5 = (\mathbf{F_5} + F_4) + \mathbf{F_5}.$$

We then computed F_4 three times:

$$F_7 = [(\mathbf{F_4} + F_3) + \mathbf{F_4}] + (\mathbf{F_4} + F_3).$$

…and so forth. So the problem isn't that the Fibonacci *inherently* requires an unreasonable amount of recursion; it's that we're wasting a lot of computations that we could reuse.

Caching. One way to avoid wasting computations is to "remember" them in what's called a **cache**. You can think of it this way: we *stash* previous results in the *cache*. To implement the cache, we use a **global** variable; that is, a variable that exists outside a function, which we can access from inside it. The reason for this is that if the cache were merely local to the function, then the same values wouldn't be available when we tackle the subproblem.

We describe two ways to cache results. One of them is to do so explicitly, with a list whose ith value is the ith Fibonacci number. We initialize the list as $F = [1, 1]$. Whenever we compute a Fibonacci number, we first check whether F already has the value. If it does, we return that value rather than recurse. If it does not, we compute it, then add it to F at the end. For instance, when computing F_5:

- We need to compute F_4 and F_3. We have neither, so we recurse.
  - For F_4, we need to compute F_3 and F_2. We don't have F_3, so we recurse.
    - For F_3, we need to compute F_2 and F_1. We have both, so we compute $F_3 = F_2 + F_1 = 1 + 1 = 2$ and store that at the end of the list F, which is now $[1, 1, 2]$. We then return it
    - For F_2, we already have it, so we return $F_2 = 1$.
    - We now have F_3 and F_2 , so we compute $F_4 = F_3 + F_2 = 2 + 1 = 3$, store that value at the end of the list, and return it.
  - For F_3, we *now* have it in the list (it was computed above) so we simply return it.
- We now have F_4 and F_3, so we compute $F_5 = F_4 + F_3 = 3 + 2 = 5$, store that value at the end of the list, and return it.

This saved us the trouble of computing F_4 more than once, and of computing F_3 more than once. What's more, the definition of the Fibonacci numbers is convenient to this method: when it comes time to store F_n, we already have F_1, F_2, …, F_{n-1} in the list, so we can store it in the right place.[89] To write pseudocode for this, we introduce a new "bold word" for our pseudocode: **global**. This heads a list of global variables used by the algorithm; for clarity, we list it after **inputs** and **outputs**.

[89] Not all recursive sequences are this lucky. If the sequence were defined by $S_n = S_{n-2} + S_{n-3}$, for instance, you'd have to make sure the cache had the right length before storing it. So we're a little lucky with the Fibonacci sequence.

algorithm *cached_fibonacci*
inputs
- $n \in \mathbb{N}$ with $n \geq 1$

outputs
- F_n, the nth Fibonacci number

globals
- F, a list of Fibonacci numbers (in order)

do
  if $n \leq |F|$
    let $result = F_n$
  else
    let $result = cached\_fibonacci(n-1) + cached\_fibonacci(n-2)$
    append $result$ to F
  return $result$

To turn this into pseudocode, we use the Sage keyword `global`, which indicates that `F` is a variable that lives outside the function. This isn't strictly necessary for Sage, but it helps alert readers that a global variable is expected.

```
sage: F = [1,1]
sage: def cached_fibonacci(n):
          global F # cache of previous values
          # if already computed, return it
          if n <= len(F):
              result = F[n-1]
          else:
              # need to recurse:  bummer
              result = cached_fibonacci(n-1) \
                       + cached_fibonacci(n-2)
              F.append(result)
          return result
```

Notice that we had to adjust one item from the pseudocode: instead of returning `F[n]`, we returned `F[n-1]`. We did this because the pseudocode assumes that lists start with index 1, but Sage starts its indices at index `0`.

You can test and compare the results between the two. It will become evident very quickly that `cached_fibonacci()` is much, *much* faster than `funny_bunnies()` once n grows to anything larger than the dinkiest of numbers:

```
sage: cached_fibonacci(10)
55
sage: cached_fibonacci(25)
75025
sage: cached_fibonacci(35)
9227465
```

To *really* drive the point home, we'll compute one that would take far too long with the recursive implementation:

```
sage: cached_fibonacci(100)
354224848179261915075
```

Don't overlook the first line of the code above, where we set `F = [1,1]`. If you forget that, you'll encounter an error. If you were a *good* reader and did not make that mistake, we will simulate the error by resetting `F`, then running `cached_fibonacci`:

```
sage: reset('F')
sage: cached_fibonacci(10)
NameError:  global name 'F' is not defined
```

If you were still a good reader and reset `F` to see what would happen, and now you want to run it again, just assign it anew:

```
sage: F = [1,1]
sage: cached_fibonacci(10)
55
```

Two last words to the wise. While this algorithm is cached, it is *still recursive;* we haven't changed the fact that, when we encounter a "cache miss" (n is larger than the size of F), the algorithm has to call itself. Since Sage has a built-in limit for recursion, we still encounter the same problem as we did with *pascals_row*:

```
sage: cached_fibonacci(1000)
RuntimeError:  maximum recursion depth exceeded while calling a
Python object
```

You can actually change this, but it's dangerous, so we won't tell you how. There are usually better options, anyway — to find one, see the next section.

Finally, you don't usually have to implement caching on your own. In the cloud, Sage will give you a cache for free; just type `@cached_function` and then start the function definition on the line below.

```
sage: @cached_function
      def decorated_fibonacci(n):
          if n == 1 or n == 2:
              result = 1
          else:
              result = decorated_fibonacci(n-1) \
                       + decorated_fibonacci(n-2)
          return result
sage: decorated_fibonacci(100)
354224848179261915075
```

Notice that we obtained the same result as with our home-grown cache.

Turning recursion into a loop. Another way to avoid wasting computations is to reformulate the recursion as a loop. Let b_{mat} count the pairs of mature bunnies, let b_{tot} count the total pairs of bunnies, and let b_{next} count the pairs of bunnies we can expect next season. Since the bunnies are immortal, b_{next} will count all b_{tot} of the bunnies we have now, but since each mature pair produces a new pair, b_{next} must also count all b_{mat} mature pairs twice to account for next season's new pairs. So

$$b_{\text{next}} = b_{\text{tot}} + b_{\text{mat}}.$$

Once the next season comes, b_{tot} will be the number of mature bunnies that are now producing pairs, while b_{next} will be the new bunny total, so we could replace b_{mat} by b_{tot} and b_{tot} by f_{next} to compute the following season's total. And so forth. Since we know the first two season's values already, we never need to count those. Keeping in mind that $n = 1$ is the first season and $n = 2$ the second, we need the loop to calculate b_{tot} for $n = 3$, $n = 4$, … That suggests the following pseudocode.

algorithm *immortal_bunnies*
inputs
- $n \in \mathbb{N}$, the number of seasons

outputs
- the number of "immortal" bunnies after n seasons

do
  let $b_{\text{mat}} = b_{\text{tot}} = 1$
  repeat $n - 2$ times
    let $b_{\text{next}} = b_{\text{tot}} + b_{\text{mat}}$
    let $b_{\text{mat}} = b_{\text{tot}}$
    let $b_{\text{tot}} = b_{\text{next}}$
  return b_{tot}

To translate this to Sage code, keep in mind that `xrange()` does *not* iterate on the last value, so we need its limit to be `n-1`. It is also more "natural" in Sage to start counting at $n = 0$, so the loop will differ slightly from the pseudocode:

```
sage: def immortal_bunnies(n):
          b_mat = b_tot = 1
          # bottom-up loop
          for each in xrange(n-1):
              b_next = b_tot + b_mat
              b_mat, b_tot = b_tot, b_next
          return b_tot
sage: [immortal_bunnies(i) for i in xrange(2,10)]
[2, 3, 5, 8, 13, 21, 34, 55]
```

Unlike caching, Sage does not provide an automatic method that turns a recursive function into a looped function. Doing this is something we have to work out on our own, reasoning from the logic of the problem. As is considered an advanced topic in computer science, we do not dwell on it further. The interested reader can find further information in computer science texts that discuss "dynamic programming." [90]

Eigenvalues and eigenvectors resolve a bunny dilemma. You may not be familiar with eigenvalues and eigenvectors, even if you've seen them in a class. We conclude this chapter with an illustration of how immensely useful they can be.

As you have probably noticed, even the non-recursive implementations of the Fibonacci sequence rely on its recursive description, whose recursive nature requires us to know some previous values. It would be nice to have a formula that didn't depend on previous values, at least not explicitly. Such a formula is called a closed form of a sequence.

Another way to describe this is as a matrix equation. The matrices

$$\begin{pmatrix} F_{\text{curr}} \\ F_{\text{prev}} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} F_{\text{next}} \\ F_{\text{curr}} \end{pmatrix}$$

represent the information we need both this season and next season to compute the following season's bunnies. Now, the sum of the numbers in the first matrix gives us the top element of the second matrix, while the top number of the first matrix gives us the bottom number of the second. We can describe this relationship using a matrix equation:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{\text{curr}} \\ F_{\text{prev}} \end{pmatrix} = \begin{pmatrix} F_{\text{next}} \\ F_{\text{curr}} \end{pmatrix}.$$

What's more, multiplying by powers of the matrix

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

[90] The inventor of dynamic programming once said that he invented the term because his company answered to a man who harbored "a pathological fear and hatred of the word research. ...You can imagine how he felt, then, about the term mathematical." Using a term like "dynamic programming" helped him steer clear of his supervisor's wrath. He added, "I thought dynamic programming was a good name... not even a Congressman could object to."

ratchets the numbers higher up the list. To wit,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^5 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 13 \\ 8 \end{pmatrix},$$

which you will recognize as the 4th and 5th numbers in the list our program produced, which are F_6 and F_7, by which we mean the number of bunnies in the 6th and 7th seasons. In general,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix},$$

since after all

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix},$$

and so forth.

If we can find a simple formula for powers of the matrix, we should be able to use it to find our desired closed form. As it happens, a very convenient result from linear algebra comes in handy here. An **eigenvector e** of a matrix M over a ring R is a vector related to an eigenvalue $\lambda \in R$ such that

$$M\mathbf{e} = \lambda\mathbf{e};$$

that is, M changes the "length" of the vector. Eigenvectors and their eigenvalues have a number of uses, one of which now comes to our aid.

EIGENDECOMPOSITION THEOREM.. *Let M be an $n \times n$ matrix with linearly independent eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$ and corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$. We can rewrite M as $Q\Lambda Q^{-1}$ where*

$$Q = (\mathbf{e}_1|\mathbf{e}_2|\cdots|\mathbf{e}_n) \quad \textit{and} \quad \Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}.$$

(The notation for Q means that the ith column of Q is the ith eigenvector of M.)

It's easy to verify this in Sage for M. Let's compute the eigenvectors first.

```
sage: evecs = M.eigenvectors_right()
sage: evecs
[(-0.618033988749895?, [(1, -1.618033988749895?)], 1),
 (1.618033988749895?, [(1, 0.618033988749895?)], 1)]
```

What does all this mean? If you study the help text…

```
sage: M.eigenvectors_right?
```

…you will see that the list holds one tuple per eigenvector, and the tuple itself contains the eigenvalue, the eigenvector, and the eigenvector's multiplicity.

For our purposes, the eigenvalues and eigenvectors suffice. It is relatively easy to extract them, but

- we want exact values, not approximate ones, and
- what do those question marks mean, anyway?

It may surprise you to learn that the question marks do *not* indicate uncertainty. They indicate that the numbers lie in a special field called the field of algebraic numbers. Sage denotes this field with the symbol `AA`.

```
sage: a, b = evecs[0][0], evecs[1][0]
sage: type(a)
<class 'sage.rings.qqbar.AlgebraicNumber'>
```

The algebraic numbers are the smallest field containing all the roots of polynomials with integer coefficients, so we can always find a "nice-looking" polynomial that has them as a root. In addition, can often rewrite them as an expression in radicals. The following methods to algebraic numbers perform these tasks:

a`.minpoly()`	returns the polynomial of smallest degree that has *a* as a root
a`.radical_expression()`	returns a radical expression equivalent to *a*

You can find many more methods in the usual manner, by typing the name of a variable whose value is an algebraic number, following it with a period, then pressing the `Tab` key.

For now, let's try these on our present eigenvector.

```
sage: a.minpoly()
x^2 - x - 1
sage: a.radical_expression()
-1/2*sqrt(5) + 1/2
sage: b.minpoly()
x^2 - x - 1
sage: b.radical_expression()
1/2*sqrt(5) + 1/2
```

Both numbers have the same minimal polynomial,[91] but their radical expressions are different. They aren't *that* different, though; we'll call them[92]

$$\psi = \frac{1-\sqrt{5}}{2} \quad \text{and} \quad \varphi = \frac{1+\sqrt{5}}{2}.$$

[91] This makes sense if you have studied this. If you haven't, watch for a linear algebra class coming soon to a semester near you!

[92] Some trivia. Since φ is a root of x^2-x-1, we know that $\varphi^2-\varphi-1=0$, so $\varphi^2=\varphi+1$, or $\varphi=\varphi+1/\varphi=1+1/\varphi$, or $1/\varphi=\varphi-1$. Notice that $\psi=1-\varphi$, so $-\psi$ and φ are actually reciprocals.

The second one in particular is wildly famous, as it is the **golden ratio**. It appears in many different places in mathematics, and arguably in many "real world" situations. Our Fibonacci bunnies may not seem an especially compelling "real world" situation, but this number suggests that the numbers may show up in "real world" situations, and indeed they do.

We return to our main problem, that of finding a closed form for the Fibonacci sequence. We first wanted to verify the Eigendecomposition Theorem. To build the matrices Q and Λ, we extract the data from `evecs` using the bracket operator `[]`, then build the matrices using comprehensions. Make sure you understand how we build these matrices.

```
sage: Q = matrix([e[1][0] for e in evecs]).transpose()
sage: L = diagonal_matrix([e[0] for e in E])
sage: M == Q*L*Q.inverse()
True
```

So the theorem works. How is it useful to our aim? Remember that

$$M^{n-2}\begin{pmatrix}1\\1\end{pmatrix}=\begin{pmatrix}F_n\\F_{n-1}\end{pmatrix}.$$

By substitution, this becomes

$$M^{n-2}=\left(Q\Lambda Q^{-1}\right)^{n-2}=\underbrace{\left(Q\Lambda Q^{-1}\right)\left(Q\Lambda Q^{-1}\right)\cdots\left(Q\Lambda Q^{-1}\right)}_{n-2\text{ times}}.$$

The associative properties allows us to rewrite this as

$$M^{n-2}=(Q\Lambda)\left(Q^{-1}Q\right)\Lambda\left(Q^{-1}Q\right)\cdots\left(Q^{-1}Q\right)\left(\Lambda Q^{-1}\right).$$

This simplifies to

$$M^{n-2}=Q\Lambda^{n-1}Q^{-1}.$$

What's more, it is "easy" to show that for a diagonal matrix like Λ

$$\Lambda^k=\begin{pmatrix}\lambda_1&&&\\&\lambda_2&&\\&&\ddots&\\&&&\lambda_n\end{pmatrix}^k=\begin{pmatrix}\lambda_1^k&&&\\&\lambda_2^k&&\\&&\ddots&\\&&&\lambda_n^k\end{pmatrix},$$

so we can write Λ^k in a simpler form.

```
sage: Lk = diagonal_matrix([L[i,i]^n for i in xrange(L.nrows())])
TypeError:  no canonical coercion from Symbolic Ring to Rational
Field
```

Well, *that's* odd. What about handing it a radical expression first?

```
sage: Lk = diagonal_matrix([L[i,i].radical_expression()^(n-2) \
             for i in xrange(L.nrows())])
sage: Lk
[(-1/2*sqrt(5) + 1/2)^(n-2)                          0]
[                         0 (1/2*sqrt(5) + 1/2)^(n-2)]
```

That worked. Sage seems to want a radical expression to proceed; we can adapt to that with Q, as well.

```
sage: Q = matrix([[Q[0,0].radical_expression(), \
                   Q[0,1].radical_expression()], \
                  [Q[1,0].radical_expression(), \
                   Q[1,1].radical_expression()] \
      ])
sage: Q
[                  1                   1]
[-1/2*sqrt(5) - 1/2 1/2*sqrt(5) - 1/2]
```

We can now compute $Q\Lambda^k Q^{-1}$ directly.

```
sage: Q*Lk*Q.inverse()*matrix([[1],[1]])
[ 1/20*sqrt(5)*(1/2*sqrt(5) + ...
```

…well, it's a bit of a mess. In fact, all we care about is the top element, and we'd prefer to have it fully simplified.

```
sage: (Q*Lk*Q.inverse()*matrix([[1],[1]]))[0,0].full_simplify()
1/5*sqrt(5)*((1/2*sqrt(5) + 1/2)^n - (-1/2*sqrt(5) + 1/2)^n)
```

Well, *that's* convenient! This is

$$\frac{1}{5}\sqrt{5}\left[\left(\frac{1}{2}\sqrt{5}+\frac{1}{2}\right)^n-\left(-\frac{1}{2}\sqrt{5}+\frac{1}{2}\right)^n\right],$$

so with a slight rewrite we find that

$$F_n=\frac{1}{\sqrt{5}}(\varphi^n-\psi^n),$$

an awfully elegant closed form!

Exercises

True/False. If the statement is false, replace it with a true statement.

1. A recursive algorithm to a problem is one where the algorithm invokes itself on a smaller case of the problem.
2. A recursive algorithm should try to exploit the well-ordering property of the integers ($\mathbb{Z}$).
3. The rows of Pascal's triangle appear in many places, such as in the expansion of trinomials.

4. Because Pascal's triangle has only one recursion, there is no danger of encountering a `RuntimeError`.
5. The growth of a population of bunnies is one of the fundamental properties of mathematics.
6. Every recursive statement can be solved using an easily-implemented function.
7. If a problem can be solved by recursion, then implementing it by recursion is the best approach.
8. A very deep recursion can indicate that something has gone awry, like an infinite loop.
9. You can get around the delay in computing the 40th Fibonacci number by buying a faster computer.
10. A cache is a local variable that stores the function's previous results.
11. Sage provides a `@cached_function` decorator that automatically caches a function's previous results.
12. You can sometimes turn a recursive function into a loop, thereby avoiding penalties associated with recursion.
13. As the golden ratio is unique to the Fibonacci numbers, they are of purely theoretical interest.
14. Eigendecomposition allows us to rewrite a matrix M in terms of other matrices that are themselves related to vectors that M rescales.
15. You can obtain a radical representation of an algebraic number using the `.radical_simplify()` method.

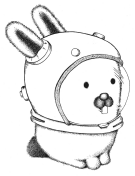
Multiple choice.

1. The well-ordering property applies to which of these sets?
   A. $\mathbb{N}$
   B. $\mathbb{Z}$
   C. $\mathbb{Q}$
   D. $\mathbb{R}$
2. Pascal's triangle is a good example of recursion because:
   A. It has many applications.
   B. Each row can be defined in terms of the previous row.
   C. It is one of the fundamental problems of mathematics.
   D. It makes an effective critique of Descartes' philosophy of systematic doubt.[93]
3. The Fibonacci numbers are a good example of recursion because:
   A. They have many applications.
   B. Each number can be defined in terms of previous numbers.
   C. They relate to one of the fundamental problems of mathematics.
   D. If you can get a sequence of numbers named after you, you have to be right — you just have to!
4. Which proof technique is essentially a kind of recursion?
   A. contradiction
   B. contrapositive
   C. induction
   D. hypothesis testing
5. What happens in Sage if you accidentally program an infinite recursion?
   A. the computer explodes
   B. the program hangs
   C. `RuntimeError`

[93] Gratuitous and irrelevant reference to modern philosophy included at no extra charge.

   D. `RecursionError`
6. Why is it actually a bad idea to implement the Fibonacci sequence with a naïve recursion?
   A. A naïve implementation repeats the vast majority of its computations.
   B. Except for base cases, every recursion doubles the number of computations needed.
   C. The time required to compute the numbers grows about as fast as the numbers themselves.
   D. All of the above.
7. Which of the following alternatives to recursion do you have to work out by analyzing the algorithm, rather than using some combination of Sage commands or variables?
   A. caching
   B. finding a closed form
   C. induction
   D. looping
8. Which of the following best defines the ring of algebraic numbers?
   A. `AA`
   B. the set of all roots of polynomials with integer coefficients
   C. the smallest field containing the roots of polynomials with integer coefficients
   D. the smallest field containing all the irrational numbers
9. The best command or method to turn an algebraic number into an exact, but easy-to-read *numerical* expression is:
   A. `.exactify()`
   B. `.full_simplify()`
   C. `.minpoly()`
   D. `.radical_expression()`
10. Which of the following best describes the meaning of an eigenvector?
   A. a vector that is useful for finding closed forms of sequences
   B. a vector that the matrix rescales, but leaves in the same direction
   C. a vector that is linearly independent of other eigenvectors
   D. a vector that can be used to decompose a matrix into a computationally useful form

Bonus. Given the recursive definition of a sequence of mathematical bunnies, the best way to find a "closed form" to describe the sequence is to
   A. acquire dogs
   B. box them up
   C. fence them in
   D. dress them up
   E. Enclose them in spacesuits, like Glenda:[94]



Short answer.

[94]Downloaded from Bell Labs' Plan 9 website and used with permission (we think — the wording's a tad vague).

1. Why would *pascals_row* not have such trouble with recursion that the Fibonacci sequence did? That is, what difference in the pseudocode (and, therefore, the Sage code) makes *pascals_row* more amenable to recursion than the Fibonacci sequence?
2. Would a cache be as useful for *pascals_row* as it was for the Fibonacci sequence? Why or why not?
3. Back on p. 77 we asked you to plot x, x^2, $\log_{10} x$, and e^x and rank them in order from largest to smallest. Recreate this graph on the interval $[1, 15]$, adding to it a curve whose points are defined by the Fibonacci numbers from 1 to 15. Again, rank these functions according to which grows fastest. *Hint:* Recall that the `line()` function graphs a "curve" defined by a finite set of points. You may need to adjust `ymax` in order to view the plot well.

Programming.

1. Suppose $y = x^m q$, where x does not divide q. For example, if $y = 12$ and $x = 2$, then we have $m = 2$. We call m the **multiplicity** of x, and can compute it the using the following recursive algorithm:

> **algorithm** *multiplicity*
> **inputs**
> - x and y, two objects such that "x divides y" makes sense
>
> **outputs**
> - the number of times x divides y
>
> **do**
> **if** x does not divide y
> $result = 0$
> **else**
> $result = 1 + multiplicity(y/x)$
> **return** *result*

   (a) Implement this as a Sage program. Check that it gives the correct results for the following:
   (i) $y = 12$, $x = 2$
   (ii) $y = 12$, $x = 3$
   (iii) $y = t$, $x = t^4 + t^2$
   (b) It is possible to solve this problem non-recursively, using a **while** loop. Write pseudocode for such an algorithm, then implement it in Sage. Test your program on the same examples.
2. Rewrite *pascals_row* so that it uses a manual cache. Do not use Sage's `@cached_function` capability; rather, use a list where the ith entry is also a list: the ith row of Pascal's triangle.
3. We can count how complicated a function's recursion is with a global variable called `invocations`. We add two lines at the beginning of the recursive function that declare `invocations` to be `global` and then add 1 to it. For instance, we could do this with `funny_bunnies()` as follows:

```
sage: def funny_bunnies(n):
          global invocations
          invocations = invocations + 1
          if n == 1 or n == 2:
              result = 1
          else:
              result = funny_bunnies(n-1) + funny_bunnies(n-2)
          return result
```

Now, every time we run it, we first set `invocations` to 0, then run the program, then print `invocations`. For example:

```
sage: invocations = 0
sage: funny_bunnies(7)
13
sage: invocations
25
```

(This is not the same as the number of recursions.)

(a) Do this for the values $n = 7$, 8, 9, 10, and 11. Would you say that this sequence behaves like the Fibonacci numbers, much as we saw with the number of recursions? If so, find a recursive formula to describe the pattern in the numbers. If not, how would you describe it?

(b) Modify the *pascals_row* program to compute the number of invocations, then run it for a few values of n. Is the number of invocations in this case Fibonacci-like? If so, find a recursive formula to describe the pattern in the numbers. If not, how would you describe it?

4. John von Neumann showed that it is possible to represent every natural number in terms of the symbols {} and ∅:

$$\begin{aligned} 0 &\longleftrightarrow \emptyset \\ 1 &\longleftrightarrow \{\emptyset\} \\ 2 &\longleftrightarrow \{\emptyset, \{\emptyset\}\} \\ 3 &\longleftrightarrow \{\emptyset, \{\emptyset\}, \{\{\emptyset, \{\emptyset\}\}\}\} \\ &\vdots \\ n &\longleftrightarrow \{n-1, \{n-1\}\} \end{aligned}$$

As you can see from the last line especially, this is a recursive definition:

$$n = (n-1) \cup \{n-1\}.$$

Implement this as Sage code.

Hint: This is not hard, but you have to be careful. First, the recursive nature of the algorithm means the function must return a frozen set. To create a frozen set R that contains another frozen set S, *as well as other elements,* first create R as a mutable set (`R = set()`), add S to it, add

the other elements (using `.add()`, `.update()`, and so forth as appropriate), and finally convert `R` to a frozen set.

5. Suppose you have a set of n objects, and you want to list its m-element subsets (that is, its subsets with exactly m elements).[95] The following algorithm would accomplish this for you:

> **algorithm** *combinations*
> **inputs**
> - S, a set of n objects
> - m, the number of objects you'd like to pick from S, with $m \leq n$
>
> **outputs**
> - the subsets of S of size n
>
> **do**
> **if** $|S| = m$
> let *result* $= \{S\}$
> **else**
> let *result* $= \emptyset$
> **for** $s \in S$
> let $U = S \backslash s$
> add *combinations*(U, m) to *result*
> **return** *result*

   (a) Implement this as a Sage program.
   Hint: This is not hard, but you have to be careful. First, the recursive nature of the algorithm means *result* must be a frozen set. However, creating a frozen set that contains a set `S` doesn't work if you use `frozenset(S)`, because that creates a frozen set whose elements are the elements of `S`, in essence *converting* `S` from a mutable set to a frozen set, rather than making a frozen set whose one element is `S`. To create a frozen set whose element is another set, first create a mutable set, add a frozen set to it, then convert the mutable set to a frozen set. Essentially, the one line of the first case of the pseudocode turns into three lines of Sage code.
   (b) Let $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and evaluate the size of *combinations*(S, i) for each $i = 0, \ldots, 8$. The resulting sizes appear earlier in the text. Where?
   (c) Do you think this is a freak accident, or does it hold in general? What makes you say this?

[95] This may seem like the sort of question only the purest of pure mathematicians would care about, but it's actually related to the reason you never seem to buy a winning ticket for PowerBall.

Making your pictures 3-dimensional

This chapter takes a look at the three-dimensional objects and graphs that you might use.

Many of the commands from two dimensions have a 3D extension, with some options carrying over as well. As before, we will not illustrate every option, expecting that options may change in future versions of Sage.

Combining graphs is again done through addition, and the `show()` command also applies to 3D plots. For 3d plots, the `show()` command adds an option that allows you to select a "viewer" or "renderer":

- When using `show()` as a command, such as `show(p, ...)`, use the `renderer` option to select one of `'webgl'` (fastest), `'canvas'` (may work better with transparency), or `'tachyon'`. This latter option gives a static image, whereas `'webgl'` and `'canvas'` are interactive.
- When using `show()` as a method, such as `p.show(...)`, use the `viewer` option to select one of `'jmol'` (requires Java), `'java3d'` (also requires Java), `'canvas3d'` (browser only, uses JavaScript), and again `tachyon'`. Again, all but the last are interactive.

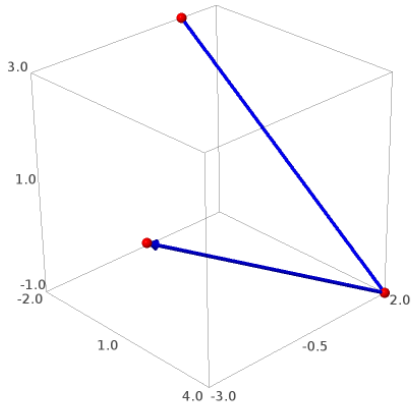
What do we mean by "interactive"? The new feature of 3D plots is that the graphs can manipulated–you can change the point of view! You'll do this by clicking to make it active and then clicking and dragging to change the viewing angle. You can also zoom by scrolling. Other viewing options are available through a menu accessible by right-clicking; we'll let you explore those.

3D objects

"Straight" stuff. Many of the 2D graphing functions that we studied previously have a corresponding function in 3 dimensions: `point()`, `line()`, `arrow()`, `polygon()`. Since all of these require one or more points, Sage will recognize that the points are given in 3D and will graph them appropriately. Many options carry over.

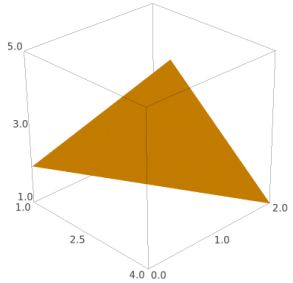
If we give the `point()` command one or more triples (rather than pairs). Lines are again graphed with the `line()` command. In 3D, the `line()` command has an `arrow_head` option; if set to *True*, the arrow head is only shown at the last point.

```
sage: pts = ((-2,1,3),(4,2,-1),(2,-3,1))
sage: p1 = point(pts,color='red',size=20)
sage: p2 = line(pts,color='blue',thickness=7,arrow_head=True)
sage: p1+p2
```



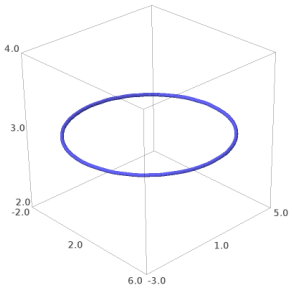
Polygons are graphed in 3D as in 2D with the `polygon()` command, which requires a collection of points. The only options available in 3D are `color` and `opacity`. Here is a triangle:

```
sage: polygon([(1,0,2),(3,1,5),(4,2,1)],color='orange')
```



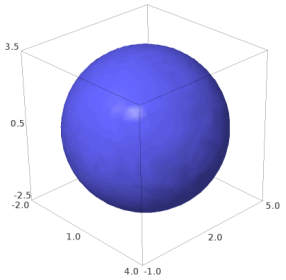
"Curvy" stuff. The circle function can also produce circles in 3 dimensions, again needing a center point and a size. Circles graphed with the `circle()` command will be placed parallel to the xy-plane; if you want other orientations, you'll need to use other methods described later. There are no `edgecolor` or `facecolor` options for circles in 3D. As in 2D, if you want a disk, set the `fill` option to *True*.

```
sage: circle((2,1,3),4,thickness=10)
```



In 3D, the set of all points equidistant from a particular point is a sphere. We can graph spheres using the `sphere()` command, which requires a center and a radius (size).

```
sage: sphere(center=(1,2,.5),size=3)
```



Basic 3D plots

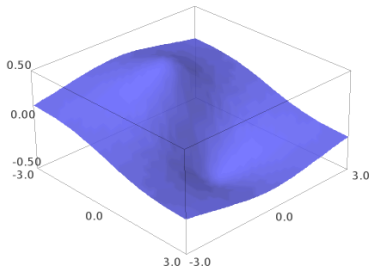
If you have taken multivariable calculus, didn't you wish your handheld calculator could have graphed those surfaces? Thankfully, Sage can! Many of the 2D graphing functions that we studied earlier have a corresponding function in 3 dimensions. There are several ways to represent 3-dimensional relations:

- Cartesian coordinates with either
  - z as a function of x and y;
  - an implicit equation in terms of x, y, and z, but not necessarily a function;
  - with x, y, and z in terms of a third variable t, a parameter;
  - with x, y, and z in terms of two other parameters u and v;
- spherical coordinates; and
- cylindrical coordinates.

Generic Cartesian plots. The most familiar three-dimensional relation defines the value of one variable (often z) in terms of two others (typically x and y). Unlike the 2D `plot()` command, the `plot3d()` command has no option to detect poles and show asymptotes.

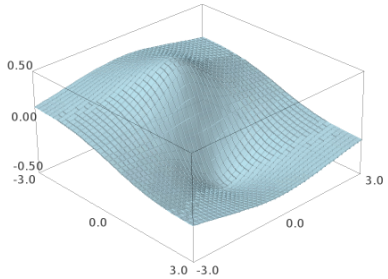
In the next few examples, we graph $f(x,y) = {}^{-x}/_{1+x^2+y^2}$ with two different styles of plots and with various options.

```
sage: f(x,y) = -x/(1+x^2+y^2)
sage: plot3d(f(x,y), (x,-3,3), (y,-3,3), opacity=.8)
```



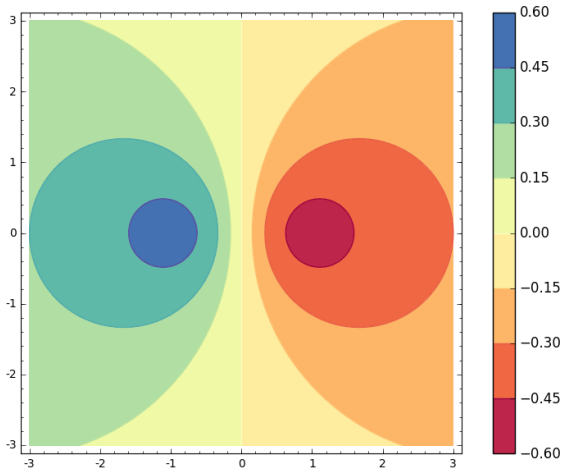
Setting the `mesh` option to *True* produces graphs with an xy-grid on the surface, like those in many textbooks:

```
sage: plot3d(f(x,y), (x,-3,3), (y,-3,3), opacity=.8, \
             color='lightblue',mesh=True)
```



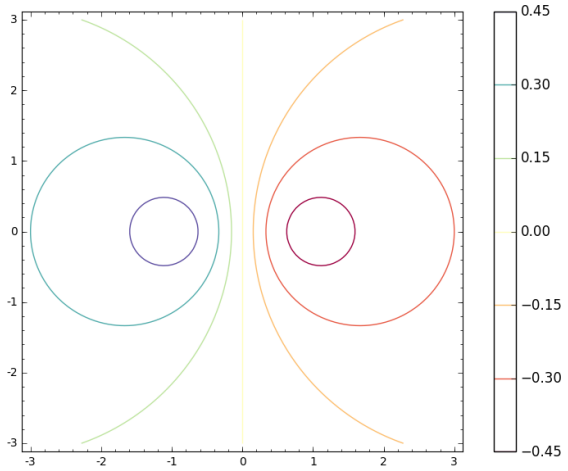
When part of the 3D graph is obscured, it may be easier to view the same surface using a "contour plot." Think of this plot as an aerial view of the graph, with points in the same color having z-values within the same range, as shown on the color bar to the right. With the `'Spectral'` color map used below, the lowest points are in red, and the highest points are in blue. The contour levels (the z-value increments shown on the color bar) were determined automatically, but they may be specified using a list for the `contours` option.

```
sage: contour_plot(f(x,y),(x,-3,3),(y,-3,3), cmap='Spectral', \
      colorbar=True)
```



The boundaries between color regions are called "level curves," meaning that $f(x,y) = k$ for some constant k. If you prefer, to see just the level curves, use the option `fill=False`. (You will notice that many textbook authors use this option.)

```
sage: var('y')
sage: contour_plot(f(x,y), (x,-3,3), (y,-3,3), \
           cmap='Spectral', fill=False, colorbar=True)
```



Tangent planes: combining 3D plots. Let's graph a function $f(x,y)$ and the plane tangent to $f(x,y)$ at a point $P_0 = (x_0, y_0, z_0)$. Recall that the tangent plane depends on the first partial derivatives of f at P_0 and is given by

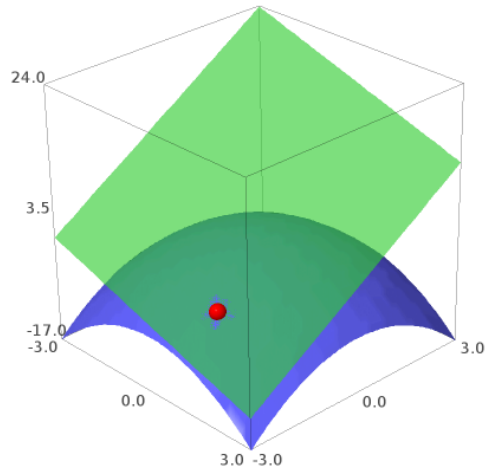
$$z = f_x(x_0,y_0)(x-x_0) + f_y(x_0,y_0)(y-y_0) + z_0,$$

where f_x and f_y are the first partial derivatives. We can easily write a Sage function to compute the tangent plane and graph both $f(x,y)$ and the tangent plane over the rectangular region $[a,b] \times [c,d]$:

```
sage: def plot_f_tanplane(f,pt,a,b,c,d):
          f(x,y)=f
          x0,y0=pt
          fx = diff(f,x)
          fy = diff(f,y)
          z0 = f(x0,y0)
          tanplane = fx(x0,y0)*(x-x0) + fy(x0,y0)*(y-y0) + z0
          p1 = plot3d(f,(x,a,b),(y,c,d))
          p2 = plot3d(tanplane,(x,a,b),(y,c,d), \
              color='limegreen',opacity=.6)
          p3 = point((x0,y0,z0),color='red',size=30)
          return p1 + p2 + p3
```

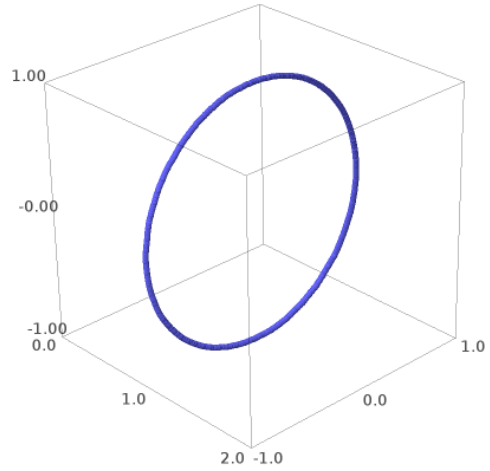
Now we can produce graphs of tangent planes for any function where the partial derivatives are defined. For example,

```
sage: var('y')
sage: plot_f_tanplane(1-x^2-y^2,(1,-2),-3,3,-3,3)
```



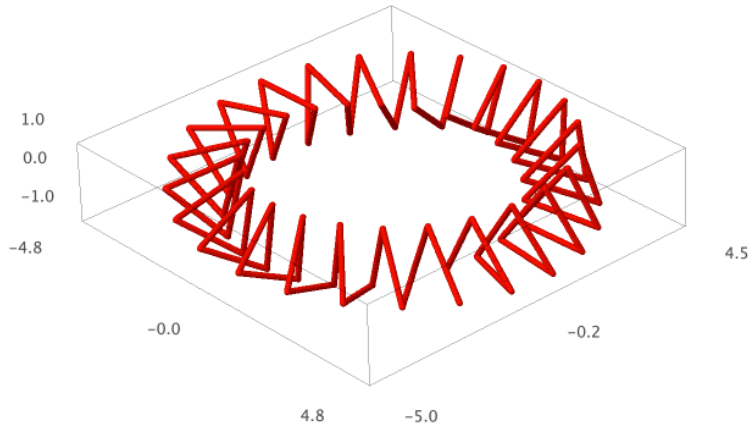
Parameterizations in 3D. Another way to plot three-dimensional objects is through parameterization. Curves are parameterized in a single variable, often called t. We saw earlier that the `circle()` command always plots a 3D circle parallel to the xy-plane. With parameterization, we can orient circles in other ways, for instance, parallel to the xz-plane:

```
sage: var('t')
sage: parametric_plot3d((1,cos(t),sin(t)),(t,0,2*pi),thickness=10)
```



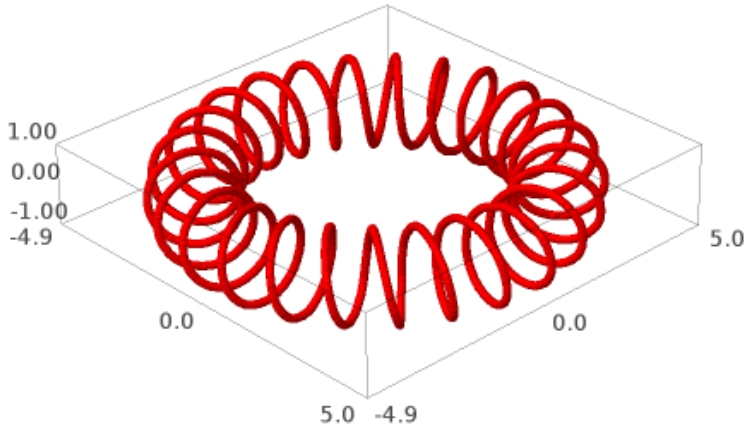
The default number of points to plot is 75, but it is often necessary to use more points to smooth out the graph. Here is a "toroidal spiral" which doesn't look especially spirally.

```
sage: var('t')
sage: p = parametric_plot3d(((4+sin(25*t))*cos(t), \
             (4+sin(25*t))*sin(t), cos(25*t)), (t,0,2*pi), \
             color='red', thickness=10)
sage: show(p,aspect_ratio=1)
```



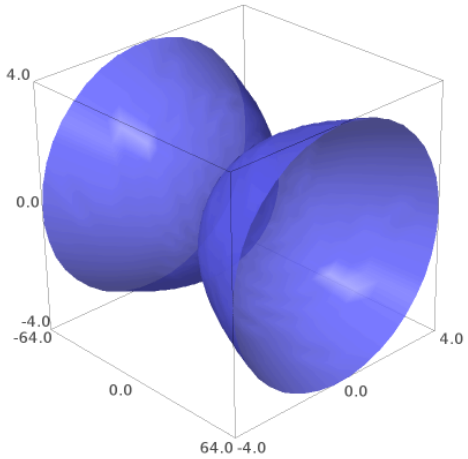
The reason is that it needs about 200 points to make it smooth.

```
sage: var('t')
sage: p = parametric_plot3d(((4+sin(25*t))*cos(t), \
             (4+sin(25*t))*sin(t), cos(25*t)), (t,0,2*pi), \
             color='red', thickness=10, plot_points=200)
sage: show(p,aspect_ratio=1)
```



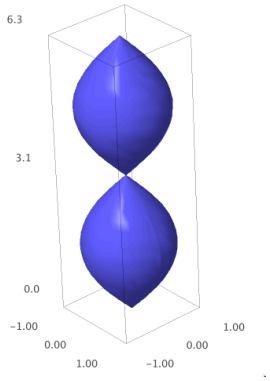
We can use the same `parametric_plot3d()` command to graph parametric surfaces. Defining parametric surfaces requires two independent variables, often called u and v. For example, we can graph the surface defined by $x = u^3$, $y = u\sin(v)$, $z = u\cos(v)$ for $-4 \leq u \leq 4$ and $0 \leq v \leq 2\pi$ in Sage:

```
sage: var('u v')
sage: parametric_plot3d((u^3,u*sin(v),u*cos(v)), \
             (u,-4,4), (v,0,2*pi), opacity=.8)
```



Cylindrical and spherical coordinates. Surfaces given in cylindrical or spherical coordinate systems can be viewed as special kinds of parameterizations. From cylindrical coordinates (r, θ, z), we convert to rectangular coordinates by $x = r\cos\theta$, $y = r\sin\theta$, and $z = z$. Below is the surface $r = \sin(z)$, for $0 \le \theta \le 2\pi$ and $0 \le z \le 2\pi$.

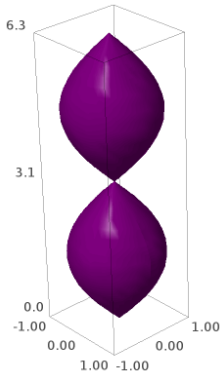
```
sage: var('u v')
sage: show(parametric_plot3d((sin(u)*cos(v),sin(u)*sin(v),u), \
             (u,0,2*pi), (v,0,2*pi)), aspect_ratio=1)
```



An alternative method uses `plot3d()` with a cylindrical transformation. The `Cylindrical()` command, as used below, will view the radius as a function of azimuth (θ) and height (z).

```
sage: S=Cylindrical('radius', ['azimuth', 'height'])
sage: var('theta, z')
sage: show(plot3d(sin(z), (theta,0,2*pi), (z,0,2*pi), \
          transformation=S, color='purple'), aspect_ratio=1)
```

Both methods produce the same graph.

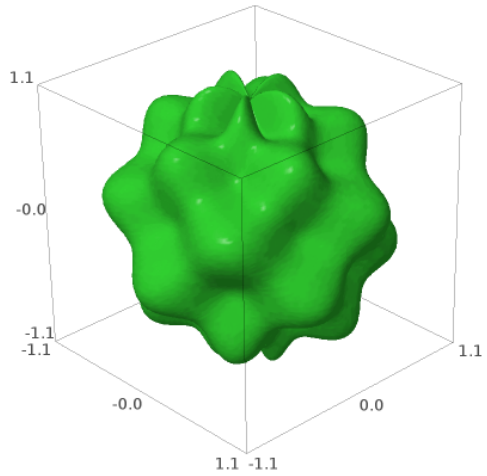


The conversion from spherical coordinates (ρ, θ, ϕ) to rectangular coordinates is given by $x = \rho \sin\phi \cos\theta$, $y = \rho \sin\phi \sin\theta$, and $z = \rho \cos\phi$. The surface defined by $\rho = 1 + \frac{1}{6}\sin(5\theta)\sin(6\phi)$ for $0 \leq \theta \leq 2\pi$ and for $0 \leq \phi \leq \pi$ is known as a "bumpy sphere" and would be graphed in Sage by

```
sage: var('theta phi')
sage: rho = 1+(1/6)*sin(5*theta)*sin(6*phi)
sage: parametric_plot3d((rho*sin(phi)*cos(theta), \
             rho*sin(phi)*sin(theta), rho*cos(phi)), \
             (theta,0,2*pi), (phi,0,pi), plot_points=[200,200], \
             color='limegreen')
```

Again, we can form the same graph using the `plot3d()` command and an appropriate transformation. The transformation is given by the `Spherical()` command, viewing the radius as a function of the azimuth (θ) and inclination (ϕ).

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
sage: var('phi theta')
sage: plot3d(1+(1/6)*sin(5*theta)*sin(6*phi), (theta,0,2*pi), \
           (phi,0,pi), transformation=T, plot_points=[200,200], \
           color='limegreen')
```



Advanced tools for 3d plots

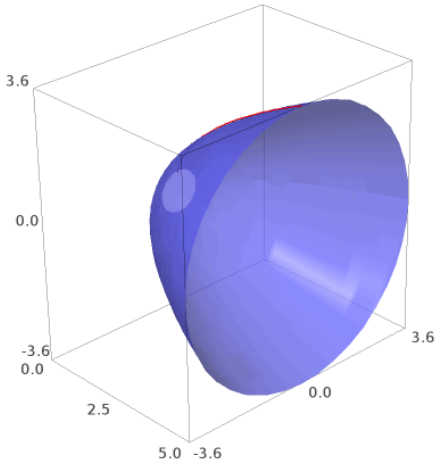
Sage offers a number of tools that assist in the production of special kinds of 3d plots in Cartesian, cylindrical, or spherical coordinates.

Surfaces of revolution. A special kind of surface is a surface of revolution. It is possible to graph these through parameterizations, but it is certainly helpful that Sage has a function especially for such graphs. Use `revolution_plot3d()` to graph these based on just the function, the axis of revolution, and the interval. Recall that if we revolve a function $f(x)$ about the x-axis, its surface area is given by

$$S = \int_a^b 2\pi f(x)\sqrt{1+f'(x)^2}\,dx$$

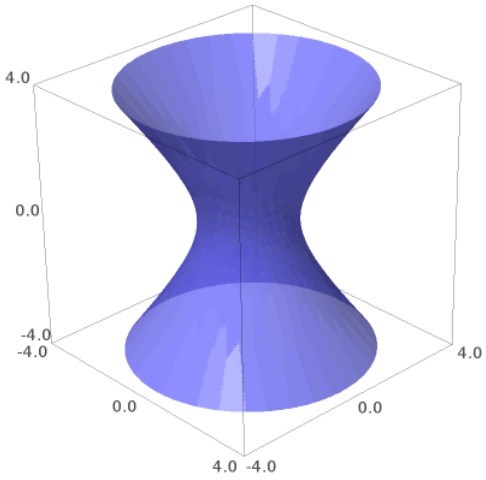
If $f(x) = \sqrt{3x-2}$,we can ask Sage to its surface area and visualize both the curve $f(x)$ and the 3D surface.

```
sage: f(x) = sqrt(3*x-2)
sage: integral(2*pi*f(x)*sqrt(1+diff(f,x)**2), x, 0, 5)
1/18*pi*(61*sqrt(61) - 1)
sage: show(revolution_plot3d(f, (x,0,5), parallel_axis='x', \
          show_curve=True, opacity=0.7), aspect_ratio=1)
```



Implicit plots in 3D. As in 2D, the most convenient way to describe some surfaces is implicitly. This time, the equations may involve one or more of the variables x, y, and z. With the `implicit_plot3d()` command, we need to specify an equation, as well as the maximum and minimum values for all three variables. In this example, the `implicit_plot3d()` command is used to produce a hyperboloid of one sheet:

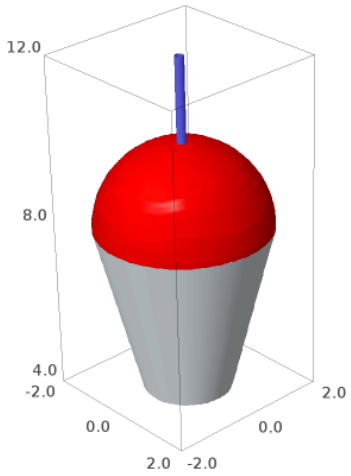
```
sage: var('x y z')
sage: implicit_plot3d(x^2/2+y^2/2-z^2/3==1, (x,-4,4), (y,-4,4), \
             (z,-4,4), opacity=.7, plot_points=200)
```



After all this, aren't you ready for a wild cherry snowball?[96]

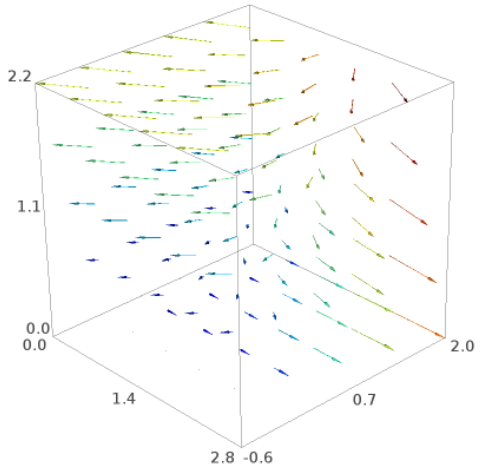
[96]Wild cherry because wedding cake will not show up.

```
sage: var('z theta r phi')
sage: bh=4 # z-coordinate for base of cup
sage: th=8 # z-coordinate for top of cup
sage: br=bh/(th-bh) # radius of base of cup
sage: tr=th/(th-bh) # radius of top of cup
sage: a = .1 #straw radius
sage: cupside=parametric_plot3d((z*cos(theta)/(th-bh), \
             z*sin(theta)/(th-bh),z), (theta,0,2*pi), \
             (z,bh,th), color='aliceblue')
sage: cupbase=parametric_plot3d((r*cos(theta), r*sin(theta), bh), \
             (theta,0,2*pi), (r,0,br),color='aliceblue')
sage: snow=parametric_plot3d((tr*sin(theta)*cos(phi), \
             tr*cos(theta)*cos(phi), tr*sin(phi)+th), \
             (theta,0,2*pi), (phi,0,pi/2),color='red')
sage: straw = parametric_plot3d((a*cos(theta),a*sin(theta),z), \
             (theta,0,2*pi),(z,bh+.01,th+2*tr))
sage: show(cupside+cupbase+snow+straw,aspect_ratio=1)
```



Vector fields. A vector field assigns to each point in the domain (either $\mathbb{R}^2$or $\mathbb{R}^3$) a 2- or 3-dimensional vector. Two-dimensional vector fields can be graphed with `plot_vector_field()`. Three-dimensional vector fields can be graphed with the `plot_vector_field3d()` command. In the example below, we graph the vector field $\mathbf{F}(x,y,z) = -2xy\mathbf{i} - 3z\mathbf{j} + z\mathbf{k}$

```
sage: plot_vector_field3d((2*x*y,-3*z,z),(x,0,2),(y,0,2),(z,0,2))
```



Ascending a hill

What is the quickest way up a hill? You should start by looking immediately around you, find the steepest path possible, proceed a little way in that direction. Then repeat this process, reevaluating your course after you go some small distance. Continue in this fashion and eventually you reach the top of the hill. You only have to look immediately around your current position at any point along the way. This strategy describes an important mathematical technique called "gradient ascent."[97]

As you know, the peak of the hill would be called a local maximum. Exact methods may be able to find the coordinates of the local maximum. When exact methods fail, the method of gradient ascent will give an excellent approximation. Better than that, it also approximates the path of steepest ascent.

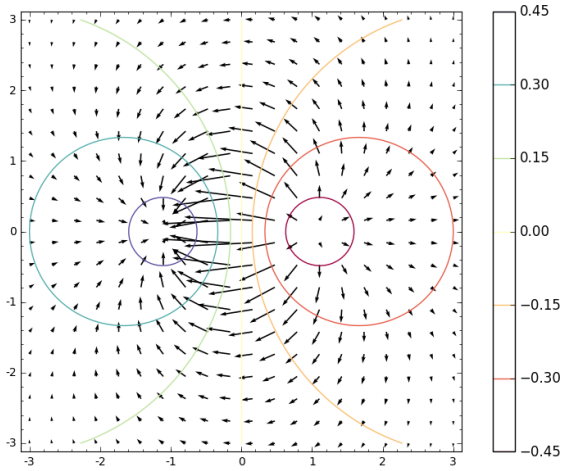
How do we know which direction to proceed in? The gradient of a function $f(x,y)$ is the vector function ∇f defined by

$$\nabla f(x,y) = \frac{\partial f}{\partial x}\overrightarrow{\mathbf{i}} + \frac{\partial f}{\partial y}\overrightarrow{\mathbf{j}}.$$

(Here, $\overrightarrow{\mathbf{i}}$ and $\overrightarrow{\mathbf{j}}$ represent the canonical basis vectors along the x- and y-axes.) When evaluated at a point (x_0, y_0), the gradient gives a vector in the the direction of the maximum rate of change. For example, if $f(x,y) = {}^{-x}/_{1+x^2+y^2}$ as before, we can plots its 2D gradient vector field. Each vector points in the direction of steepest ascent from its tail point.

```
sage: var('y')
sage: f(x,y)=-x/(1+x^2+y^2)
sage: cp = contour_plot(f(x,y),(x,-3,3),(y,-3,3), \
             cmap='Spectral', fill=False,colorbar=True)
sage: vf = plot_vector_field(f.gradient(),(x,-3,3),(y,-3,3))
sage: show(cp+vf)
```

[97] After going up the hill, you may want to go back down — or perhaps you never wanted to climb it in the first place! In this case, you can use the method of gradient *de*scent. Want to guess how that works?



Back to our gradient ascent: If we compute the gradient at our starting point, take a small step in that direction, we can repeat the process from that next point until we are at the top of the hill. The small steps are just gradient vectors scaled down using a small value of ε. From any point (x_n, y_n), we move to the next point (x_{n+1}, y_{n+1}) by the formula

$$(x_{n+1}, y_{n+1}) = (x_n, y_n) + \varepsilon \nabla F(x_n, y_n).$$

Since each point depends only on the previous point, an implementation needs to remember only the current position in order to compute the next one. Rather than use subscripts as above, we can rewrite the formula more simply using $\mathbf{q}$ as the current point and $\mathbf{p}$ as the previous point:

$$\mathbf{q} = \mathbf{p} + \varepsilon \nabla F(\mathbf{p})$$

How do we know when we've reached the top? Near the local maximum, the magnitude of the gradient vector will be nearly zero. (This means the tangent plane will be nearly horizontal.) When this short gradient vector is scaled down by ε, adding the vector $\varepsilon \nabla F(\mathbf{p})$ to $\mathbf{p}$ will keep $\mathbf{q}$ very close to $\mathbf{p}$. So we can say that we have reached the local maximum when our steps become very small, i.e., when the distance between points $\mathbf{p}$ and $\mathbf{q}$ is smaller than some desired accuracy. Since we cannot predict the number of steps before starting, we use a `while` loop for the repetition.

We can take advantage of the nature of the condition in the while loop by starting p and q appropriately.

algorithm gradient_ascent
inputs
- $f(x,y)$, a differentiable function given as an expression or function in x and y
- $\mathbf{p}$, the point (x_0, y_0)
- d, the desired number of decimal places of accuracy
- ε, the scaling factor of a step

outputs
- a list L of the points along the approximate path of steepest ascent

do
convert $\mathbf{p}$ to a vector
let $L = (\mathbf{p})$
let $\mathbf{q} = \mathbf{p}$
add $\overrightarrow{i}$ **to** $\mathbf{p}$
repeat as long as $\mathbf{p}$ and $\mathbf{q}$ are farther apart than the desired accuracy:
  let $\mathbf{p} = \mathbf{q}$
  let $\mathbf{q} = \mathbf{p} + \epsilon \nabla f(\mathbf{p})$
  append $\mathbf{q}$ to the list
return the list

This translates to the following Sage code:

```
sage: def gradient_ascent(f,p,eps,d):
          p = vector(p).n()
          L = [p]
          q = p
          p = p + vector([1,0])
          grad = f.gradient()
          while round(norm(p-q), d) > 0:
              p = q
              q = p + eps*grad(x=p[0], y=p[1])
              L.append(q)
          return L
```

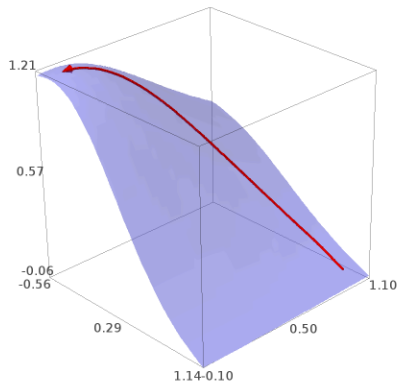
Let's add a visualization of our steps along the path. If we have a path as a list of (x,y) points output from the gradient ascent algorithm, then we can graph those points on the 3d plot with an arrow toward the last point. In order to know the dimensions of the plot of $f(x,y)$, we will need the minimum and maximum x- and y-values from all points on the path. A helper function to find these values is given here:

```
sage: def xyminmax(L):
          xvals = [pt[0] for pt in L]
          yvals = [pt[1] for pt in L]
          xmin = min(xvals)
          xmax = max(xvals)
          ymin = min(yvals)
          ymax = max(yvals)
          return xmin, xmax, ymin, ymax
```

In our visualization, we will include the 3D plot of $f(x,y)$ and the line segments with an arrowhead. We extend the graph of $f(x,y)$ to include a bit more beyond the minimum and maximum x- and y-values.

```
sage: def visual_gradient_ascent3d(f,p,eps,d):
          L = gradient_ascent(f,p,eps,d)
          xmin, xmax, ymin, ymax = xyminmax(L)
          xrange = xmax - xmin
          yrange = ymax - ymin
          g = plot3d(f, (x, xmin - .1*xrange, xmax+.1*xrange), \
              (y, ymin - .1*yrange, ymax+.1*yrange), opacity=.5)
          g = g + line3d([(pt[0], pt[1], f(x=pt[0], y=pt[1])) \
              for pt in L], color='red', arrow=True, size=30)
          show(g)
          q = L[-1]
          return q, f(x=q[0], y=q[1])
```

Starting from $(1,1)$ on $g(x,y) = \frac{1-x}{1+x^2+y^2}$, the algorithm finds that 25 steps are needed for 3 decimal places of accuracy with $\varepsilon = .2$. The maximum value is approximately $z = 1.2071$ at $(-0.4141, 0.00069)$, and the path has this shape:



Exercises

True/False. If the statement is false, replace it with a true statement.

1. Parametric curves and parametric surfaces can be graphed using the same Sage command.
2. Every 3D graphing function in Sage has a corresponding 2D function.
3. In 3D, graphs are combined by appending them.
4. Sage's `detect_poles=True` option will detect the asymptote in $f(x,y) = 3/x^2+y^2$.
5. If you want to graph $\rho = \theta/\phi$ for $\pi/12 \leq \phi \leq \pi$ and $0 \leq \theta \leq 6\pi$, the easiest way is with the `plot3d()` command.
6. The surface $r = z\sin(2\theta)$ for $0 \leq z \leq 1$ and $0 \leq \theta \leq 2\pi$ is the same as for $0 \leq \theta \leq 8\pi$, so Sage's graph of `plot3d(z*sin(2*theta), (theta,0,2*pi), (z,0,2), transformation=C)` is the same as that of `plot3d(z*sin(2*theta), (theta,0,8*pi), (z,0,2), transformation=C)`, after letting `C=Cylindrical('radius', ['azimuth', 'height'])`,
7. The best way to graph the paraboloid $y = x^2 + z^2$ is to isolate z by taking the square root, plot the positive and negative branches separately using `plot3d()`, and combine them using addition.
8. If you want to see the level curves of $f(x,y)$, you use `plot3d()` with the `mesh=True` option.
9. The domain of $\nabla f(x,y)$ is a subset of $\mathbb{R}^2$.
10. If the function has no local maximum, the gradient ascent algorithm returns the initial point (x_0, y_0).

Multiple choice.

1. Suppose `pts` is a list of 3D points. If you want to use those vertices to plot an unfilled, outlined polygon in 3D, what should you do?
   A. `polygon(pts, fill=False)`
   B. `polygon(pts, color=None, edgecolor='green')`
   C. `line(pts+[pts[0]])`
   D. ask your instructor
2. How many parameters are needed to parametrically define 3-dimensional curves?
   A. 1
   B. 2
   C. 3
   D. 4
3. How many parameters are needed to parametrically define 3-dimensional surfaces?
   A. 1
   B. 2
   C. 3
   D. 4
4. Compare a gradient vector to the level curve from which it originates.
   A. The gradient vector is always tangential to the level curve.
   B. The gradient vector is always perpendicular to the level curve.
   C. The gradient vector has no relationship to the level curve.
   D. The gradient vector always has the same color as the level curve.
5. The error produced by `plot3d(sin(x*y)/(x*y), (x,5,-5), (y,-3,3))` is due to
   A. the function being undefined at the origin
   B. misplaced parentheses
   C. the region not being square

D. the reversal of `xmin` and `xmax`

6. If you are graphing a 3D surface with `plot3d()`, how many pairs of min and max values do you need to give?
   A. 1
   B. 2
   C. 3
   D. 4
7. If you are graphing a 3D surface with `implicit_plot3d()`, how many pairs of min and max values do you need to give?
   A. 1
   B. 2
   C. 3
   D. 4
8. To modify the gradient ascent algorithm to do gradient descent, we should let
   A. $\mathbf{q} = -\mathbf{p} + \varepsilon \nabla f(\mathbf{p})$
   B. $\mathbf{q} = \mathbf{p} + \varepsilon \nabla f(-\mathbf{p})$
   C. $\mathbf{q} = -\mathbf{p} - \varepsilon \nabla f(\mathbf{p})$
   D. $\mathbf{q} = \mathbf{p} - \varepsilon \nabla f(\mathbf{p})$
9. The gradient ascent algorithm could be modified to find a local maximum of functions of how many variables?
   A. 1
   B. 3
   C. 4
   D. All of these
10. Which of the following best describes the meaning of the gradient?
    A. a number that tells you whether you are at a local maximum
    B. a number that tells you whether to ascend or descend
    C. a vector-valued function that can be evaluated to tell you which direction to go to stay at the same level
    D. a vector-valued function that can be evaluated to tell you which direction to go to ascend most quickly

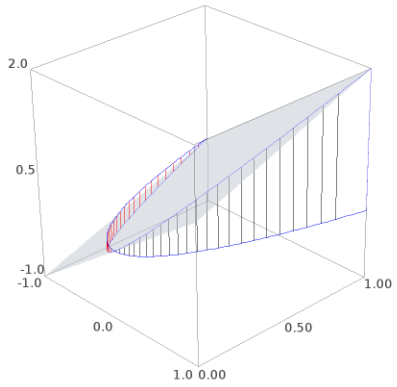
Short answer.

1. Are different colors necessary for visualizing a vector field or could you still understand the vector field if all vectors were shown in the same color?
2. What can go wrong with the gradient descent algorithm? Explore with various functions and initial points. How might you modify the code to avoid these issues?
3. Plot the ellipsoid $2x^2 + y^2 + z^2 = 3$ with each of
   (a) `plot3d()`
   (b) `implicit_plot3d()`
   (c) `revolution_plot3d()`
   (d) `parametric_plot3d()`.
   Which produces the best graph, even with more plotted points? What issues do you see with the others?

Programming.

1. Write a Sage function that would graph line segments with arrow heads on all segments. Make sure it works for collections of 2D or 3D points.
2. Rewrite the gradient descent algorithm (without plotting) recursively.
3. Rewrite the graphical gradient descent algorithm to plot on the surface using `contour_plot()` rather than `plot3d()`. Also, include the plot of the gradient vector field in the visualization.
4. Rewrite your code from problem 3 to show an animation on the contour plot, where each frame of animation contains one more step than the previous frame.
5. Write a Sage function to visualize the region whose area is computed by a line integral with respect to arc length $\int_C f(x,y)\,ds$ in the case that the curve C parameterized by $x(t)$ and $y(t)$ for $a \leq t \leq b$. The graph should be formed by combining
   - the boundary in blue–including vertical lines from the xy-plane to f at $x = a$ and $x = b$ as well as the curve C, plotted on the xy-plane and projected onto the surface $f(x,y)$,
   - the interior of the region, not shaded, but displayed using at least 50 vertical lines, showing any positive area with black lines and any negative area with red lines, and
   - the graph of $f(x,y)$ in a half-transparent `aliceblue` color.

   For example, with 80 vertical lines, `visualize_line_int(x+y, (t,t^2), -1, 1)` should produce a graph like the one below.



Advanced Techniques

This last chapter introduces two techniques that the serious Sage-user will find useful, and probably necessary, from time to time.

Making your own objects

Sage includes a lot of mathematics. — No, that's not right. Sage includes

A LOT

of mathematics. You will probably never, ever need to use anything that is not in Sage.

Still, some mathematicians sometimes work with a group of objects that Sage does not offer immediately. If you're studying combinatorial games, which are nothing short of awesome[98] then Sage does not immediately offer a way to perform what is called *Nimber arithmetic.*[99] This section introduces Nimber arithmetic and shows how you can *create your own type* to enable Sage to do Nimber arithmetic.

Background. The basic idea of Nimbers is born from a game named Nim, first described in a mathematical journal early in the 20th century. The rules are simple:

- The game is played with pebbles arranged in rows.
- Each player may take as many pebbles as desired from *one* row.
- The last player to remove a pebble wins.

For instance, suppose David and Emmy decide to play a game of Nim with three rows of pebbles, where the first row has seven pebbles, the second row has five, and the last row has three.

David goes first; suppose he takes four pebbles from the first row.

[98] Hackenbush, Nim, Chomp, Ideal Nim, Dots, Sprouts, … see [3] or [2] for more information than you can shake a stick at.

[99] …as of the version being used by the authors at the time of this writing. There has been some discussion of including the Combinatorial Game Suite, which would probably fix that in a jiffy, and offer even more than nimbers: even surreal numbers.

This was not a very smart move on David's part, as Emmy can take all the pebbles from the second row and leave this configuration.

David now sees that whatever move he makes in one row, Emmy can "mirror" that move in the other row. In other words, he has already lost.

Nimber arithmetic. The idea of Nimber arithmetic flows from this observation that, once the game works itself down to two rows that are visually symmetric, the game is essentially over. The arithmetic works like this:

- The smallest Nimber is 0, followed by 1, 2, ...
- Given a set S of Nimbers, the smallest Nimber *not* in S is called the minimum excludant, or mex for short.
- To add two Nimbers, write each as a sum of powers of 2, cancel identical powers, then simplify.
- Subtraction is equivalent to addition.
- To multiply two Nimbers m and n, find $\operatorname{mex}\{in + mj + ij : i < m, j < n\}$.

So, for example, addition works like so:

$$\begin{aligned}
0 + x &= x \\
1 + 1 &= 0 \\
1 + 2 &= 3 \\
2 + 2 &= 0 \\
1 + 3 &= 1 + (1 + 2) = 2 \\
2 + 3 &= 2 + (1 + 2) = 1 \\
3 + 3 &= 0
\end{aligned}$$

while multiplication works like so:

$$\begin{aligned} 2\times 3 &= \operatorname{mex}\{0\times 3+2\times 0+0\times 0, 1\times 3+2\times 0+1\times 0, 0\times 3+2\times 1+0\times 1,\\ &\qquad 1\times 3+2\times 1+1\times 1, 0\times 3+2\times 2+0\times 2, 1\times 3+2\times 2+1\times 2\}\\ &= \operatorname{mex}\{0,3,2\}\\ &= 1. \end{aligned}$$

This is burdensome to do by hand, so it's a good idea to automate it in Sage.

We could, of course, write a sequence of functions that simply perform the operations directly, but it's also possible to give Sage a way of dealing with the numbers automatically, and using them in a natural way. For instance, if we create functions `nim_add()` and `nim_mult()`, then to add and multiply $a\times b+c\times d$, we'd have to type

```
nim_add(nim_mult(a,b),nim_mult(c,d)) .
```

This is hard to understand. It is much more natural to type

```
a*b + c*d
```

and then, so long as Sage recognizes `a`, `b`, `c`, and `d` as Nimbers, it performs the arithmetic automatically.

Do it with class! The way to get Sage to do this is with a **class**. A class describes an object's type to a computer, and tells the computer what methods can be sent to an object of this type. Classes allow us to organize programs around data and the things we can do with that data, and help to keep things in one place. In Sage, the data associated to a class, also called its **attributes**, are accessed inside the class using the *`self`* constant, which we must write as the first argument to each of the class's methods.[100] You will see this in a moment.

To create a class, use the `class` keyword:

```
sage: class ClassName:
          #list of functions associated with this class
```

As the colon suggests, the functions associated with this class should be indented. When the indentation stops, so does the class. Each class should have at least one method named `.__init__()`, which tells Sage how to initialize an element of the class. This method is called a **constructor**, and is called silently whenever we create a new object of a class.[101] This new object is called an **instance** of the class.

Initialization: What information is proper to the class? We will define a class named `Nimber`. A nimber is just a number whose arithmetic works differently than normal, so we can simply initialize it with a number, which really ought to be an integer, so we'll test for that and raise an exception if not. In addition, Nimber arithmetic is based on powers of 2, so it would be a good idea to keep track of the nimber's representations in terms of powers of 2; we might as well put that into the initialization code. Our `.__init__()` method should, therefore, initialize two items of data with the class `Nimber`:

- *`self`*`.value`, the integer associated with this value; and

[100] We don't actually have to use the name *`self`*; we could use `a` if we wanted, or `this` or `me`, but *`self`* is the convention Sage inherits from Python.

[101] It is also possible to call it again after we create a new object `a` by typing `a.__init__(...)`, though this should be done sparingly, if at all.

- `self.powers`, the powers of 2 that sum to `self.value`.

To figure out which powers of 2 sum to `self.value`, we will test whether the number is divisible by 2, then divide by 2 to remove powers, keeping only the quotient. For instance,

- 11 is not divisible by 2, so 1 is a power of 2 that sums to 11. If we divide $11 = 1+2+8$ by 2, we now have $5 = 0+1+4$.
- 5 is not divisible by 2, so 1 is a power of 2 that sums to 5. We had divided by 2, so $1\times 2 = 2$ is a power of 2 that sums to 11. If we divide $5 = 0+1+4$ by 2, we now have $2 = 0+0+2$.
- 2 is divisible by 2, so 1 is not a power of 2 that sums to 2. We had divided by 2 twice, so $1\times 2^2 = 4$ is not a power of 2 that sums to 11. If we divide $0 = 0+0+2$ by 2, we now have $1 = 0+0+1$.
- 1 is not divisible by 2, so 1 is a power of 2 that sums to 1. We had divided by 2 three times, so $1\times 2^3 = 8$ is a power of 2 that sums to 11. If we divide $1 = 0+0+1$ by 2, we now have $0 = 0+0+0$.

Once we hit 0, we are done, so the powers of 2 that sum to 11 correspond to divisions that gave us remainder 1: that is, 1, 2, and 8.

That suggests the following initialization code.

```
sage: class Nimber:
          def __init__(self, n):
              # check for valid type
              if type(n) != Integer and type(n) != int or n < 0:
                  raise ValueError, \
                        'Nimbers must be nonnegative integers'
              self.value = n
              self.powers = set()
              # find powers of 2 that add to n
              i = 1
              while n != 0:
                  if n % 2 == 1:
                      self.powers.add(i)
                  n = n // 2
                  i = i * 2
```

If you are working with a worksheet, go ahead and type this into one cell. Later we will add more methods, and you should type those into the same cell. If you are working from the command line, it is easiest to write this as a script and attach it; as you add more methods to the class and save the modified file, Sage will automatically re-load it.

Once we type the above and execute or attach it to Sage, how do we create a `Nimber` instance? We use the class name as if it were a function, whose arguments are appropriate to pass to its `__init__()` method.

```
sage: a = Nimber(4)
sage: b = Nimber(1/2)
ValueError:  Nimbers must be nonnegative integers
```

So far, so good. Let's look at `a` a little more closely.

```
sage: a
<__main__.Nimber instance at 0x161c9e5a8>
```

That's not very helpful output, is it?

Representations of an object. To give more useful output, you can add a `.__repr__()` method. Probably the best thing to do is print *`self`*`.value`. Sage wants `__repr__()` to return a string value, and you can convert *`self`*`.value` to a string using the `str()` command:

```
sage: class Nimber:
          ...
          def __repr__(self):
              return str(self.value)
```

(Don't forget that you need the `.__init__()` definition in the space indicated by the ellipses.) Now when initialize a nimber, we can print its value:

```
sage: a = Nimber(4)
sage: a
4
```

On the other hand, it could be useful to see which powers of 2 the class has found that sum to the number. The methods `__init__()` and `__repr__()` are called "special method names," but there isn't a special method name that's appropriate for an alternate, more detailed representation of an object. For this we can define a method with a name of our own devising; we will call it `.power_repr()`, and it will simply return *`self`*`.powers`.[102]

```
sage: class Nimber:
          ...
          def power_repr(self):
              return self.powers
```

We can now see how this works:

```
sage: a = Nimber(28)
sage: a.power_repr()
{4, 8, 16}
```

So far, so good.

[102] There is no need to surround `power_repr` with underscores, as this is not a "special method." The purpose of the underscores is to make sure people don't accidentally redefine a special method.

Implementing arithmetic. What we *really* need, though, is a way to automate the addition and multiplication of Nimbers — preferably in a way that allows us to use ordinary arithmetic operators. Again, Sage offers us special methods for this; we will use `.__add__()` and `.__mul__()`. But how will we implement them?

For addition, it should be a simple matter of choosing the powers of 2 that are not replicated in both nimbers' `powers`. There is an easy way to do this: since we kept the *set* of powers that add to the number, we can use the `.symmetric_difference()` method to a set. For example, the nimbers $14 = 2+4+8$ and $20 = 4+16$ should add to $2+8+16 = 26$. Applying the symmetric difference to the sets of powers of 2 gives us that result:

```
sage: {2, 4, 8}.symmetric_difference({4,16})
{2, 8, 16}
```

Addition, therefore, is relatively straightforward to implement: return the `Nimber` defined by sum of the symmetric difference.

```
sage: class Nimber:
          ...
          def __add__(self, b):
              return Nimber(sum( \
                  self.powers.symmetric_difference( \
                      b.powers \
                 )))
```

Let's test this on our example:

```
sage: Nimber(14) + Nimber(20)
26
```

Excellent!

Multiplication is a little tougher, for two reasons. First, recall the definition of nimber multiplication:

$$m \times n = \operatorname{mex}\{in + mj + ij : i < m, j < n\}.$$

This requires us to compute the mex of a set with a lot of numbers that we have to generate. Building the set is actually not that hard: we can use a list comprehension to take care of that. However, Sage does not have a built-in mex function, so we need to define a function to compute it. This should *not* be part of the `Nimber` class, because while the mex works on nimbers, it is not a property of a nimber. We will implement it as a separate function.

How should we implement the mex function? We have to find the smallest nimber that is *not* in the set, so why not start with $i = 0$, test to see if i is in the set, and increasing and repeating if so? This approach will work:

```
sage: def mex(S):
          i = 0
          while Nimber(i) in S:
              i = i + 1
          return Nimber(i)
sage: mex([Nimber(0), Nimber(2), Nimber(4)])
sage: 1
sage: mex([Nimber(0), Nimber(1), Nimber(2)])
sage: 3
```

This new function allows us to define multiplication for our class. The definition is recursive; that is, we first have to compute the products of smaller pairs. Our recursion needs base cases; we might as well use 0 and 1, as the definition of nimber multiplication shows that $0 \times x = 0$ and $1 \times x = x$ for every nimber x:

```
sage: class Nimber:
          ...
          def __mul__(self, b):
              if self.value == 0 or b.value == 0:
                  return Nimber(0)
              elif self.value == 1:
                  return b
              elif b.value == 1:
                  return self
              else:
                  return mex({self*Nimber(j) + Nimber(i)*b \
                              + Nimber(i)*Nimber(j) \
                              for i in xrange(self.value)
                              for j in xrange(b.value)})
```

If you actually try this, you may encounter the following error:

```
sage: a, b = Nimber(4), Nimber(2)
sage: a*b
TypeError:  'builtin_function_or_method' object is not iterable
```

PANIC!

…well, no, don't. Let's think about what might cause this.

Were you to experiment a little more, you would find deeper problems:

```
sage: a, b = Nimber(2), Nimber(2)
sage: a*b
0
```

The answer *ought* to be 3.[103] What's going on here? The basic problem is that `.__mul__()` creates a set of nimbers, then sends that to `mex()`, which tries to find a minimum element. There are two problems with this approach:

- Sage has no way to tell if two `Nimber`'s are equal, so we encounter replication in the set.
- Sage has no way to tell if one `Nimber` is less than another, so it can only guess at the set's minimum excludant.

To see this directly,

```
sage: S = {Nimber(0), Nimber(1), Nimber(2), Nimber(1), Nimber(2)}
sage: S
{0, 1, 1, 2, 2}
sage: mex(S)
0
sage: Nimber(2) == Nimber(2)
False
```

The replication of elements serves as a hint that Sage cannot tell that they are equal.

Comparing Nimbers. **We can fix this. We have the technology.** We can also fix the problem with mex, by adding special methods that teach Sage how to compare `Nimber`s. These are:

`.__eq__(`*self*`, `*other*`)`	tests whether *self* and *other* are the same
`.__ne__(`*self*`, `*other*`)`	tests whether *self* and *other* are different
`.__lt__(`*self*`, `*other*`)`	tests whether *self* is less than *other*
`.__le__(`*self*`, `*other*`)`	tests whether *self* is less than or equal to *other*
`.__ge__(`*self*`, `*other*`)`	tests whether *self* is greater than or equal to *other*
`.__gt__(`*self*`, `*other*`)`	tests whether *self* is greater than *other*

We have to define all six, but fortunately this isn't too hard, as we can simply compare `value`s.

[103] $2 \times 2 = \text{mex}\{0\times 2+2\times 0+0\times 0, 1\times 2+2\times 0+1\times 0, 0\times 2+2\times 1+0\times 1, 1\times 2+2\times 1+1\times 1\} = \{0,2,1\} = 3$. If you're wondering why the last element of the set isn't 5, remember that nimber addition is self-canceling, so $1\times 2+2\times 1=0$.

```
sage: class Nimber:
          ...
          def __eq__(self, b):
              return self.value == b.value
          def __ne__(self, b):
              return self.value != b.value
          def __lt__(self, b):
              return self.value < b.value
          def __le__(self, b):
              return self.value <= b.value
          def __ge__(self, b):
              return self.value >= b.value
          def __gt__(self, b):
              return self.value > b.value
```

This at least allows us to do the following:

```
sage: Nimber(2) == Nimber(2)
True
sage: Nimber(2) == Nimber(3)
False
```

Unfortunately, sets still won't work:

```
sage: S = {Nimber(0), Nimber(1), Nimber(2), Nimber(1), Nimber(2)}
TypeError:  unhashable instance
```

Ouch. What's a "hashable instance?"

An object is *hashable* when you can associate an integer (of `int` type, not `Integer` type) to it that Sage then uses to create sets and dictionaries. This integer is called a *hash* and should never change, as changing it would make things weird in sets and dictionaries. *That* means the object itself should never change. Both of these criteria apply to `Nimber`s, as we have not created any methods that change its data, and the `value` of a `Nimber` is an integer that will not change. To create this as its hash, we simply set it up as the `.__hash__()` method:

```
sage: class Nimber:
          ...
          def __hash__(self):
              return int(self.value)
```

…and now sets work great:

```
sage: S = {Nimber(0), Nimber(1), Nimber(2), Nimber(1), Nimber(2)}
sage: S
{0, 1, 2}
```

Now that we have sets of Nimbers working, we can finally multiply `Nimber`s. How about a `Nimber` multiplication table?

```
sage: for i in xrange(10):
          for j in xrange(10):
              print Nimber(i) * Nimber(j)
          print
```

You may not have much luck getting it to print more than a few lines. Why? Once we get beyond a certain point, the recursion in multiplication makes it very, *very* slow. Sage has a `@cached_method` decorator that should work like the `@cached_function` decorator, only for class methods. However, it doesn't work in this situation, because the cache is bound to each object, and our loop is constantly creating new `Nimber`s. A better approach would be to create a global cache; we leave this as an exercise to the reader.

Stuff we've left out. We've omitted a number of things that classes allow us to do, with *inheritance* and *overloading* being two major topics.

The idea of *inheritance* is to avoid code duplication by automatically inheriting methods for one type that legitimately apply to another type, as well. For instance, the set of nimbers forms a field; so, if `Nimber` were to inherit Sage's `Field` class, it would automatically acquire methods that are pre-defined for `Field`. The downside to this is that a field is a rather abstract object, so while `Field` provides a lot of methods, it doesn't implement most of them, but raises `NotImplementedError` when you try to access them. This means you have to implement that yourself, which isn't always easy. (For instance: How do you find a nimber's multiplicative inverse?)

The idea of *overloading* is that we can implement a function with different types. For instance, you might want to multiply a `Nimber` by a regular integer, rather than allow multiplication by `Nimber`s alone. In this case, you'd have to modify the `__mul__()` function to test its inputs. In other cases, a method might take two arguments sometimes, and three arguments other times. This is accomplished in Sage by providing default values for some arguments and checking argument types.

Cython

Sage is built on Python, an interpreted language. We spoke of the difference between interpreted and other kinds of code on p. 13; essentially, interpreting code incurs a penalty to translate lines over and over again.

In addition, Python relies on what is called "dynamic typing." That means Python does not know a variable's type until it assigns a value to it. For instance, in the `.__mul__()` method for `Nimber`, Sage does not know what type `b` has until it actually executes `.__mul__()`. This means that Sage must frequently stop to check a variable's type before executing an operation; after all, it cannot evaluate the expression `b.value` on an object `b` that lacks an attribute named `value`.

The other major approach to typing used in programming is "static typing." In this approach, the programmer must declare every variable's type of *before* the program is run. This can be a hassle to the programmer, but it saves the computer the trouble of checking each time whether it actually has to do this. That can lead to massive increases in speed.

Test case: the Method of Bisection. To see what a difference this can make, we re-present our `method_of_bisection()` function from page 161 and discuss how to time it.

```
sage: def method_of_bisection(a, b, n, f, x=x):
          c, d = a, b
          f(x) = f
          for i in xrange(n):
              # compute midpoint, then compare
              e = (c + d)/2
              if (f(c) > 0 and f(e) > 0) or (f(c) < 0 and f(e) < 0)
                  c = e
              elif f(e) == 0:
                  c = d = e
                  break
              else:
                  d = e
          return (c, d)
```

Recall that `a` and `b` are endpoints, `n` is the number of times to run the loop, `f` is a function, and `x` the indeterminate. To get an idea of how to time the code, we'll use a special function called `%timeit`. This function repeats a statement that follows it several times, and returns the fastest of those times.

```
sage: %timeit method_of_bisection(1, 2, 40, x**2 - 2)
100 loops, best of 3:  17 ms per loop
```

(You may get a number different from 17 ms; this is okay, as it depends on a lot of things besides the program: CPU speed, bus speed, ….) What happened? Sage ran the program in three sets of 100 loops, something akin to this:

```
sage: for i in xrange(3):
          for j in xrange(100):
              method_of_bisection(1, 2, 40, x**2 - 2)
```

For each of those three sets, it recorded how long it took to perform all 100 loops, obtaining three timings. Finally, it reported the best of those three timings: 17 milliseconds. In short, *one* execution of `method_of_bisection()` took roughly .17 milliseconds, or 170 microseconds.

To see what effect compiling can have, type the following, either into a Sage cell or into a script to attach to the command line. (If you use a worksheet, type `%cython` on the first line. If you use the command line, save the script with the suffix `.spyx` instead of `.sage` .) Don't worry about the changes just yet; we'll explain them in a moment.

```
from sage.symbolic.ring import SR
from sage.symbolic.callable import \
CallableSymbolicExpressionRing
def method_of_bisection_cython(a, b, n, f, x=SR.var('x')):
    c, d = a, b
    # f(x) = f
    f = CallableSymbolicExpressionRing([x])(f)
    for i in xrange(n):
        # compute midpoint, then compare
        e = (c + d)/2
        if (f(c) > 0 and f(e) > 0) or \
                (f(c) < 0 and f(e) < 0):
            c = e
        elif f(e) == 0:
            c = d = e
            break
        else:
            d = e
        return (c, d)
```

When you execute this cell or attach this script, Sage will pause a moment to compile the code. If you typed something wrong, you are likely to see one or more errors listed. In that case, look through this list carefully and try to fix the errors. Once it successfully compiles (you'll know because Sage reports no errors), you can time it just as you did before.

```
sage: %timeit method_of_bisection_cython(1, 2, 40, x**2 - 2)
100 loops, best of 3:  16 ms per loop
```

The code has sped up a bit: $16/17 \approx 94.1\%$ as long, or a speedup of roughly 5%. We don't blame you if this doesn't impress you. It is possible to do better in some cases; the Cython website gives one such example.

Before describing how we can improve this code, let's explain a bit about the differences in the Cython code from the original Sage code.

- ***Cython needs us to "import" objects that Sage ordinarily provides for free.***
  The statements that do this use the keywords `from` and `import`. We had to import the symbolic ring `SR` and a curious new type called `CallableSymbolicExpressionRing` that we'll discuss a little further down.
- ***Cython needs us to declare our indeterminates — even `x`!***
  …and to declare our indeterminates, we need the `var()` command, which is itself not available immediately to Cython; we have access it as a method to `SR`.
- ***Cython needs us to define mathematical functions in a different way.***
  When Sage reads anything you type in a worksheet, on the command line, or in a Sage script, it rewrites some of it from the convenient form we use to a form more convenient to its Python foundation. One example is the declaration of a function; as you might guess from the code above, the declaration `f(x) = ...` is a convenient shorthand for `f`

`= CallableSymbolicExpressionRing([x])(...)`. Another example is the "carat" operator `^`; this has a completely different meaning in Python, which considers the "double-product" operator `**` the true exponentiation operator. This is the reason we have always advised you to use the double-product for exponentiation, but if habit were to trip us up, and we type a carat, Sage will silently accept it as exponentiation – *unless* it appears in a Cython script.

These changes are therefore necessary to make the code compile as Cython.

We can fix this. We have the technology. Remember how we mentioned that a *lot* of time is wasted on checking the type of an object. What if we assign a type to the items that are reused the most? The code spends a lot of its time working on computing `f(c)` and `f(e)`, so perhaps we would do well to compute them once, then store their values. Let's change the program as follows:

```
from sage.symbolic.ring import SR
from sage.symbolic.callable import \
CallableSymbolicExpressionRing
def method_of_bisection_cython(a, b, n, f, x=SR.var('x')):
    c, d = a, b
    # f(x) = f
    f = CallableSymbolicExpressionRing([x])(f)
    for i in xrange(n):
        # compute midpoint, then compare
        e = (c + d)/2
        fc, fe = f(c), f(e)
        if (fc > 0 and fe > 0) or \
                (fc < 0 and fe < 0):
            c = e
        elif fe == 0:
            c = d = e
            break
        else:
            d = e
        return (c, d)
```

This gives us a noticeable improvement in performance.

```
sage: %timeit method_of_bisection_cython(1, 2, 40, x**2 - 2)
100 loops, best of 3:  11.6 ms per loop
```

That's an improvement of a little more than 25% over the previous version, so we're doing better.

But! — the thoughtful reader will complain — *we could have pre-computed those values without Cython, as well. What difference does that make?* It makes a healthy difference indeed. We won't show the code here, but if you modify `method_of_bisection()` to pre-compute those values, we see a timing along these lines:

```
sage: %timeit method_of_bisection(1, 2, 40, x**2 - 2)
100 loops, best of 3:  12.7 ms per loop
```

That's still slower than the pre-computing Cython version, but it's faster than the first Cython version! Again, you may find such a mild improvement less than impressive.

Fair enough, but we still have one trick up our sleeve.

Assigning static types. As we mentioned earlier, Python is a dynamically-typed language. Cython is more of a hybrid. As we saw above, you don't *have* to type anything, but you can type *some* things. You're about to see that this can have a significant effect.

In addition to computing `f(c)` and `f(e)` only once, let's try assigning a type to them. What type should they have? Some of the usual options are

- `int` and `long`, which correspond to the exact Python types for "smaller" and "larger" integers;
- `float` and `double`, which correspond to the exact Python types for "smaller" and "larger" floating-point numbers;
- `Integer` and `Rational`, which correspond to the exact Sage types for integers and rational numbers.

There are many, many other types, some of which are very useful in various circumstances. Remember that you can generally find the type of an object by asking sage: `type(2/3)`, for instance, will give us `<type 'sage.rings.rational.Rational'>`. Make a note of that, as we'll return to it in a moment.

If you look back to where we introduced the Method of Bisection, we mentioned that it is a kind of "exact approximation," in that it specifies an interval (so, approximation) on which the root certainly lies (so, exact). You'll remember that the Method of Bisection gives us fractions, which more or less rules out `int`, `long`, and `Integer`. Should we use `float`, `double`, or `Rational`, then? Any one of them will work, and the first two are much, much faster than the third, but the authors have a fondness for exact numbers in all their long, complicated glory, so we'll opt for that.

Recall that the `type()` command told us a `Rational` is `sage.rings.rational.Rational`. Of those four words, the first three tell us *from where* we import, and the last tells us *what* we import. We'll add a line to our import list that takes care of this:

```
from sage.rings.rational cimport Rational
```

If you look carefully, you'll notice we used the `cimport` keyword instead of the `import` keyword as we did with the other two. This is because we are importing a type implemented as a class in another Cython script. *You cannot import types implemented as classes in Sage or Python scripts.*

Once we have that, we add a line at the beginning of the code that defines `fc` and `fe` as `Rational`. We then have to *coerce* the computations `f(c)` and `f(e)` to `Rational`, because they are by default a mere `Expression`. These are all the changes we have to make, so we can list the modified code below:

```
from sage.symbolic.ring import SR
from sage.rings.rational cimport Rational
from sage.symbolic.callable import \
CallableSymbolicExpressionRing
def method_of_bisection_cython(a, b, n, f, x=SR.var('x')):
    cdef Rational fc, fe
    c, d = a, b
    f = CallableSymbolicExpressionRing([x])(f)
    for i in xrange(n):
        # compute midpoint, then compare
        e = (c + d)/2
        fc, fe = Rational(f(c)), Rational(f(e))
        if (fc > 0 and fe > 0) or (fc < 0 and fe < 0):
            c = e
        elif fe == 0:
            c = d = e
            break
        else:
            d = e
    return (c, d)
```

The timing on this code is *remarkably* faster:

```
sage: %timeit method_of_bisection_cython(1, 2, 40, x**2 - 2)
100 loops, best of 3:  6.27 ms per loop
```

That's almost half as fast as it was before, and definitely half as fast as the fastest Python time we've seen. Altogether the code now takes about one-third as long as the original.

What if we add this type information to the Python script? — oh, wait. We *can't*.

A caveat. Adding type information for the remaining variables does not improve performance in any measurable way and in some cases it actually slows the program down. (We checked to be sure.) If you find it necessary to go down the route of writing Cython scripts, then optimizing them by adding type information, you will want to learn about profiling your code to see what is taking longest. We give only a few brief hints here.

Sage offers a profiling tool called `prun()`. It counts the number of times each function is called, measures the amount of time Sage spends in each function, and guesses from there which functions would be best to compute. This works with both ordinary Sage/Python functions as well as with Cython functions.

When writing this chapter, we tried `prun()` on the original Cythonized `method_of_bisection()` as follows:

```
sage: prun(method_of_bisection_cython(1, 2, 30, f))
```

Sage ran the program and immediately provided output in the form of a table, with each column indicating

- number of calls (`ncalls`);
- total time spent in the function (`tottime`);
- average amount of time spent in each call (first `percall`);
- cumulative total time spent in that function and all functions it calls (`cumtime`);
- average cumulative time spent in each call (second `percall`); and
- the location of the call.

When we first ran it, three of the first four lines indicated that Sage made

- 106 calls to the `substitute` method for `sage.symbolic.expression.Expression` objects;
- 107 calls to `callable.py:343(_element_constructor_)`; and
- 106 calls to `callable.py:448(_call_element_)`.

Without having seen those lines before, it wasn't hard to guess that Sage was spending a lot of time computing the value of a function at a point. This motivated us to try avoiding re-computation of `f(c)` and `f(e)`. When that was done, the number of calls to these three functions was cut by more than a third, respectively to 60, 61, and 60 again.

With that out of the way, the hint that we needed to assign a type to *something* would be useful is that there were 209 calls to`{isinstance}`. That, along with mysterious invocations of functions in the file `complex_interval_field.py`, indicated that a lot of type checking was going on, and in fact the final version of the program reports *only one* innovation of `{ininstance}`, and *no* invocations from `complex_interval_field.py` .

Deciding which precise variables need type information remains something of an art form with skills acquired by experience and trial-and-error, which is why we classify it, along with the creation of classes, under "Advanced Techniques." If you go far enough into the wondrous world of mathematics, you may find yourself needing them one day.

Exercises

True/False. If the statement is false, replace it with a true statement.

1. Most people who work with Sage will, at some point, need to make their own mathematical objects.
2. Combinatorial games are nothing short of awesome.
3. Nimber addition is self-canceling; that is, $x + x = 0$ for any nimber x.
4. Nimber multiplication makes use of the maximum element of a set of numbers.
5. One reason to encapsulate Nimbers in a class is to write the arithmetic operations in a more convenient and natural format.
6. A class's attributes are variables that include data that is proper to the class.
7. Modification of a class's basic behaviors is subject to special methods: `__init__()`, `__repr__()`, `__eq__()`, and so forth.
8. In Sage, a `set` automatically figures out when two objects of a class are equivalent.
9. A hash is a technical term for an object whose class is too complicated to add to a `set`.
10. The angry dragon of recursion flies into this chapter and sets at least one class method aflame.
11. Cython allows us to speed up Sage programs by *both* compiling *and* assigning static types.
12. Dynamic languages know every object's type before the program is executed; static languages remain agnostic until the value is assigned.

13. We can only make use of Cython from the worksheet, by placing the `%cython` statement before a block of code.
14. We can "Cythonize" Sage code without having to worry about modifying it.
15. We cannot import types implemented as classes in Cython scripts, only types that are native to Sage like `int` and `Rational`.

Multiple Choice.

1. The methods used to initialize the data proper to a class are called:
   A. attributors
   B. constructors
   C. initializers
   D. makers
2. The correct term to refer to a variable whose type is of class `C` is:
   A. an attribute of `C`.
   B. a constructor for `C`.
   C. a descendant of `C`.
   D. an instance of `C`.
3. The value of $\operatorname{mex}\{1,2,4,5,7,8,9\}$ is:
   A. 0
   B. 3
   C. 6
   D. 9
4. In Nimber arithmetic, the value of $3 + 10$ is:
   A. 0
   B. 2
   C. 9
   D. 13
5. Why is Nimber multiplication difficult to do by hand?
   A. it requires recursion
   B. it's new and different
   C. self-cancellation of addition
   D. it *isn't* difficult to do by hand
6. The proper way to create a new instance of a class named `C`, whose constructor takes an integer argument which we want to be 2, is by typing:
   A. `a = C(2)`
   B. `a = new C(2)`
   C. `a = new(C, 2)`
   D. `a = C.__init__(2)`
7. The term for the specially-named methods Sage expects for certain "natural" behaviors of a class are:
   A. attribute methods
   B. magic methods
   C. special methods
   D. dot methods
8. An object is hashable when we can associate it with a value of what type?
   A. any well-ordered type
   B. `hash`

C. `Integer`
D. `int`

9. Which keyword allows us to assign a type to a variable in Cython?
   A. `cdef`
   B. `def`
   C. `type`
   D. no keyword; just place the type before the variable name
10. What keyword(s) do we use to access Sage objects and types defined in other Cython files?
   A. `access...from...`
   B. `from...import...`
   C. `#include`
   D. PANIC!

Short Answer.

1. Summarize why it can be useful to create a class for a new mathematical object, rather than implementing operations as functions with the names `my_object_add()`, `my_object_mul()`, and so forth.
2. Summarize the similarities and differences between attributes and instance variables.
3. While Cython compiles Python statements to true machine code, the result cannot be run independently of a Python interpreter (like Sage). Do you think this makes Cython a truly compiled language? Why or why not?
4. The Method of Bisection can be run with `float` types instead of `Rational` types. Do you think this will make it faster or slower? Try it and discuss how the result compares with your guess.

Programming.

1. Another special method for a class is `__nonzero__()`, which allows Sage to decide whether an instance of a class is, well, "nonzero." Implement this method for `Nimber`, returning `True` if and only if `self.value==0`.
2. It is *somewhat* more correct in Python that `__repr__()` should return a string with which you could re-create the object; that is, the following sequence of commands should actually have the given output, not just `2`.

```
sage: a = Nimber(2)
sage: a
Nimber(2)
```

   (a) Modify `Nimber`'s `__repr__()` method to return the "proper" string. *Hint:* You can join strings by "adding".
   (b) Python recommends that the `__str__()` method be used to return an "informal," "more convenient," or "concise" representation of the object. This value is returned whenever we use the `str()` function. Add a `__str__()` method to the `Nimber` class that returns what we originally wrote for `__repr__()`. Once you're done with this problem, the class should work as follows:

```
sage: a = Nimber(2)
sage: a
Nimber(2)
sage: str(a)
2
```

3. We said that `Nimber` multiplication could be improved by using a cache variable. Modify your `Nimber` class to introduce a global variable named `cached_multiplications`, which is initialized as an empty dictionary. Then modify `Nimber`'s `.__mul__()` method to do the following:
   - Let `c=self.value` and `d=b.value` .
   - If `c < d`, swap the two.
   - Try to assign to `result` the value associated with the key `(c,d)` in `cached_multiplications`.
   - If this raises an error, catch it, perform the multiplication in the usual way, assign its value to `result`, and record it in the dictionary.
   - Finally, return `result`.

   Now try to generate the multiplication table that took too long in the text.
4. It is not unreasonable to imagine that someone might want to instantiate a `Nimber` using a set of powers of 2, saving us the trouble of having the computer figure it out. Modify the `__iter__()` function so that it tests `n`'s type. If `n` is not a tuple, list or set, it proceeds as before; otherwise, it converts `n` to a set, assigns that value to `self.powers`, then computes `self.value`. The correct implementation should provide the following results (using our original implementation of `__repr__()`, not the one assigned in a previous exercise):

```
sage: a = Nimber({1, 4})
sage: a
5
sage: a.power_repr()
{1, 4}
```

   Bonus: Add a check that the collection contains only powers of 2, so that the input `{1, 3}` would raise a *`ValueError`*.

Useful LaTeX

Many places of this text use LaTeX, and quite a few exercises and labs direct you to use it. This chapter summarizes all the commands you should need for what we do in this text, plus a few extra if you want to experiment. It will *not* serve as a general introduction to LaTeX, for which there are many, much more appropriate texts whose entire purpose is to expound on this wonderful tool.

Basic commands

Figure 24 gives a handy list of LaTeX commands. To use them in a string of text, you must enclose in dollar signs, e.g. `$x \in \mathbb{R}$`. You can also use `\(` and `\)` as delimiters; for example, `\(x \in \mathbb{R}\)`.

Delimiters

If you have a complex expression, you might want the delimiters (parentheses, brackets, etc.) to grow with it. You can see this in the difference between

$$(x^{x^{x^{x^{x^{\cdots}}}}}) \qquad \text{and} \qquad \left(x^{x^{x^{x^{x^{\cdots}}}}}\right).$$

To do this, place the command `\left` or the command `\right` immediately before the delimiter. **Every `\left` must match a `\right`, but if you only want one, you can place a dot after the other to indicate that you want nothing.**

The second expression above comes from typing:

```
$\left( x^{x^{x^{x^{x^{\iddots}}}}} \right)$.
```

You could obtain the interval $\left[2^{\ln 5}, \infty\right)$ by typing the following:

```
$\left[2^{\ln5},\infty\right)$.
```

That will looks better than `$[2^{\ln5},\infty)$` (which gives $[2^{\ln 5}, \infty)$), because in the first example the delimiters stretched to match the height of the objects involved.

Matrices

It's best to set matrix expressions on separate lines; we can do this using the delimiters `\[` ("begin math display") and `\]` ("end math display").

LaTeX notation	concept represented	example in LaTeX	result
`{...}`	grouping	see below	see below
`^`	superscript	`x^2`	x^2
`\sqrt`	square root	`sqrt{x^2+1}`	$\sqrt{x^2+1}$
`_`	subscript	`x_{\mathrm{next}}`	x_{next}
`\in`	element of	`x\in S`	$x \in S$
`\{...\}`	a set containing ...	`\{1,5,7\}`	$\{1,5,7\}$
`\frac{a}{b}`	fraction of a over b	`\frac{2}{5}`	$\frac{2}{5}$
`\alpha`, `\beta`, etc.	Greek letters	`2\pi`	2π
`\infty`	infinity	`(-\infty,\infty)`	$(-\infty,\infty)$
`\sin`, `\cos`, etc.	properly formatted functions	`\sin(\frac{pi}{6})`	$\sin(\frac{\pi}{6})$
`\rightarrow`, `\leftarrow`, etc.	arrows	`\lim_{x\rightarrow2}`	$\lim_{x\to 2}$
`\sum`, `\int`, `\prod`	sum, integral,	`\int_a^b f(x)dx`	$\int_a^b f(x)\,dx$
	product	`\lim_ {n\rightarrow\infty} \sum_{i=1}^n f(x_i) \Delta x`	$\lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x$[104]
`\leq`, `\geq`	$\leq, \geq$	`a\leq b`	$a \leq b$
`\notin`, `\neq`	$\notin, \neq$	`a\not\in S`	$a \notin S$
`\subset`, `\not\subset`	$\subset, \not\subset$	`S\not\subset T`	$S \not\subset T$
`\ldots`, `\cdots`	$\ldots, \cdots$	`\mathbb{N} =\{1,2,\ldots\}`	$\mathbb{N} = \{1,2,\ldots\}$
`\cap`, `\cup`	intersection, union	`S\cap(T\cup U)`	$S \cap (T \cup U)$
`\mathrm{...}`	don't italicize ...	`\mathrm{next}`	next
`\mathbb{...}`	write ... in "blackboard bold"	`\mathbb{R}`	$\mathbb{R}$
`\mathbf{...}`	write ... in bold font	`a\mathbf{F}`	$a\mathbf{F}$
`\mathcal{...}`	write ... in calligraphic font	`\mathcal{S}`	$\mathcal{S}$

FIGURE 24. Useful LaTeX markup for Sage

Matrices require a LaTeX environment, called an array. We start a matrix using the command `\begin{matrix}{`*format*`}`, and end it using the command `\end{matrix}`. For the *format*, we indicate whether you want the columns of the matrix aligned left (`l`), center (`c`), or right (`r`). We do this *for each column*, as you will see below.

Finally, we specify the entries of the matrix. Columns are separated by the ampersand (`&`), while rows are separated by a double backslash (`\\`).

For example, we can obtain the matrix

$$\begin{pmatrix} \cos x^2 - 1 & & & & x \\ & e^{2x} & & & \\ x+1 & & \sin x^2 - 1 & & \\ & 1-x & & e^{-2x} & \\ & & & x & \end{pmatrix}$$

by typing the following:

```
\[ \left(\begin{array}{clccr}
      \cos x^{2}-1 & & & & x\\
      & e^{2x}\\
      x+1 & & & \sin x^{2}-1\\
      & 1-x & & & e^{-2x}\\
      & & & x
\end{array}\right) \]
```

Part 2

Encyclopædia Laboratorica

Prerequisites for each lab

The labs in this part are organized according to the general field of mathematics to which they most apply. Here we indicate which chapters of the text are required for a student to be able to complete that lab *according to the approach we had in mind.* If the instructor has a different approach in mind, the prerequisites might not be so firm.

Various kinds of plots: This lab requires just the chapters "Basic computations" and "Pretty (and not-so-pretty) pictures."

Continued fractions: This lab requires the chapter "Repeating yourself inductively."

An important difference quotient: This lab requires just a few basic computations from the chapter of that name.

Illustrating Calculus: This lab requires calculus, which appears in the chapter "Basic computations," and plotting, which appears in the chapter "Pretty (and not-so-pretty) pictures."

Simpson's Rule: This lab requires calculus, which appears in the chapter "Basic computations," interactive worksheets, which appear at the end of the chapter "Writing your own functions," and **for** loops, which appears in the chapter "Repeating yourself with collections."

The Runge-Kutta method: This lab requires calculus, which appears in the chapter "Basic computations," and the most basic loops, which appear in the chapter "Repeating yourself definitely with collections."

Maclaurin coefficients: This lab requires calculus, which appears in the chapter "Basic computations," and **for** loops, which appears in the chapter "Repeating yourself with collections."

***p*-series:** This lab requires plots and definite loops.

Maxima and minima in 3d: This lab requires solving equations from the chapter of that name, creating 3d plots from the chapter of that name, and dictionaries, which appear in the chapter "Repeating yourself with collections."

One-to-one functions: This lab requires dictionaries, which appear in the chapter "Repeating yourself definitely with collections."

The Set Game: This lab requires collections, which appear in the chapter "Repeating yourself definitely with collections."

The number of ways to select *m* elements from a set of *n*: This project requires the chapter "Repeating yourself inductively."

Properties of finite rings: This lab requires just a few basic computations from the chapter of that name. It would be easier with for loops, which appear in "Repeating yourself definitely with collections," but the numbers are kept small enough so that it's not strictly necessary (be warned, though, that your students may hate you for assigning it before covering loops).

The geometry of radical roots: This lab requires just a few basic computations from the chapter of that name. It would be easier with for loops, which appear in "Repeating yourself definitely with collections," but the numbers are kept small enough so that it's not strictly necessary. (Unlike another lab, they shouldn't hate you if you assign this lab before covering loops.)

Lucas sequences: This lab requires the section "Eigenvalues and eigenvectors resolve a bunny dilemma" from the chapter "Repeating yourself inductively."

Introduction to Group Theory: This lab requires the chapters "Repeating yourself definitely with collections" and "Decision-making."

Coding theory and cryptography: Most of this lab can be done after "Repeating yourself inductively," but the bonus requires the section on Cython from "Advanced techniques."

Continued fractions: This lab requires the chapters "Repeating yourself definitely with collections" and "Decision-making."

General mathematics

Various kinds of plots

1. Create a new worksheet. Set the title to, "Lab: Various kinds of plots". Add other information to identify you, as necessary.

Part 1: Implicit plots.

2. Select a problem according to the following schema.

If your ID ends with…	…use this equation.
1,2	$x^3 + x = y^2$
4,5	$\left(x^2+y^2\right)^2 = \frac{25}{4}xy^2$
6,7	$x^4 - x^2y + y^4 = 1$
other	weird: see me

3. First, use at least 300 points to create and display an implicit plot of the equation on the region $[-2.25, 2.25] \times [-2.25, 2.25]$. The curve can be any color you like, as long as it's black.[105]
4. Choose an x-value in the interval $[-2, 2]$. Using your math skills (not Sage's, nor anyone else's — getting help from me is okay) find the equation of a line tangent to the curve at that point. Write the equation of this line in a text cell. Use basic LaTeX commands to make sure it looks nice. *Note:* There may be more than one y-value for that x-value, so there may be more than one line tangent to the curve for that value of x, but you need plot only one. Don't pick a point where the tangent line is horizontal or vertical; that would be cheating.
5. Verify your claim by combining the plot of the curve with a plot of the tangent line. The line should be red. Put a red dot at the point where the line and curve intersect. Make the dot large enough to stand out; I don't think the default dot size stands out.

Part 2: Parametric plots.

6. Plot a black Bézier curve. Remember that this involves four control points (x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3) and the equations

$$\begin{cases} x(t) &= x_0(1-t)^3 + 3x_1 t(1-t)^2 + 3x_2 t^2(1-t) + x_3 t^3 \\ y(t) &= y_0(1-t)^3 + 3y_1 t(1-t)^2 + 3y_2 t^2(1-t) + y_3 t^3 \end{cases}$$

   where $t \in [0, 1]$. You can choose the four control points at random, but don't use any you've seen before. Feel free to make a loop, if you can. Add red dashed lines that connect the 1st and 2nd control points (x_0, y_0) and (x_1, y_1), and the 3rd and 4th control points (x_2, y_2) and (x_3, y_3), allowing the viewer to see how the control points relate to tangent lines.
7. Use your graph to write a short description of the relationship between the control points and the lines.

Part 3: Animate!

8. Animate the Bézier curve:
   - Change the values of the 2nd and 3rd points 12 times. (Don't change them by very much, only a little, so the curve doesn't jump around too abruptly.)
   - For each change, create a new graph. Each graph should be identical to the one you created in step 6, except that the 2nd and 3rd control points are different.
   - Animate all the graphs using the `animate()` command, and `show()` it.

[105] That's a joke, but make it black, anyway. No apologies to Henry Ford.

Part 4: Make it look good!

9. Add text cells throughout the document that clearly delineate the parts. It should be clear where each part begins and ends. In Part 1, a text cell should state specifically what value of x you chose; in Part 2, a text cell should state specifically what control points you used; in Part 3, a text cell should state clearly the sequence of adjustments made to the 2nd and 3rd control points. You should also add some explanatory text before each plot: "This is the first plot, with points…", "This is the second plot, with points…", … and finally "This is the animation of all the frames." **Use LaTeX for mathematical expressions and statements;** don't forget the reference of useful LaTeX commands on the class website.

Calculus and Differential Equations

An important difference quotient

1. Create a new worksheet. Set the title to, "Lab: An important difference quotient".
2. Create a text cell (shift+click on blue line). Write your name, and this semester. Change it to some color. You can choose any color you like, as long as it's not black. — Or white. White would be bad, too.
3. In the first computational cell, use the `var()` command to define variables x and h.
4. In the next few computational cells, have Sage expand the product $(x+h)^n$ for several values of n. The `expand()` command was shown on p. , and you should pick several sequential values of n.
5. In a text cell that follows these computational cells, make a conjecture as to what the last two terms of $(x+h)^n$ will always be, and what common factor the remaining terms always have.
6. In Calculus, you are told that
   - the definition of $\frac{d}{dx}f(x)$ is
   $$\lim_{h\to 0}\frac{f(x+h)-f(x)}{h},$$
   - and $\frac{d}{dx}x^n = nx^{n-1}$.

   In a final text cell, explain why your answers to steps 4 and 5 demonstrate this fact.

Illustrating Calculus

1. Create a new worksheet. Set the title to, "Lab: Illustrating Calculus". Add other information to identify you, as necessary.
2. Select a problem according to the following schema.

If your ID ends with…	…use this function…	…over this interval.
0,1,2	$f(x) = \sin x$	$\left[-\frac{\pi}{3}, \frac{2\pi}{3}\right]$
3,4,5	$f(x) = \cos x$	$\left[-\frac{\pi}{3}, \frac{2\pi}{3}\right]$
6,7,8,9	$f(x) = \tan x$	$\left[-\frac{\pi}{6}, \frac{\pi}{3}\right]$

Part 1: Derivatives.

3. Find the equation of the line tangent to f at $x = \pi/4$. Any computation that can be done with Sage should be evident in your worksheet!
4. Combine the plots of both f and the line tangent to it over the interval given. The curve for f should be black, and have a width of 2. The line should be blue, and have a width of 2.
5. Create an animation with at least 8 frames that shows the approach of the secant line to the tangent line as $x \to \pi/4$ from the left. Reuse the plots of f and the tangent line from above. The secant lines should be red, and have a width of 1. You are free to choose any points you like for the secant, just so long as $x \to \pi/4$ from the left. When you are done, your animation should resemble the one on the course syllabus: for instance, the secant line should proceed back and forth, not just in one direction.

Part 2: Exact integrals.

6. Compute the area between f and $g(x) = 1 - x^2$ over the interval given.
7. Combine the plots of both f and g over the interval given. Fill in the area between f and g. The curves for both f and g should be black, with a width of 2. The filling can be any color you like, but make it half-transparent. Add a text label inside the filling which contains the area. Use LaTeX so that the text label looks nice.

Part 3: Approximate integrals.

8. Go back to your Calculus text and review the calculation of arclength with integrals. Write the formula, and in a text cell explain briefly what tool from high school geometry is used to derive the formula.
9. Use Sage to ***approximate*** the arclength of the ellipse $x^2/4 + y^2/9 = 1$. Limit the approximation to 5 sample points, and round your answer to 5 decimal places.
10. Repeat problem 9, this time limiting the approximation to 10 sample points. What part of the answer indicates that you have a more accurate answer?

Part 4: BONUS! (for those with exceptional time and/or motivation). Animate approximations of the arclength, where each frame shows

- no axes;
- the ellipse, in black, with the curve's width 2;
- six frames of 5 dashed line segments, then 6 dashed line segments, …, and finally 10 dashed line segments, in red, of width 1;

- a text label in each frame with the corresponding approximation to the arclength, at the center of the ellipse, in black.

This bonus is worth as much as *the entire assignment*. If you wish, you may do Part 4 instead of Parts 1–3. Be sure you know what you are doing; this can take a while.

Simpson's Rule

In Calculus II you should have learned Simpson's rule to approximate the value of an integral over an interval.

1. In a text cell,
   (a) state the formula for Simpson's Rule;
   (b) briefly summarize the idea that gives rise to the formula.
2. Write a Sage function that accepts as inputs an integrable function f, the endpoints a and b of an interval, and a positive integer n, then uses Simpson's Rule to estimate $\int_a^b f(x)\,dx$.
3. Write Sage function that accepts as inputs an integrable function f, the endpoints a and b of an interval, and a positive integer n, then illustrates Simpson's Rule with a plot of both f and the figures whose areas are used to estimate the value of the integral. Have the function invoke the solution to the previous question, and print the area as a LaTeX'd label somewhere on the curve.
4. Make an interactive function for the solution to the previous question. In addition to accepting f, a, b, and n as inputs, the interactive function should allow the user to select the color for the figures used to estimate the value of the integral.

The Runge-Kutta method

The Runge-Kutta method of approximating the solution to a differential equation is similar to Euler's Method. The following pseudocode describes Runge-Kutta:

algorithm *Runge_Kutta*
inputs
- df, the derivative of a function
- (x_0, y_0), initial values of x and y
- Δx, step size
- n, number of steps to take

do
  let $a = x_0$, $b = y_0$
  repeat n times
    let $k_1 = df\,(a, b)$
    let $k_2 = df\,(a + {}^{\Delta x}/_2, b + {}^{\Delta x}/_2 \cdot k_1)$
    let $k_3 = df\,(a + {}^{\Delta x}/_2, b + {}^{\Delta x}/_2 \cdot k_2)$
    let $k_4 = df\,(a + \Delta x, b + \Delta x \cdot k_3)$
    add ${}^{\Delta x}/_6(k_1 + 2k_2 + 2k_3 + k_4)$ to b
    add Δx to a
  return (a, b)

Implement Runge-Kutta in Sage code. Try it on the same differential equation and initial condition we tried in Euler's method, and compare the result.

Maclaurin coefficients

In Calculus III you should have learned about Maclaurin expansions. **Maclaurin coefficients** are the numerical coefficients of the terms in the Maclaurin expansion.

1. Write a function `maclaurin_coeffs()` that accepts as inputs a function `f` and a positive integer `d`. It returns a list of the Maclaurin coefficients of $f(x)$ up to and including degree d. *Hint:* You should probably use the Sage functions `taylor()` and `coefficient()`.
2. Test your function with known elementary functions such as e^x, $\sin x$, and $\cos x$. Verify that your answers correspond to those that appear in a calculus textbook.
3. Use your function to find the Maclaurin coefficients of $f(x) = \arcsin^2(x)$ up to degree 20.
4. Look at the nonzero Maclaurin coefficients of $f(x)$ for degree 6 and higher. What common factor do you see in the numerators?
5. Using your answer to the previous problem, modify your invocation to

   ```
   maclaurin_coeffs(arcsin(???)**2, 20)
   ```

   so that all numerators are 1. Save this list as `L1`. Why does it make sense that you can do this?
6. From `L1`, use a list comprehension to make `L2`, a list of the reciprocals of the nonzero coefficients.
7. What is the smallest nontrivial common factor of the elements of `L2`? Write another list comprehension to create `L3`, which contains the elements of `L2`, but divided by this common factor.
8. Look up this new list at the Online Encyclopedia of Integer Sequences. What number sequence do you find?
9. Work backwards through your last few steps to answer the original problem:

$$\arcsin^2 x = \sum_{k=1}^{\infty} ??? \, .$$

   (We haven't *proved* that this is the correct series; we've only conjectured it based on a few coefficients.)

p-series

In the following, $\lceil x \rceil$ denotes the ceiling of x, and $\lfloor x \rfloor$ denotes the floor of x.

1. Write code to compute the sum
$$\sum_{k=1}^{n} \frac{1}{k^p},$$
and use loops to generate lists of the first 50 sums, for $p = 1, 2, 3$.
2. Generate plots of these sums for $p = 1, 2, 3$ by plotting the step functions
$$s_p(x) = \sum_{k=1}^{\lceil x \rceil} \frac{1}{k^p}$$
on the interval $[1, 50]$. Using your plots, guess whether or not the p-series
$$\sum_{k=1}^{\infty} \frac{1}{k^p}$$
converges, for $p = 1, 2, 3$.
3. Generate plots of the sequences for $p = 1, 2, 3$ by plotting the functions
$$f_p(x) = \frac{1}{\lfloor x \rfloor^p},\ g_p(x) = \frac{1}{x^p},\ h_p(x) = \frac{1}{\lceil x \rceil^p}$$
together (using different colors) on the interval $[1, 10]$.
4. What do these plots suggest about
$$\sum_{k=2}^{n} \frac{1}{k^p},\ \int_1^n \frac{1}{x^p}\,dx,\ \text{and}\ \sum_{k=1}^{n-1} \frac{1}{k^p},$$
and what do they suggest about
$$\sum_{k=1}^{\infty} \frac{1}{k^p}\ \text{and}\ \int_1^{\infty} \frac{1}{x^p}\,dx?$$
5. Using your answers to problem 4, determine whether or not
$$\sum_{k=1}^{\infty} \frac{1}{k^p}$$
converges, for $p = 1, 2, 3$.

Maxima and Minima in 3D

1. Write a function named `critical_points()` which accepts as input a twice-differentiable function f in two variables x and y, and returns a list of points (x_i, y_i) that are critical points of f. Make sure you find only real-valued critical points; you can use the `imag_part()` function to test each point.
2. Write a second function, `second_derivs_test()`, which accepts as input a twice-differentiable function f in two variables x and y, as well as a critical point (x, y), and returns one of the following:
   - `'max'` if the point is a maximum;
   - `'min'` if the point is a minimum;
   - `'saddle'` if the point is a saddle point; and
   - `'inconclusive'` if the second derivative test is inconclusive.
3. Write a third function, `plot_critical_points()`, which accepts as input a twice-differentiable function f in two variables x and y, and returns a 3-dimensional plot of the following:
   - $f(x, y)$ in the default color;
   - local maxima in red;
   - local minima in yellow;
   - saddle points in green;
   - other critical points in black.

   Use a dictionary to look up the appropriate color for the points. Use Sage's `max()` and `min()` functions to determine the plot's minimum and maximum x- and y-values to make sure that all the critical points appear in the plot.

Linear Algebra

Algebraic and geometric properties of linear systems

This lab requires linear algebra, which appears in the chapter "Solving equations."

A system of equations.

1. Suppose your university ID is $abcdef$. For instance, if your ID is 123456, then $a = 1$, $b = 2$, $\ldots$, $f = 6$. Use that to define the following system of equations:

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$$

2. Plot each equation on the x-y plane. Make one line blue, the other red. (It doesn't matter which is which.) In all likelihood, the two lines will not be parallel, but will intersect at exactly one point. Adjust the x and y axes so that this intersection is visible. *If the lines are parallel or coincident, modify one of a, b, $\ldots$, f so that the lines intersect at exactly one point.*
3. Use Sage to find the *exact* solution to the system of equations. This should be a point (x_0, y_0). Create a *new* plot with both lines (still different colors) and a big, fat yellow point on top of their intersection. (Not too fat, but fat enough to see.) Make sure the point lies *on top* of the lines.

An invariant.

4. Define variables for a, b, $\ldots$, f. Use that to define the following system of equations *without any numbers at all:*

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$$

5. Solve the system above for x and y.
6. You may have noticed that the solution has a common denominator. (If you didn't notice it, this would be a good time to notice.) What is sort-of-amazing, but not-really-that-amazing about that denominator?
   Hint: Think about some basic matrix operations on the matrix that corresponds to the left sides of the original equations. It's something you should have computed in high school algebra, and *definitely* would have computed in linear algebra.
7. Suppose that a and b have known, concrete values. Use your answer to #6 to explain why the existence of a solution dependes *entirely* on d and e — and has nothing to do with c and f!!!
   Hint: I'm asking about the *existence* of a solution, not the *value* of the solution once it exists. The value most certainly depends on c and f, but the existence depends only on d and e. So the question asks you to use the previous answer to explain this question of *existence,* not *value.*
8. Let a and b have the same values they had in #1. Let $g(x)$ be the denominator of the solution found in #6. Substitute these values of a and b into $g(x)$. You should get a linear eauation in two variables, d and e. Change d to x and e to y.
9. Plot the line determined (no pun intended) by this equation, which has y-intercept `0`. Also plot the original equation $ax + by = c$. How are these two lines related? (It may not be clear unless you use the plot option `aspect_ratio=1` to make it clear.)

10. Would this relationship between the two lines hold *regardless of the values of a and b?* That is, if the only thing you changed were a and b in both the line and the function $g(x)$, would the line $ax + by = c$ still have the same relationship to the corresponding value of $g(x)$?

A point at infinity.

11. We return to your original system of equations. Define lists D and E so that the values in D move in 10 steps from d *almost* to a and the values in E move in 10 steps from e *almost* to b. (The difference between the final values and a or b should be miniscule.) For instance, if your original system is
$$\begin{cases} 1x + 2y = 3 \\ 4x + 5y = 6 \end{cases}$$
then you could have something like $D = (4, 3, 2, 2.5, 2.1, 2.01, 2.001, 2.0001, 2.00001, 2.000001)$ and $E = (5, 4, 3, 2.5, 2.1, 2.01, 2.001, 2.0001, 2.00001, 2.000001)$.
12. Loop through the lists D and E to create a plot for each system
$$\begin{cases} ax + by = c \\ d_i x + e_i y = f \end{cases}.$$
(Here, d_i and e_i refer to the ith elements of D and E, respectively.) Combine the plots into a sequential animation. Animate it.
13. Describe the *eventual* relationship between the lines, especially if you let $d_i \to a$ and $e_i \to b$.
14. The field of **projective geometry** introduces a new point so that *all lines, even parallel lines, intersect at least once.* Use the animation to explain why it is appropriate to call this point a "point at infinity."
Hint: You may want to adjust the minimum and maximum x- and y-values of your animation to see this more clearly.

Transformation matrices

This lab requires linear algebra, which appears in the chapter "Solving equations."

Define two symbolic variables `a` and `b`. Let

$$A = \begin{pmatrix} \cos a & \sin a \\ -\sin a & \cos a \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} \cos b & \sin b \\ -\sin b & \cos b \end{pmatrix}.$$

1. Compute AB in Sage. Use your knowledge of trigonometry to specify a simpler form than what Sage gives. Use LaTeX to write this simpler form in a text box below the computation of AB.
2. Extract, and give a name to, the entry in the first row and column of AB.

$$\begin{pmatrix} \textbf{this one} & \text{not this one} \\ \text{nor this one} & \text{certainly not this one} \end{pmatrix}$$

   Try to give it a meaningful name, not just x, which is a bad idea, anyway.
3. In a new computational cell, type the name you just created for that entry. Then, type a dot (period). Then press tab. Look through the names that pop up for a command that might reduce the trigonometric expression to something simpler. Use that command to confirm your result in part (b).
4. Define a new matrix, C, obtained by substituting the value $a = \pi/3$ into A.
5. Let $\mathbf{v}$ be the vector defined by $(5, 3)$.
6. Compute the vectors $C\mathbf{v}$, $C^2\mathbf{v}$, $C^3\mathbf{v}$, $C^4\mathbf{v}$, $C^5\mathbf{v}$.
7. Plot the vectors $\mathbf{v}$, $C\mathbf{v}$, $C^2\mathbf{v}$, $C^3\mathbf{v}$, $C^4\mathbf{v}$, $C^5\mathbf{v}$ in different colors (your choice). Plot them as arrows or points, but not as step functions. Please combine them into a single plot, rather than making six different plots.
8. What is the geometric effect of multiplying $\mathbf{v}$ by the matrix C repeatedly? What do you predict $C^{60}\mathbf{v}$ would look like? What about $C^{1042}\mathbf{v}$?

Visualizing eigenvalues and eigenvectors

This lab requires linear algebra, which appears in the chapter "Solving equations."

Let M be the 2×2 matrix defined by the first four numbers in your student ID.

1. Use Sage to find the eigenvalues and eigenvectors of M. There should be two distinct eigenvectors, but it's possible you'll get only one, with a multiplicity of 2. In that case, modify the entries of your matrix *very slightly* so that it produces two distinct eigenvectors.
2. Extract the eigenvectors and name them `v1` and `v2`.
   Hint: For full credit, *extract* them using the bracket operator; do not define new vectors.
3. Let `K` be the list of vectors obtained from the 10 products $M^i v_1$ for $i = 1, 2, \ldots, 10$. Use a **for** loop to define this list; you will lose the vast majority of points on this part if you do them one at a time.
   Hint: **Be careful.** The expressions `range(10)` and `xrange(10)` commands start at `0`; you want to start at 1.
4. Let `L` be the list of vectors obtained from the 10 products $M^i v_2$ for $i = 1, 2, \ldots, 10$. Use a **for** loop to define this list; you will lose the vast majority of points on this part if you do them one at a time.
5. Define `p=Graphics()` and `q=Graphics()`. (This defines an "empty" plot.)
6. Use a **for** loop to add a plot of each vector in `K` to `p`. The first vector should be red; successive vectors should shift increasingly towards blue, until the last one is completely blue. Here's how I'd suggest shifting from red to *green:*
   ```
   for i in xrange(len(L)):
   p += plot(L[i],color=((10-i)/10,i/10,0),zorder=-i)
   ```
7. Use a **for** loop to add a plot of each vector in `L` to `q`. Vectors should shift from purple (red and blue) to yellow (red and yellow).
8. Show `p` in one cell; show q in another.
9. Describe how the result is consistent with what we said in class about eigenvalues and eigenvectors. If it isn't clear what I mean, try adjusting the `xmin`, `xmax`, `ymin`, `ymax` values to see a smaller portion of the graph.

Least squares averaging

This lab requires linear algebra, which appears in the chapter "Solving equations."

A system of two linear equations in two variables will find an exact solution for exact data. Unfortunately, the real world often lacks exact data, so scientists and engineers compensate by collecing extra data and "averaging" the results. For instance, in the case of two unknowns x and y, we might take m measurements that correspond to the system

$$\begin{aligned} x + a_1 y &= b_1 \\ x + a_2 y &= b_2 \\ &\vdots \\ x + a_m y &= b_m. \end{aligned}$$

The best solution (x_0, y_0) minimizes the error; that is, it minimizes

$$\left\| \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} - \begin{pmatrix} x_0 + a_1 y_0 \\ \vdots \\ x_0 + a_m y_0 \end{pmatrix} \right\|^2 = [b_1 - (x_0 + a_1 y_0)]^2 + \cdots + [b_m - (x_0 + a_m y_0)]^2.$$

We can minimize this in the following way: let

$$A = \begin{pmatrix} 1 & a_1 \\ \vdots & \vdots \\ 1 & a_m \end{pmatrix} \quad , \qquad X = \begin{pmatrix} x \\ y \end{pmatrix} \quad , \qquad B = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

The system of equations above translates to

$$AX = B.$$

If we multiply both sides on the left by A^T, we have

$$\left(A^T A\right) X = A^T B.$$

The product of A^T and A is a square matrix, hopefully invertible. If it is invertible, we can solve it by multiplying on both sides:

$$X = \left(A^T A\right)^{-1} A^T B.$$

It can be shown that this X minimizes the error (though we will not show it).

1. Write pseudocode for a function that accepts two lists of corresponding measurements $a_1, \ldots a_m$ and $b_1, \ldots, b_m$ and solves for $x_1, \ldots, x_m$.
2. Implement the pseudocode as a Sage function named `least_squares_2d()`.
3. Write a Sage function that accepts two lists of corresponding measurements $a_1, \ldots a_m$ and $b_1, \ldots, b_m$, plots each point (a_i, b_i), and plots the line that lies along the solution vector (x_0, y_0). Optionally, the function should accept two additional arguments `color_points` and `color_line`, which respectively indicate the colors of the points and the color of the line. Be sure to use Sage's `min()` and `max()` commands to set `ymin`, `ymax`, `xmin`, and `xmax` appropriately.

Bareiss' Method

This lab requires linear algebra, which appears in the chapter "Solving equations."

The Bareiss method to compute a determinant can be describe in pseudocode as follows:

algorithm *Bareiss_determinant*
inputs
- an $n \times n$ matrix M

outputs
- $\det M$

do
  let D be a copy of M
  let $a = 1$
  for $k \in \{1, \ldots, n-1\}$
    for $i \in \{k+1, \ldots, n\}$
      for $j \in k+1, \ldots, n$
        let $d_{i,j} = \frac{d_{i,j}d_{k,k} - d_{i,k}d_{k,j}}{a}$
    let $a = d_{k,k}$
  return $D_{n,n}$

1. Implement this pseudocode as a Sage program, and test it on the following matrices. When it produces a result, is the result actually correct? If it doesn't produce a result, what exactly does it do? In this latter case, study the entries of the matrix to see what goes wrong.

$$\text{(a) } M = \begin{pmatrix} 3 & 2 & 1 \\ 5 & 4 & 3 \\ 6 & 3 & 4 \end{pmatrix} \qquad \text{(b) } M = \begin{pmatrix} 1 & -4 & 1 & 2 \\ -1 & 4 & 4 & 1 \\ 3 & 3 & 3 & 4 \\ 2 & 5 & 2 & -1 \end{pmatrix} \qquad \text{(c) } M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \\ 1 & 1 & 3 & 3 \end{pmatrix}$$

2. As with the Gaussian method, we can salvage a computation of the Bareiss algorithm when one encounters failure — and in more or less the same way. Modify your Bareiss implementation to take this into account, then test it on the three matrices again to make sure it behaves correctly.

Dodgson's Method

This lab requires linear algebra, which appears in the chapter "Solving equations."

Dodgson's Method is an "intuitively easy" way to compute a determinant.[106] The following pseudocode describes it precisely. (Here, numbering of rows and columns starts at 1, rather than at 0 as in Sage.)

algorithm *Dodgsons_method*
inputs
- M, an $n \times n$ matrix

outputs
- $\det M$

do
  let L be an $(n-1) \times (n-1)$ matrix of 1's
  for $i \in \{1, \ldots, n-1\}$
    let m be the number of rows of M
    let N be an $(m-1) \times (m-1)$ matrix
    for $j \in \{1, \ldots, m-1\}$
      for $k \in \{1, \ldots, m-1\}$
        let $N_{j,k} = \left(M_{j,k} \times M_{j+1,k+1} - M_{j+1,k} \times M_{j,k+1}\right) / L_{j,k}$
    let L be the $(m-2) \times (m-2)$ submatrix of M starting in row 2, column 2
    replace M with N
  return M

1. Implement this pseudocode as a Sage program, and test it on the following matrices. (Something will go wrong with one of them.)

(a) $M = \begin{pmatrix} 3 & 2 & 1 \\ 5 & 4 & 3 \\ 6 & 3 & 4 \end{pmatrix}$ (b) $M = \begin{pmatrix} 1 & -4 & 1 & 2 \\ -1 & 4 & 4 & 1 \\ 3 & 3 & 3 & 4 \\ 2 & 5 & 2 & -1 \end{pmatrix}$ (c) $M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \\ 1 & 1 & 3 & 3 \end{pmatrix}$

2. When it produces a result, is the result actually correct?
3. If it doesn't produce a result, what exactly does it do?
4. In this latter case, try to perform the computation by hand. Explain what goes wrong.[107]

[106] Named for its inventor, Charles Lutwidge Dodgson, a deacon of the Church of England who also held a chair in mathematics. He is better known for a pair of children's books he wrote under the pen name "Lewis Carroll," *Alice and Wonderland* and *Alice Through the Looking Glass*.

[107] Don't pursue the temptation to fix it unless you want a challenging research project.

Discrete Mathematics

One-to-one functions

In this lab, we view a dictionary as a map or function from a set A, the **domain**, the set of the dictionary's keys, to another set B, the **range**, sometimes called the **codomain**. The range may actually contains more elements than those than appear in the dictionary's values.

1. Write a Sage function which accepts a dictionary D as input and returns `True` if D is a one-to-one function and `False` otherwise.
2. Write a Sage function which accepts a dictionary D as input and returns `True` if D is a one-to-one function; if not, it returns *both* `False` *and* a counterexample.
3. Write a Sage function which accepts a dictionary D and a set B as inputs and returns `True` if D is onto B, and `False` otherwise. In the latter case, it should also return all elements of B that are not values of D.
4. Write a Sage function which accepts a dictionary D and a set B as inputs and returns `True` if D is both one-to-one and onto B, and `False` if not. In the latter case, it prints (not returns!) which property fails and the appropriate counterexample.

The Set Game

The game Set consists of 81 cards, each of which has four properties:

- The number of objects on the card is 1, 2, or 3.
- The color of the object(s) is identical: red, green, or purple.
- The shading of the object(s) is identical: solid, striped, or empty.
- The shape of the object(s) is identical: diamonds, ovals, or squiggles.

Notice that each property has 3 possibilities, explaining why there are $81 = 3^4$ cards.

The object of the game is to identify as many "sets" there are in a group of cards. A "set" is any collection of cards in which each property is the same or different.

1. Write a Sage function `make_cards()` which creates and returns a list of all the cards. Each card should be a tuple of four integers, where each integer indicates the property. Thus, $(1,1,3,2)$ would represent a card with 1 red empty oval.
2. Write a Sage function `is_set()` which accepts one argument, `L`, which is a list of 3 cards. It returns *`True`* if `L` forms a set, and *`False`* otherwise. You may want to have this function call other functions which test whether a property is the same among all cards or different among all cards.
3. In a new cell, type `import itertools`. Write a Sage function `has_set()` which accepts one argument, `B`, a list of at least 3 cards, and returns *`True`* if `B` has a set, and *`False`* otherwise. You can use the command `BI = itertools.combinations(B,3)` to create an iterator that will choose all possible combinations of 3 cards from `B`. You can then loop over `BI` using a `for` loop.
4. Write a Sage function `prob_set_in(N)` which takes an integer of at least 3 and returns the probability that a randomly selected board of size N contains a set. You will not want to try this on a large value of N, as it will take a ***very long time*** to run. Test it only with $N = 3$ (a few seconds, max) and $N = 4$ (a couple of minutes, max). You can use the command `CN = itertools.combinations(CL,N)` to create an iterator that will choose all possible combinations of N cards from a list `CL` (in this case, a list of cards).

The number of ways to select m elements from a set of n

Suppose you have a bag with n distinct objects, and you want to select m of them. Such a choice of balls is called a *combination*, as opposed to a *permutation* where order matters. There are many applications where we want to count the number of ways to select m balls from a bag of n distinct objects.[108] We write this as ${}_nC_m$ or $\binom{n}{m}$. The formula for this value is

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}.$$

1. Write a Sage function that accepts as input n and m and computes $\binom{n}{m}$.
2. Use your function to compute $\binom{n}{m}$ for $n = 2, 3, \ldots, 8$ and $m = 0, \ldots, n$.
3. Do you notice any pattern to the numbers? *Hint:* Look back at Pascal's Triangle.
4. Describe how one could use combinations to implement a function that computes Pascal's Triangle *without* using recursion.

[108] For instance, the PowerBall lottery awards prizes according to whether one selects the same five numbered balls out of 69 total balls as are selected on a TV show, not necessarily in the same order. (There's another criterion as well, but at the very least one has to know the number of ways to select 5 of 69 balls.

Algebra and Number Theory

Properties of finite rings

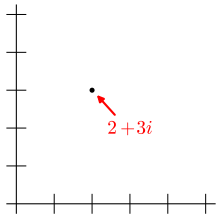
1. Create a new worksheet. Set the title to, "Lab: Properties of finite rings". Add other information to identify you, as necessary.

Reread the Section on "Algebra" in Lecture 2, "Introduction to Sage."

2. Create a section labeled, "Modular arithmetic: demonstration," then:
   (a) Define a ring R to be $\mathbb{Z}_{10}$, the finite ring of 10 elements.
   (*Hint:* The revised notes show a simpler way to do this than the in-class demonstration.)
   (b) Define m to be the value of 2 in R, and compute $1 \times m$, $2 \times m$, $3 \times m$, ..., $10 \times m$.
   (*Hint:* If your answer to the last product is 20, you're doing it wrong. You have to convert 2 to a value of the ring R. The notes show how to do that.)
   (c) Define n to be the value of 3 in R, and compute $1 \times n$, $2 \times n$, $3 \times n$, ..., $10 \times n$.
   (d) Define r to be the value of 5 in R, and compute $1 \times r$, $2 \times r$, $3 \times r$, ..., $10 \times r$.
   (e) Define s to be the value of 7 in R, and compute $1 \times s$, $2 \times s$, $3 \times s$, ..., $10 \times s$.
   (f) Define t to be the value of 9 in R, and compute $1 \times t$, $2 \times t$, $3 \times t$, ..., $10 \times t$.
3. For the following, write your answer into a *text* box at the end of the worksheet. The top of the textbox should have the heading, "Modular arithmetic: analysis."
   (a) Notice that $10 \times 2 = 10 \times 3 = \cdots = 10 \times 9$ in this ring. Why does that make sense? (Your answer should address what I said in class about arithmetic in $\mathbb{Z}_n$, illustrated on slide 44 with $\mathbb{Z}_7$.)
   (b) Which of m, n, r, s, t lists *all* the numbers from 0 to 9?
   (c) What property do the numbers you listed in (b) share that the other numbers do not?

The geometry of radical roots

REMARK. In this assignment, we view all solutions as complex numbers $a+bi$, where $i^2 = -1$. When you are asked to "plot $a+bi$ on the complex plane," plot it as the point (a,b). So, for instance, a plot of the complex number $2+3i$ would give you the point $(2,3)$:



1. Create a new worksheet. Set the title to, "Lab: The geometry of radical roots". Add other information to identify you, as necessary.
2. Use Sage to solve the equation $x^2 - 1 = 0$. Plot all solutions on one complex plane. It's not very interesting, is it?
3. Use Sage to solve the equation $x^3 - 1 = 0$. Plot all solutions on one complex plane, not the same as before. This is a little more interesting.
4. Use Sage to solve the equation $x^4 - 1 = 0$. Plot all solutions on a third complex plane. This might be a little boring, again.
5. Use Sage to solve the equation $x^5 - 1 = 0$. Plot all solutions on a fourth complex plane. (You will probably need to use `real_part()` and `imag_part()` here.) This figure should be arresting, especially if you connect the dots, but don't do that; just plot the points, note the result, and move on.
6. Can you guess? Yep, use Sage to solve the equation $x^6 - 1 = 0$. Plot all solutions on yet another complex plane. This is probably getting boring, if only because you can probably guess the result not only of this one, but also of…
7. Use Sage to solve the equation $x^7 - 1 = 0$. Plot all solutions on another complex plane. If you understand the geometric pattern, move on; if not, complain. Or, if you want to save time, you can probably guess that I'll tell you to plot the solutions for a few more examples of $x^n - 1 = 0$, only with larger values of n. Sooner or later, you should perceive a geometric pattern.
8. In a text cell, explain why the geometric pattern of the images justifies the assertion that the solutions to $x^n - 1$ all have the form
$$\cos\frac{2\pi k}{n} + i\sin\frac{2\pi k}{n}.$$
Be sure to explain what on earth k and n have to do with it. If it helps, look first at the parametric plot of $\cos(2\pi t) + i\sin(2\pi t)$ for $t \in [0,1]$, remembering as before that you plot $a+bi$ as (a,b).

   Before you raise your hand in panic, take a moment, then a deep breath, and think back to the meaning of sine and cosine. Look in a textbook if you have to. You can do this, honest!

   Likewise, don't ask me how to create a parametric plot. *We went over it,* so if you don't remember, look it up in the notes.
9. Define $\omega = \cos\frac{2\pi}{6} + i\sin\frac{2\pi}{6}$. Using a **for** loop in Sage, compute ω, ω^2, …, ω^6. Plot each number on the complex plane, and describe how the result is consistent with your answers to #6 and #8.

10. Now suppose $\omega = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$. *By hand,* work out an explanation why

$$\omega^k = \cos\frac{2\pi k}{n} + i\sin\frac{2\pi k}{n}. \tag{1}$$

Type this explanation in your Sage worksheet, using LaTeX to make it all purty-like. If you know proof by induction, that will work. Otherwise, take the following approach:

- Explain why equation (1) is true for $k = 1$.
- Expand ω^2 and use some trigonometric identities you're supposed to remember to simplify to the value of equation (1) for $k = 2$.
- Do the same for ω^3.
- Finally, point to a pattern in the previous two steps that you can repeat *ad infinitum*, so that if equation (1) is true for *some* value of k, it's also true for the *next* value of k.

11. Time to up our game.[109] Experiment with $x^n - a$ for some nice values of a and a sufficiently large value of n. What do you see? Formulate a conjecture as to the geometric form of these solutions.

    (The term "nice values of a" it can mean one of two things. *First,* it can mean numbers of the form $a = b^n$; for instance, if $n = 6$, then you could let $b = 3$, and you would have $a = 3^6 = 729$. You'd then work with the equation $x^6 - 729$, which is not as scary as it looks. Honest! *Second,* you could take the Bob Ross approach, in which case any number is a nice value because, really, all numbers are nice, happy numbers. That's harder, though.)

12. As before, make it all look nice, with sectioning, commentary in text boxes, at least a *little* LaTeX, etc.

[109] I'd rather say, "Let's generalize," as "up our game" just don't roll off the tongue in the same pleasant way, but They tell me that "up our game" is more current. Don't ask who They are, and don't look at me like I'm crazy. Of course I am.

Lucas sequences

The **Fibonacci sequence** is defined as

$$f_1 = 1, \quad f_2 = 1, \quad f_{n+2} = f_n + f_{n+1}.$$

The Fibonacci numbers are one example of what mathematicians now call a **Lucas sequence**. (More information at the link.) We usually define Lucas sequences recursively, but you can find a "closed formula" in a manner similar to what we did in class for the Fibonacci sequence.

Let a, b, c, and d be the first two numbers of your student ID. If any two numbers are the same, change them so that all four numbers differ. The sequence

$$\ell_1 = a, \quad \ell_2 = b, \quad \ell_{n+2} = c\ell_n + d\ell_{n+1}$$

is a Lucas sequence that we'll call the "*[insert your last name here]* sequence."

1. In a Sage text cell, state the definition of the *[insert your last name here]* sequence. Use LaTeX!
2. In the same cell, list the first five numbers of the *[insert your name here]* sequence.
3. Define a matrix L and a vector $\mathbf{v}$ which generate the sequence. For instance, if the first four digits of your ID are 1, 2, 8, and 9 then

$$L = \begin{pmatrix} 9 & 8 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{v} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

   Compute $L\mathbf{v}$, $L^2\mathbf{v}$, $L^3\mathbf{v}$, $L^4\mathbf{v}$, and $L^5\mathbf{v}$ in Sage, and compare the results to #2. If they differ, either #2 or #3 is wrong. *Or* I have a typo. Ask me, and/or fix it before continuing.
4. Compute L's "eigendata." Extract the eigenvectors and eigenvalues and have Sage convert them to *radical* form. (Numbers should no longer end in question marks.)
5. Construct matrices Q and Λ such that $L = Q\Lambda Q^{-1}$. Use Sage to verify that $L = Q\Lambda Q^{-1}$.
6. Construct the matrix $M = \left(Q\Lambda Q^{-1}\right)^n$.
   Hint: The lecture notes discuss this; it requires some knowledge of linear algebra.
7. Use the product of M and $\mathbf{v}$ to find the closed form of the *[insert your name here]* sequence.
   Hint: Again, the notes should come in handy here if you need help.
8. Use the closed form to compute the first five numbers of the *[insert your name here]* sequence, and compare your results to what you found in #2 and #3. If they differ, you have a problem or I have a typo; ask me and/or fix it!

Introduction to Group Theory

Background.

DEFINITION. If a set S and an operation $\otimes$ satisfy the **closure**, **associative**, **identity**, and **inverse** properties, then we call S a **group** under $\otimes$. These properties are defined in the following way:

- *closure:* $x \otimes y \in S$ for all $x, y \in S$;
- *associative:* $x \otimes (y \otimes z) = (x \otimes y) \otimes z$ for all $x, y, z \in S$;
- *identity:* we can find $\iota \in S$ such that $x \otimes \iota = x$ and $\iota \otimes x = x$ for any $x \in S$;
- *inverse:* for any $x \in S$, we can find $y \in S$ such that $x \otimes y = y \otimes x = \iota$.

EXAMPLE. The integers $\mathbb{Z}$ form a group under addition, because

- adding any two integers gives you an integer ($x + y \in \mathbb{Z}$ for all $x, y \in \mathbb{Z}$);
- addition of integers is associative;
- there is an additive identity ($x + 0 = x$ and $0 + x = x$ for all $x \in \mathbb{Z}$); and
- every integer x has an additive inverse *that is also an integer* ($x + (-x) = (-x) + x = 0$).

EXAMPLE. The integers $\mathbb{Z}$ *do not* form a group under multiplication, for two reasons:

- 0 has no multiplicative inverse 0^{-1}; and
- the other integers a have multiplicative inverses $1/a$, but *most are not integers*. A group only satisfies the inverse property if it contains the inverses of each element.

In this lab you will use pseudocode to write code to test whether a finite set is a group under *multiplication*. You will then test it on three sets, two of which succeed, and one of which does not. A complication in this project is that the function has to depend on the operation, so you can't just write a function for one operation, only.

Pseudocode.

Closure. We must check every pair $x, y \in S$. We can test whether this is true for "every" element of a finite set using definite loops.

```
algorithm is_closed
inputs
   S, a finite set
outputs
   true if S is closed under multiplication; false otherwise
do
   for s ∈ S
      for t ∈ S
         if s t ∉ S
            print "fails closure for", s, t
            return false
   return true
```

Associative. We must check every triplet $x, y, z \in S$, requiring definite loops. The pseudocode is an exercise.

Identity. We can test whether "we can find" an identity using a special variable called a **flag** with Boolean value (sometimes called a **signal**). We adjust the flag's value depending on whether a candidate continues to satisfy a known property. When the loop ends, the flag indicates whether we're done (i.e., whether we've found an identity). The quantifiers' structure ("we can find… for any…") requires the pseudocode to presume an identity exists until proved otherwise.

```
algorithm find_identity
inputs
  S, a finite set
outputs
  an identity, if it can find it; otherwise, ∅
do
  for s ∈ S
    let maybe_identity = true
    for t ∈ S
      if s t ≠ t or t s ≠ t
        let maybe_identity = false
    if maybe_identity = true
      return s
  print "no identity"
  return ∅
```

Inverse. We are looking for an inverse for each element. Here, again, we use a flag a **flag**, as the logic requires us to find an inverse. Unlike the previous pseudocode, we presume an inverse *does not* exist until proved otherwise; this is because the order of the quantifiers is switched ("for any… we can find…" instead of "we can find… for any…"). This pseudocode also requires that we identify the set's identity in the input.

```
algorithm has_inverses
inputs
  S, a finite set
  ι, an identity of S under multiplication
outputs
  true if every element of S has a multiplicative inverse; false otherwise
do
  for s ∈ S
    let found_inverse = false
    for t ∈ S
      if s t = ι and t s = ι
        let found_inverse = true
    if found_inverse = false
      print "no inverse for", s
      return false
  return true
```

Putting them together. This pseudocode tests whether a set is a group under an operation by invoking all four algorithms defined above.

```
algorithm is_a_group
inputs
   S, a finite set
outputs
   true if S is a group under multiplication; false otherwise
do
   if is_closed(S) and is_associative(S)
      let ι = find_identity(S)
      if ι ≠ ∅ and has_inverses(S, ι)
         return true
   return false
```

Your tasks. Use LaTeX in your Sage worksheets wherever appropriate. **Two of the sets in 3–5 are groups; one is not.**

1. Study the pseducode for closure, and write pseudocode for an algorithm named *is_associative* that tests whether a set S is associative under multiplication. You essentially modify the pseudocode for *is_closed* with a third loop, and change the condition for the **if** appropriately.
2. Write Sage code for each of the five algorithms defined above in pseudocode. You will test them on the following sets.
3. Define a ring R to be $\mathbb{Z}_{101}$, the finite ring of 101 elements. (You will want to revisit Lab #2 if you forgot how to do this.) Let $S = \{1, 2, \ldots, 100\} \subsetneq R$; that is, S should include every element of R *except* 0. Be sure to define S using elements of R, and not plain integers. (Again, you will want to revisit Lab #2 if you forgot how to do this.) Test your Sage code on S; is S a group under multiplication? If not, which property fails?
4. Redefine the ring R to be $\mathbb{Z}_{102}$, the finite ring of 102 elements. Let $S = \{1, 2, \ldots, 101\} \subsetneq R$; that is, S should include every element of R *except* 0. Be sure to define S using elements of R, and not plain integers. Test your Sage code on S; is S a group under multiplication? If not, which property fails?
5. Define the matrices

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \mathbf{i} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$$
$$\mathbf{j} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \qquad \mathbf{k} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}$$

   and the set

$$Q = \{I_2, -I_2, \mathbf{i}, -\mathbf{i}, \mathbf{j}, -\mathbf{j}, \mathbf{k}, -\mathbf{k}\}.$$

   Test your Sage code on Q; is Q a group under multiplication? If not, which property fails?

   REMARK. This set is sometimes called the **set of quaternions**.

6. Using the matrices of problem #4, define the set

$$S = \{I_2, -I_2, \mathbf{j}, -\mathbf{j}\}.$$

(a) You've probably noticed that $S \subseteq Q$. Is S also a group? If so, we call S a **subgroup** of Q. If not, which property fails?
(b) The set S actually consists of matrices of the form A from Lab #6, Problem #1. Indicate in a text cell the correct value of a for each matrix.

Coding theory and cryptography

Two applications of algebra and number theory are **coding theory** and **cryptography**:

- The aim of *coding theory* is transmit information *reliably*, detecting errors and, when possible, correcting them.
- The aim of *cryptography* is to transmit information *securely*, so that an eavesdropper can neither understand nor work out the meaning of the transmission.

To apply these mathematical ideas, both require a preliminary stage of transforming text into numbers, and vice versa. You will write two or more functions to do help someone this; one will be interactive.

When transforming text into numbers, it is necessary first to group the text into conveniently-sized blocks. For instance, the phrase

```
GET OUT OF DODGE
```

can be grouped several different ways. One way is to group it into blocks of two:

```
GE TO UT OF DO DG EX
```

and another is to group it into blocks of four:

```
GETO UTOF DODG EXXX.
```

In both cases, the message's length does not divide the group's length evenly, so we pad the message with `X`'s.

How does one do this? Python has a command that lets you convert each *character* into a number. (If you've forgotten it, it appears in the book, and you used it in a previous assignment, so it would be good to review that, as I might ask about it on the exam!) This command would convert our phrase into the numbers

$$71, 69, 84, 79, 85, 84, 79, 70, 68, 79, 68, 71, 69\,.$$

You'll notice I've removed the spaces, and haven't yet created the groups. How do we then group them? First, turn them into the numbers 0–25 by subtracting the number corresponding to `A`. That gives us

$$6, 4, 19, 14, 20, 19, 14, 5, 3, 14, 3, 6, 4\,.$$

To group them, we reason the following:

- We don't need lower-case letters or punctuation to get our meaning across.[110]
- Thus, every value to encode lies between 1 and 26.
- We can use a base-26 number system to encode any group of letters.

So, to encode a group of four letters with numerical values a, b, c, d, we can compute

$$m = a + 26b + 26^2c + 26^3d\,.$$

In general, to encode a group of n letters with numerical values $a_1, a_2, \ldots, a_n$, compute

$$m = a_1 + 26a_2 + 26^2a_3 + \cdots + 26^{n-1}a_n\,.$$

To *decode* an encoded group m of n characters, do the following n times:

- determine the remainder of dividing m by 26, and call it a;
- convert a back into a character; and finally,
- replace m by $^{m-a}/_{26}$.

[110] Related trivia: Ancient written languages typically used all upper-case letters with no punctuation or even spacing.

You need to write at least two functions.

1. The first function, `encode()`, takes two inputs: an integer n, and a list L of characters. It converts each character into a number, organizes the numbers into groups of n characters, then converts each group into one number using the formula above. It returns a list M of numbers, one for each group. **You may assume that n divides the length of L.** (It's a bit harder if you have to figure out the padding.)
2. The second function is interactive. It offers the user a text box and a slider. In the text box, the user inputs a message. In the slider, the user selects from 2 to 6 characters. The function takes these values and sends them to encode, then prints the list numbers that `encode()` returns.

For extra credit, you can also:

3. Implement a third function, `decode()`, takes two inputs: an integer n, and a list M of integers. It converts each integer into a group of n integers, then converts each integer into a character.
4. You can Cythonize either `encode()` or `decode()`. At least one variable's type must be declared.

Continued fractions

A continued fraction representation of a number has the following form:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}} \quad .$$

Continued fractions have a number of fascinating properties, some of which you will explore in this lab. The following pseudocode will compute a continued fraction up to m places:

algorithm *continued_fraction*
inputs
- b, a real number
- m, the number of places to expand the continued fraction

outputs
- a list $(a_0, a_1, \ldots, a_m)$ listing the entries of the continued fraction

do
```
let L be a list of m + 1 zeros
let i = 0
while i ≤ m and b ≠ 0
    set L_i to the floor of b
    let d = b − L_i
    if d ≠ 0
        replace d by 1/d
    let b = d
    add 1 to i
return L
```

1. Implement this pseudocode as a Sage function. *Hint:* Sage's `floor()` function could prove useful here.
2. Use your code to compute the continued fraction approximations to at most 20 places for $12/17$, $17/12$, $4/15$, $15/4$, $729/1001$, and $1001/729$.
3. Write down at least two properties you notice about the continued fractions above. Explain why these properties make sense.
4. Use your code to compute the continued fraction approximations to at most 20 places for $\sqrt{41}$, $1/\sqrt{41}$, $1+\sqrt{5}/2$, $2/1+\sqrt{5}$, $1+\sqrt{7}/\sqrt{3}$, and $\sqrt{3}/1+\sqrt{7}$.
5. Write down at least two properties you notice about these continued fractions. One of the properties may be the same as with the rational numbers, but the other definitely should not.
6. Use your code to compute the continued fraction approximations to at most 20 places for e, $\sqrt{e}$, and $\sqrt[3]{e}$.
7. Write down any patterns you notice about these continued fractions. (It is unlikely you find one that is common to the previous numbers, but finding one would well be interesting.)

Bibliography

[1] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Harmony Books, New York, 1979.

[2] Michael H. Albert, Richard J. Nowakowski, and David Wolfe. *Lessons in Play*. AK Peters, Ltd., Wellesley, Massachusetts, 2007.

[3] Elwyn Berlekamp, John Conway, and Richard Guy. *Winning Ways for Your Mathematical Plays*. A K Peters / CRC Press, second edition, 2001.

[4] Anna M. Bigatti. Computation of Hilbert-Poincaré Series. *Journal of Pure and Applied Algebra*, 119(3):237–253, 1997.

[5] Anna M. Bigatti, Massimo Caboara, and Lorenzo Robbiano. Computing inhomogeneous Gröbner bases. *Journal of Symbolic Computation*, 46(5):498–510, 2010.

[6] Alberto Damiano, Graziano Gentili, and Daniele Struppa. *Journal of Symbolic Computation*, 45:38–45, January 2010.

[7] Vladimir Gerdt and Yuri Blinkov. Involutive Bases of Polynomial Ideals. *Mathematics and Computers in Simulation*, 45(5–6):419–541, March 1998.

[8] Maxima. *Maxima, a Computer Algebra System. Version 5.38.1*. `http://maxima.sourceforge.net/`, 2016.

[9] Marin Mersenne. *Cogitata Physica-Mathematica*. 1644.

[10] Diophantus of Alexandria. *Arithmetica*. 1670. With marginal notes by Pierre de Fermat.

[11] Euclid of Alexandria. *Elements*. c. 300 BC.

[12] Jamie Uys. *The Gods Must Be Crazy*, 1980. film distributed by 20th Century Fox.

Index