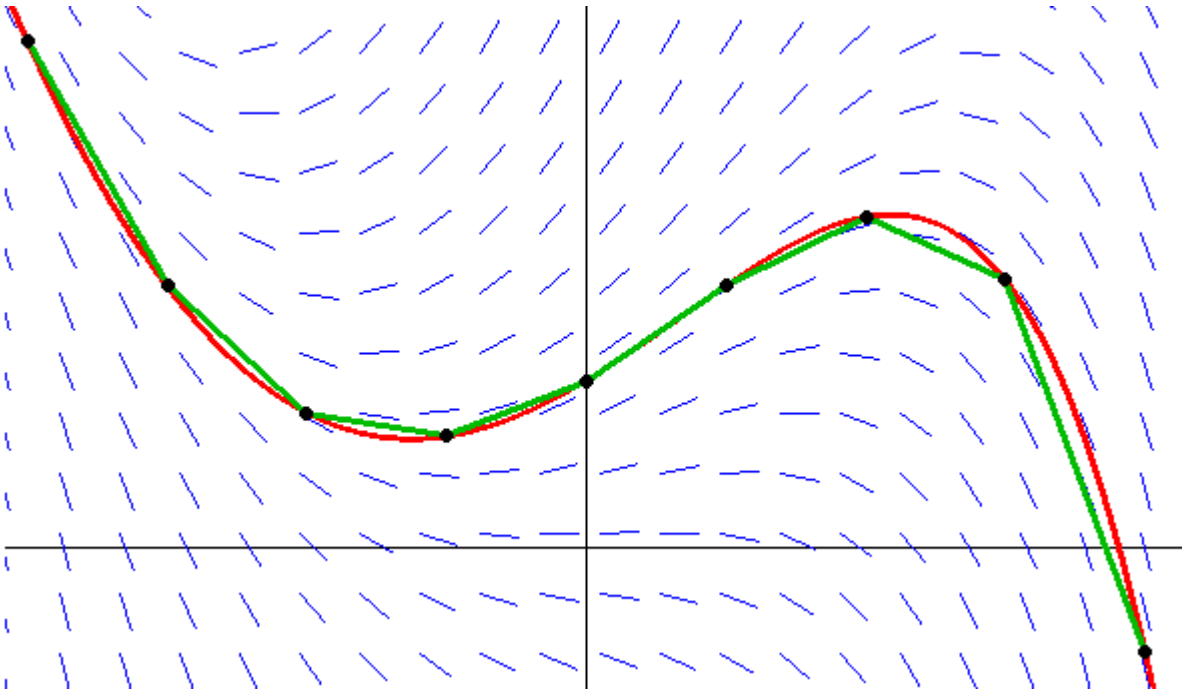
An Intuitive Guide to Numerical Methods



Brian Heinold

Department of Mathematics and Computer Science
Mount St. Mary's University

Contents

Preface

These are notes I wrote up for my Fall 2013 Numerical Methods course. These notes are meant as a supplement to a traditional textbook, not as a replacement. I wanted something that explains the mathematical ideas behind the various methods without getting into formal proofs. I find that many numerical analysis proofs show that something is true without giving much insight into why it is true. If students can gain an intuitive understanding of how a method works, why it works, and where it's useful, then they will be better users of those methods if they need them, and, besides, the mathematical ideas they learn may be useful in other contexts.

One thing that is missing here is linear algebra. Most of my students hadn't yet taken linear algebra, so it wasn't something I covered.

I have included Python code for many of the methods. I find that sometimes it is easier for me to understand how a method works by looking at a program. Python is easy to read, being almost like pseudocode (with the added benefit that it actually runs). However, everything has been coded for clarity, not for efficiency or robustness. In real-life there are lots of edge cases that need to be checked, but checking them clutters the code, and that would defeat the purpose of using the code to illustrate the method. In short, do not use the code in this book for anything important.

In writing this, I relied especially on *Numerical Analysis* by Timothy Sauer, which I have taught from a couple of times. For practical advice, I relied on *Numerical Recipes* by Press, Teukolsky, Vetterling, and Flannery. From time to time, I also consulted *A Friendly Introduction to Numerical Analysis* by Brian Bradie and *Numerical Methods* by Faires and Burden. I also read through a number of Wikipedia articles and online lecture notes from a variety of places.

I am not a numerical analyst, so I am sure there are mistakes in here. It is also hard for me to catch my own typos. So if you see anything wrong here, please send me a note at |heinold@msmary.edu|. I am also open to suggestions for improvement.

Last updated June 11, 2018.

Chapter 1

Introductory material

1.1 What Numerical Methods is about

Equations like $3x+4=7$ or $x^2-2x+9=0$ can be solved using high school algebra. Equations like $2x^3+4x^2+3x+1=0$ or $x^4+x+1=0$ can be solved by some more complicated algebra.

But the equation $x^5-x+1=0$ cannot be solved algebraically. There are no algebraic techniques that will give us an answer in terms of anything familiar. If we need a solution to an equation like this, we use a *numerical method*, a technique that will give us an approximate solution.

Here is a simple example of a numerical method that estimates $\sqrt{5}$: Start with a guess for $\sqrt{5}$, say $x=2$ (since $\sqrt{4}=2$ and 5 is pretty close to 4). Then compute

$$\frac{2+5/2}{2}=\frac{9}{4}=2.25.$$

This is the average of our guess, 2, and $5/2$. Next compute

$$\frac{2.25+5/2.25}{2}=2.2361111\ldots.$$

We get this by replacing our initial guess, 2, in the previous equation with 2.25. We can then replace 2.25 with 2.2361... and compute again to get a new value, then use that to get another new value, etc. The values 2, 2.25, 2.2361..., and so on, form a sequence of approximations that get closer and closer to the actual value of $\sqrt{5}$, which is 2.236067....

This is what a numerical method is—a process that produces approximate solutions to some problem. The process is often repeated like this, where approximations are plugged into the method to generate newer, better approximations. Often a method can be run over and over until a desired amount of accuracy is obtained.

It is important to study how well the method performs. There are a number of questions to ask: Does it always work? How quickly does it work? How well does it work on a computer?

Now, in case a method to find the square root seems silly to you, consider the following: How would you find the square root of a number without a calculator? Notice that this method allows us to estimate the square root using only the basic operations of addition and division. So you could do it by hand if you wanted to. In fact it's called the Babylonian method, or Hero's method, as it was used in Babylon and ancient Greece. It used to be taught in school until calculators came along.[2]

[2] Calculators themselves use numerical methods to find square roots. Many calculators use efficient numerical methods to compute e^x and $\ln x$ and use the following identity to obtain $\sqrt{x}$ from e^x and $\ln x$:

$$\sqrt{x}=x^{1/2}=e^{\ln(x^{1/2})}=e^{\frac{1}{2}\ln x}.$$

1.2 Floating-point arithmetic

Most numerical methods are implemented on computers and calculators, so we need to understand a little about how computers and calculators do their computations.

Computer hardware cannot not represent most real numbers exactly. For instance, π has an infinite, non-repeating decimal expansion that would require an infinite amount of space to store. Most computers uses something called *floating-point arithmetic*, specifically the IEEE 754 standard. We'll just give a brief overview of how things work here, leaving out a lot of details.

The most common way to store real numbers uses 8 bytes (64 bits). It helps to think of the number as written in scientific notation. For instance, write 25.394 as 2.5394×10^2. In this expression, 2.5394 is called the mantissa. The 2 in 10^2 is the exponent. Note that instead of a base 10 exponent, computers, which work in binary, use a base 2 exponent. For instance, .3 is written as 1.2×2^{-2}.

Of the 64 bits, 53 are used to store the mantissa, 11 are for the exponent, and 1 bit is used for the sign (+ or −). These numbers actually add to 65, not 64, as a special trick allows an extra bit of info for the mantissa. The 53 bits for the mantissa allow for about 15 digits of precision. The 11 bits for the exponent allow numbers to range from as small as 10^{-307} to as large as 10^{308}.

Consequences of limited precision

The precision limits mean that we cannot distinguish 123.000000000000007 from 123.000000000000008. However, we would be able to distinguish .00000000000000007 and .00000000000000008 because those numbers would be stored as 7×10^{-18} and 8×10^{-18}. The first set of numbers requires more than 15 digits of precision, while the second pair requires very little precision.

Also, doing something like 1000000000 + .00000004 will just result in 1000000000 as the 4 falls beyond the precision limit.

When reading stuff about numerical methods, you will often come across the term *machine epsilon*. It is defined as the difference between 1 and the closest floating-point number larger than 1. In the 64-bit system described above, that value is about 2.22×10^{-16}. Machine epsilon is a good estimate of the limit of accuracy of floating-point arithmetic. If the difference between two real numbers is less than machine epsilon, it is possible that the computer would not be able to tell them apart.

Roundoff error

One interesting thing is that many numbers, like .1 or .2, cannot be represented exactly using IEEE 754. The reason for this is that computers store numbers in binary (base 2), whereas our number system is decimal (base 10), and fractions behave differently in the two bases. For instance, converting .2 to binary yields .00110011001100..., an endlessly repeating expansion. Since there are only 53 bits available for the mantissa, any repeating representation has to be cut off somewhere.[1] This is called *roundoff error*. If you convert the cut-off representation back to decimal, you actually get .19999999999999998.

If you write a lot of computer programs, you may have run into this problem when printing numbers. For example, consider the following simple Python program:

```
print(.2 + .1)
```

We would expect it to print out .3, but in fact, because of roundoff errors in both .2 and .1, the result is `0.30000000000000004`.

Actually, if we want to see exactly how a number such as .1 is represented in floating-point, we can do the following:

[1] It's interesting to note that the only decimal fractions that have terminating binary expansions are those whose denominators are powers of 2. In base 10, the only terminating expansions come from denominators that are of the form $2^j 5^k$. This is because the only proper divisors of 10 are 2 and 5.

```
print("{:.70f}".format(.1))
```

The result is `0.1000000000000000055511151231257827021181583404541015625000000000000000`.

This number is the closest IEEE 754 floating-point number to .1. Specifically, we can write .1 as $.8 \times 2^{-3}$, and in binary, .8 is the has a repeating binary expansion .110011001100.... That expansion gets cut off after 53 bits and that eventually leads to the strange digits that show up around the 18th decimal place.

Floating-point problems

One big problem with this is that these small roundoff errors can accumulate, or *propagate*. For instance, consider the following Python program:

```
s = 0
for i in range(10000000):
    s = s + .1
```

This program adds .1 to itself ten million times, and should return 1,000,000. But the result is `999999.9998389754`. Roundoff errors from storing .1 and the intermediate values of `s` have accumulated over millions of iterations to the point that the error has crept all the way up to the fourth decimal place. This is a problem for many numerical methods that work by iterating something many times. Methods that are not seriously affected by roundoff error are called *stable*, while those that are seriously affected are *unstable*.

Here is something else that can go wrong: Try computing the following:

```
x = (1.000000000000001 - 1) * 100000000000000
```

The exact answer should be .1, but the program gives 0.11102230246251565, only correct to one decimal place. The problem here is that we are subtracting two numbers that are very close to each other. This causes all the incorrect digits beyond the 15th place in the representation of .1 to essentially take over the calculation. Subtraction of very close numbers can cause serious problems for numerical methods and it is usually necessary to rework the method, often using algebra, to eliminate some subtractions.

This is sometimes called the *golden rule of numerical analysis* — don't subtract nearly equal numbers.

Using floating-point numbers in calculations

The 64-bit storage method above is called IEEE 754 *double-precision* (this is where the data type `double` in many programming languages gets its name). There is also 32-bit single-precision and 128-bit quadruple-precision specified in the IEEE 754 standard.

Single-precision is the `float` data type in many programming languages. It uses 24 bits for the mantissa and 8 for the exponent, giving about 6 digits of precision and values as small as 10^{-128} to as large as 10^{127}. The advantages of single-precision over double-precision are that it uses half as much memory and computations with single-precision numbers often run a little more quickly. On the other hand, for many applications these considerations are not too important.

Quadruple-precision is implemented by some programming languages.[1] One big difference between it and single- and double-precision is that the latter are usually implemented in hardware, while the former is usually implemented in software. Implementing in software, rather than having specially designed circuitry to do computations, results in a significant slowdown.

For most scientific calculations, like the ones we are interested in a numerical methods course, double-precision will be what we use. Support for it is built directly into hardware and most programming languages. If we need more precision, quadruple-precision and arbitrary precision are available in libraries for most programming languages. For instance, Java's `BigDecimal` class and Python's `Decimal` class can represent real numbers to very high precision. The drawback to these is that they are implemented in software and thus are much slower than a native double-precision type. For many numerical methods this slowdown makes the methods too slow to use.

[1] There is also an 80-bit extended precision method on x86 processors that is accessible via the `long double` type in C and some other languages.

See the article, “What Every Computer Scientist Should Know About Floating-Point Arithmetic” at http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html for more details about floating-point arithmetic.

Chapter 2

Solving equations numerically

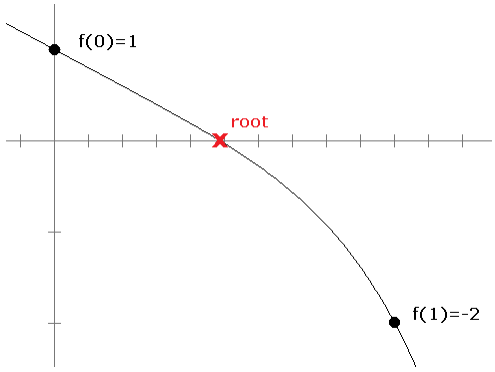
Solving equations is of fundamental importance. Often many problems at their core come down to solving equations. For instance, one of the most useful parts of calculus is maximizing and minimizing things, and this usually comes down to setting the derivative to 0 and solving.

As noted earlier, there are many equations, such as $x^5 - x + 1 = 0$, which cannot be solved exactly in terms of familiar functions. The next several sections are about how to find approximate solutions to equations.

2.1 The bisection method

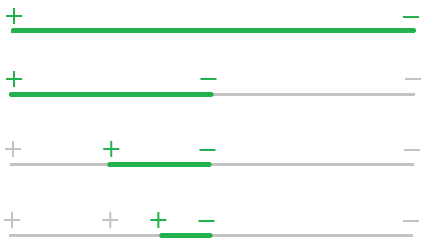
The *bisection method* is one of the simpler root-finding methods. If you were to put someone in a room and tell them not to come out until they come up with a root-finding method, the bisection method is probably the one they would come up with.

Here is the idea of how it works. Suppose we want to find a root of $f(x) = 1-2x-x^5$. First notice that $f(0) = 1$, which is above the x-axis and $f(1) = -2$, which is below the axis. For the function to get from 1 to -2, it must cross the axis somewhere, and that point is the root.[2] So we know a root lies somewhere between $x = 0$ and $x = 1$. See the figure below.



Next we look at the midpoint of 0 and 1, which is $x = .5$ (i.e., we bisect the interval) and check the value of the function there. It turns out that $f(.5) = -0.03125$. Since this is negative and $f(0) = 1$ is positive, we know that the function must cross the axis (and thus have a root) between $x = 0$ and $x = .5$. We can then bisect again, by looking at the value of $f(.25)$, which turns out to be around .499. Since this is positive and $f(.5)$ is negative, we know the root must be between $x = .25$ and $x = .5$. We can keep up this bisection procedure as long as we like, and we will eventually zero in on the location of the root. Shown below is a graphical representation of this process.

[2] This is common sense, but it also comes from the Intermediate Value Theorem.



In general, here is how the bisection method works: We start with values a and b such that $f(a)$ and $f(b)$ have opposite signs. If f is continuous, then we are guaranteed that a root lies between a and b. We say that a and b *bracket* a root.

We then compute $f(m)$, where $m = (a+b)/2$. If $f(m)$ and $f(b)$ are of opposite signs, we set $a = m$; if $f(m)$ and if $f(a)$ are of opposite signs, we set $b = m$. (In the unlikely event that $f(m) = 0$, then we stop because we have found a root.) We then repeat the process again and again until a desired accuracy is reached. Here is how we might code this in Python:

```
def bisection(f, a, b, n):
    for i in range(n):
        m = (a + b) / 2
        if f(a)*f(m) < 0:
            b = m
        else:
            a = m
    return m
```

For simplicity, we have ignored the possibility that $f(m) = 0$. Notice also that an easy way to check if $f(a)$ and $f(m)$ are of opposite signs is to check if $f(a)f(m) < 0$. Here is how we might call this function to do 20 bisection steps to estimate a root of $x^2 - 2$:

```
print(bisection(lambda x:x*x-2, 0, 2, 20))
```

In Python, `lambda` creates an anonymous function. It's basically a quick way to create a function that we can pass to our bisection function.

Analysis of the algorithm

Let a_0 and b_0 denote the starting values of a and b. At each step we cut the previous interval exactly in half, so after n steps the interval has length $(b_0 - a_0)/2^n$. At any step, our best estimate for the root is the midpoint of the interval, which is no more than half the length of the interval away from the root, meaning that after n steps, the error is no more than $(b_0 - a_0)/2^{n+1}$.

For example, if $a_0 = 0$, $b_0 = 1$, and we perform $n = 10$ iterations, we would be within $1/2^{11} \approx .00049$ of the actual root.

If we have a certain error tolerance ϵ that we want to achieve, we can solve $(b_0 - a_0)/2^{n+1} = \epsilon$ for n to get

$$n = \left\lceil \log_2\left(\frac{b_0 - a_0}{\epsilon}\right) - 1 \right\rceil.$$

So in the example above, if we wanted to know how many iterations we would need to get to within $\epsilon = 10^{-6}$ of the correct answer, we would need $\lceil \log_2(1/10^{-6}) - 1 \rceil = 19$ iterations.

In the world of numerical methods for solving equations, the bisection method is very reliable, but very slow. We essentially get about one new correct decimal place for every three iterations (since $\log_2(10) \approx 3$). Some of the methods we will consider below can double and even triple the number of correct digits at every step. Some can do even better. But this performance comes at the price of reliability. The bisection method, however slow it may be, will usually work.

However, it will miss the root of something like x^2 because that function is never negative. It can also be fooled around vertical asymptotes, where a function may change from positive to negative but have no root, such as with $1/x$ near $x = 0$.

False position

The false position method is a lot like the bisection method. The difference is that instead of choosing the midpoint of a and b, we draw a line between $(a, f(a))$ an $(b, f(b))$ and see where it crosses the axis. In particular, the crossing point can be found by finding the equation of the line, which is $y - f(a) = \frac{f(b)-f(a)}{b-a}(x-a)$, and setting $y = 0$ to get its intersection with the x axis. Solving and simplifying, we get

$$m = \frac{af(b) - bf(a)}{f(b) - f(a)}.$$

This is the point we use in place of the midpoint. Doing so usually leads to faster convergence than the bisection method, though it can sometimes be worse.

2.2 Fixed point iteration

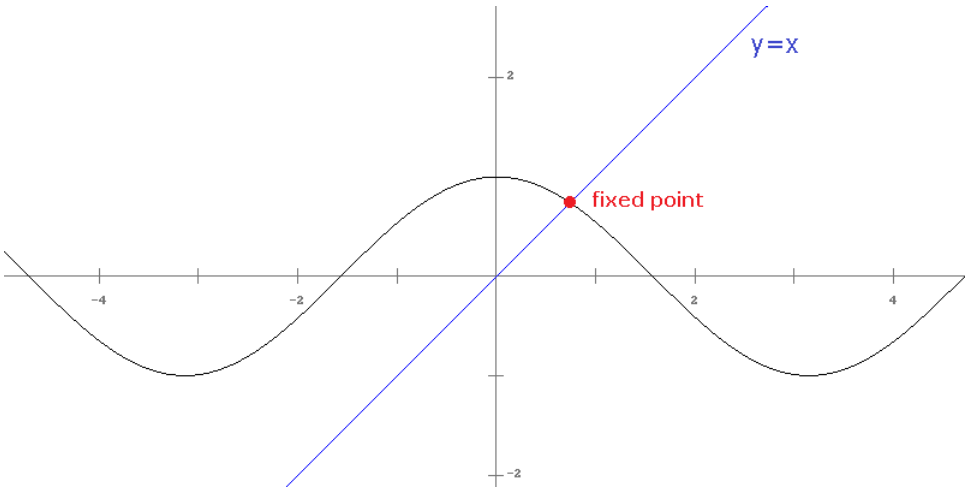
Try the following experiment: Pick any number. Take the cosine of it. Then take the cosine of your result. Then take the cosine of that. Keep doing this for a while. On a TI calculator, you can automate this process by typing in your number, pressing enter, then entering `cos(Ans)` and repeatedly pressing enter. Here's a Python program to do this along with its output:

```
from math import cos
x = 2
for i in range(20):
    x = cos(x)
    print(x)
```

```
-0.4161468365471424
0.9146533258523714
0.6100652997429745
0.8196106080000903
0.6825058578960018
0.7759946131215992
0.7137247340083882
0.7559287135747029
0.7276347923146813
0.7467496017309728
0.7339005972426008
0.7425675503014618
0.7367348583938166
0.740666263873949
0.7380191411807893
0.7398027782109352
0.7386015286351051
0.7394108086387853
0.7388657151407354
0.7392329180769628
```

We can see that the numbers seem to be settling down around .739. If we were to carry this out for a few dozen more iterations, we would eventually get stuck at (to 15 decimal places) `0.7390851332151607`. And we will eventually settle in on this number no matter what our initial value of x is.

What is special about this value, .739..., is that it is a *fixed point* of cosine. It is a value which is unchanged by the cosine function; i.e., $\cos(.739\ldots) = .739\ldots$. Formally, a fixed point of a function $g(x)$ is a value p such that $g(p) = p$. Graphically, it's a point where the graph crosses the line $y = x$. See the figure below:

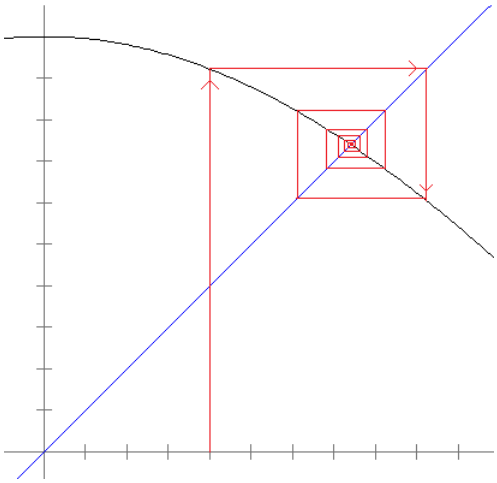


Some fixed points have the property that they attract nearby points towards them in the sort of iteration we did above. That is, if we start with some initial value x_0, and compute $g(x_0)$, $g(g(x_0))$, $g(g(g(x_0)))$, etc., those values will approach the fixed point. The point .739... is an attracting fixed point of $\cos x$.

Other fixed points have the opposite property. They repel nearby points, so that it is very hard to converge to a repelling fixed point. For instance, $g(x) = 2\cos x$ has a repelling fixed point at $p = 1.02985\ldots$. If we start with a value, say $x = 1$, and iterate, computing $g(1)$, $g(g(1))$, etc., we get 1.08, .94, 1.17, .76, 1.44, numbers which are drifting pretty quickly away from the fixed point.

Here is a way to picture fixed points: Imagine a pendulum rocking back and forth. This system has two fixed points. There is a fixed point where the pendulum is hanging straight down. This is an attracting state as the pendulum naturally wants to come to rest in that position. If you give it a small push from that point, it will soon return back to that point. There is also a repelling fixed point when the pendulum is pointing straight up. At that point you can theoretically balance the pendulum, if you could position it just right, and that fixed point is repelling because just the slighted push will move the pendulum from that point. Phenomena in physics, weather, economics, and many other fields have natural interpretations in terms of fixed points.

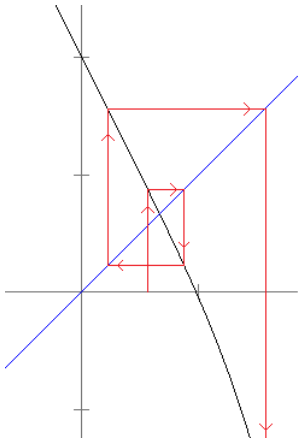
We can visualize the fact that the fixed point of $\cos x$ is attracting by using something called a *cobweb diagram*. Here is one:



In the diagram above, the function shown is $g(x) = \cos(x)$, and we start with $x_0 = .4$. We then plug that into the function to get $g(.4) = .9210\ldots$. This is represented in the cobweb diagram by the line that goes from the x-axis to the cosine graph. We then compute $g(g(.4))$ (which is $g(.9210\ldots)$). What we are doing here is essentially taking the previous y-value, turning it into an x-value, and plugging that into the function. Turning the y-value into an x-value is represented graphically by the horizontal line that goes from our previous location on the graph to the line $y = x$. Then plugging in that value into the function is represented by the vertical line that goes down to the graph of the function.

We then repeat the process by going over to $y = x$, then to $y = \cos(x)$, then over to $y = x$, then to $y = \cos(x)$, etc. We see that the iteration is eventually sucked into the fixed point.

Here is an example of a repelling fixed point, iterating $g(x) = 1 - 2x - x^5$.



The key to whether a fixed point p is attracting or repelling turns out to be the value of $g'(p)$, the slope of the graph at the fixed point. If $|g'(p)| < 1$ then the fixed point is attracting. If $|g'(p)| > 1$, the fixed point is repelling.[1] Graphically, if the function is relatively flat near the fixed point, the lines in the cobweb diagram will be drawn in towards the fixed point, whereas if the function is steeply sloped, the lines will be pushed away. If $|g'(p)| = 1$, a more sophisticated analysis is needed to determine the point's behavior, as it can be (slowly) attracting, (slowly) repelling, possibly both, or divergent.

The value of $|g'(p)|$ also determines the speed at which the iterates converge towards the fixed point. The closer $|g'(p)|$ is to 0, the faster the iterates will converge. A very flat graph, for instance, will draw the cobweb diagram quickly into the fixed point.

One caveat here is that an attracting fixed point is only guaranteed to attract nearby points to it, and the definition of "nearby" will vary from function to function. In the case of cosine, "nearby" includes all of $\mathbb{R}$, whereas with other functions "nearby" might mean only on a very small interval.

Using fixed points to solve equations

The process described above, where we compute $g(x_0)$, $g(g(x_0))$, $g(g(g(x_0)))$, ... is called *fixed point iteration*. It can be used to find an (attracting) fixed point of f. We can use this to find roots of equations if we rewrite the equation as a fixed point problem.

For instance, suppose we want to solve $1 - 2x - x^5 = 0$. We can rewrite this as a fixed point problem by moving the $2x$ over and dividing by 2 to get $(1 - x^5)/2 = x$. It is now of the form $g(x) = x$ for $g(x) = (1 - x^5)/2$. We can then iterate this just like we iterated $\cos(x)$ earlier.

We start by picking a value, preferably one close to a root of $1 - 2x - x^5$. Looking at a graph of the function, $x_0 = .5$ seems like a good choice. We then iterate. Here is result of the first 20 iterations:

```
0.484375
0.48666851269081235
0.48634988700459447
0.48639451271206546
0.4863882696148348
0.48638914315650295
0.4863890209322015
0.48638903803364786
0.4863890356408394
0.48638903597563754
```

[1] This result is sometimes called the Banach fixed point theorem and can be formally proved using the Mean Value Theorem.

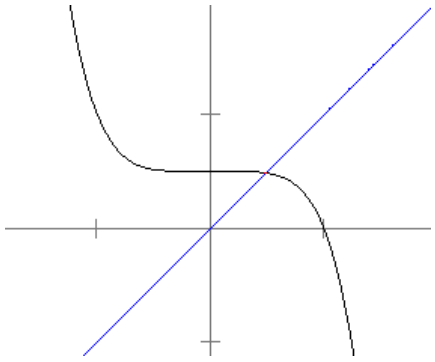
```
0.4863890359287931
0.4863890359353475
0.48638903593443045
0.48638903593455873
0.4863890359345408
0.4863890359345433
0.48638903593454297
0.486389035934543
0.48638903593454297
0.486389035934543
```

We see that the iterates pretty quickly settle down to about `.486389035934543`.

On a TI calculator, you can get the above (possibly to less decimal places) by entering in .5, pressing enter, then entering in `(1-Ans^5)/2` and repeatedly pressing enter. In Excel you can accomplish the same thing by entering .5 in the cell A1, then entering the formula `=(1-A1^5)/2` into cell A2, and filling down. Here also is a Python program that will produce the above output:

```
x = .5
for i in range(20):
    x = (1-x**5)/2
    print(x)
```

There are a couple of things to note here about the results. First, the value of $|g'(p)|$ is about .14, which is pretty small, so the convergence is relatively fast. We add about one new correct digit with every iteration. We can see in the figure below that the graph is pretty flat around the fixed point. If you try drawing a cobweb diagram, you will see that the iteration gets pulled into the fixed point very quickly.



Second, the starting value matters. If our starting value is larger than about 1.25 or smaller than -1.25, we are in the steeply sloped portion of the graph, and the iterates will be pushed off to infinity. We are essentially outside the realm of influence of the fixed point.

Third, the way we rewrote $1-2x-x^5=0$ was not the only way to do so. For instance, we could instead move the x^5 term over and take the fifth root of both sides to get $\sqrt[5]{1-2x}=x$. However, this would be a bad choice, as the derivative of this function at the fixed point is about -7, which makes the point repelling. A different option would be to write $1=2x+x^5$, factor out an x and solve to get $\frac{1}{x^4+2}=x$. For this iteration, the derivative at the fixed point is about $-.11$, so this is a good choice.

2.3 Newton's method

Here is a clever way to turn $f(x)=0$ into a fixed point problem: divide both sides of the equation by $-f'(x)$ and then add x to both sides. This gives

$$x-\frac{f(x)}{f'(x)}=x.$$

To see why this is useful, let's look at the derivative of the expression on the left side. We get

$$1-\frac{f'(x)^2-f(x)f''(x)}{f'(x)^2}.$$

Since we have turned $f(x) = 0$ into a fixed point problem, a fixed point p must satisfy $f(p) = 0$. Plugging p into the expression above gives

$$1 - \frac{f'(p)^2 - 0}{f'(p)^2} = 0.$$

So the derivative at the fixed point is 0. Remember that the flatter the graph is around a fixed point, the faster the convergence, so we should have pretty fast convergence here.

Iterating $x - \frac{f(x)}{f'(x)}$ to find a root of f is known as *Newton's method*.[1] As an example, suppose we want a root of $f(x) = 1 - 2x - x^5$. To use Newton's method, we first compute $f'(x) = -2 - 4x^5$. So we will be iterating

$$x - \frac{1 - 2x - x^5}{2 - 4x^5}.$$

Then we pick a starting value for the iteration, preferably something we expect to be close to the root of f, like .5. We then iterate the function. Using a calculator, computer program, spreadsheet, or whatever (even by hand), we get the following sequence of iterates:

```
0.4864864864864865
0.4863890407290883
0.486389035934543
0.486389035934543
0.486389035934543
```

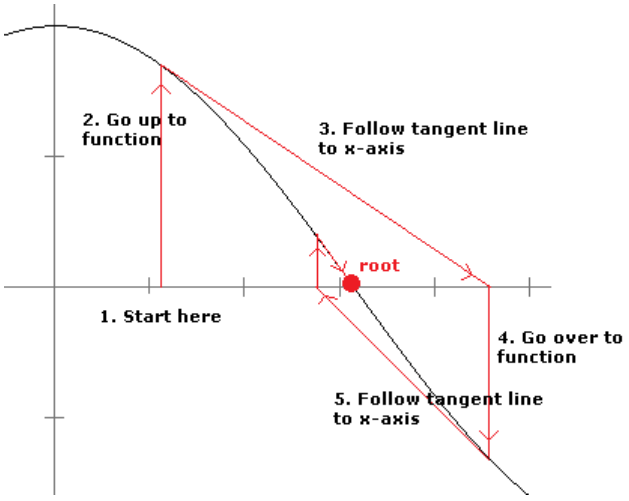
After just a few iterations, Newton's method has already found as many digits as possible in double-precision floating-point arithmetic.

To get this on a TI calculator, put in .5, press enter, then put in `Ans-(1-2*Ans-Ans^5)/(-2-5*Ans^4)` and repeatedly press enter. In Excel, .5 in the cell A1, put the formula `=A1-(1-2*A1-A1^5)/(-2-5*A1^4)` into cell A2, and fill down. Here also is a Python program that will produce the above output, stopping when the iterates are equal to about 15 decimal places:

```
oldx = 0
x = .5
while abs(x - oldx) > 1e-15:
    oldx, x = x, x-(1-2*x-x**5)/(-2-5*x**4)
    print(x)
```

Geometric derivation of Newton's method

The figure below shows a nice way to picture how Newton's method works. The main idea is that if x_0 is sufficiently close to a root of f, then the tangent line to the graph at $(x_0, f(x_0))$ will cross the x-axis at a point closer to the root than x_0.



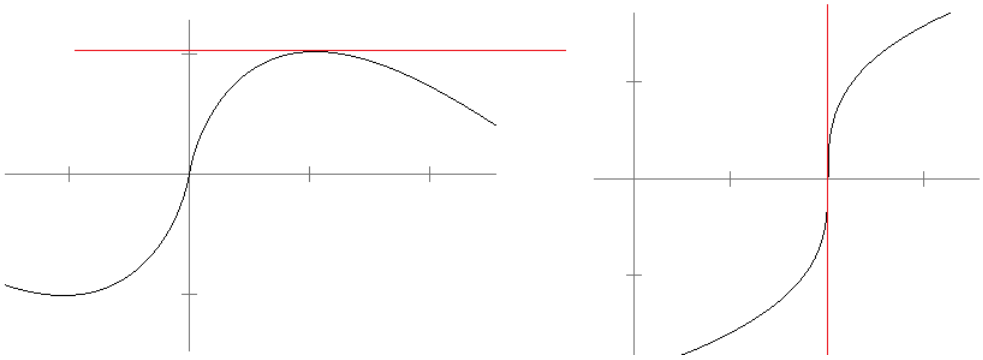
[1] It is also called the Newton-Raphson method.

What we do is start with an initial guess for the root and draw a line up to the graph of the function. Then we follow the tangent line down to the x-axis. From there, we draw a line up to the graph of the function and another tangent line down to the x-axis. We keep repeating this process until we (hopefully) get close to a root. It's a kind of slanted and translated version of a cobweb diagram.

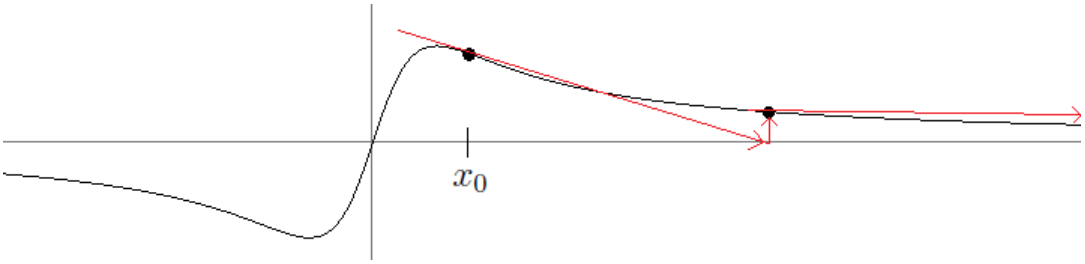
This geometric process helps motivate the formula we derived for Newton's method. If x_0 is our initial guess, then $(x_0, f(x_0))$ is the point where the vertical line meets the graph. We then draw the tangent line to the graph. The equation of this line is $y - f(x_0) = f'(x_0)(x - x_0)$. This line crosses the x-axis when $y = 0$, so plugging $y = 0$ and solving for x, we get that the new x value is $x_0 - \frac{f(x_0)}{f'(x_0)}$.

What can go wrong with Newton's method

There are a few things that can go wrong with Newton's method. One thing is that $f'(x)$ might be 0 at a point that is being evaluated. In that case the tangent line will be horizontal and will never meet the x-axis (note that the denominator in the formula will be 0). See below on the left. Another possibility can be seen with $f(x) = \sqrt[3]{x-1}$. Its derivative is infinite at its root, $x = 1$, which has the effect of pushing tangent lines away from the root. See below on the right.



Also, just like with fixed-point iteration, if we choose a starting value that is too far from the root, the slopes of the function might push us away from the root, never to come back, like in the figure below:



There are stranger things that can happen. Consider using Newton's method to find a root of $f(x) = \sqrt[3]{\frac{3-4x}{x}}$.

After a fair amount of simplification, the formula $x - \frac{f(x)}{f'(x)}$ for this function becomes $4x(1-x)$. Now, a numerical method is not really needed here because it is easy to see that the function has a root at $x = 3/4$ (and this is its only root). But it's interesting to see how the method behaves. One thing to note is that the derivative goes vertical at $x = 3/4$, so we would expect Newton's method to have some trouble. But the trouble it has is surprising.

Here are the first 36 values we get if we use a starting value of .4. There is no hint of this settling down on a value, and indeed it won't ever settle down on a value.

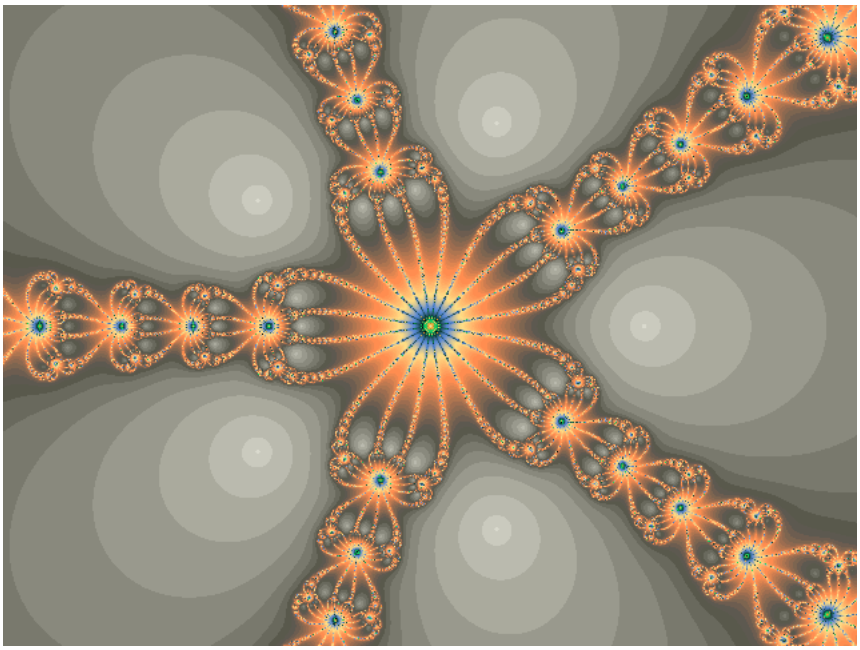
```
.960, .154, .520, .998, .006, .025, .099, .358, .919, .298, .837, .547,
.991, .035, .135, .466, .995, .018, .071, .263, .774, .699, .842, .532,
.996, .016, .064, .241, .732, .785, .676, .876, .434, .983, .068, .252,
.754, .742, .765, .719, .808, .620, .942, .219, .683, .866, .464, .995
```

But things are even stranger. Here are the first 10 iterations for two very close starting values:

.400	**.401**
.960	.960
.154	.151
.520	.512
.998	.999
.006	.002
.025	.009
.099	.036
.358	.137
.919	.474

We see these values very quickly diverge from each other. This is called *sensitive dependence on initial conditions*. Even if our starting values were vanishingly close, say only 10^{-20} apart, it would only take several dozen iterations for them to start to diverge. This is a hallmark of *chaos*. Basically, Newton's method is hopeless here.

Finally, to show a little more of the interesting behavior of Newton's method, the figure below looks at iterating Newton's method on x^5-1 with starting values that could be real or complex numbers. Each point in the picture corresponds to a different starting value, and the color of the point is based on the number of iterations it takes until the iterates get within 10^{-5} of each other.



The five light areas correspond to the five roots of x^5-1 (there is one real root at $x=1$ and four complex roots). Starting values close to those roots get sucked into those roots pretty quickly. However, points in between the two roots bounce back and forth rather chaotically before finally getting trapped by one of the roots. The end result is an extremely intricate picture. It is actually a fractal in that one could continuously zoom in on that figure and see more and more intricate detail that retains the same chaining structure as in the zoomed-out figure.

Newton's method and the Babylonian method for roots

We can use Newton's method to derive a method for estimating square roots. The trick is to find an appropriate function to apply Newton's method to. The simplest possibility is $f(x)=x^2-2$. Its roots are $\pm\sqrt{2}$. We have $f'(x)=2x$ and so the Newton's method formula is

$$x-\frac{f(x)}{f'(x)}=x-\frac{x^2-2}{2x}=\frac{x^2+2}{2x}=\frac{x+\frac{2}{x}}{2}.$$

And we see this is actually the Babylonian method formula.

2.4 Rates of convergence

The numerical methods we have examined thus far try to approximate a root r of a function f by computing approximations $x_1, x_2, \ldots, x_n$, which are hopefully getting closer and closer to r. We can measure the error at step i with $|x_i - r|$, which is how far apart the approximation is from the root.

Suppose we are using FPI to estimate a solution of $x^3+x-1=0$. One way to rewrite the problem as a fixed-point problem is to write $x(x^2+1)-1=0$ and solve for x to get $x = 1/(x^2+1)$. Let's start with $x_0 = 1$ and iterate. Below are the iterates x_1 through x_{12} along with their corresponding errors:

x_i	e_i	e_{i+1}/e_i
0.54030	0.1988	0.762
0.85755	0.1185	0.596
0.65429	0.0848	0.716
0.79348	0.0544	0.641
0.70137	0.0377	0.693
0.76396	0.0249	0.660
0.72210	0.0170	0.683
0.75042	0.0113	0.667
0.73140	0.0077	0.678
0.74424	0.0052	0.671
0.73560	0.0035	0.676
0.74143	0.0023	0.672

The third column is the ratio of successive errors. If we look carefully at the errors, it looks like each error is roughly 2/3 of the previous error, and the ratio e_{i+1}/e_i bears that out. In fact, after enough iterations it ends up being true that $e_{i+1} \approx .674e_i$. When the errors go down by a nearly constant factor at each step like this, we have what is called *linear convergence*.

For some methods, like Newton's method, the error can go down at a faster than constant rate. For instance, if we apply Newton's method to the same problem as above, starting with $x_0 = 1$, we get the following sequence of errors:

e_i
0.067672196171981
0.003718707799888
0.000011778769295
0.000000000118493

The errors are going down at a faster than linear rate. It turns out that they are going down roughly according to the rule $e_{i+1} \approx .854e_i^2$. This is an example of *quadratic convergence*.

Formally, if $\lim_{i\to\infty} \frac{e_{i+1}}{e_i} = s$ and $0 < s < 1$, the method is said to converge linearly.[1] If $\lim_{i\to\infty} \frac{e_{i+1}}{e_i^2} = s$ and $0 < s < 1$, the method is said to converge quadratically. Cubic, quartic, etc. convergence are defined similarly. The exponent in the denominator is the key quantity of interest. It is called the *order of convergence*. (The order of linear convergence is 1, the order of quadratic convergence is 2, etc.)

An order of convergence greater than 1 is referred to as *superlinear*.

The difference between linear and quadratic convergence

Let's examine the difference between linear and quadratic convergence: Suppose we have linear convergence of the form $e_{i+1} \approx .1e_i$. Each error is about 10% of the previous, so that if for example, $e_1 = .1$, then we have

$e_2 = .01$
$e_3 = .001$
$e_4 = .0001$
$e_5 = .00001$

[1] If $s = 0$, the convergence will be faster than linear.

We add about one new decimal place of accuracy with each iteration. If we had a larger constant, say $e_{i+1} = .5e^i$, then the errors would go down more slowly, but we would still add about one new decimal place of accuracy every couple of iterations.

On the other hand, suppose we have quadratic convergence of the form $e_{i+1} \approx e_i^2$. If we start with $e_1 = .1$, then we have

$e_2 = .01$
$e_3 = .0001$
$e_4 = .00000001$
$e_5 = .0000000000000001$

So we actually double the number of correct decimal places of accuracy at each step. So there is a big difference between linear and quadratic convergence. For the linear convergence described above, it would take 1000 iterations to get 1000 decimal places of accuracy, whereas for the quadratic convergence described above, it would take about 10 iterations (since $2^{10} = 1024$).

Similarly, cubic convergence roughly triples the number of correct decimal places at each step, quartic quadruples it, etc.

Rates of convergence of bisection, FPI and Newton's method

Suppose we are solving $f(x) = 0$ and that f' and f'' exist and are continuous. Then the following hold:

1. The bisection method converges linearly. In particular, we have the error relation $e_{i+1} \approx .5e_i$. That is, the error is cut in half with each iteration of the bisection method.
2. If we rewrite $f(x) = 0$ as a fixed point problem $g(x) = x$, and r is a fixed point of g (and hence a root of f) then FPI will converge at least linearly to r for initial guesses close to r. In particular, for large enough values of i, we have $e_{i+1} \approx |g'(r)|e_i$. If $|g'(r)| = 0$, the convergence turns out to be quadratic or better.
3. If $f'(r) \neq 0$ then Newton's method converges quadratically for initial guesses close to r.[1] If $f'(r) = 0$, then Newton's method converges linearly for initial guesses close to r.[2]

In the above, "close to r" formally means that there is some interval around r for which if the initial guess falls in that interval, then the iterates will converge to r. The formal term for this is *local convergence*. The size of that interval varies with the function.

In summary, if a few assumptions are met, then we know that FPI converges linearly with rate $|g'(r)|$, bisection converges linearly with rate .5, and Newton's method converges quadratically (at simple roots). And those assumptions can't entirely be ignored. For instance, $f(x) = \sqrt{x}$ has a root at $x = 0$, and $f''(0)$ does not exist. For this function, Newton's method turns out to converge very slowly to the root at 0, not quadratically.

Generalizations of Newton's method

There are generalizations of Newton's method that converge even more quickly. For instance, there is *Halley's method*, which involves iterating

$$x - \frac{2f(x)f'(x)}{2f'(x)^2 - f(x)f''(x)}.$$

Halley's method converges cubically. Halley's method and Newton's method are special cases of a more general class of algorithms called *Householder's methods*, which have convergence of all orders. However, even though

[1] In particular, $e_{i+1} \approx Ce_i^2$, where $C = \left|\frac{f''(r)}{2f'(r)}\right|$. This result requires that f'' exist and be continuous.

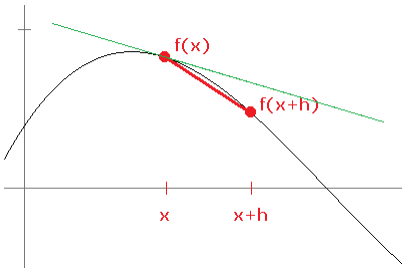
[2] In this case, we can still get quadratic convergence by changing the iteration formula to $x - mf(x)/f'(x)$, where m is the smallest integer for which $f^{(m)}(r) \neq 0$ (i.e., m is the multiplicity of the root).

those methods converge very quickly, that fast convergence comes at the cost of a more complex formula, requiring more operations to evaluate. The amount of time to compute each iterate may outweigh the gains from needing less iterations.

2.5 The Secant method

One weakness of Newton's method is that it requires derivatives. This is usually not a problem, but for certain functions, getting and using the derivative might be inconvenient, difficult, or computationally expensive. The *secant method* uses the same ideas from Newton's method, but it approximates the derivative (slope of the tangent line) with the slope of a secant line.

Remember that the derivative of a function f at a point x is the slope of the tangent line at that point. The definition of the derivative, given by $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{x+h}$, is gotten by approximating the slope of that tangent line by the slopes of secant lines between x and a nearby point $x+h$. As h gets smaller, $x+h$ gets closer to x and the secant lines start to look more like tangent lines. In the figure below a secant line is shown in red and the tangent line in green.



The formula we iterate for Newton's method is

$$x - \frac{f(x)}{f'(x)}.$$

It is helpful to explicitly describe the iteration as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

That is, the next iterate comes from applying the Newton's method formula to the previous iterate. For the secant method, we replace the derivative with slope of the secant line created from the previous two iterates and get:

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}}.$$

We can then apply a little algebra to simplify the formula into the following:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n-x_{n-1})}{f(x_n)-f(x_{n-1})}.$$

Note that this method actually requires us to keep track of two previous iterates.

As an example, let's use the secant method to estimate a root of $f(x) = 1-2x-x^5$. Since the formula needs two previous iterations, we will need two starting values. For simplicity, we'll pick $x_0 = 2$ and $x_1 = 1$. We have $f(x_0) = -35$ and $f(x_1) = -2$. We then get

$$x_2 = x_1 - \frac{f(x_1)(x_1-x_0)}{f(x_1)-f(x_0)} = 1 - \frac{-2(1-2)}{-2--35} = .939393\ldots.$$

We then compute x_3 in the same way, using x_1 and x_2 in place of x_0 and x_1. Then we compute x_4, x_5, etc. Here are the first few iterates:

```
0.9393939393939394
0.6889372446790889
0.5649990748325172
0.49771298490941546
0.48690106666448396
0.4863920162110657
0.4863890367053524
0.48638903593454413
0.486389035934543
0.486389035934543
```

We can see that this converges quickly, but not as quickly as Newton's method. By approximating the derivative, we lose some efficiency. Whereas the order of convergence of Newton's method is usually 2, the order of convergence of the secant method is usually the golden ratio, $\phi = 1.618\ldots$.[1]

An implementation in Python

Here is a simple implementation of the secant method in Python:

```
def secant(f, a, b, toler=1e-10):
    while f(b)!=0 and abs(b-a)>toler:
        a,b = b, b - f(b)*(b-a)/(f(b)-f(a))
    return b
```

Here is how we might call this to estimate a solution to $x^5 - x + 1 = 0$ correct to within 10^{-10}:

```
secant(lambda x:1-2*x-x**5, 2, 1)
```

In the function, a and b are used for x_{n-1} and x_n in the formula. The loop runs until either $f(b)$ is 0 or the difference between two successive iterations is less than some (small) tolerance. The default tolerance is 10^{-10}, but the caller can specify a different tolerance. The assignment line sets a to the old value of b and b to the value from the secant method formula.

Our implementation is simple, but it is not very efficient or robust. For one thing, we recompute $f(b)$ (which is $f(x_n)$) at each step, even though we already computed it when we did $f(a)$ (which is $f(x_{n-1})$ at the previous step. A smarter approach would be to save the value. Also, it is quite possible for us to get caught in an infinite loop, especially if our starting values are poorly chosen. See *Numerical Recipes* for a faster and safer implementation.

An alternative formula

A close relative of Newton's method and the secant method is to iterate the formula below:

$$x - \frac{f(x)}{\frac{f(x+h)-f(x-h)}{2h}},$$

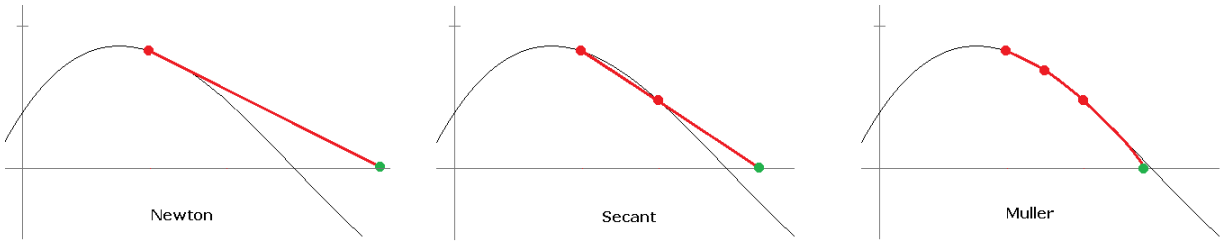
where h is a small, but not too small, number (10^{-5} is a reasonably good choice). The denominator is a numerical approximation to the derivative. However, the secant method is usually preferred to this technique. The numerical derivative suffers from some floating-point problems (see Section 4.1), and this method doesn't converge all that much faster than the secant method.

2.6 Muller's method, inverse quadratic interpolation, and Brent's Method

Muller's method is a generalization of the secant method. Recall that Newton's method involves following the tangent line down to the x-axis. That point where the line crosses the axis will often be closer to the root than the

[1] Note we are only guaranteed converge if we are sufficiently close to the root. We also need the function to be relatively smooth — specifically, f'' must exist and be continuous. Also, the root must be a simple root (i.e. of multiplicity 1). Function are pretty flat near a multiple root, which makes it slow for tangent and secant lines to approach the root.

previous point. The secant method is similar except that it follows a secant line down to the axis. What Muller's method does is it follows a parabola down to the axis. Whereas a secant line approximates the function via two points, a parabola uses three points, which can give a closer approximation.



Here is the formula for Muller's (where f_i denotes $f(x_i)$ for any integer i):

$$\begin{aligned} D &= (x_{n-2}-x_n)(x_{n-1}-x_n)(x_{n-2}-x_{n-1}) \\ a &= \frac{(x_{n-1}-x_n)(f_{n+2}-f_n)-(x_n-x_{n-2})(f_{n-1}-f_n)}{D} \\ b &= \frac{(x_{n-2}-x_n)^2(f_{n+1}-f_n)-(x_{n-1}-x_n)^2(f_{n-2}-f_n)}{D} \\ c &= f_n \\ x_{n+1} &= x_n - \frac{2c}{b+\operatorname{sgn}(b)\sqrt{b^2-4ac}}. \end{aligned}$$

This comes from finding the equation of the parabola through the three points and solving it using the quadratic formula. The form of the quadratic formula used here is one that suffers less from floating-point problems caused by subtracting nearly equal numbers.

Inverse quadratic interpolation is a relative of Muller's method. Like Muller's, it uses a parabola, but it approximates f^{-1} instead of f. This is essentially the same as approximating f by the inverse of a quadratic function. After some work (which we omit), we get the following iteration:

$$a = \frac{f_{n-1}}{f_n}, \quad b = \frac{f_{n-1}}{f_{n-2}}, \quad c = \frac{f_{n-2}}{f_n}$$

$$x_{n+1} = x_{n-1} + \frac{b\,(c(a-c)(x_n-x_{n-1})-(1-a)(x_{n-1}-x_{n-2}))}{(a-1)(b-1)(c-1)}$$

Both Muller's method and inverse quadratic interpolation have an order of convergence of about 1.839.[1]

Brent's method

Recall that we have a and b with $f(a)$ and $f(b)$ of opposite signs, like in the bisection method, we say a root is *bracketed* by a and b. That is, the root is contained inside that interval.

Brent's method is one of the most widely used root-finding methods. You start by giving it a bracketing interval. It then uses a combination of the bisection method and inverse quadratic interpolation to find a root in that interval. In simple terms, Brent's method uses inverse quadratic interpolation unless it senses something is going awry (like an iterate leaving the bracketing interval, for example), then it switches to the safe bisection method for a bit before trying inverse quadratic interpolation again. In practice, it's a little more complicated than this. There are a number of special cases built into the algorithm. It has been carefully designed to ensure quick, guaranteed convergence in a variety of situations. It converges at least as fast as the bisection method and most of the time is better, converging superlinearly.

[1] Like with the secant method, this result holds for starting values near a simple root, and the function must be relatively smooth. Note also that the order of the secant method, 1.618…, is the golden ratio, related to the Fibonacci numbers. The order of Muller's and IQI, 1.839…, is the tribonacci constant, related to the tribonacci numbers, defined by the recurrence $x_{n+1} = x_n + x_{n-1} + x_{n-2}$. That number is the solution to $x^3 = x^2 + x + 1$, whereas the golden ratio is the solution to $x^2 = x + 1$.

There are enough special cases to consider that the description of the algorithm becomes a bit messy. So we won't present it here, but a web search for Brent's method will turn up a variety of resources.

Ridder's method

Ridder's method is a variation on bisection/false position that is easy to implement and works fairly well. *Numerical Recipes* recommends it as an alternative to Brent's method in many situations.

The method starts out with a and b, like in the bisection method, where one of $f(a)$ and $f(b)$ is positive and the other is negative. We then compute the midpoint $m = (a+b)/2$ and the expression

$$z = m \pm \frac{(m-a)f(m)}{\sqrt{f(m)^2 - f(a)f(b)}}.$$

The sign of the $\pm$ is positive if $f(a) > f(b)$ and negative otherwise. We then adjust a or b just like in the bisection method, setting $a = z$ if $f(a)$ and $f(z)$ have the same sign and $b = z$ otherwise. Repeat the process as many times as needed.

This equation gives us an improvement over the bisection method, which uses m as its estimate, leading to quadratic convergence. It is, however, slower than Newton's method, as it requires two new function evaluations at each step as opposed to only one for Newton's method. But it is more reliable than Newton's method.

2.7 Some more details on root-finding

Speeding up convergence

There is an algorithm called Aitken's Δ^2 method that can take a slowly converging sequence of iterates and speeds things up. Given a sequence $\{x_n\}$ create a new sequence $\{y_n\}$ defined as below:

$$y_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_n - 2x_{n+1} + x_{n+2}}.$$

This is essentially an algebraic transformation of the original sequence. It comes from fact that once we are close to a root, the ratio of successive errors doesn't change much. That is,

$$\frac{x_{n+2} - r}{x_{n+1} - r} \approx \frac{x_{n+1} - r}{x_n - r}.$$

Treating this as an equation and solving for r gives the formula above (after a fair bit of algebra). Here is a Python function that displays an FPI sequence and the sped up version side by side.

```
def fpid2(g,s,n):
    x,x1,x2 = s,g(s),g(g(s))
    for i in range(n):
        x,x1,x2 = x1,x2,g(x2)
        print(x,(x2*x-x1*x1)/(x2-2*x1+x))
```

Here are the first few iterations we get from iterating $\cos x$.

FPI	sped up FPI
0.87758	0.73609
0.63901	0.73765
0.80269	0.73847
0.69478	0.73880
0.76820	0.73896
0.71917	0.73903
0.75236	0.73906
0.73008	0.73907
0.74512	0.73908
0.73501	0.73908

The fixed point of $\cos x$ is .73908 to five decimal places and we can see that after 10 iterations, the sped-up version is correct to that point. On the other hand, it takes the regular FPI about 20 more iterations to get to that point (at which point the sped-up version is already correct to 12 places).

Finding roots of polynomials

The root-finding methods considered in previous sections can be used to find roots of all sorts of functions. But probably the most important and useful functions are polynomials. There are a few methods that are designed specifically for finding roots of polynomials. Because they are specifically tailored to one type of function, namely polynomials, they can be faster than a method that has to work for any type of function. An important method is Laguerre's method. Here is how it works to find a root of a polynomial p:

Pick a starting value x_0. Compute $G = \frac{p'(x_0)}{p(x_0)}$ and $H = G^2 - \frac{p''(x_0)}{p(x_0)}$. Letting n be the degree of the polynomial, compute

$$x_1 = x_0 - \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}.$$

In the denominator, choose whichever sign makes the absolute value of the denominator larger. Keep iterating to find x_2, x_3, etc. The method converges cubically, so we usually won't need many iterations.

Laguerre's method actually comes from writing the polynomial in factored form $p(x) = c(x-r_1)(x-r_2)\dots(x-r_n)$. If we take the logarithm of both sides and use the multiplicative property of logarithms, we get

$$\ln|p(x)| = \ln|c| + \ln|x-r_1| + \ln|x-r_2| + \cdots + \ln|x-r_n|.$$

Then take the derivative of both sides of this expression twice to get

$$G(x) = \frac{p'(x)}{p(x)} = \frac{1}{x-r_1} + \frac{1}{x-r_2} + \cdots + \frac{1}{x-r_n}$$

$$H(x) = \left(\frac{p'(x)}{p(x)}\right)^2 - \frac{p''(x)}{p(x)} = \frac{1}{(x-r_1)^2} + \frac{1}{(x-r_2)^2} + \cdots + \frac{1}{(x-r_n)^2}.$$

Now, for a given value of x, $|x-r_i|$ represents the distance between x and r_i. Suppose we are trying to estimate r_1. Once we get close to r_1, it might be true that all the other roots will seem far away by comparison, and so in some sense, they are all the same distance away, at least in comparison to the distance from r_1. Laguerre's method suggests that we pretend this is true and assume $|x-r_2| = |x-r_3| = \cdots = |x-r_n|$. If we do this, the expressions for G and H above simplify considerably, and we get an equation that we can solve. In particular, let $a = |x-r_1$ and $b = |x-r_2| = |x-r_3| = \cdots = |x-r_n|$. Then the expressions for G and H simplify into

$$G(x) = \frac{1}{a} + \frac{n-1}{b} \qquad H(x) = \frac{1}{a^2} + \frac{n-1}{b^2}.$$

A little algebra allows us to solve for a to get the formula from the iterative process described above. See *A Friendly Introduction to Numerical Analysis* by Bradie for a more complete derivation.[1]

Laguerre's method works surprisingly well, usually converging to a root and doing so cubically.

One useful fact is that we can use a simple algorithm to compute $p(x_0)$, $p'(x_0)$, and $p''(x_0)$ all at once. The idea is that if we use long or synthetic division to compute $p(x)/(x-x_0)$, we are essentially rewriting $p(x)$ as $(x-x_0)q(x)+r$ for some polynomial q and integer r (the quotient and remainder). Plugging in $x = x_0$ gives $p(x_0) = r$. Taking a derivative of this and plugging in x_0 also gives $p'(x_0) = q(x_0)$. So division actually gives both the function value and derivative at once. We can then apply this process to $q(x)$ to get $p''(x_0)$. Here is a Python function that does this. The polynomial is given as a list of its coefficients (with the constant term first and the coefficient of the highest term last).

```
def eval_poly(P, x):
    p, q, r = P[-1], 0, 0

    for c in P[-2::-1]:
```

[1] Most of the discussion about Laguerre's method here is based off of Bradie's on pages 128-131.

```
        r = r*x + q
        q = q*x + p
        p = p*x + c

    return p, q, 2*r
```

There are many other methods for finding roots of polynomials. One important one is the Jenkins-Traub method, which is used in a number of software packages. It is quite a bit more complicated than Laguerre's method, so we won't describe it here. But it is fast and reliable.

There is also a linear-algebraic approach that involves constructing a matrix, called the companion matrix, from the polynomial, and using a numerical method to find its eigenvalues (which coincide with the roots of the polynomial). This is essentially the inverse of what you learn in an introductory linear algebra class, where you solve polynomials to find eigenvalues. Here we are going in the reverse direction and taking advantage of some fast eigenvalue algorithms.

Another approach is to use a root isolation method to find small disjoint intervals, each containing one of the roots and then to apply a method, such as Newton's, to zero in on it. One popular, recent algorithm uses a result called Vincent's theorem is isolate the roots. Finally, there is the Lindsey-Fox method, uses the Fast Fourier Transform to compute the roots.

Finding all the roots of an equation

Most of the methods we have talked about find just one root. If we want to find all the roots of an equation, one option is to find one (approximate) root r, divide the original equation by $x-r$ and then apply the method above to find another. This process is called *deflation*.

Finding complex roots

Some of the methods above, like the bisection method, will only find real roots. Others, like Newton's method or the secant method can find complex roots if the starting values are complex. For instance, starting Newton's method with $x_0 = .5 + .5i$ will lead us to the root $x = -.5 + \frac{\sqrt{3}}{2}i$ for $f(x) = x^3 - 1$. Muller's method can find complex roots, even if the starting value is real, owing to the square root in its formula. Another method, called Bairstow's method can find complex roots using only real arithmetic.

Systems of nonlinear equations

Some of the methods, like Newton's method and the secant method, can be generalized to numerically solve systems of nonlinear equations.

2.8 Measuring the accuracy of root-finding methods

There are a couple of ways to measure how accurate an approximation is. One way, assuming we know what the actual root is, would be to compute the difference between the root r and the approximation x_n, namely, $|x_n - r|$. This is called the *forward error*. In practice, we don't usually have the exact root (since that is what we are trying to find). For some methods, we can derive estimates of the error. For example, for the bisection method we have $|x_n - r| < |a_0 - b_0|/2^{n+1}$.

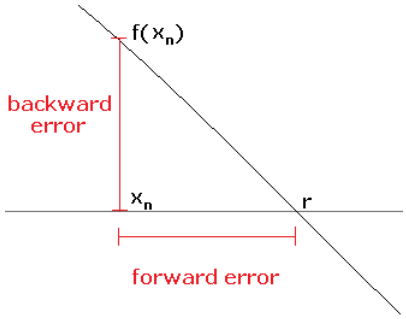
If the difference between successive iterates, $|x_n - x_{n-1}|$ is less than some tolerance ϵ, then we can be pretty sure (but unfortunately not certain) that the iterates are within ϵ of the actual root. This is often used to determine when to stop the iteration. For example, here is how we might code this stopping condition in Python:

```
while abs(x - oldx) > .0000001:
```

Sometimes it is better to compute the relative error: $\left|\frac{x_n - x_{n-1}}{x_{n-1}}\right|$.

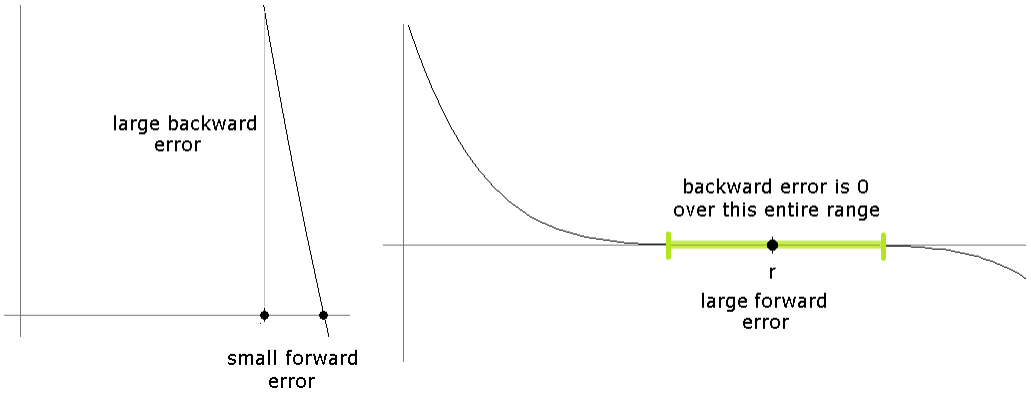
Another way to measure accuracy is the *backward error*. For a given iterate x_n, the backward error is $|f(x_n)|$. For example, suppose we are looking for a root of $f(x) = x^2 - 2$ and we have an approximation $x_n = 1.414$. The backward error here is $|f(1.414)| = 0.000604$. We know that at the exact root r, $f(r) = 0$, and the backward error is measuring how far away we are from that.

The forward error is essentially the error in the x-direction, while the backward error is the error in the y-direction.[1] See the figure below:



We can also use backward error as a stopping condition.

If a function is almost vertical near a root, the forward error will not be a good measure as we could be quite near a root, but $f(x_n)$ will still be far from 0. On the other hand, if a function is almost horizontal near a root, the backward error will not be a good measure as $f(x_n)$ will be close to 0 even for values far from the root. See the figure below.

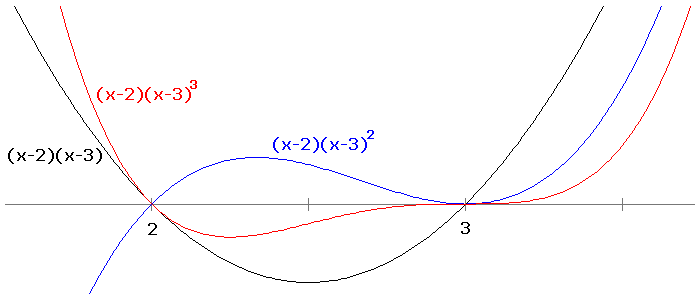


2.9 What can go wrong with root-finding

These methods generally don't do well with multiple roots

There are two types of roots: *simple roots* and *multiple roots*. For a polynomial, we can tell whether a root is simple or multiple by looking at its factored form. Those roots that are raised to the power of 1 are simple and the others are multiple. For instance, in $(x-3)^4(x-5)^2(x-6)$, 6 is a simple root, while 3 and 5 are multiple roots. The definition of simple and multiple roots that works for functions in general involves the derivative: namely, a root r is simple if $f'(r) \neq 0$ and multiple otherwise. The multiplicity of the root is the smallest m such that $f^{(m)}(r) \neq 0$. Graphically, the higher the multiplicity of a root, the flatter the graph is around that root. Shown below are the graphs of $(x-2)(x-3)$, $(x-2)(x-3)^2$, and $(x-2)(x-3)^3$.

[1] Some authors use a different terminology, using the terms *forward* and *backward* in the opposite way that we do here.



When the graph is flat near a root, it means that it will be hard to distinguish the function values of nearby points from 0, making it hard for a numerical method to get close to the root. For example, consider $f(x) = (x-1)^5$. In practice, we might encounter this polynomial in its expanded form, $x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$. Suppose we try Newton's method on this with a starting value of 1.01. After about 20 iterations, the iterates stabilize around `1.0011417013971329`. We are only correct to two decimal places. If we use a computer (using double-precision floating-point) to evaluate the function at this point, we get $f(1.0011417013971329) = 0$ (the actual value should be about 1.94×10^{-15}, but floating-point issues have made it indistinguishable from 0). For x-values in this range and closer, the value of $f(x)$ is less than machine epsilon away from 0. For instance, $f(1.000001) = 10^{-30}$, so there is no chance of distinguishing that from 0 in double-precision floating point.

The lesson is that the root-finding methods we have considered can have serious problems at multiple roots. More sophisticated approaches or more precision can overcome these problems.

Sensitivity to small changes

The Wilkinson polynomial is the polynomial $W(x) = (x-1)(x-2)(x-3)\cdots(x-20)$. In this factored form, we can see that it has simple roots at the integers 1 through 20. Here are a few terms of its expanded form:

$$x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + \cdots - 8752948036761600000x + 2432902008176640000$$

James Wilkinson (who investigated it), examined what happened if the second coefficient was changed from -210 to -210.0000001192, which was a change on the order of machine epsilon on the machine he was working on. He discovered that even this small change moved the roots quite a bit, with the root at 20 moving to roughly 20.85 and the roots at 16 and 17 actually moving to roughly $16.73 \pm 2.81i$.

Such a problem is sometimes called *ill-conditioned*. An ill-conditioned problem is one in which a small change to the problem can create a big change in the result. On the other hand, a well-conditioned problem is one where small changes have small effects.

The methods we have talked about will not work well if the problem is ill-conditioned. The problem will have to be approached from a different angle or else higher precision will be needed.

2.10 Root-finding summary

At this point, it's good to remind ourselves of why this stuff is important. The reason is that a ton of real-life problems require solutions to equations that are too hard to solve analytically. Numerical methods allow us to approximate solutions to such equations, often to whatever amount of accuracy is needed for the problem at hand.

It's also worth going back over the methods. The bisection method is essentially a binary search for the root that is probably the first method someone would come up with on their own, and the surprising thing is that it is one of the most reliable methods, albeit a slow one. It is a basis for faster approaches like Brent's method and Ridder's method.

Fixed point iteration is not the most practical method to use to find roots since it requires some work to rewrite the problem as a fixed point problem, but an understanding of FPI is useful for further studies of things like dynamical systems.

Newton's method is one of the most important root-finding methods. It works by following the tangent line down to the axis, then back up to the function and back down the tangent line, etc. It is fast, converging quadratically for a simple root. However, there is a good chance that Newton's method will fail if the starting point is not sufficiently close to the root. Often a reliable method is used to get relatively close to a root, and then a few iterations of Newton's method are used to quickly add some more precision.

Related to Newton's method is the secant method, which, instead of following tangent lines, follows secant lines built from the points of the previous two iterations. It is a little slower than Newton's method (order 1.618 vs. 2), but it does not require any derivatives. Building off of the secant method are Muller's method, which uses a parabola in place of a secant line, and inverse quadratic interpolation, which uses an inverse quadratic function in place of a secant line. These methods have order around 1.839. Brent's method, which is used in some software packages, uses a combination of IQI and bisection as a compromise between speed and reliability. Ridder's method is another reliable and fast algorithm that has the benefit of being much simpler than Brent's method. If we restrict ourselves to polynomials, there are a number of other, more efficient, approaches we can take.

We have just presented the most fundamental numerical root-finding methods here. They are prerequisite for understanding the more sophisticated methods, of which there are many, as root-finding is an active and important field of research. Hopefully, you will find the ideas behind the methods helpful in other, completely different contexts.

Many existing software packages (such as Mathematica, Matlab, Maple, Sage, etc.) have root-finding methods. The material covered here should hopefully give you an idea of what sorts of things these software packages might be using, as well as their limitations.

Chapter 3

Interpolation

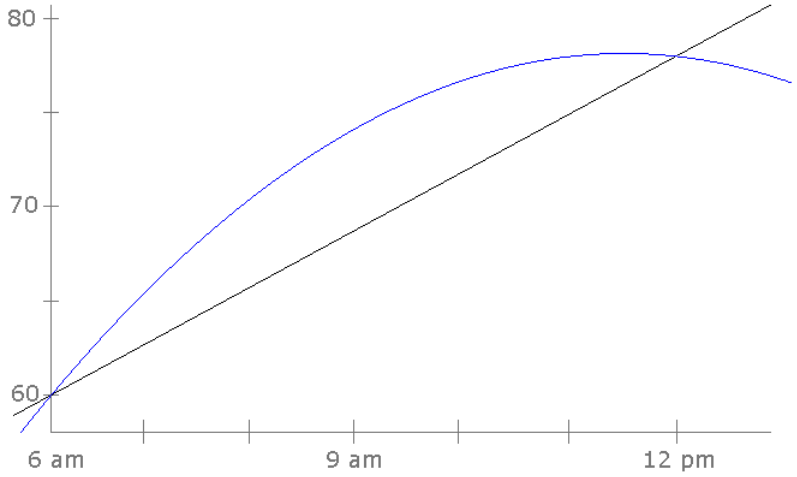
Very roughly speaking, interpolation is what you use when you have a few data points and want to use them to estimate the values between those data points. For example, suppose one day the temperature at 6 am is 60° and at noon it is 78°. What was the temperature at 10 am? We can't know for sure, but we can make a guess. If we assume the temperature rose at a constant rate, then 18° in 6 hours corresponds to a 3° rise per hour, which would give us a 10 am estimate of $60° + 4 \cdot 3° = 72°$. What we just did here is called *linear interpolation*: if we have a function whose value we know at two points and we assume it grows at a constant rate, we can use that to make guesses for function values between the two points.

However, a linear model is often just a rough approximation to real life. Suppose in the temperature example that we had another data point, namely that it was 66° at 7 am. How do we best make use of this new data point? The geometric fact that is useful here is that while two points determine a unique line, three points determine a unique parabola. So we will try to find the parabola that goes through the three data points.

One way to do this is to assume the equation is of the form $ax^2 + bx + c$ and find the coefficients a, b, and c. Take 6 am as $x = 0$, 7 am as $x = 1$, and noon as $x = 6$, and plug these values in to get the following system of equations:

$$\begin{aligned} c &= 60 \\ a + b + c &= 66 \\ 36a + 6b + c &= 78 \end{aligned}$$

We can solve this system using basic algebra or linear algebra to get $a = -3/5$, $b = 33/5$, and $c = 60$. So the formula we get is $-\frac{3}{5}x^2 + \frac{33}{5}x + 60$, and plugging in $x = 4$, we get that at 10 am the temperature is predicted to be 76.8°. The linear and quadratic models are shown below.



Notice that beyond noon, the models lose accuracy. The quadratic model predicts temperature will start dropping at noon, while linear model predicts the temperature will never stop growing. The "inter" in interpolation means "between" and these models may not be accurate outside of the data range.

3.1 The Lagrange form

The geometric facts we implicitly used in the example above are that two points determine a unique line (linear or degree 1 equation) and three points determine a unique parabola (quadratic or degree 2 equation). This fact extends to more points. For instance, four points determine a unique cubic (degree 3) equation, and in general, n points determine a unique degree $n-1$ polynomial.

So, given n points, (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), we want to find a formula for the unique polynomial of degree $n-1$ that passes directly through all of the points. One approach is a system of equations approach like the one we used above. This is fine if n is small, but for large values of n, the coefficients get very large leading to an ill-conditioned problem.[1]

A different approach is the *Lagrange formula* for the interpolating polynomial. It is given by

$$L(x) = y_1 L_1(x) + y_2 L_2(x) + \cdots + y_n L_n(x),$$

where for each $k = 1, 2, \ldots n$, we have

$$L_k(x) = \frac{(x-x_1)(x-x_2)\ldots\widehat{(x-x_k)}\ldots(x-x_n)}{(x_k-x_1)(x_k-x_2)\ldots\widehat{(x_k-x_k)}\ldots(x_k-x_n)}.$$

A hat over a term means that that term is not to be included.

As an example, let's use this formula on the temperature data, $(0, 60)$, $(1, 66)$, $(6, 78)$, from the previous section. To start, $L_1(x)$, $L_2(x)$, and $L_3(x)$ are given by

$$L_1(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}, \quad L_2(x) = \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}, \quad L_3(x) = \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}.$$

We then plug in the data to get

$$L_1(x) = \frac{(x-1)(x-6)}{(0-1)(0-6)}, \quad L_2(x) = \frac{(x-0)(x-6)}{(1-0)(1-6)}, \quad L_3(x) = \frac{(x-0)(x-1)}{(6-0)(6-1)}.$$

Plug these into

$$L(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x),$$

and simplify to get

$$L(x) = 10(x-1)(x-6) - \frac{66}{5}x(x-6) + \frac{13}{5}x(x-1).$$

We could further simplify this, but sometimes a factored form is preferable.[2] Lagrange's formula is useful in situations where we need an actual formula for some reason, often in theoretical circumstances. For practical computations, there is a process, called *Newton's divided differences*, that is less cumbersome to use.[3]

3.2 Newton's divided differences

Newton's divided differences is a method for finding the interpolating polynomial for collection of data points. It's probably easiest to start with some examples. Suppose we have data points $(0, 2)$, $(2, 2)$, and $(3, 4)$. We create the following table:

[1] There are numerical methods for finding solutions of systems of linear equations, and the large coefficients can cause problems similar to the ones we saw with polynomial root-finding.

[2] If we were to simplify it, we would get $-\frac{3}{5}x^2 + \frac{33}{5}x + 60$, which is what we got in the previous section using the system of equations.

[3] Things like this often happen in math, where there is a simple process to find the answer to something, while an explicit formula is messy. A good example is solving cubic equations. There is a formula akin to the quadratic formula, but it is rather hideous. On the other hand, there is a process of algebraic manipulations that is much easier to use and yields the same solutions.

$$
\begin{array}{c|c|c|c|c}
0 & \boxed{1} & & & \\
 & & \frac{2-1}{2-0}=\boxed{1/2} & & \\
2 & 2 & & \frac{2-1/2}{3-0}=\boxed{1/2} & \\
 & & \frac{4-2}{3-2}=2 & & \\
3 & 4 & & & \\
\end{array}
$$

The first two columns are the x- and y-values. The next column's values are essentially slopes, computed by subtracting the two adjacent y-values and dividing by the corresponding x-values. The last column is computed by subtracting the adjacent values in the previous column and dividing the result by the difference $3-0$ from the column of x-values. In this column, we subtract x-values that are two rows apart.

We then create the polynomial like below:

$$
\begin{aligned}
p(x) &= 1+\frac{1}{2}(x-0)+\frac{1}{2}(x-0)(x-2) \\
&= 1+\frac{1}{2}x+\frac{1}{2}x(x-2).
\end{aligned}
$$

The coefficients come from the top "row" of the table, the boxed entries. The x terms are progressively built up using the x-values from the first column.

Let's try another example using the points $(0,2)$, $(3,5)$, $(4,10)$ and $(6,8)$. Here is the table:

$$
\begin{array}{c|c|c|c|c}
0 & \boxed{2} & & & \\
 & & \frac{5-2}{3-0}=\boxed{1} & & \\
3 & 5 & & \frac{5-1}{4-0}=\boxed{1} & \\
 & & \frac{10-5}{4-3}=5 & & \frac{-2-1}{6-0}=\boxed{-1/2} \\
4 & 10 & & \frac{-1-5}{6-3}=-2 & \\
 & & \frac{8-10}{6-4}=-1 & & \\
6 & 8 & & & \\
\end{array}
$$

Notice that in the third column, the x-values that are subtracted come from rows that are two apart in the first column. In the fourth column, the x-values come from rows that are three apart. We then create the polynomial from the x-coordinates and the boxed entries:

$$
\begin{aligned}
p(x) &= 2+1(x-0)+1(x-0)(x-3)-\frac{1}{2}(x-0)(x-3)(x-4) \\
&= 2+x+x(x-3)-\frac{1}{2}x(x-3)(x-4).
\end{aligned}
$$

One nice thing about this approach is that if we add a new point, we can add it to the bottom of the previous table and not have to recompute the old entries. For example, if we add another point, $(5,8)$, here is the resulting table:

$$
\begin{array}{c|c|c|c|c|c}
0 & \boxed{2} & & & & \\
 & & \frac{5-2}{3-0}=\boxed{1} & & & \\
3 & 5 & & \frac{5-1}{4-0}=\boxed{1} & & \\
 & & \frac{10-5}{4-3}=5 & & \frac{-2-1}{6-0}=\boxed{-1/2} & \\
4 & 10 & & \frac{-1-5}{6-3}=-2 & & \frac{1-\!-1/2}{7-0}=\boxed{3/14} \\
 & & \frac{8-10}{6-4}=-1 & & \frac{2-\!-2}{7-3}=1 & \\
6 & 8 & & \frac{5-\!-1}{7-4}=2 & & \\
 & & \frac{13-8}{7-6}=5 & & & \\
7 & 13 & & & & \\
\end{array}
$$

Here is the polynomial we get:

$$\begin{aligned} p(x) &= 2 + 1(x-0) + 1(x-0)(x-3) - \frac{1}{2}(x-0)(x-3)(x-4) + \frac{3}{14}(x-0)(x-3)(x-4)(x-6) \\ &= 2 + x + x(x-3) - \frac{1}{2}x(x-3)(x-4) + \frac{3}{14}x(x-3)(x-4)(x-6). \end{aligned}$$

A general definition

Here is a succinct way to describe the process: Assume the data points are (x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1}). We can denote the table entry at row r, column c (to the right of the line) by $N_{r,c}$. The entries of the first column are the y-values, i.e., $N_{i,0} = y_i$, and the other entries are given by

$$N_{r,c} = \frac{N_{r+1,c-1} - N_{r,c-1}}{x_{r+c} - x_r}.$$

The notation here differs from the usual way Newton's Divided Differences is defined. The Wikipedia page http://en.wikipedia.org/wiki/Divided_differences has the usual approach.

A practical example

Here is world population data for several years:

Year	Millions of people
1960	3040
1970	3707
1990	5282
2000	6080

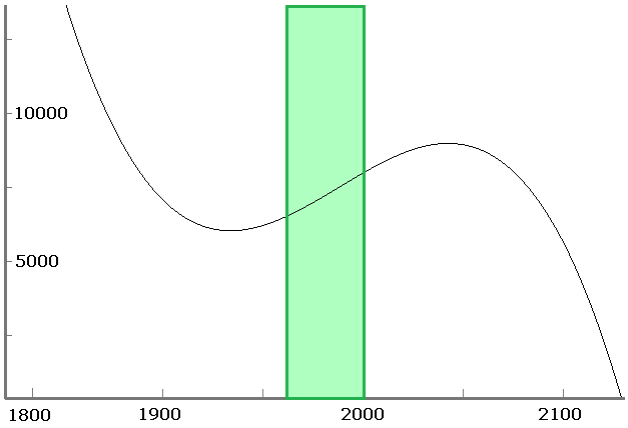
Let's interpolate to estimate the population in 1980 and compare it to the actual value. Using Newton's divided differences, we get the following:

1960	3040			
		66.7		
1970	3707		.4017	
		78.75		−.0092
1990	5282		.0350	
		79.8		
2000	6080			

This gives

$$f(x) = 3040 + 66.7(x-1960) + .4017(x-1960)(x-1970) - .0092(x-1960)(x-1970)(x-1990).$$

Plugging in $x = 1980$ gives 4473 million people in 1980, which is not too far off from the actual value, 4454. Note that this function interpolates pretty well, but it should not be used for extrapolating to years not between 1960 and 2000. While it does work relatively well for years near this range (e.g., for 2010 it predicts 6810, which is not too far off from the actual value, 6849), it does not work well for years far from the range. For instance, it predicts a population of about 50 billion in 1800. The figure below shows the usable range of the function in green:



Neville's algorithm

Related to Newton's divided differences is Neville's algorithm, which uses a similar table approach to compute the interpolating polynomial at a specific point. The entries in the table are computed a little differently, and the end result is just the value at the specific point, not the coefficients of the polynomial. This approach is numerically a bit more accurate than using Newton's divided differences to find the coefficients and then plugging in the value. Here is a small example to give an idea of how it works. Suppose we have data points $(1, 11)$, $(3, 15)$, $(4, 20)$, and $(5, 22)$, and we want to interpolate at $x = 2$. We create the following table:

$$\begin{array}{c|c|c|c|c} 1 & 11 & & & \\ & & \frac{(2-1)\cdot 15-(2-3)\cdot 11}{3-1}=13 & & \\ 3 & 15 & & \frac{(2-1)\cdot 10-(2-4)\cdot 13}{4-1}=12 & \\ & & \frac{(2-3)\cdot 20-(2-4)\cdot 15}{4-3}=10 & & \frac{(2-1)\cdot 7-(2-5)\cdot 12}{5-1}=\boxed{10.75} \\ 4 & 20 & & \frac{(2-3)\cdot 16-(2-5)\cdot 10}{5-3}=7 & \\ & & \frac{(2-4)\cdot 22-(2-5)\cdot 20}{5-4}=16 & & \\ 5 & 22 & & & \end{array}$$

The value of the interpolating polynomial at 2 is 10.75, the last entry in the table. The denominators work just like with Newton's divided differences. The numerators are similar in that they work with the two adjacent values from the previous column, but there are additional terms involving the point we are interpolating at and the x-values.

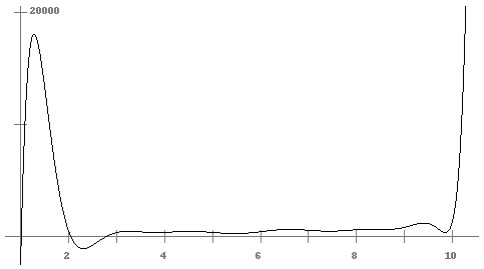
We can formally describe the entries in the table in a similar way to what we did with Newton's divided differences. Assume the data points are (x_0, y_0), (x_1, y_1), $\ldots$, (x_{n-1}, y_{n-1}). We can denote the table entry at row r, column c (to the right of the line) by $N_{r,c}$. The entries of the first column are the y-values, i.e., $N_{i,0} = y_i$, and the other entries are given by the following, where x is the value we are evaluating the interpolating polynomial at:

$$N_{r,c} = \frac{(x - x_r)N_{r+1,c-1} - (x - x_{r+c})N_{r,c-1}}{x_{r+c} - x_r}.$$

3.3 Problems with interpolation

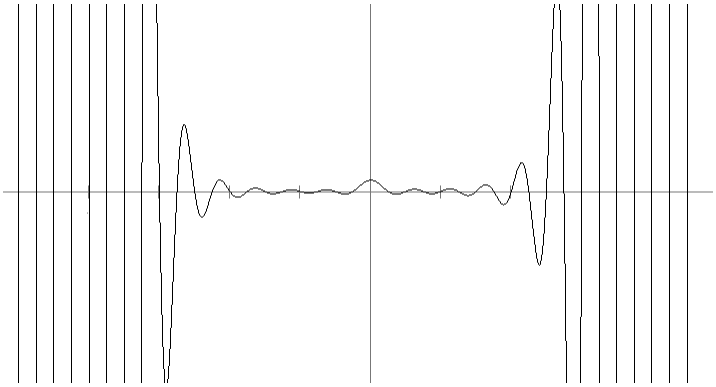
At www.census.gov/population/international/data/worldpop/table_history.php there is a table of the estimated world population by year going back several thousand years. Suppose we take the data from 1000 to 1900 (several dozen data points) and use it to interpolate to years that are not in the table. Using Newton's divided differences we get the polynomial that is graphed below.[1]

[1]The units for x-values are 100s of years from 1000 AD. For instance $x = 2.5$ corresponds to 1250. We could have used the actual years as x-values but that would make the coefficients of the polynomials extremely small. Rescaling so that the units are closer to zero helps keep the coefficients reasonable.

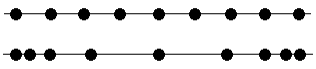


The thing to notice is the bad behavior between $x = 0$ and $x = 3$ (corresponding to years 1000 through 1200). In the middle of its range, the polynomial is well behaved, but around the edges there are problems. For instance, the interpolating polynomial gives a population of almost 20000 million people (20 billion) around 1025 and a negative population around 1250.

This is a common problem and is known as the *Runge phenomenon*. Basically, polynomials want to bounce up and down between points. We can control the oscillations in the middle of the range, but the edge can cause problems. For example, suppose we try to interpolate a function that is 0 everywhere except that $f(0) = 1$. If we use 20 evenly spaced points between -5 and 5, we get the following awful result:



The solution to this problem is to pick our points so that they are not evenly spaced.[1] We will use more points near the outside of our range and fewer inside. This will help keep the edges of the range under control. Here is an example of evenly-spaced points versus our improvement:



We will see exactly how to space those points shortly. First, we need to know a little about the maximum possible error we get from interpolation.

When interpolating we are actually trying to estimate some underlying function f. We usually don't know exactly what f is. However, we may have some idea about its behavior. If so, the following expression gives an upper bound for the error between the interpolated value at x and its actual value:

$$\left| \frac{(x-x_1)(x-x_2)\dots(x-x_n)}{n!} \max f^{(n)} \right|.$$

Here, $f^{(n)}$ refers to the nth derivative of f. The max is taken over the range that contains x and all the x_i. Here is an example of how the formula works: suppose we know that function changes relatively slowly, namely that $|f^{(n)}(x)| < 2$ for all n and all x. Suppose also that the points we are using have x-values at $x = 1$, 2, 4, and 6. The interpolation error at $x = 1.5$ must be less than

$$\left| \frac{(1.5-1)(1.5-2)(1.5-4)(1.5-6)}{4!} \times 2 \right| = .234375.$$

[1] In fact, evenly spaced points are usually one of the worst choices, which is unfortunate since a lot of real data is collected in that way.

With a fixed number of points, the only part of the error term that we can control is the $(x-x_1)(x-x_2)\dots(x-x_n)$ part. We want to find a way to make this as small as possible. In particular, we want to minimize the worst-case scenario. That is, we want to know what points to choose so that $(x-x_1)(x-x_2)\dots(x-x_n)$ never gets too out of hand.

3.4 Chebyshev polynomials

In terms of where exactly to space the interpolation points, the solution involves something called the *Chebyshev polynomials*. They will tell us exactly where to position our points to minimize the effect of the Runge phenomenon. In particular, they are the polynomials that minimize the $(x-x_1)(x-x_2)\dots(x-x_n)$ term in the error formula.

The nth Chebyshev polynomial is defined by

$$T_n(x) = \cos(n \arccos(x)).$$

This doesn't look anything like a polynomial, but it really is one. For the first few values of n, we get

$$\begin{aligned} T_0(x) &= \cos(0 \cdot \arccos x) = 1 \\ T_1(x) &= \cos(1 \cdot \arccos x) = x \\ T_2(x) &= \cos(2 \cdot \arccos x) = 2\cos^2(\arccos x) - 1 = 2x^2 - 1 \end{aligned}$$

The formula for $T_2(x)$ comes from the identity $\cos(2\theta) = 2\cos^2\theta - 1$. We can keep going to figure out $T_3(x)$, $T_4(x)$, etc., but the trig gets messy. Instead, we will make clever use of the trig identities for $\cos(\alpha \pm \beta)$. Let $y = \arccos x$. Then we have

$$T_{n+1}(x) = \cos((n+1)y) = \cos(ny + y) = \cos(ny)\cos y - \sin(ny)\sin y.$$

Similarly, we can compute that

$$T_{n-1}(x) = \cos((n-1)y) = \cos(ny - y) = \cos(ny)\cos y + \sin(ny)\sin y.$$

If we add these two equations, we get

$$T_{n+1}(x) + T_{n-1}(x) = 2\cos(ny)\cos y = 2T_n(x) \cdot x.$$

In other words, we have found a recursive formula for the polynomials. Namely,

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Using the fact that $T_0(x) = 1$ and $T_1(x) = x$, we can use this formula to compute the next several polynomials:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ T_5(x) &= 16x^5 - 20x^3 + 5x \\ T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \\ T_7(x) &= 64x^7 - 112x^5 + 56x^3 - 7x. \end{aligned}$$

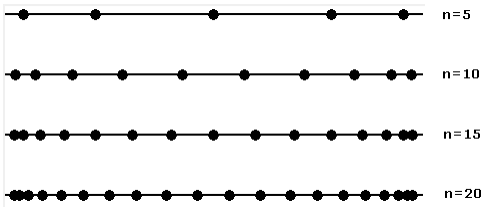
There a few things we can quickly determine about the Chebyshev polynomials:

1. For $n > 0$, the leading coefficient of $T_n(x)$ is 2^{n-1}. This can easily be seen from the recursive formula.
2. Because $T_n(x)$ is $\cos(\arccos x)$, and cosine stays between -1 and 1, $T_n(x)$ will stay between -1 and 1, at least if $-1 \le x \le 1$ (which is the domain of the arccosine.)

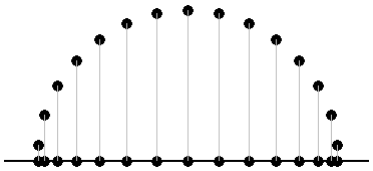
3. The roots of the polynomials can be determined as follows: We know the roots of $\cos x$ are $\pm\frac{\pi}{2}$, $\pm\frac{3\pi}{2}$, $\pm\frac{5\pi}{2}$, ... Therefore, to find the roots of $T_n(x) = \cos(n \arccos x)$, we set $n \arccos x$ equal to these values. Solving, we get that the roots are
$$x_i = \cos\left(\frac{(2i-1)\pi}{2n}\right) \quad \text{for } i = 1, 2, \ldots, n.$$
For example, if $n = 8$, the roots are $\cos\frac{\pi}{16}$, $\cos\frac{3\pi}{16}$, $\cos\frac{5\pi}{16}$, $\cos\frac{7\pi}{16}$, $\cos\frac{9\pi}{16}$, $\cos\frac{11\pi}{16}$, $\cos\frac{13\pi}{16}$, and $\cos\frac{15\pi}{16}$. Note that in the formula above, if $i > n$, then we would fall outside of the interval $[-1, 1]$.

The roots of the Chebyshev polynomials (which we will call *Chebyshev points* tell us exactly where to best position the interpolation points in order to minimize interpolation error and the Runge effect.

Here are the points for a few values of n. We see that the points become more numerous as we reach the edge of the range.



In fact, the points being cosines of linearly increasing angles means that they are equally spaced around a circle, like below:



If we have a more general interval, $[a, b]$, we have to scale and translate from $[-1, 1]$. The key is that to go from $[-1, 1]$ to $[a, b]$, we have to stretch or contract $[-1, 1]$ by a factor of $\frac{b-a}{1--1} = \frac{b-a}{2}$ and translate it so that its center moves to $\frac{b+a}{2}$. Applying these transformations to the Chebyshev points, we get the following
$$x_i = \frac{b-a}{2}\cos\left(\frac{(2i-1)\pi}{2n}\right) + \frac{b+a}{2}, \quad \text{for } i = 1, 2, \ldots, n.$$

Example

As an example, suppose we want to interpolate a function on the range $[0, 20]$ using 5 points. The formulas above tell us what points to pick in order to minimize interpolation error. In particular,
$$\begin{aligned}
x_i &= \frac{20-0}{2}\cos\left(\frac{(2i-1)\pi}{2\cdot 5}\right) + \frac{20+0}{2} \quad \text{for } i = 1, 2, 3, 4, 5 \\
&= 10\cos\left(\frac{(2i-1)\pi}{10}\right) + 10 \quad \text{for } i = 1, 2, 3, 4, 5 \\
&= 10\cos\tfrac{\pi}{10} + 10, \quad 10\cos\tfrac{3\pi}{10} + 10, \quad 10\cos\tfrac{5\pi}{10} + 10, \quad 10\cos\tfrac{7\pi}{10} + 10, \quad 10\cos\tfrac{9\pi}{10} + 10 \\
&\approx 19.51, \quad 15.88, \quad 10.00, \quad 4.12, \quad .49
\end{aligned}$$

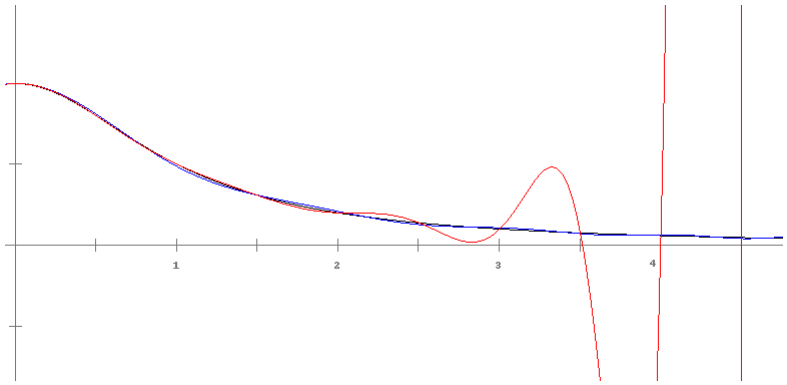
Error formula

Plugging the Chebyshev points into the error formula, we are guaranteed that for any $x \in [a, b]$ the interpolation error will be less than
$$\frac{\left(\frac{b-a}{2}\right)^n}{n!2^{n-1}}\left|\max f^{(n)}\right|$$

where the max is taken over the interval $[a, b]$.

A comparison

Below is a comparison of interpolation using 21 evenly-spaced nodes (in red) versus 21 Chebyshev nodes (in blue) to approximate the function $f(x) = 1/(1+x^2)$ on $[-5,5]$. Just the interval $[0,5]$ is shown, as the other half is the similar. We see that the Runge phenomenon has been mostly eliminated by the Chebyshev nodes.



3.5 Approximating functions

One use for interpolation is to approximate functions with polynomials. This is a useful thing because a polynomial approximation is often easier to work with than the original function.

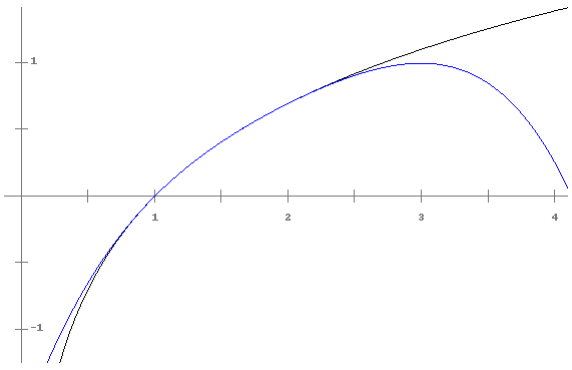
As an example, let's approximate $\ln x$ on the interval $[1,2]$ using five points. The Chebyshev points are

$$x_i = \frac{1}{2}\cos\left(\frac{(2i-1)\pi}{10}\right) + \frac{3}{2} \quad \text{for } i = 1,2,3,4,5.$$

Then using Newton's divided differences, we get the following polynomial (with all the coefficients shown to four decimal places for brevity):

$$\begin{aligned} &0.0242 \\ &+0.8986(x-1.0245) \\ &-0.3294(x-1.0245)(x-1.206) \\ &+0.1336(x-1.0245)(x-1.2061)(x-1.5) \\ &-0.0544(x-1.0245)(x-1.2061)(x-1.5)(x-1.7939). \end{aligned}$$

The polynomial and $\ln x$ are shown below.



We can see that they match up pretty well on $[1,2]$. Using the error formula to compute the maximum error over the range, we see that the error must be less than $4!/(5!\cdot 2^4) = 1/80$. (The 4! comes from finding the maximum of the fourth derivative of $\ln x$.) But in fact, this is an overestimate. The maximum error actually happens at $x = 1$ and we can compute it to be roughly 7.94×10^{-5}.

You might quibble that we actually needed to know some values of $\ln x$ to create the polynomial. This is true, but it's just a couple of values and they can be computed by some other means. Those couple of values are then used to estimate $\ln x$ over the entire interval. In fact, to within an error of about 7.94×10^{-5}, we are representing the entire range of values the logarithm can take on in $[1,2]$ by just 9 numbers (the coefficients and interpolation points).

But we can actually use this to get $\ln x$ on its entire range, $(0,\infty)$. Suppose we want to compute $\ln(20)$. If we keep dividing 20 by 2, eventually we will get to a value between 1 and 2. In fact, $20/2^4$ is between 1 and 2. We then have

$$\ln(20) = \ln\left(2^4 \cdot \frac{20}{2^4}\right) = 4\ln(2) + \ln\left(\frac{20}{16}\right).$$

We can use our polynomial to estimate $\ln(20/16)$ and we can compute $\ln 2$ by other means and get an approximate answer that turns out to be off by about 3.3×10^{-5}.

This technique works in general. Given any $x \in (2,\infty)$, we repeatedly divide x by 2 until we get a value between 1 and 2. Say this takes d divisions. Then our estimate for $\ln(x)$ is $d\ln 2 + P(x/2^d)$, where P is our interpolating polynomial. A similar method can give us $\ln(x)$ for $x \in (0,1)$: namely, we repeatedly multiply x until we are between 1 and 2, and our estimate is $P(2^d x) - d\ln 2$.

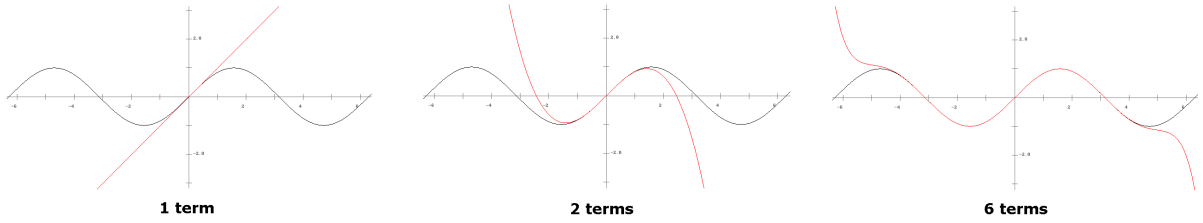
General techniques for approximating functions

The example above brings up the question of how exactly computers and calculators compute functions.

One simple way to approximate a function is to use its Taylor series. If a function is reasonably well-behaved, it can be written as an infinitely long polynomial. For instance, we can write

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

In other words, we can write the sine function as a polynomial, provided we use an infinitely long polynomial. We can get a reasonably good approximation of $\sin x$ by using just the few terms from the series. Shown below are a few such approximations.



The first approximation, $\sin x \approx x$, is used quite often in practice. It is quite accurate for small angles.

In general, the Taylor series of a function $f(x)$ centered at a is

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{f'(a)}{2!}(x-a)^2 + \frac{f''(a)}{3!}(x-a)^3 + \ldots.$$

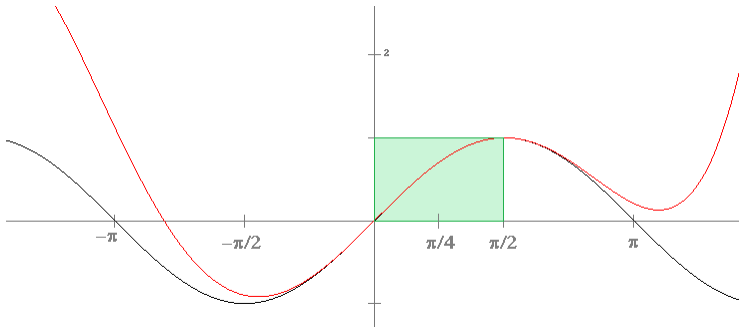
For example, if we want the Taylor series of $f(x) = \cos x$ centered at $a = 0$, we would start by computing derivatives of $\cos x$. The first few are $f'(x) = -\sin x$, $f''(x) = -\cos x$, $f'''(x) = \sin x$, and $f^{(4)}(x) = \cos x$. Since $f^{(4)}(x) = f(x)$, we will have a nice repeating pattern in the numerator of the Taylor series formula. Plugging $a = 0$ into each of these and using the Taylor series formula, we get

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots.$$

Taylor series are often very accurate near the center but inaccurate as you move farther away. This is the case for $\cos x$, but there things we can do to improve this. To start, let's move the center to $\pi/4$. An approximation using six terms is

$$\frac{\sqrt{2}}{2}\left(1+(x-\pi/4)-\frac{(x-\pi/4)^2}{2!}-\frac{(x-\pi/4)^3}{3!}+\frac{(x-\pi/4)^4}{4!}+\frac{(x-\pi/4)^5}{5!}\right).$$

Its graph is shown below:



It is a very close approximation on the interval $[0,\pi/2]$. But this is all we need to get $\cos x$ for any real number x. Looking at figure above, we see that $\cos x$ between $-\pi/2$ and π is a mirror image of $\cos x$ between 0 and $\pi/2$. In fact, there is a trig identity that says $\cos(\pi/2+x)=\cos(\pi/2-x)$. So, since we know $\cos x$ on $[0,\pi/2]$, we can use that to get $\cos x$ on $[\pi/2,\pi]$. Similar use of trig identities can then get us $\cos x$ for any other x.

Fourier series

Taylor series are simple, but they often aren't the best approximation to use. One way to think of Taylor series is we are trying to write a function using terms of the form $(x-a)^n$. Basically, we are given a function $f(x)$ and we are trying to write

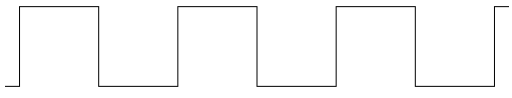
$$f(x)=\sum_{n=0}^{\infty}c_n(x-a)^n$$

where we want to find what the coefficients c_n are. For Taylor series, those coefficients are determined using derivatives and factorials. We can generalize this idea and try to write $f(x)$ using different types of terms. For example, a very important type of expansion is the Fourier series expansion, where we replace the terms of the form $(x-a)^n$ with sines and cosines. Specifically, we try to write

$$f(x)=c_0+\sum_{n=1}^{\infty}a_n\cos(nx)+b_n\sin(nx).$$

The goal is to determine c_0 and the a_n and b_n. There are simple formulas for them involving integrals that we won't go into here. We can then cut off the series after a few terms to get an approximation to our function. Fourier series are especially useful for approximating periodic functions. Practical applications include signal processing, where we take a complicated signal and approximate it by a few Fourier coefficients, thereby compressing a lot of information into a few coefficients.

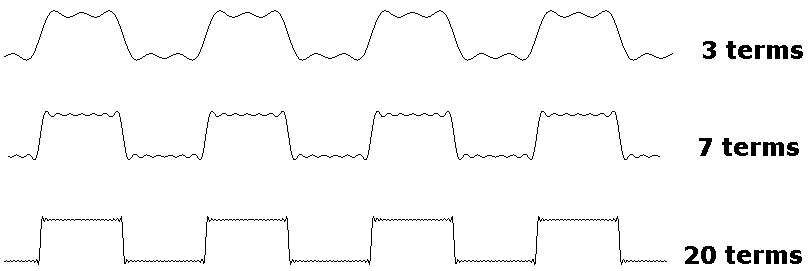
For example, the function shown partially below is called a square wave.



Choosing the frequency and amplitude of the wave to make things work out, the Fourier series turns out to be

$$\sin t+\frac{1}{3}\sin 3t+\frac{1}{5}\sin 5t+\frac{1}{7}\sin 7t+\ldots.$$

Here are what a few of the approximations look like:



Fourier series are used extensively in signal processing.

Other types of approximations

One of the most accurate approaches of all is simply to apply Newton's Divided Differences to the Chebyshev points, $(x_i, f(x_i))$, where $x_i = \frac{b-a}{2}\cos\left(\frac{(2i-1)\pi}{2n}\right) + \frac{b+a}{2}$ for $i = 1, 2, \ldots, n$.

Another way of looking at this is that we are trying to write the function in terms of the Chebyshev polynomials, namely

$$f(x) = \sum_{n=0}^{\infty} c_n T_n(x).$$

The goal is to find the coefficients c_n. This is called Chebyshev approximation. To approximate $f(x)$ on $[a, b]$ using N terms of the series, set $s = (b-a)/2$, $m = (b+a)/2$, and compute

$$f(x) \approx -\frac{c_0}{2} + \sum_{k=0}^{N-1} c_k T_k(sx + m),$$

where

$$c_j = \frac{2}{N}\sum_{k=0}^{N-1} f\left(\left(\cos\left(\frac{\pi(k+\frac{1}{2})}{N} - m\right)/s\right)\right)\cos\left(\frac{\pi j(k+\frac{1}{2})}{N}\right).$$

Chebyshev approximation has the property that you don't need very many terms to get a good approximation. See Sections 5.8 - 5.11 of *Numerical Recipes* for more particulars on this.

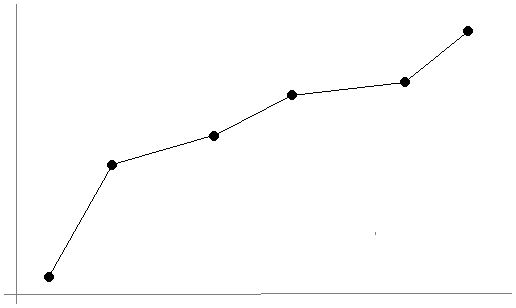
The CORDIC algorithm

At the CPU level, many chips use something called the CORDIC algorithm to compute trig functions, logarithms, and exponentials. It approximates these functions using only addition, subtraction, multiplication and division by 2, and small tables of precomputed function values. Multiplication and division by 2 at the CPU level correspond to just shifting bits left or right, which is something that can be done very efficiently.

3.6 Piecewise linear interpolation

One problem with using a single polynomial to interpolate a large collection of points is that polynomials want to oscillate, and that oscillation can lead to unacceptable errors. We can use the Cheybshev nodes to space the points to reduce this error, but real-life constraints often don't allow us to choose our data points, like with census data, which comes in ten-year increments.

A simple solution to this problem is *piecewise linear interpolation*, which is where we connect the points by straight lines, like in the figure below:



One benefit of this approach is that it is easy to do. Given two points, (x_1, y_1) and (x_2, y_2), the equation of the line between them is $y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$.

The interpolation error for piecewise linear interpolation is on the order of h^2, where h is the maximum of $x_{i+1} - x_i$ over all the points, i.e., the size of the largest gap in the x-direction between the data points. So unlike polynomial interpolation, which can suffer from the Runge phenomenon if large numbers of equally spaced points are used, piecewise linear interpolation benefits when more points are used. In theory, by choosing a fine enough mesh of points, we can bring the error below any desired threshold.

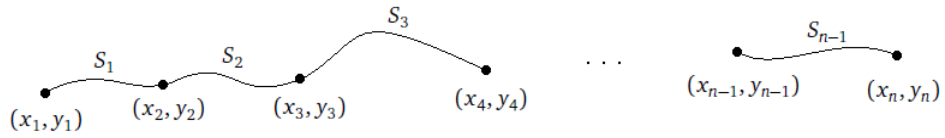
3.7 Cubic spline interpolation

By doing a bit more work, we can improve the error term of piecewise linear interpolation from h^2 to h^4 and also get a piecewise interpolating function that is free from sharp turns. Sharp turns are occasionally a problem for physical applications, where smoothness (differentiability) is needed.

Instead of using lines to approximate between (x_i, y_i) and (x_{i+1}, y_{i+1}), we will use cubic equations of the form

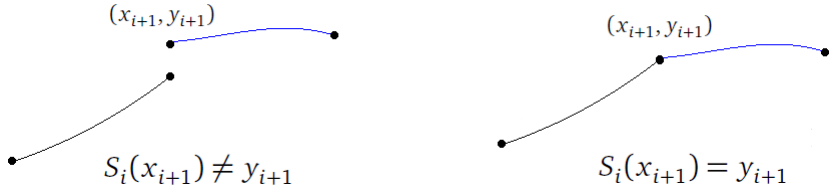
$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

Here is what this might look like:



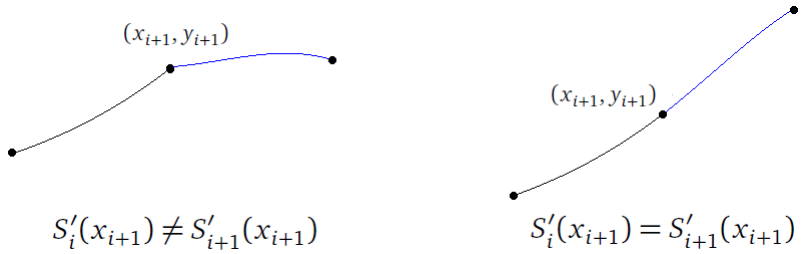
If we have n points, then we will have $n-1$ equations, and we will need to determine the coefficients a_i, b_i, c_i, and d_i for each equation. However, right away we can determine that $a_i = y_i$ for each i, since the constant term a_i tells the height that curve S_i starts at, and that must be the y-coordinate of the corresponding data value, (x_i, y_i). So this leaves b_i, c_i, and d_i to be determined for each curve, a total of $3(n-1)$ coefficients. To uniquely determine these coefficients, we will want to find a system of $3(n-1)$ equations that we can solve to get the coefficients.

We will first require for each $i = 1, 2, \ldots n$ that $S_i(x_{i+1}) = y_{i+1}$. This says that at each data point, the curves on either side must meet up without a jump. This implies that the overall combined function is continuous. See the figure below.



This gives us a total of $n-1$ equations to use in determining the coefficients.

Next, for each $i = 1, 2, \ldots n$, we require that $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$. This says that where the individual curves come together, there can't be a sharp turn. The functions have to meet up smoothly. This implies that the overall function is differentiable. See the figure below.



$S_i'(x_{i+1}) \neq S_{i+1}'(x_{i+1})$ $\qquad$ $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$

This gives us $n-2$ more equations for determining the coefficients.

We further require for each $i = 1, 2, \ldots n$, that $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$. This adds a further smoothness condition at each data point that says that the curves have meet up very smoothly. This also gives us $n-2$ more equations for determining the coefficients.

At this point, we have a total of $(n-1)+(n-2)+(n-2) = 3n-5$ equations to use for determining the coefficients, but we need $3n-3$ equations. There are a few different ways that people approach this. The simplest approach is the so-called natural spline equations, which are $S_1''(x_1) = 0$ and $S_{n-1}''(x_n) = 0$. Geometrically, these say the overall curve has inflection points at its two endpoints.

Systems of equations like this are usually solved with techniques from linear algebra. With some clever manipulations (which we will omit), we can turn this system into one that can be solved by efficient algorithms. The end result for a natural spline is the following[1]:

Set $H_i = y_{i+1} - y_i$ and $h_i = x_{i+1} - x_i$ for $i = 1, 2 \ldots, n-1$. Then set up the following matrix.

$$\left[\begin{array}{cccccccccc|c}
1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
h_1 & 2(h_1+h_2) & h_2 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 3\left(\frac{H_2}{h_2} - \frac{H_1}{h_1}\right) \\
0 & h_2 & 2(h_2+h_3) & h_3 & 0 & \cdots & 0 & 0 & 0 & 0 & 3\left(\frac{H_3}{h_3} - \frac{H_2}{h_2}\right) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} & 3\left(\frac{H_{n-1}}{h_{n-1}} - \frac{H_{n-2}}{h_{n-2}}\right) \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 & 0
\end{array}\right]$$

This matrix is tridiagonal (in each row there are only three nonzero entries, and those are centered around the diagonal). There are fast algorithms for solving tridiagonal matrix equations. The solutions to this system are $c_1, c_2, \ldots, c_n$. Once we have these, we can then get b_i and d_i for $i = 1, 2, \ldots n-1$ as follows:

$$b_i = \frac{H_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}), \qquad d_i = \frac{c_{i+1} - c_i}{3h_i}.$$

Example:

Suppose we want the natural cubic spline through $(1, 2)$, $(3, 5)$, $(4, 8)$, $(5, 7)$, and $(9, 15)$.

The h's are the differences between the x-values, which are $h_1 = 2$, $h_2 = 1$, $h_3 = 1$, and $h_4 = 4$. The H's are the differences between the y-values, which are $H_1 = 3$, $H_2 = 3$, $H_3 = -1$, and $H_4 = 8$. We then put these into the

[1] The derivation here follows Sauer's *Numerical Analysis* section 3.4 pretty closely.

following matrix:

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2(2+1) & 1 & 0 & 0 & 3\left(\frac{3}{1}-\frac{3}{2}\right) \\ 0 & 1 & 2(1+1) & 1 & 0 & 3\left(\frac{-1}{1}-\frac{3}{1}\right) \\ 0 & 0 & 1 & 2(1+4) & 4 & 3\left(\frac{8}{4}-\frac{-1}{1}\right) \\ 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right].$$

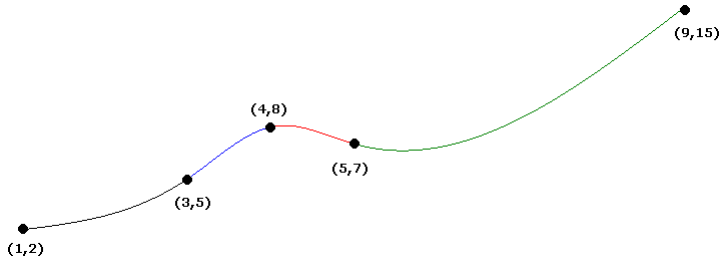
This can be solved by row reduction or a more efficient tridiagonal algorithm to get $c_1 = 0$, $c_2 = 87/64$, $c_3 = -117/32$, $c_4 = 81/64$, and $c_5 = 0$. Row reduction can be done using a graphing calculator, various online tools, or a computer algebra system (or by hand if you know the technique and are patient). Once we have these coefficients, we can plug into the formulas to get

$$b_1 = \frac{H_1}{h_1} - \frac{h_1}{3}(2c_1 + c_2) = \frac{19}{32} \qquad d_1 = \frac{c_2 - c_1}{3h_1} = \frac{29}{128}$$
$$b_2 = \frac{H_2}{h_2} - \frac{h_2}{3}(2c_2 + c_3) = \frac{53}{16} \qquad d_2 = \frac{c_3 - c_2}{3h_2} = \frac{-107}{64}$$
$$b_3 = \frac{H_3}{h_3} - \frac{h_3}{3}(2c_3 + c_4) = \frac{65}{64} \qquad d_3 = \frac{c_4 - c_3}{3h_3} = \frac{105}{64}$$
$$b_4 = \frac{H_4}{h_4} - \frac{h_4}{3}(2c_4 + c_5) = \frac{-11}{8} \qquad d_4 = \frac{c_5 - c_4}{3h_4} = \frac{-27}{256}$$

This can be done quickly with spreadsheet or a computer program. The equations are thus

$$\begin{aligned} S_1(x) &= 2 + \frac{19}{32}(x-1) + \frac{29}{128}(x-1)^3 \\ S_2(x) &= 5 + \frac{53}{16}(x-3) + \frac{87}{64}(x-3)^2 - \frac{107}{64}(x-3)^3 \\ S_3(x) &= 8 + \frac{65}{64}(x-4) - \frac{117}{32}(x-4)^2 + \frac{105}{64}(x-4)^3 \\ S_4(x) &= 7 - \frac{11}{8}(x-5) + \frac{81}{64}(x-4)^2 - \frac{27}{256}(x-5)^3. \end{aligned}$$

They are graphed below:



Other types of cubic splines

Besides the natural cubic spline, there are a few other types. Recall that the natural spline involves setting $S_1''(x_1) = 0$ and $S_{n-1}''(x_n) = 0$. If we require these second derivatives to equal other values, say u and v, we have what is called a curvature-adjusted cubic spline (curvature being given by the second derivative). The procedure above works nearly the same, except that the first and last rows of the matrix become the following:

$$\begin{array}{ccccc|c} 2 & 0 & 0 & 0 & 0 & \cdots \\ 2 & 0 & 0 & 0 & 0 & \cdots \end{array} \quad \begin{array}{cccc|c} 0 & 0 & 0 & 0 & u \\ 0 & 0 & 0 & 0 & v. \end{array}$$

If the derivative of the function being interpolated is known at the endpoints of the range, we can incorporate that information to get a more accurate interpolation. This approach is called a clamped cubic spline. Assuming the values of the derivative at the two endpoints are u and v, we set $S'(x_1) = u$ and $S'(x_n) = v$. After running through everything, the only changes are in the second and second-to-last rows of the matrix. They become:

$$\left[\begin{array}{ccccccccc|c} 2h_1 & h_1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 3\left(\frac{H_1}{h_1} - u\right) \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & h_{n-1} & 2h_{n-1} & 3\left(v - \frac{H_{n-1}}{h_{n-1}}\right). \end{array}\right]$$

One final approach that is recommended in the absence of any other information is the not-a-knot cubic spline. It is determined by setting $S_1'''(x_2) = S_2'''(x_2)$ and $S_{n-2}'''(x_{n-1} = S_{n-1}'''(x_{n-1})$. These are equivalent to requiring $d_1 = d_2$ and $d_{n-2} = d_{n-1}$, which has the effect of forcing S_1 and S_2 to be identical as well as forcing S_{n_2} and S_{n-1} to be identical. The only changes to the procedure are in the second and second-to-last rows:

$$\left[\begin{array}{ccccccccc|c} h_1 & -(h_1+h_2) & h_2 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & h_{n-2} & -(h_{n-2}+h_{n-1}) & h_{n-1} & 0. \end{array}\right]$$

Interpolation error

The interpolation error for cubic splines is on the order of h^4, where h is the maximum of $x_{i+1} - x_i$ over all the points x_i (i.e., the largest gap between the x coordinates of the interpolating points). Compare this with piecewise linear interpolation, whose interpolation error is on the order of h^2.

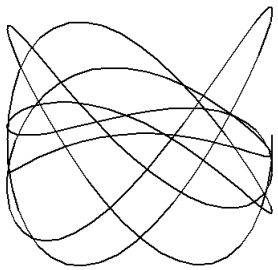
3.8 Bézier curves

Bézier curves were first developed around 1960 by mathematician Paul de Casteljau and engineer Pierre Bézier, working at competing French car companies. Nowadays, Bézier curves are very important in computer graphics. For example, the curve and path drawing tools in many graphics programs use Bézier curves. They are also used for animations in Adobe Flash and other programs to specify the path that an object will move along. Another important application for Bézier curves is in specifying the curves in modern fonts. We can also use Bézier curves in place of ordinary cubic curves in splines.

Before we define what a Bézier curve is, we need to review a little about parametric equations.

Parametric equations

Many familiar curves are defined by equations like $y = x$, $y = x^2$, or $x^2 + y^2 = 9$. But consider the curve below:

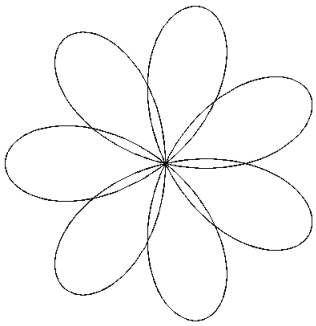


Maybe it represents a trace of the motion of an object under the influence of several forces, like a rock in the asteroid belt. Such a curve is not easily represented in the usual way. However, we can define it as follows:

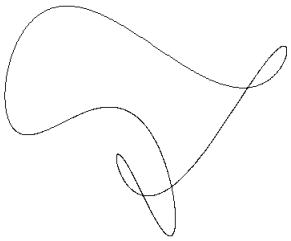
$$x(t) = 2\cos(\sqrt{2}t), \quad y(t) = \sin(\sqrt{3}t) + \sin(\sqrt{5}t) \qquad \text{for } 0 \le t \le 20.$$

Thinking of this as the motion of a physical object, t stands for time, and the equations $x(t)$ and $y(t)$ describe the position of the particle in the x and y directions at each instant in time.

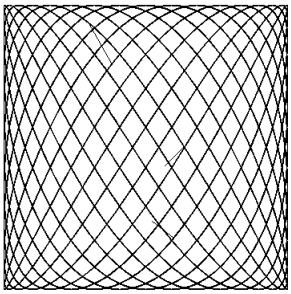
Parametric equations are useful for describing many curves. For instance, a circle is described by $x = \cos(t)$, $y = \sin t$, $0 \le t \le 2\pi$. A function of the form $y = f(x)$ is described by the equations $x = t$, $y = f(t)$. In three-dimensions, a helix (the shape of a spring and DNA) is given by $x = \cos t$, $y = \sin t$, $z = t$. There are also parametric surfaces, where the equations for x, y, and z are each functions of two variables. Various interesting shapes, like toruses and Möbius strips can be described simply by parametric surface equations. Here a few interesting curves that can be described by parametric equations.



$x = \sin 5t + \cos 2t$
$y = \sin 2t + \cos 5t$



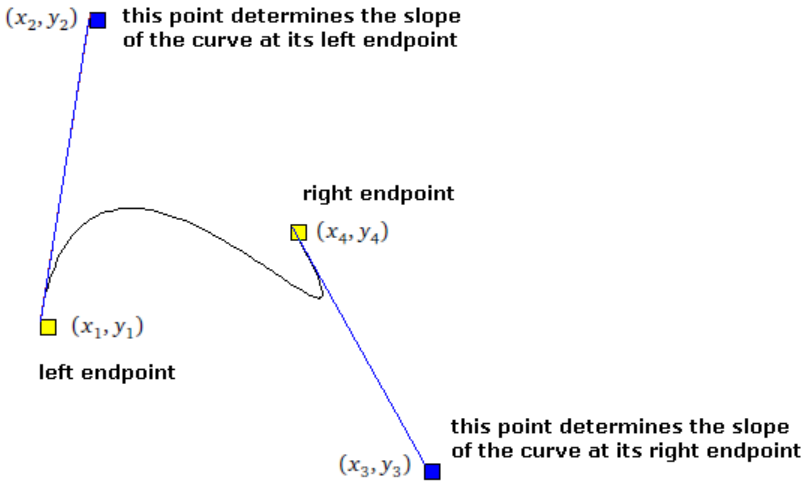
$x = \sin t - \cos(t/2)$
$y = \sin(t/2) - \cos t \sin t$



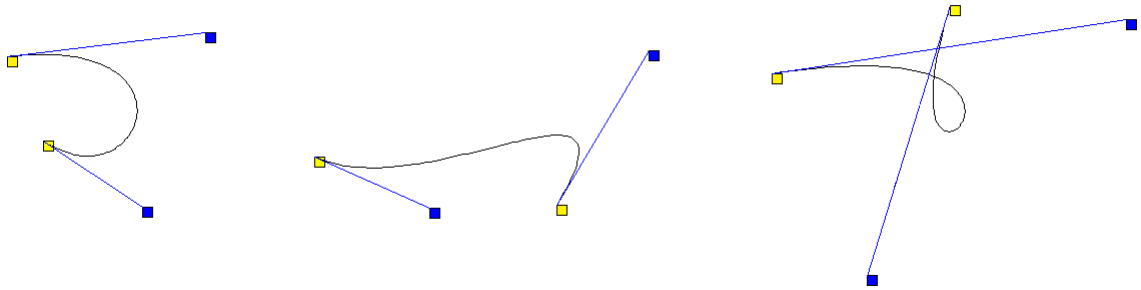
$x = \sin 11t$
$y = \cos 17t$

How Bézier curves are defined

A cubic Bézier curve is determined by its two endpoints and two *control points* that are used to determine the slope at the endpoints. In the middle, the curve usually follows a cubic parametric equation. See the figure below:



Here are a few more examples:



The parametric equations for a cubic Bézier curve are given below:

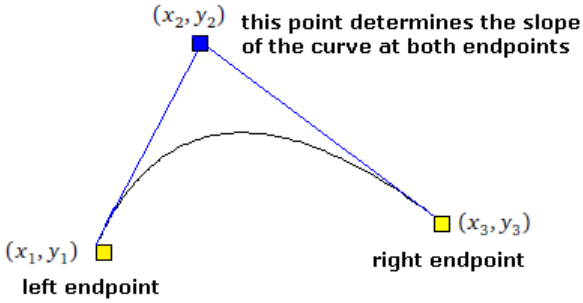
$$x = x_1 + b_x t + c_x t^2 + d_x t^3$$
$$y = y_1 + b_y t + c_y t^2 + d_y t^3$$

$$b_x = 3(x_2 - x_1) \qquad b_y = 3(y_2 - y_1)$$
$$c_x = 3(x_3 - x_2) - b_x \qquad c_y = 3(y_3 - y_2) - b_y$$
$$d_x = x_4 - x_1 - b_x - c_x \qquad d_y = y_4 - y_1 - b_y - c_y$$

These equations are valid for $0 \le t \le 1$, where $t = 0$ gives the left endpoint and $t = 1$ gives the right.

Quadratic Bézier curves

Sometimes quadratic Bézier curves are used. The parametric equations for these curves are quadratic and only one control point is used to control the slopes at both endpoints, as shown below:



The parametric equations for a quadratic Bézier curve are given below:

$$x = x_1 + b_x t + c_x t^2$$
$$y = y_1 + b_y t + c_y t^2$$

$$b_x = 2(x_2 - x_1) \qquad b_y = 2(y_2 - y_1)$$
$$c_x = (x_3 - x_2) - b_x \qquad c_y = (y_3 - y_2) - b_y$$

These equations are valid for $0 \le t \le 1$, where $t = 0$ gives the left endpoint and $t = 1$ gives the right.

One can also define quartic and higher order Bézier curves, with more control points specifying the slope elsewhere on the curve, but, in practice, the added complexity is not worth the extra precision.

3.9 Summary of interpolation

Just to summarize, the point of interpolation is that we have data that come from some underlying function, which we may or may not know. When we don't know the underlying function, we usually use interpolation to estimate the values of the function at points between the data points we have. If we do know the underlying function, interpolation is used to find a simple approximation to the function.

The first approach to interpolation we considered was finding a single polynomial goes through all the points. The Lagrange formula gives us a formula for that interpolating polynomial. It is useful in situations where an explicit formula for the coefficients is needed. Newton's divided differences is a process that finds the coefficients of the interpolating polynomial. A similar process, Neville's algorithm, finds the value of the polynomial at specific points without finding the coefficients.

When interpolating, sometimes we can choose the data points and sometimes we are working with an existing data set. If we can choose the points, and we want a single interpolating polynomial, then we should space the points out using the Chebyshev nodes.

Equally-spaced data points are actually one of the worst choices as they are most subject to the Runge phenomenon. And the more equally-spaced data points we have, the worse the Runge phenomenon can get. For this reason, if interpolating from an existing data set with a lot of points, spline interpolation would be preferred.

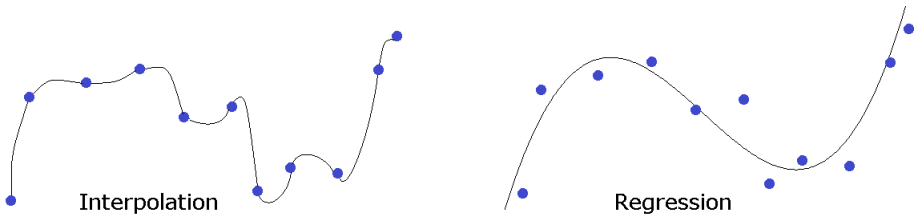
We also looked at Bézier curves which can be used for interpolation and have applications in computer graphics.

Further information

Just like with root-finding, we have just scratched the surface here. There are a variety of ways to interpolate that we haven't covered, each with its own pros and cons. For instance, one can interpolate using rational functions (functions of the form $p(x)/q(x)$ where $p(x)$ and $q(x)$ are polynomials). Rational functions are better than polynomials at interpolating functions with singularities.

Another type of interpolation is Hermite interpolation, where if there is information about the derivative of the function being interpolated, a more accurate polynomial interpolation can be formed. There is also interpolation with trigonometric functions (such as with Fourier series), which has a number of important applications.

Finally, we note that interpolation is not the only approach to estimating a function. Regression (often using least-squares technique) involves finding a function that doesn't necessarily exactly pass through the data points, but passes near the points and minimizes the overall error. See the figure below.



Chapter 4

Numerical differentiation

Taking derivatives is something that is generally easy to do analytically. The rules from calculus are enough in theory to differentiate just about anything we have a formula for. But sometimes we don't have a formula, like if we are performing an experiment and just have data in a table. Also, some formulas can be long and computationally very expensive to differentiate. If these are part of an algorithm that needs to run quickly, it is often good enough to replace the full derivative with a quick numerical estimate. Numerical differentiation is also an important part of some methods for solving differential equations. And often in real applications, functions can be rather complex things that are defined by computer programs. Taking derivatives of such functions analytically might not be feasible.

4.1 Basics of numerical differentiation

The starting point for many methods is the definition of the derivative

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

From this, we see a simple way to estimate the derivative is to pick a small value of h and use it to evaluate the expression inside the limit. For example, suppose we have $f(x) = x^2$ and we want the derivative at $x = 3$. If we try $h = .0001$, we get

$$f'(3) \approx \frac{3^2 - 3.0001^2}{.0001} = 6.0001.$$

This is not too far off from the exact value, 6. The expression, $\frac{f(x+h)-f(x)}{h}$, that we use to approximate the derivative, is called the *forward difference formula*.

In theory, a smaller values of h should give us a better approximations, but in practice, floating-point issues prevent that. It turns out that there are two competing errors here: the mathematical error we get from using a nonzero h, and floating-point error that comes from breaking the golden rule (we are subtracting nearly equal terms in the numerator).

Floating-point error

Here is a table of some h values and the approximate derivative of x^2 at 3, computed in Python.

h	$\frac{(x+h)^2-x^2}{h}$
0.1	6.100000000000012
0.01	6.009999999999849
0.001	6.000999999999479
0.0001	6.000100000012054
10^{-5}	6.000009999951316
10^{-6}	6.000001000927568
10^{-7}	6.000000087880153
10^{-8}	5.999999963535174
10^{-9}	6.000000496442226
10^{-10}	6.000000496442226
10^{-11}	6.000000496442226
10^{-12}	6.000533403494046
10^{-13}	6.004086117172844
10^{-14}	6.217248937900877
10^{-15}	5.329070518200751

We see that the approximations get better for a while, peaking around $h = 10^{-7}$ and then getting worse. As h gets smaller, our violation of the golden rule gets worse and worse. To see what is happening, here are the first 30 digits of the floating-point representation of $(3+h)^2$, where $h = 10^{-13}$:

```
9.000000000000600408611717284657
```

The last three "correct" digits are 6 0 0. Everything after that is an artifact of the floating-point representation. When we subtract 9 from this and divide by 10^{-13}, all of the digits starting with the that 6 are "promoted" to the front, and we get 6.004086..., which is only correct to the second decimal place.

Mathematical error

The mathematical error comes from the fact that we are approximating the slope of the tangent line with slopes of nearby secant lines. The exact derivative comes from the limit as h approaches 0, and the larger h is, the worse the mathematical error is. We can use Taylor series to figure out this error.

First, recall the Taylor series formula for $f(x)$ centered at $x = a$ is

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots.$$

Replacing x with $x+h$ and taking $a = x$ in the formula above gives the following:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \dots.$$

This is a very useful version of the Taylor series.[1] Solving for $f'(x)$ gives

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{f''(x)}{2!}h - \frac{f'''(x)}{3!}h^2 - \dots.$$

All the terms beyond the h^2 term are small in comparison to the h^2 term, so we might consider dropping them. More formally, Taylor's theorem tells us that we can actually replace all of the terms from h^2 on with $\frac{f''(c)}{2!}h$ for some constant $c \in [x, x+h]$. This gives us

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{f''(c)}{2!}h$$

[1] It tells us that if we know the value of $f(x)$ and we perturb x by a small amount to $x+h$, then the function value at that point is $f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \dots$. Often we approximate the full series by just taking the first couple of terms. For instance, $\sqrt{4} = 2$ and if we want $\sqrt{4.1}$, we could use three terms of the series to estimate it as $2 + \frac{1}{2\sqrt{4}}(.1) - \frac{1}{4\sqrt{4}}(.1)^2 = 2.02374$, which is only off by about .001 from the exact value. Using only the first two terms, $f(x) + f'(x)h$ is called a *linear approximation* and is a common topic in calculus classes.

We can read this as saying that $f'(x)$ can be approximated by $\frac{f(x+h)-f(x)}{h}$ and the error in that approximation is given by $\frac{f''(c)}{2!}h$. The most important part of the error term is the power of h, which is 1 here. This is called the *order* of the method.

Just like with root-finding, a first order method is sometimes called linear, a second order quadratic, etc. The higher the order, the better, as the higher order allows us to get away with smaller values of h. For instance, all other things being equal, a second order method will get you twice as many correct decimal places for the same h as a first order method.

Combining the two types of errors

One the one hand, pure mathematics says to use the smallest h you possibly can in order to minimize the error. On the other hand, practical floating-point issues say that the smaller h is, the less accurate your results will be. This is a fight between two competing influences, and the optimal value of h lies somewhere in the middle. In particular, one can work out that for the forward difference formula, the optimal choice of h is $x\sqrt{\epsilon}$, where ϵ is machine epsilon, which is roughly 2.2×10^{-16} for double-precision floating-point.

For instance, for estimating $f'(x)$ at $x = 3$, we should use $h = 3\sqrt{2.2 \times 10^{-16}} \approx 4 \times 10^{-8}$. We see that this agrees with the table of approximate values we generated above.

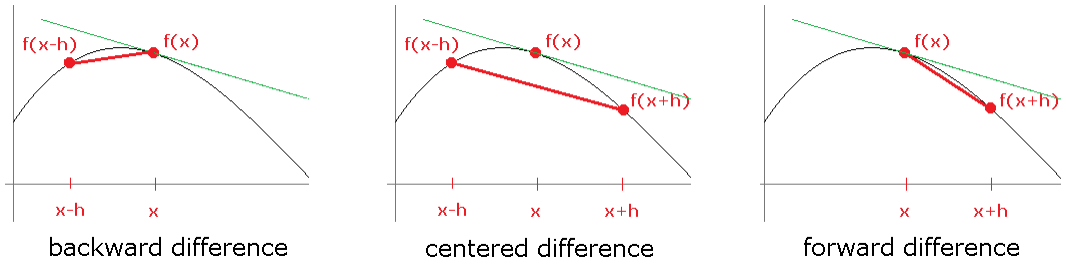
In general, if for an nth order method, the optimal h turns out to be $x\sqrt[n+1]{\epsilon}$.

4.2 Centered difference formula

Thinking of $\frac{f(x+h)-f(x)}{h}$ as the slope of a secant line, we are led to another secant line approximation:

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}.$$

Whereas the former approximation looks "forward" from $f(x)$ to $f(x+h)$, the latter approximation is centered around $f(x)$, so it is called the *centered difference formula*. There is also a *backward difference formula*, $\frac{f(x)-f(x-h)}{h}$. See the figure below:



The centered difference formula is actually more accurate than the forward and backward ones. To see why, use Taylor series to write

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \ldots$$
$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 + \ldots$$

Subtracting these two equations, dividing by $2h$ and applying Taylor's formula[1], we see that the f'' term cancels out leaving

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + \frac{f'''(c)}{3!}h^2.$$

[1]There's an additional wrinkle here in that there are actually two different c values from each series, but things do work out so that we only need one value of c in the end. And if you don't want to worry about formalities at all, just imagine that we drop all the terms past the h^2 term, since we're only really concerned with the order of the method and not the constants.

So this is a second order method.

For the sake of comparison, here are the forward difference and centered difference methods applied to approximate the derivative of e^x at $x = 0$ (which we know is equal to 1):

h	forward difference	centered difference
.01	1.0050167084167950	1.0000166667499921
.001	1.0005001667083846	1.0000001666666813
.0001	1.0000500016671410	1.0000000016668897
10^{-5}	1.0000050000069650	1.0000000000121023
10^{-6}	1.0000004999621837	0.9999999999732445
10^{-7}	1.0000000494336803	0.9999999994736442
10^{-8}	0.9999999939225290	0.9999999939225290
10^{-9}	1.0000000827403710	1.0000000272292198

We see that for small values of h, the centered difference formula is far better, though it starts suffering from floating-point problems sooner than the forward difference formula does. However, the centered difference formula's best estimate (1.0000000000121023) is better than the forward difference formula's best estimate (0.9999999939225290).

Generally, the centered difference formula is preferred. However, one advantage that the forward difference formula has over the centered difference formula is that it if you've already have $f(x)$ computed for some other reason, then you can get a derivative estimate with only one more function evaluation, whereas the centered difference formula requires two. This is not a problem if f is a simple function, but if f is computationally very expensive to evaluate (and this is one of the times where we might need numerical differentiation), then the extra function call might not be worth the increase in order from linear to quadratic.

Another advantage of the forward and backward difference formulas is that they can be used at the endpoints of the domain of a function, whereas the centered difference formula would require us to step over an endpoint and hence would not work.

4.3 Using Taylor series to develop formulas

The technique we used to develop the centered difference formula can be used to develop a variety of other formulas. For instance, suppose we want a second-order formula for $f'(x)$ using $f(x)$, $f(x+h)$, and $f(x+2h)$. Our goal is to something like this:

$$f'(x) = \frac{af(x) + bf(x+h) + cf(x+2h)}{h} + Kh^2,$$

The first part is the numerical differentiation formula that we want and the second part is the error term. We need to find a, b, and c. We start by writing the Taylor expansions for $f(x+h)$ and $f(x+2h)$:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \frac{f^{(4)}(x)}{4!}h^4 + \frac{f^{(5)}(x)}{5!}h^5 + \dots$$
$$f(x+2h) = f(x) + 2f'(x)h + 4\frac{f''(x)}{2!}h^2 + 8\frac{f'''(x)}{3!}h^3 + 16\frac{f^{(4)}(x)}{4!}h^4 + 32\frac{f^{(5)}(x)}{5!}h^5 + \dots$$

We want to multiply the first series by b, the second by c, add them together, and solve for $f'(x)$. When we solve for $f'(x)$, we will end up dividing by h. This means that if we want an error term of the form Kh^2, it will come from the h^3 terms in the Taylor series. Also, since there are no f'' terms in our desired formula, we are going to need those to cancel out when we combine the series. So to cancel out the f'' terms, we will choose $b = -4$ and $c = 1$. Then we combine the series to get:

$$-4f(x+h) + f(x+2h) = -3f(x) + -2f'(x)h + Kh^3.$$

Note that we could get an expression for K if we want, but since we're mostly concerned with the order, we won't bother.[1]. We can then solve this for $f'(x)$ to get

$$f'(x) = \frac{-3f(x)+4f(x+h)-f(x+2h)}{2h} + Kh^2.$$

And to see if our formula works, suppose we try to estimate $f'(3)$ using $f(x) = x^2$ and $h = .00001$. We get

$$f'(3) \approx \frac{-3(3)^2 + 4(3.00001)^2 = f(3.00002)^2}{.00002} = 5.999999999861671,$$

which is pretty close to the exact answer, 6.

Higher derivatives

We can use the Taylor series technique to estimate higher derivatives. For example, here are the Taylor series for $f(x+h)$ and $f(x-h)$:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \frac{f^{(4)}(x)}{4!}h^4 + \dots$$
$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 + \frac{f^{(4)}(x)}{4!}h^4 + \dots$$

If we add these two we get

$$f(x+h) + f(x-h) = 2f(x) + 2\frac{f''(x)}{2!}h^2 + 2\frac{f^4(x)}{4!}h^4 + \dots.$$

The key here is that we have canceled out the f' terms (and as a bonus, the f''' terms have canceled out as well). Solving for $f''(x)$, we get

$$f''(x) = \frac{f(x+h)-2f(x)+f(x-h)}{h^2} + 2\frac{f^{(4)}(x)}{4!}h^2.$$

This is a second order approximation for f''.

Another way to get an approximation for f'' might be $\frac{f'(x+h)-f'(x)}{h}$. And if we use the forward and backward difference formulas to approximate f', we end up with the same result as above:

$$f''(x) \approx \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} = \frac{f(x+h)-2f(x)+f(x-h)}{h^2}.$$

Note

It is worth noting that all the formulas we've obtained via Taylor series can also be obtained by differentiating the Lagrange interpolating polynomial. For instance, if we write the Lagrange interpolating polynomial through the points $(a, f(a)$ and $(a+h, f(a+h))$, and differentiate the result, the forward difference formula results. Further, if we write the interpolating polynomial through $(a-h, f(a-h)$, $(a, f(a)$, and $(a+h, f(a+h)$, differentiate it, and plug in $x = a$, then we obtain the centered difference formula. And plugging in $x = a-h$ or $x = a+h$ gives a second-order backward and forward difference formulas.

4.4 Richardson extrapolation

Earlier we saw Aitken's Δ^2 method, which was an algebraic way to speed up the convergence of a root-finding method. *Richardson extrapolation* is another algebraic technique that can increase the order of a numerical method. Here is the idea of how it works.

[1] The actual value of K is $\frac{4f'''(c)}{3!}$ for some $c \in [x, x+h]$.

Let's start with the centered difference formula with its full error series (not dropping any higher order terms):

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + c_1h^2 + c_2h^4 + c_3h^6 + \dots.$$

We don't care about the exact values of c_1, c_2, etc. What we do now is replace h with $h/2$. We get

$$f'(x) = \frac{f(x+h/2)-f(x-h/2)}{h} + c_1\frac{h^2}{4} + c_2\frac{h^4}{16} + c_3\frac{h^6}{64} + \dots.$$

If we multiply the second series by 4 and subtract the first series, we can cancel out the h^2 terms. This gives us

$$4'f(x) - f'(x) = 4\frac{f(x+h/2)-f(x-h/2)}{h} - \frac{f(x+h)-f(x-h)}{2h} + Kh^4$$

This is now a fourth order method. (Note that we don't care about the exact coefficients of h^4, h^6, etc., so we just coalesce all of them into Kh^4.) We can simplify things down to the following

$$f'(x) = \frac{f(x-h)-8f(x-h/2)+8f(x+h/2)-f(x+h)}{6h} + Kh^4.$$

This method works in general to increase the order of a numerical method. Suppose we have an nth order method $F(h)$. If we apply the same procedure as above, we can derive the following new rule:

$$G(h) = \frac{2^nF(h/2)-F(h)}{2^n-1}.$$

This rule has order greater than n.[1] For example, let's apply extrapolation to the first-order backward difference formula $\frac{f(x)-f(x-h)}{h}$. We compute

$$\frac{2^1\frac{f(x)-f(x-h/2)}{h/2} - \frac{f(x)-f(x-h)}{h}}{2^1-1} = \frac{3f(x)-4f(x-h/2)+f(x-h)}{h}.$$

This method has order 2. This comes from the fact that the power series for the backward difference formula has terms of all orders — $c_1h+c_2h^2+c_3h^3+\dots$ — and the extrapolation only knocks out the first term, leaving the h^2 and higher terms. On the other hand, sometimes we are lucky, like with the centered difference formula, where the power series has only even terms and we can jump up two orders at once.

If we try this method on $f(x) = x^2$ with $x = 3$ and $h = .00001$, we get

$$\frac{3(3)^2-4(2.999995)^2+(2.99999)^2}{.00001} = 5.999999999239946.$$

If we want to, we can repeatedly apply Richardson extrapolation. For instance, if we apply extrapolation twice to the centered difference formula, we get the following 6th order method:

$$f'(x) = \frac{-f(x-h)+40f(x-h/2)-256f(x-h/4)+256f(x+h/4)-40f(x+h/2)+f(x+h)}{90h}.$$

However, as we increase the order, we see that the number of function evaluations increases. This has the effect of slowing down calculation of the numerical derivative. With many numerical methods, there are two competing influences: higher order and number of function evaluations. For numerical differentiation, a higher order means that we don't have to use a very small value of h to get a good approximation (remember that smaller h values cause floating point problems), but we pay for it in terms of the number of function evaluations. The number of function evaluations is a good measure of how long the computation will take. For numerical differentiation, second- or fourth-order methods usually provide a decent tradeoff in terms of accuracy for a given h and the number of required function evaluations.

[1] The term $2^nF(h/2)-F(h)$ is used to cancel out the term of the form Ch^n in the power series expansion. Note also that one can write Richardson extrapolation in the form $Q = \frac{2^nF(h)-F(2h)}{2^n-1}$.

A tabular approach

Rather than using Richardson extrapolation to create higher order methods, we can apply a tabular approach. We start by using a rule, like the centered difference formula with h values that go down by a factor of 2. So we would use the formula with $h = 1, 1/2, 1/4, 1/8, 1/16, \ldots$. This gives us a sequence of values that will make up the first column of our table. We'll call them a_{11}, a_{21}, a_{31}, etc.

Then we apply the Richardson extrapolation formula to these values. Since the centered difference formula has order 2, the formula becomes $G(h) = (4F(h/2) - F(h))/3$. The values $F(h/2)$ and $F(h)$ are consecutive values that we calculated. That is, for the second column of our table, we compute $a_{22} = (4a_{21} - a_{11})/3$, $a_{32} = (4a_{31} - a_{21})/3$, etc.

Then we can extrapolate again on this data. The data in this second column corresponds to a fourth order method, so the Richardson extrapolation formula becomes $G(h) = (16F(h/2) - F(h))/15$. This gives us our third column. Then we can extrapolate on this column using the formula $G(h) = 32F(h/2) - F(h))/31$. We can keep building columns this way, with the approximations in each column becoming more and more accurate. Here is what the start of the table will look like:

$a_{11} = \frac{f(x+1)-f(x-1)}{2\cdot 1}$				
$a_{21} = \frac{f(x+\frac{1}{2})-f(x-\frac{1}{2})}{2\cdot\frac{1}{2}}$	$a_{22} = \frac{4a_{21}-a_{11}}{3}$			
$a_{31} = \frac{f(x+\frac{1}{4})-f(x-\frac{1}{4})}{2\cdot\frac{1}{4}}$	$a_{32} = \frac{4a_{31}-a_{21}}{3}$	$a_{33} = \frac{4^2a_{32}-a_{22}}{4^2-1}$		
$a_{41} = \frac{f(x+\frac{1}{8})-f(x-\frac{1}{8})}{2\cdot\frac{1}{8}}$	$a_{42} = \frac{4a_{41}-a_{31}}{3}$	$a_{43} = \frac{4^2a_{42}-a_{32}}{4^2-1}$	$a_{44} = \frac{4^3a_{43}-a_{33}}{4^3-1}$	
$a_{51} = \frac{f(x+\frac{1}{16})-f(x-\frac{1}{16})}{2\cdot\frac{1}{16}}$	$a_{52} = \frac{4a_{51}-a_{41}}{3}$	$a_{43} = \frac{4^2a_{52}-a_{42}}{4^2-1}$	$a_{54} = \frac{4^3a_{53}-a_{43}}{4^3-1}$	$a_{55} = \frac{4^4a_{54}-a_{44}}{4^4-1}$

For example, suppose we want to estimate the derivative of $f(x) = -\cos x$ at $x = 1$. We know the exact answer is sin(1), which is 0.8414709848078965 to 16 decimal places, so we can see how well the method does. For the first column, we use the centered difference formula with $h = 1, 1/2, 1/4, 1/8, 1/16$, and $1/32$. All of this is quick to program into a spreadsheet. Here are the results.

.708073418273571					
.806845360222670	.839769340872369				
.832733012957104	.841362230535249	.841468423179441			
.839281365458636	.841464149625813	.841470944231850	.841470984248555		
.840923259124113	.841470557012606	.841470984171726	.841470984805692	.841470984807877	
.841334033327051	.841470958061364	.841470984797948	.841470984807888	.841470984807896	.841470984807896

The bottom right entry is the most accurate, being correct to 15 decimal places, which is about as correct as we can hope for.

4.5 Automatic differentiation

This section is about something that can almost be described as magic. It is a numerical differentiation technique that returns answers accurate down to machine epsilon with seemingly no work at all. It is not a standard topic in most numerical methods books, but it is an important technique used in the real world. The math behind it is a little weird, but understandable.

Let's start with a familiar concept: imaginary numbers. The number i is defined to be a value that satisfies $i^2 = -1$. It is not a real number; it doesn't fit anywhere on the number line. But mathematically, we can pretend that such a thing exists. We can then define rules for arithmetic with i. We start by creating complex numbers like $2 + 3i$

that are combinations of real and imaginary numbers. Then we can define addition of complex numbers as simply adding their real and imaginary parts separately, like $(2+3i)+(6+9i) = 8+12i$. We can define multiplication of complex numbers by foiling out their product algebraically. For instance, $(2+3i)(4+5i) = 8+12i+10i+15i^2$, which simplifies to $-7+22i$ using the fact that $i^2 = -1$. Various other operations can also be defined.

This is what renaissance mathematicians did. They didn't believe that complex numbers existed in any real-world sense, but they turned out to be convenient for certain problems. In particular, they were used as intermediate steps in finding real roots of cubic polynomials. Gradually, more and more applications of complex numbers were found, even to practical things such as electronic circuits. In short, the number i might seem like a mathematical creation with no relevance to the real world, but it turns out to have many applications.

Similarly, here we will introduce another seemingly made-up number, ϵ. Just like i is a nonreal number with the property that $i^2 = -1$, we will define ϵ as a nonreal number with the property that $\epsilon^2 = 0$, but with the proviso that ϵ is not itself 0. That is, ϵ is a kind of infinitesimal number that is smaller than any real number but not 0. To repeat: its key property is that $\epsilon^2 = 0$.[1]

Then just like we have complex numbers of the form $a+bi$, we will introduce numbers, called *dual numbers*, of the form $a+b\epsilon$. We can do arithmetic with them just like with complex numbers. For instance, $(2+3\epsilon)+(6+9\epsilon) = 8+12\epsilon$. And we have $(2+3\epsilon)(4+5\epsilon) = 8+12\epsilon+10\epsilon+15\epsilon^2$. Since $\epsilon^2 = 0$, this becomes $8+22\epsilon$.

Now let's look at the Taylor series expansion for $f(x+\epsilon)$:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + \frac{f''(x)}{2!}\epsilon^2 + \frac{f'''(x)}{3!}\epsilon^3 + \ldots.$$

All of the higher order terms are 0, since ϵ^2, ϵ^3, etc. are all 0. So the following equation is exact:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon.$$

With a little more work, we can show that

$$f(a+b\epsilon) = f(a) + bf'(a)\epsilon,$$

This tells us how to compute function values of dual numbers. For instance,

$$\sin\left(\frac{\pi}{3}+5\epsilon\right) = \sin\left(\frac{\pi}{3}\right) + 5\cos\left(\frac{\pi}{3}\right)\epsilon = \frac{\sqrt{3}}{2} + \frac{5}{2}\epsilon.$$

Here's where things start to get interesting: Let's look at the product of two functions.

$$\begin{aligned}(fg)(x+\epsilon) &= f(x+\epsilon)g(x+\epsilon)\\ &= (f(x)+f'(x)\epsilon)(g(x)+g'(x)\epsilon)\\ &= f(x)g(x) + \big(f'(x)g(x) + f(x)g'(x)\big)\epsilon\end{aligned}$$

Look at how the product rule shows up as the component of ϵ. Next, let's look at the composition of two functions.

$$f(g(x+\epsilon)) = f(g(x)+g'(x)\epsilon) = f(g(x)) + g'(x)f'(g(x))\epsilon.$$

The chain rule somehow shows up as the component of ϵ. We could similarly show that the quotient rule pops out of dividing two functions.

Finally, going back to the important equation $f(x+\epsilon) = f(x) + f'(x)\epsilon$, if we solve for the derivative, we get

$$f'(x)\epsilon = f(x+\epsilon) - f(x).$$

This tells us that if we want the (real) derivative of a function, we can subtract $f(x+\epsilon)-f(x)$ and the ϵ component of that will equal the derivative. The upshot of all this is that we can program all the arithmetical rules for dual numbers as well as the definitions of some common functions, and then our program will almost magically be able to take derivatives of any algebraic combination of those functions, including compositions. Here is a Python implementation of this:

[1] In math ϵ is generally the symbol used for very small quantities.

```
class Dual:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return str(self.a)+(" + "+str(self.b) if self.b>=0 else " - "+str(-self.b))+chr(949)

    def __str__(self):
        return self.__repr__()

    def __add__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(self.a + y, self.b)
        else:
            return Dual(y.a+self.a, y.b+self.b)

    def __radd__(self, y):
        return self.__add__(y)

    def __sub__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(self.a-y, self.b)
        else:
            return Dual(self.a-y.a, self.b-y.b)

    def __rsub__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(y-self.a, -self.b)

    def __mul__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(self.a*y, self.b*y)
        else:
            return Dual(y.a*self.a, y.b*self.a + y.a*self.b)

    def __rmul__(self, y):
        return self.__mul__(y)

    def __pow__(self, e):
        return Dual(self.a ** e, self.b*e*self.a ** (e-1))

    def __truediv__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(self.a/y, self.b/y)
        else:
            return Dual(self.a/y.a, (self.b*y.a-self.a*y.b)/(y.a*y.a))

    def __rtruediv__(self, y):
        if type(y) == int or type(y) == float:
            return Dual(y/self.a, -y*self.b/(self.a*self.a))

def create_func(f, deriv):
    return lambda D: Dual(f(D.a), D.b*deriv(D.a)) if type(D)==Dual else f(D)

def autoderiv_nostring(f,x):
    return (f(Dual(x,1))-f(Dual(x,0))).b

def autoderiv(s, x):
    f = eval('lambda x: ' + s.replace("^", "**"))
    return (f(Dual(x,1))-f(Dual(x,0))).b

sin = create_func(math.sin, math.cos)
```

```
cos = create_func(math.cos, lambda x:-math.sin(x))
tan = create_func(math.tan, lambda x:1/math.cos(x)/math.cos(x))
sec = create_func(lambda x:1/math.cos(x), lambda x:math.sin(x)/math.cos(x)/math.cos(x))
csc = create_func(lambda x:1/math.sin(x), lambda x:-math.cos(x)/math.sin(x)/math.sin(x))
cot = create_func(lambda x:1/math.tan(x), lambda x:-1/math.sin(x)/math.sin(x))
arcsin = create_func(lambda x:math.asin(x), lambda x:1/math.sqrt(1-x*x))
arccos = create_func(lambda x:math.acos(x), lambda x:-1/math.sqrt(1-x*x))
arctan = create_func(lambda x:math.atan(x), lambda x:1/(1+x*x))
exp = create_func(math.exp, math.exp)
ln = create_func(math.log, lambda x:1/x)
```

To use this, if we call `autoderiv("sin(1/x)",2)`, the program will automatically differentiate $\sin\left(\frac{1}{x}\right)$ at $x = 2$ and give us an answer as accurate as if we were to analytically do the derivative and plug in 2 ourselves.

Here's a little about how the code above works. It creates a class to represent dual numbers and overrides the $+$, $-$, etc. operators using the rules for dual arithmetic. This is done by writing code for the Python "magic" methods `__add__`, `__sub__`, etc. Then the `create_func` function is used to build a dual function according to the rule $f(a+b\epsilon) = f(a) + bf'(a)\epsilon$. After that, we use this to program in definitions of common functions like sines and cosines. We do this simply by specifying the function and its derivative. Finally, to compute the derivative, we apply the rule that $f'(x)\epsilon = f(x+\epsilon) - f(x)$, which tells us that the derivative comes from the ϵ component of $f(x+\epsilon) - f(x)$.

4.6 Summary of numerical differentiation

Because we can easily differentiate most functions, numerical differentiation would appear to have limited usefulness. However, numerical differentiation is useful if we just have a table of function values instead of a formula, it is useful if our function is computationally expensive to evaluate (where finding the derivative analytically might be too time-consuming), it is an integral part of some numerical methods for solving differential equations, and it is useful for functions defined by computer programs.

The approaches we considered in detail above involve variations on the difference quotient $\frac{f(x+h)-f(x)}{h}$ that can be obtained by manipulating Taylor series or applying Richardson extrapolation to previously obtained formulas.

We can use those techniques to create methods of any order, though as the order increases, the number of terms in the estimate also increases, negating some of the benefits of higher order. In general, the centered difference formula or the fourth order method we looked at should suffice. Extrapolation can also be used to improve their results. The forward and backward difference formulas have order 1, but if you already have $f(x)$ calculated for some other purpose, then they only require one new function evaluation, so if a function is *really* expensive to evaluate, then these may be the only options. They (and higher order versions of them) are also useful for estimating the derivative at the endpoint of a function's domain, where the centered difference formula could not be used.

One thing to be careful of, as we saw, was to choose the value of h carefully. As all of our estimates involve subtracting nearly equal terms, floating-point problems can wreck our estimates. Specifically, $h = x\sqrt[n+1]{\epsilon}$ is recommended, where x is the point the derivative is being evaluated at, n is the order of the method, and ϵ is machine epsilon, roughly 2.2×10^{-16} for double-precision arithmetic.

Automatic differentiation is a technique that deserves to be better known. It totally avoids the problems that the formulas above suffer from.

There are other methods for numerically estimating derivatives. For instance, we could use cubic spline interpolation to approximate our function and then differentiate the splines. Another method actually uses integration to compute derivatives. Specifically, it does numerical integration of a contour integral in the complex plane to approximate the derivative. It is based on the Cauchy integral formula.

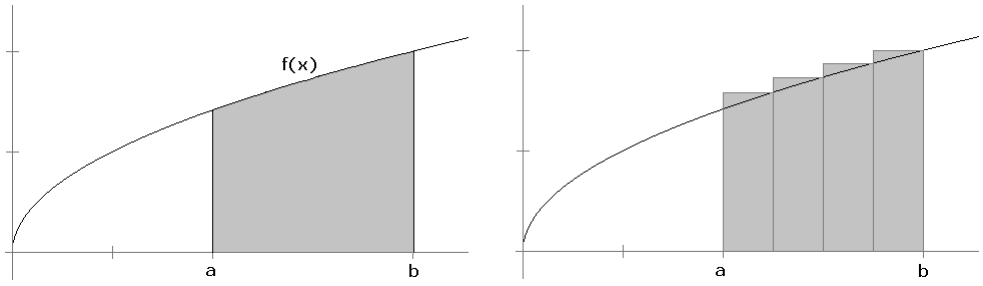
Chapter 5

Numerical integration

We can analytically take the derivative anything we encounter in calculus, but taking integrals is a different story. Consider, for example, the function $\sin(x^2)$. Using the chain rule, we can quickly find that its derivative is $2x\cos(x^2)$. On the other hand, $\int \sin(x^2)\,dx$ cannot be done using the techniques of calculus. The integral does exist, but it turns out that it cannot be written in terms of elementary functions (polynomials, rational functions, trig functions, logs, and exponentials). There are many other common functions that also can't be done, like e^{-x^2}, which is of fundamental importance in statistics. Even something as innocent-looking at $\sqrt[3]{1+x^2}$ can't be done. So numerical techniques for estimating integrals are important.

5.1 Newton-Cotes formulas

The starting point for numerical integration is the definition of the integral. Recall that the integral $\int_a^b f(x)\,dx$ gives the area under $f(x)$ between $x = a$ and $x = b$. That area is estimated with rectangles, whose areas we can find easily. See the figure below:

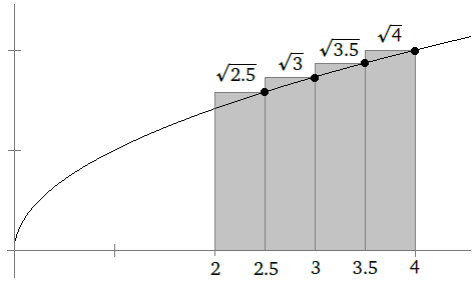


As an example, suppose we want to estimate $\int_2^4 \sqrt{x}\,dx$ using 4 rectangles. The width of each rectangle is $(4-2)/4 = .5$ units. A simple choice for the height of each rectangle is to use the function value at the rectangle's right endpoint. Then using length · width for the area of each rectangle, we get the following estimate:

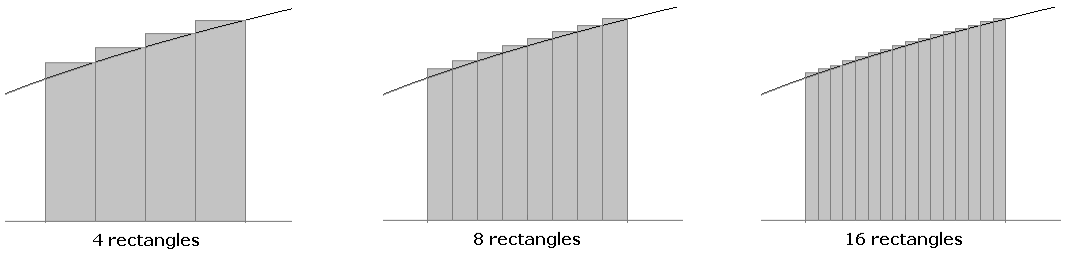
$$\int_2^4 \sqrt{x}\,dx \approx .5\cdot\sqrt{2.5} + .5\cdot\sqrt{3} + .5\cdot\sqrt{3.5} + .5\cdot\sqrt{4} \approx 3.59\ldots.$$

The exact answer[2] is given by $\frac{2}{3}x^{3/2}\Big|_2^4 \approx 3.45$. See the figure below.

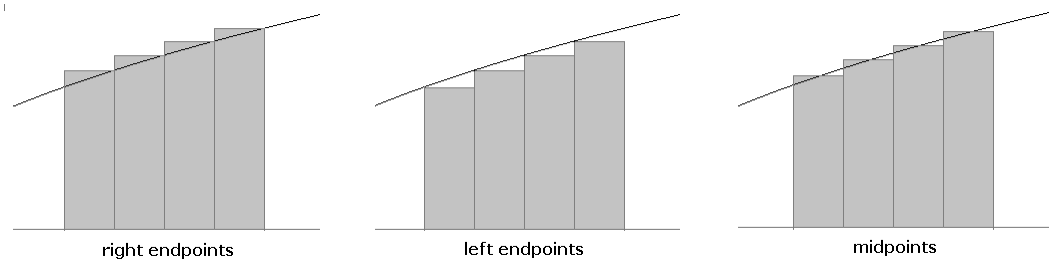
[2] It is useful to test out numerical methods on easy things so we can compare the estimate with the correct value.



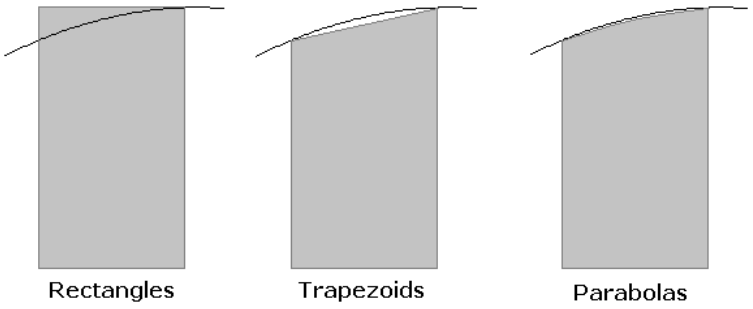
We can improve this estimate in several different ways. The simplest approach is to use more rectangles, like below. The more rectangles we have, the closer we can fit them to the curve, thus reducing the error.



As another approach, we notice that, at least for the function in the figure, right endpoints give an overestimate of the area and left endpoints give an underestimate. Averaging them might give a better estimate. A related, but different approach is to use midpoints of the rectangles for the height instead of left or right endpoints.



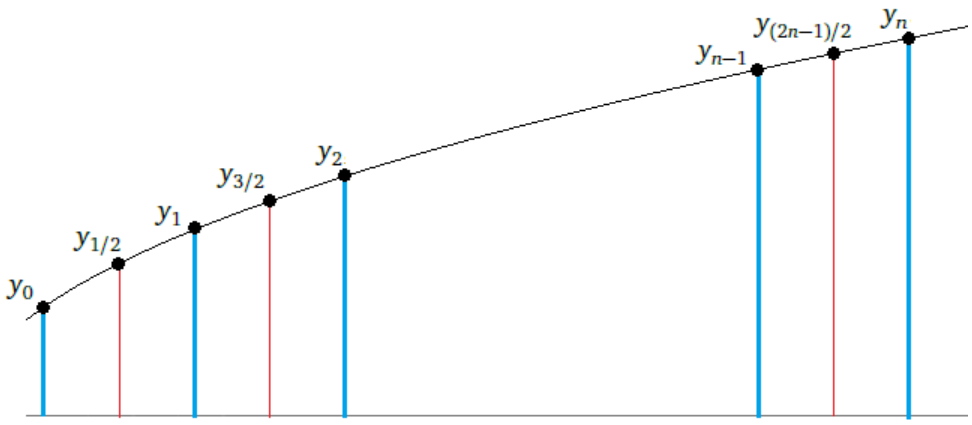
We can also change the shape of the rectangles. Once nice improvement comes from connecting the left and right endpoints by a straight line, forming a trapezoid.[1] One can also connect the endpoints by a higher order curve, like a parabola or cubic curve. See the figure below.



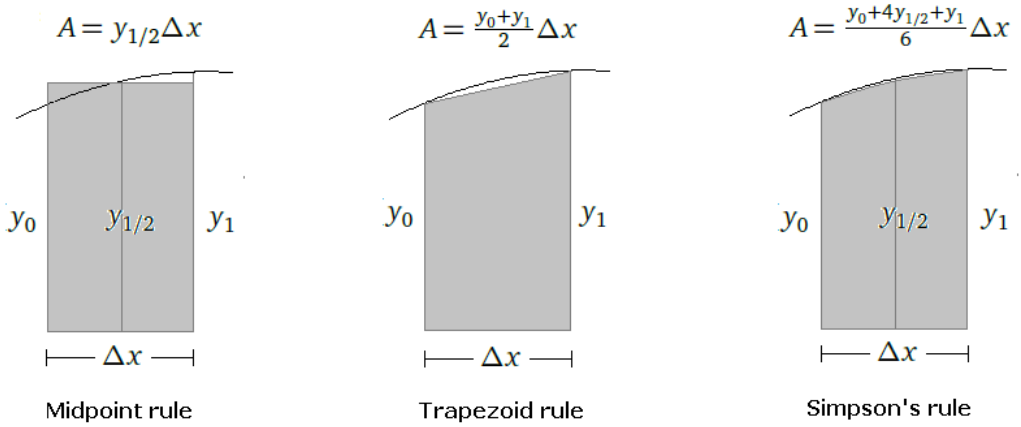
Connecting the endpoints by a straight line gives rise to the *trapezoid rule*. Using a parabola, like on the right above, gives rise to *Simpson's rule*.

[1] It is not too hard to show that using trapezoids is equivalent to averaging the left and right rectangle approximations since the area of a trapezoid comes from averaging the heights at its two ends.

We can use a little geometry to work out formulas for each of these rules. To estimate $\int_a^b f(x)\,dx$, we break up the region into n rectangles/trapezoids/parabolic rectangles of width $\Delta x = (b-a)/2$ and sum up the areas of each. In our formulas, the values of the function at the endpoints of the rectangles are denoted by $y_0, y_1, y_2, \ldots, y_n$. The function values at the midpoints of the rectangles will be denoted by $y_{1/2}, y_{3/2}, y_{5/2}, \ldots, y_{(2n-1)/2}$. See the figure below:



Below is a figure showing the areas of the building-block shapes we will use to estimate the area under the curve:



The trapezoid's area formula comes from looking at a trapezoid as equivalent to a rectangle whose height is $(y_1 + y_2)/2$, the average of the heights of the two sides.

For the parabolic rectangle, recall that we need three points to determine a parabola; we use the left and right endpoints and the midpoint. From there, the area can be determined by finding the equation of the interpolating polynomial (using the Lagrange formula, for instance) and integrating it.[1]

Composite Midpoint Rule Summing up the areas of rectangles whose heights are given by the midpoints gives

$$M = \Delta x(y_{1/2} + y_{3/2} + \cdots + y_{(2n-1)/2}).$$

Composite Trapezoid Rule Summing up the areas of the trapezoids for the trapezoid rule, we get after a little simplification

$$T = \frac{\Delta x}{2}\left(y_0 + 2y_1 + 2y_2 + \cdots + 2y_{n-1} + y_n\right).$$

[1]Please note that the presentation here differs from what you might see elsewhere. Many authors write area as $A = \frac{h}{3}(y_0 + 4y_1 + y_2)$, where their y_0, y_1, y_2 are equivalent to our $y_0, y_{1/2}, y_1$ and $h = \Delta x/2$.

Composite Simpson's Rule For Simpson's rule, summing up the areas of the parabolic rectangles, we get (after simplifying)

$$S = \frac{\Delta x}{6}\left(y_0 + y_n + 2(y_1 + y_2 + \ldots y_{n-1}) + 4(y_{1/2} + y_{3/2} + \cdots + y_{(2n-1)/2})\right).$$

Note that the left endpoint of the ith rectangle is $x_i = a + i\Delta x$ and the right endpoint is $x_{i+1} = a + (i+1)\Delta x$. The midpoint of that rectangle is $x_{(2i+1)/2} = a + \frac{2i+1}{2}\Delta x$. And each y_i is just $f(x_i)$.

The twos in the trapezoid rule formula (and in Simpson's) come from the fact that $y_1, y_2, \ldots y_{n-1}$ show up as the left endpoint of one trapezoid and the right endpoint of another. The fours in the Simpson's rule formula come from the area of the parabolic rectangles.

It is worth noting that

$$S = \frac{2M + T}{3},$$

where S, M, and T denote the Simpson's, midpoint, and trapezoid rules, respectively. This says that Simpson's rule is essentially a weighted average of the midpoint and trapezoid rules.

It is also worth noting that Simpson's rule requires about twice as many function evaluations as the midpoint and trapezoid rules, so it will take about twice as long as either of those methods. However, it turns out that Simpson's rule with n rectangles is more accurate than either of the other methods with $2n$ rectangles. That is, if we keep the number of function evaluations constant, Simpson's rule is more efficient than the others.

Example

As an example, suppose we want to estimate $\int_0^3 e^x\,dx$. We don't need a numerical method for this integral. It's just something simple to demonstrate the methods. If we use 6 rectangles, then each will have width $\Delta x = (3-0)/6 = .5$. The endpoints of the rectangles are 0, .5, 1, 1.5, 2, 2.5, and 3 and the midpoints are at .25, .75, 1.25, 1.75, 2.25, and 2.75. The calculations for the midpoint, trapezoid, and Simpson's rules are shown below:

$$\begin{aligned}
M &= \tfrac{1}{2}\left(e^{.25} + e^{.75} + e^{1.25} + e^{1.75} + e^{2.25} + e^{2.75}\right)\\
T &= \tfrac{1/2}{2}\left(e^0 + 2e^{.5} + 2e^1 + 2e^{1.5} + 2e^2 + 2e^{2.5} + e^3\right)\\
S &= \tfrac{1/2}{6}\left(e^0 + e^3 + 2\left(e^{.5} + e^1 + e^{1.5} + e^2 + e^{2.5}\right) + 4\left(e^{.25} + e^{.75} + e^{1.25} + e^{1.75} + e^{2.25} + e^{2.75}\right)\right)
\end{aligned}$$

The exact answer turns out to be $e^3 - 1 \approx 19.08554$. The midpoint estimate is 18.99, the trapezoid estimate is 19.48, and the Simpson's rule estimate is 19.08594, the most accurate of the three by far.

A program

It's pretty quick to write Python programs to implement these rules. For instance, here is the trapezoid rule:

```
def trapezoid(f, a, b, n):
    dx = (b-a) / n
    return dx/2 * (f(a) + sum(2*f(a+i*dx) for i in range(1,n)) + f(b))
```

Errors and degree of precision

The errors of the various rules for estimating $\int_a^b f(x)\,dx$ are summarized below[1]:

[1] The form these are given in differs a bit from the standard presentation in numerical analysis texts.

Rule	Error	Degree of precision
Simple rectangles	$\frac{b-a}{n}\lvert\max f'\rvert$	0
Midpoint rule	$\frac{(b-a)^3}{24n^2}\lvert\max f''\rvert$	1
Trapezoid rule	$\frac{(b-a)^3}{12n^2}\lvert\max f''\rvert$	1
Simpson's rule	$\frac{(b-a)^5}{2880n^4}\lvert\max f^{(4)}\rvert$	3

In each of the error terms, the max is to be taken over the range $[a,b]$. These error terms are generally overestimates.

The power of n, the number of rectangles, is the *order* of the rule. It is the most important part, as it is the one thing we have control over when we numerically integrate. The jump from n^2 to n^4 as we go from the midpoint and trapezoid rules to Simpson's rule accounts for the big difference in accuracy in the example above.

The *degree of precision* of an integration rule is the highest degree polynomial that the rule will integrate exactly. For example, a rule whose degree of precision is 3 would give the exact answer to $\int_a^b p(x)\,dx$, where $p(x)$ is any polynomial of degree 3 or less, like x^2+3 or x^3-x+1. In general, the degree of precision is 1 less than the order.

The trapezoid rules's degree of precision is 1 because it integrates linear and constant equations exactly. This is clear because the trapezoid rule itself uses linear equations for its estimates. What is more surprising is that Simpson's rule has a degree of precision of 3. We would expect a degree of precision of 2, considering that Simpson's rule uses parabolas in its estimates. But those estimates using parabolas actually give the exact answer for integrals of cubic equations as well. It's a bit like getting something for nothing.

As an example of where these formulas are useful, we could consider how many trapezoids we would need to estimate $\int_0^3 \sqrt{1+x^3}\,dx$ correct to within .000001. We have $a=0$, $b=3$, and using calculus or a computer algebra system, we can find that $\max f''$ on $[1,3]$ occurs at $x=1$ and is $15/(8\sqrt{2})\approx 1.3$ We then want to solve the following for n:

$$\frac{(3-0)^3}{12n^2}|1.3| < .000001.$$

Solving this, we get $n > 1710$. So we would need at least 1711 trapezoids to be guaranteed of an error less than .000001, though since the error formula is an overestimate, we could probably get away with fewer.

Higher orders

The formulas we have been considering in this section are referred to as *Newton-Cotes formulas*. There are Newton-Cotes formulas of all orders. For instance, if we use cubic curves on top of our rectangles instead of the parabolas that Simpson's rule uses, we get something called *Simpson's 3/8 rule*. A cubic rectangle has area $A=\frac{\Delta x}{8}(y_0+3y_{1/3}+3y_{2/3}+y_1)$ and the areas of all the little cubic rectangles these can be summed to give a composite formula like the ones we came up with for the other rules.[1] Simpson's 3/8 rule has degree of precision 3, the same as Simpson's rule. Its error term is $\frac{(b-a)^5}{6480n^4}|\max f^{(4)}|$.

If we use quartic curves, we get *Boole's rule*. A quartic rectangle has area $A=\frac{\Delta x}{90}(7y_0+32y_{1/4}+12y_{1/2}+32y_{3/4}+7y_1)$. We can sum the areas of the quartic rectangles to get a composite rule. Note that Boole's rule has degree of precision 5, a jump of 2 orders over Simpson's 3/8 rule. In general, as we progress through higher degree polynomials on the top of our rectangles, the degrees of precision alternate not changing and jumping up by 2. The error term for Boole's rule is $\frac{(b-a)^7}{1935360n^6}|\max f^{(6)}|$.

[1] Simpson's 3/8 rule gets its name from an alternate form of the cubic rectangle's area: $A=\frac{3}{8}(y_0+3y_1+3y_2+y_3)h$, where $h=x_1-x_0$.

5.2 The iterative trapezoid rule

Let's start with the trapezoid rule to estimate $\int_a^b f(x)\,dx$ with just one trapezoid of width $h = b-a$. Our estimate is

$$T_1 = h\frac{y_0 + y_1}{2}.$$

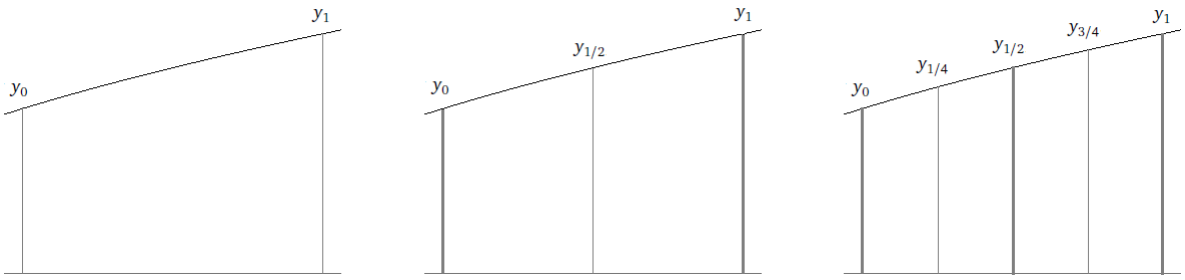
Then let's cut the trapezoid exactly in half and use the two resulting trapezoids to estimate the integral:

$$T_2 = \frac{h}{2}\left(\frac{y_0 + y_{1/2}}{2} + \frac{y_{1/2} + y_1}{2}\right) = \frac{1}{2}T_1 + \frac{h}{2}y_{1/2}.$$

The key here is that we already computed part of this formula when we did T_1, so we can make use of that. If we cut things in half again, we get

$$T_3 = \frac{h}{4}\left(\frac{y_0 + y_{1/4}}{2} + \frac{y_{1/4} + y_{1/2}}{2} + \frac{y_{1/2} + y_{3/4}}{2} + \frac{y_{3/4} + y_1}{2}\right) = \frac{1}{2}T_2 + \frac{h}{4}(y_{1/4} + y_{3/4}).$$

See the figure below.



We can continue halving things in this way, doubling the number of trapezoids at each step, until we get to our desired accuracy. We can describe the algorithm as follows:

Start with $h_1 = b-a$ and $T_1 = h_1\frac{f(a)+f(b)}{2}$.

At each subsequent step, we compute

$$h_n = \frac{1}{2}h_{n-1}$$
$$T_n = \frac{1}{2}T_{n-1} + h_n\left(f(a+h_n) + f(a+3h_n) + f(a+5h_n) + \cdots + f(a+(2^{n-1}-1)h_n)\right).$$

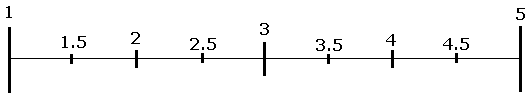
This method is fairly reliable, even if the function being integrated is not very smooth. In that case, though, it may take a good number of halvings to get to the desired tolerance, leaving an opportunity for roundoff error to creep in. The authors of *Numerical Recipes* note that this method is usually accurate up to at least 10^{-10} (using double-precision floating-point).

An example

Let's try this method on $\int_1^5 e^{-x^2}\,dx$. For simplicity, we will round to 4 decimal places. Here are the first four steps of the method:

$$h_1 = 4 \qquad T_1 = 4(e^{-1^2} + e^{-5^2})/2 = .7358$$
$$h_2 = 2 \qquad T_2 = \tfrac{7.358}{2} + 2e^{-3^2} = .3681$$
$$h_3 = 1 \qquad T_3 = \tfrac{.3681}{2} + 1\left(e^{-2^2} + e^{-4^2}\right) = .2024$$
$$h_4 = .5 \qquad T_4 = \tfrac{.2024}{2} + .5\left(e^{-1.5^2} + e^{-2.5^2} + e^{-3.5^2} + e^{-4.5^2}\right) = .1549$$

Shown below are x-values used at each step:



If we continue the process, the next several trapezoid estimates we get are .1432, .1404, .1396, .1395, .1394, .1394, .1394. We see that it is correct to four decimal places after a total of nine steps. By this point, as the number of trapezoids doubles at each step, we are approximating the integral with $2^8 = 256$ trapezoids. It takes about 21 steps for the method to be correct to 10 decimal places. By this point, there are 2^{20} trapezoids, corresponding to $2^{19} \approx 500{,}000$ new function evaluations. This continued doubling means that we will eventually reach a point where there are too many computations to finish in a reasonable amount of time.

A program

Here is a short Python program implementing this method:

```
def iterative_trap(f, a, b, toler=1e-10):
    h = b - a
    T = h * (f(a)+f(b))/2
    n = 1
    print(T)
    while n==1 or abs(T - oldT) > toler:
        h = h/2
        n += 1
        T, oldT = T/2 + h * sum(f(a + i*h) for i in range(1, 2**(n-1), 2)), T
        print(T)
```

5.3 Romberg integration

We can apply Richardson extrapolation to each of the iterative trapezoid approximations to obtain a higher order extrapolation. Recall that for a numerical method, $F(h)$ of order n, Richardson extrapolation produces a higher order numerical method via the formula

$$\frac{2^n F(h/2) - F(h)}{2^n - 1}.$$

Applying Richardson extrapolation to the trapezoid rule (which is of order 2), we get for $k = 1, 2, \ldots$

$$S_k = \frac{2^2 T_{k+1} - T_k}{2^2 - 1} = \frac{4T_{k+1} - T_k}{3}.$$

We have chosen the symbol S here because extrapolating the trapezoid rule actually gives us Simpson's rule, an order 4 method. From here, we can extrapolate again to get

$$B_k = \frac{2^4 S_{k+1} - S_k}{2^4 - 1} = \frac{16S_{k+1} - S_k}{15}.$$

Here, we have chosen the symbol B because extrapolating Simpson's rule gives us Boole's rule, an order 6 method.

We can continue this process. Each time we extrapolate, we increase the order by 2. We can record all of this in a table, like below:

T_1

$T_2 \quad S_1$

$T_3 \quad S_2 \quad B_1$

$\ldots$

This process is called *Romberg integration*. The T, S, and B notation becomes cumbersome to extend, so instead we will use the entry R_{ij} to refer to the entry in row i, column j. Here are the first five rows of the table.

R_{11}				
R_{21}	$R_{22}=\frac{4R_{21}-R_{11}}{3}$			
R_{31}	$R_{32}=\frac{4R_{31}-R_{21}}{3}$	$R_{33}=\frac{4^2R_{32}-R_{22}}{4^2-1}$		
R_{41}	$R_{42}=\frac{4R_{41}-R_{31}}{3}$	$R_{43}=\frac{4^2R_{42}-R_{32}}{4^2-1}$	$R_{44}=\frac{4^3R_{43}-R_{33}}{4^3-1}$	
R_{51}	$R_{52}=\frac{4R_{51}-R_{41}}{3}$	$R_{43}=\frac{4^2R_{52}-R_{42}}{4^2-1}$	$R_{54}=\frac{4^3R_{53}-R_{43}}{4^3-1}$	$R_{55}=\frac{4^4R_{54}-R_{44}}{4^4-1}$

The table can be extended as far as we want. The first column is gotten by the iterative trapezoid rule given earlier, namely $h_1=b-a$, $R_{11}=h_1(f(a)+f(b)/2$, and for $n>1$

$$h_n=\frac{1}{2}h_{n-1}$$
$$R_{n,1}=\frac{1}{2}R_{n-1,1}+h_n\left(f(a+h_n)+f(a+3h_n)+f(a+5h_n)+\cdots+f(a+(2^{n-1}-1)h_n)\right).$$

The rest of the entries in the table are then developed from these according to the given extrapolation formulas. One thing to note in the table is that since the order of each extrapolation goes up by 2, the 2^n term in the extrapolation formula will become 4^n.

Example

Let's try this method on $\int_1^5 e^{-x^2}\,dx$. This is the same function we tried with the iterative trapezoid method. The values we found in that example form the first column in Romberg integration. We then use the formulas above to calculate the rest of the values.

Here is the first six rows of the Romberg table, shown to nine decimal places:

.735758882					
.368126261	.245582054				
.202378882	.147129755	.140566269			
.154856674	.139015938	.138475016	.138441822		
.143242828	.139371546	.139395253	.139409860	.139413656	
.140361311	.139400805	.139402756	.139402875	.139402847	.139402837

Here is the calculation of R_{22}, for example:

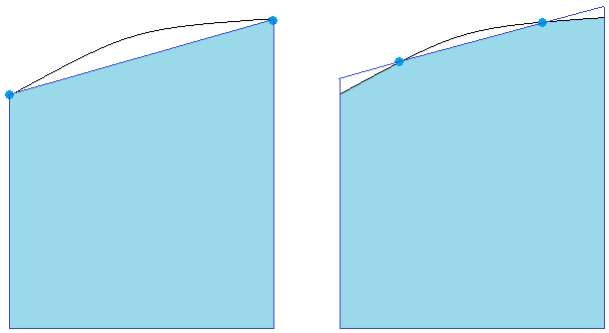
$$R_{22}=\frac{4R_{21}-R_{11}}{3}=\frac{4\times .368126261-.735758882}{3}=.245582054.$$

If we extend the table by a few more rows, we can get the answer correct to 15 decimal places: 0.1394027926389689. In general, $R_{n,n}$ will be a much more accurate estimate than $R_{n,1}$, with very little additional computation needed.

5.4 Gaussian quadrature

Recall that the simple way to estimate the area under a curve is to use rectangles. Using the left or right endpoints to give the rectangle's height is the simplest way, but it is not very accurate. We saw that it was more accurate to use the midpoints of the interval as the height.

For the trapezoid rule, we connect the two endpoints of the interval by a line. But maybe we could connect different points to get a more accurate estimation. See the figure below:



To make things concrete, let's suppose we are estimating $\int_{-1}^{1} f(x)\,dx$. The estimate using a single trapezoid is $f(-1)+f(1)$. We want to replace 1 and -1 with some other values, call them c and d, to see if we can get a more accurate method.

Recall that there is a close relation between the error term (accuracy) and the degree of precision of a method. That is, if a numerical method integrates polynomials up to a certain degree exactly, then for any function that is reasonably well approximated by polynomials, we would expect the numerical method to be pretty accurate. For example, near $x = 0$, $\sin x$ is well-approximated by the first two terms of its Taylor series, $x - x^3/6$. If we have a numerical method whose degree of precision is 3, then $x - x^3/6$ would be integrated exactly, and so an integral like $\int_{-1/4}^{1/4} \sin x\,dx$ would be pretty accurately estimated by that numerical method.

The trapezoid rule has degree of precision 1. By choosing c and d well, we can create a rule with a higher degree of precision. In particular, suppose we want our rule to have degree of precision 2; that is, it should exactly integrate degree 0, 1, and 2 polynomials. In particular, for $f(x) = x$ and $f(x) = x^2$, we must have

$$\int_{-1}^{1} f(x)\,dx = f(c) + f(d).$$

Plugging in, we get the following two equations:

$$\begin{aligned} 0 &= \int_{-1}^{1} x = c + d \\ \frac{2}{3} &= \int_{-1}^{1} x^2 = c^2 + d^2. \end{aligned}$$

We can quickly solve these to get $c = -\sqrt{1/3}$, $d = \sqrt{1/3}$. So for any polynomial of degree 2 or less, this rule will exactly integrate it. For example, if $f(x) = 4x^2 + 6x - 7$, we have

$$\int_{-1}^{1} 3x^2 + 6x - 7\,dx = x^3 + 3x^2 - 7x\Big|_{-1}^{1} = -12.$$

But we also have

$$f(-\sqrt{1/3}) + f(\sqrt{1/3}) = \left[4\left(-\sqrt{1/3}\right)^2 + 6\left(-\sqrt{1/3}\right) - 7\right] + \left[4\left(\sqrt{1/3}\right)^2 + 6\sqrt{1/3} - 7\right] = -12.$$

So we have the interesting fact that, we can compute the integral of any quadratic on $[-1, 1]$ just by evaluating it at two points, $\pm\sqrt{1/3}$.

A nice surprise is that this method actually has degree of precision 3. It exactly integrates polynomials up to degree three. We can see this because

$$\int_{-1}^{1} x^3\,dx = 0 = \left(-\sqrt{1/3}\right)^3 + \left(\sqrt{1/3}\right)^3.$$

The method that we just considered is called two-point *Gaussian quadrature*.[1]

Extending from $[-1,1]$ to $[a,b]$

The method developed above works specifically for $[-1,1]$, but we can generalize it to any interval $[a,b]$. The key is that the transformation of $[a,b]$ to $[-1,1]$ is given by first subtracting $(b+a)/2$ to center things, and then scaling by $(b-a)/2$, which is the ratio of the lengths of the intervals. Given an integral $\int_a^b f(x)\,dx$, we make the transformation by using the substitution $u = (x - \frac{b+a}{2})/\frac{b-a}{2}$. Under this transformation $x = a$ becomes $u = -1$ and $x = b$ becomes $u = 1$. From this, we get the following:

$$\int_a^b f(x)\,dx = \int_{-1}^{1} f\left(\frac{b-a}{2}u + \frac{b+a}{2}\right)\frac{b-a}{2}\,du = p\Big[f(-p\sqrt{1/3}+q) + f(p\sqrt{1/3}+q)\Big],$$

where $p = \frac{b-a}{2}$ and $q = \frac{b+a}{2}$.

General rule for two-point Gaussian quadrature

To estimate $\int_a^b f(x)\,dx$, compute $p = \frac{b-a}{2}$, $q = \frac{b+a}{2}$ and then

$$\int_a^b f(x)\,dx \approx p\Big[f(-p\sqrt{1/3}+q) + f(p\sqrt{1/3}+q)\Big].$$

An example Suppose we want to estimate $\int_1^4 \sqrt{1+x^3}\,dx$. We compute $p = 1.5$ and $q = 2.5$. So we have

$$\int_1^4 \sqrt{1+x^3}\,dx = 1.5\left[\sqrt{1+(-1.5\sqrt{1/3}+2.5)^3} + \sqrt{1+(1.5\sqrt{1/3}+2.5)^3}\right] \approx 12.857.$$

To three decimal places, the exact answer is 12.871, so we have done pretty well for just two function evaluations.

A composite method

We can turn this method into a composite method, using n of these special trapezoids. We break up the region from a to b into n evenly spaced regions of width $\Delta x = (b-a)/n$. The left and right endpoints of the ith region are $a + i\Delta x$ and $a + (i+1)\Delta x$. We apply Gaussian quadrature to each of these regions. For the ith region, we have $p = \Delta x/2$ and $q = a + (i + \frac{1}{2})\Delta x$. Summing up all of these regions gives

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\sum_{i=1}^{n} f\left(\frac{\Delta x}{2}\left(2i+1-\sqrt{1/3}\right)\right) + f\left(\frac{\Delta x}{2}\left(2i+1-\sqrt{1/3}\right)\right).$$

The error term for this method $\frac{(b-a)^5}{4320n^4}|\max f^{(4)}|$, which is actually a little better than Simpson's rule, and with one less function evaluation.

[1] "Quadrature" is an older term for "integration."

Gaussian quadrature with more points

We can extend what we have developed to work with more points. Suppose we want to work with three points. Take as an example Simpson's rule, which uses three points. We can write it as $\frac{1}{6}y_0 + \frac{2}{3}y_{1/2} + \frac{1}{6}y_1$. The three points are evenly spaced, but we can do better, and actually get a method whose degree of precision is 5, if we move the points around. But we will also have to change the coefficients.

In particular, our rule must be of the form $c_1 f(x_1) + c_2 f(x_2) + c_3 f(x_3)$. There are six values to find. Since we know this is to have a degree of precision of 5, we can set up a system of six equations (one for each degree from 0 to 5) and solve to get our values. However, this is a system of nonlinear equations and is a bit tedious to solve.

Instead, it turns out the values are determined from a family of polynomials called the *Legendre polynomials*. (Notice the parallel with interpolation, where the ideal placement of the interpolating points was given by the roots of another class of polynomials, the Chebyshev polynomials.)

Like the Chebyshev polynomials, the Legendre polynomials can be defined recursively. In particular, they are defined by $P_0(x) = 1$, $P_1(x) = x$, and

$$P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x).$$

The next two are $P_2(x) = (3x^2 - 1)/2$ and $P_3(x) = (5x^3 - 3x)/2$.

The roots of the Legendre polynomials determine the x_i. For instance, $P_2(x) = (3x^2 - 1)/2$ has roots at $\pm\sqrt{1/3}$, which are exactly the points we found earlier for 2-point Gaussian quadrature. The roots of $P_3(x) = (5x^3 - 3x)/2$ are 0 and $\pm\sqrt{3/5}$. These are the x_1, x_2, and x_3, we want to use for 3-point Gaussian quadrature. The coefficients, c_i, are given as follows:

$$c_i = \frac{2}{(1 - x_i^2)(P_n'(x_i))^2}.$$

Fortunately, the x_i and c_i are fixed, so we just need someone to calculate them once to reasonably high accuracy, and then we have them. Here is a table of the values for a few cases:

n	x_i	c_i
2	$-\sqrt{1/3}$, $\sqrt{1/3}$	1, 1
3	$-\sqrt{3/5}$, 0, $\sqrt{3/5}$	5/9, 8/9, 5/9
4	-.861136, -.339981, .339981, .861136	.347855, .652145, .652145, .347855

Note that for $n = 4$, the x_i are given by $\pm\sqrt{(3 - 2\sqrt{6/5})/7}$, and the c_i are given by $(18 \pm \sqrt{30})/36$. People have tabulated the x_i and c_i to high accuracy for n-point Gaussian quadrature for many values of n.[1]

Note that the same transformation from $[a, b]$ to $[-1, 1]$ that was made for two-point Gaussian quadrature applies in general. Here is the general (non-composite) formula:

$$\int_a^b f(x) \approx p\Big[c_1 f(px_1 + q) + c_2 f(px_2 + q) + \cdots + c_n f(px_n + q)\Big],$$

where $p = \frac{b-a}{2}$ and $q = \frac{b+a}{2}$. We could generalize this to a composite rule if we like, but we won't do that here.

An example Suppose we want to use 3-point Gaussian quadrature to estimate $\int_1^4 \sqrt{1 + x^3}\, dx$. We have $p = 1.5$ and $q = 2.5$ and we get to 5 decimal places

$$\int_1^4 \sqrt{1 + x^3}\, dx = 1.5\left[\frac{5}{9}\sqrt{1 + (-1.5\sqrt{3/5} + 2.5)^3} + \frac{8}{9}\sqrt{1 + (0 + 2.5)^3} + \frac{5}{9}\sqrt{1 + (1.5\sqrt{3/5} + 2.5)^3}\right] \approx 12.87085.$$

The exact answer is 12.87144 to five decimal places.

[1] See http://pomax.github.io/bezierinfo/legendre-gauss.html, for example.

A little bit of theory

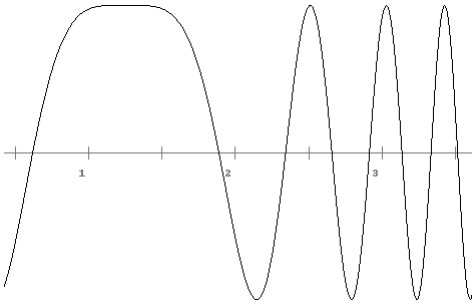
What we have left out is why the Legendre polynomials have anything to do with Gaussian quadrature. It turns out that the Legendre polynomials have a property called orthogonality.[1] What this means is that $\int_{-1}^{1} P_i(x)P_j(x)\,dx = 0$ if $i \neq j$. This property is key to making the math work out (though we will omit it).

The higher the degree of our approximation, the more accurate it is for certain types of functions, but not for all types of functions. In particular, these high degree methods are good if our integrand is something that can be approximated closely by polynomials. However, if our function is something that has a vertical asymptote, then it is not well approximated by polynomials, and to get better accuracy, we need a different approach.

The approach turns out to be to rewrite $\int_a^b f(x)\,dx$ in the form $\int_a^b w(x)g(x)\,dx$, where $w(x)$ is what is called a *weight function*. For ordinary Gaussian quadrature, $w(x) = 1$, but we can take $w(x)$ to be other useful things like e^{-x}, e^{-x^2}, or $1/\sqrt{1+x^2}$. Different weight functions lead to different classes of polynomials. For instance, $w(x) = 1$ we saw leads to the Legendre polynomials. For $w(x) = e^{-x}$, we are led to a family called the Laguerre polynomials. For $w(x) = e^{-x^2}$, we are led to a family called to the Hermite polynomials, and with $w(x) = 1/\sqrt{1+x^2}$, we are led to the Chebyshev polynomials. There are many other weight functions possible and there are techniques that can be used to figure out the x_i and c_i for them.

5.5 Adaptive quadrature

With some graphs, spacing our regions evenly is inefficient. Consider the graph of $\sin(4x^2 - 10x + 3/2)$ shown below:



Suppose we are using the trapezoid rule. The region near $x = 1$ will be integrated pretty accurately using only a few trapezoids, whereas the region around $x = 3$ will need quite a bit more. If, for instance, we use 1000 trapezoids to approximate the integral from 1 to 3, we should get a reasonably accurate answer, but we would be using way more trapezoids than we really need around $x = 1$, wasting a lot of time for not much additional accuracy.

We could manually break the region up and apply the trapezoid rule separately on each region, but it would be nice if we had a method that could automatically recognize which areas need more attention than others. The trick to doing this is based off of the iterative trapezoid rule we considered a few sections back.

Recall that the iterative trapezoid rule involves applying the trapezoid rule with one trapezoid, then breaking things into two trapezoids, then four, then eight, etc., at each step making use of the results from the previous calculation. In the example from that section, the approximations we got were .1432, .1404, .1396, .1395, .1394, .1394, .1394. We see that they are settling down on a value. It looks like we will be within ϵ of the exact answer once successive iterates are within ϵ of each other.[2] This suggests the following solution to our problem.

Adaptive quadrature process To integrate $\int_a^b f(x)\,dx$, correct to within ϵ, using a variable number of trapezoids depending on the region of the function, do the following:

[1] Orthogonality is a concept covered in linear algebra. It is a generalization of the geometrical notion of being perpendicular.

[2] This is not always true, but it is true a lot of the time, and it gives us something to work with.

1. Start with the trapezoid rule, $T_{[a,b]}$, with one trapezoid on $[a,b]$.
2. Then cut $[a,b]$ in half at $m=(a+b)/2$ and run the trapezoid rule on each half, getting $T_{[a,m]}$ and $T_{[m,b]}$.
3. Compute the difference $T_{[a,b]}-(T_{[a,m]}+T_{[m,b]})$. If it is less than 3ϵ, then we're done. Otherwise apply the same procedure on $[a,m]$ and $[m,b]$ with ϵ replaced with $\epsilon/2$.

This procedure is called *adaptive quadrature* – the "adaptive" part meaning that the method adapts the number of trapezoids to what each region of the graph demands.

There is one odd thing in the method above, and that is why do we stop when the difference is less than 3ϵ? Where does the 3 come from? Here is the idea: Write

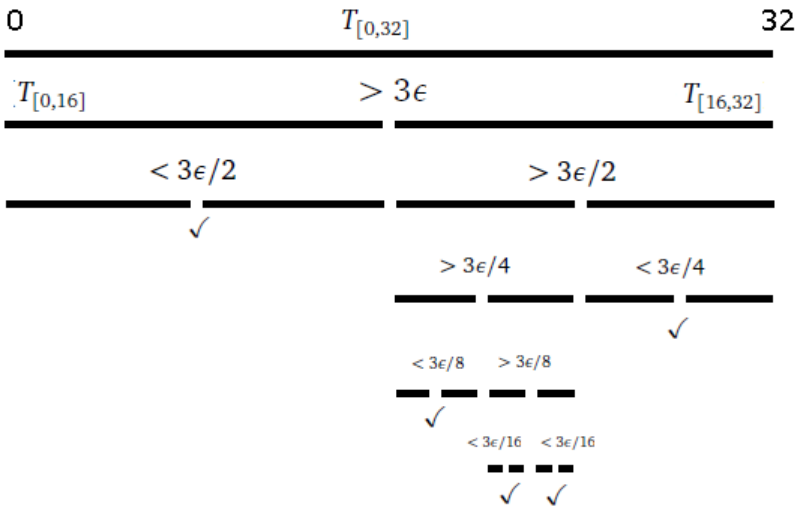
$$\int_a^b f(x)\,dx = T_{[a,b]}+C_1h^3 \qquad \int_a^b f(x)\,dx = T_{[a,m]}+C_2\left(\frac{h}{2}\right)^3 \qquad \int_a^b f(x)\,dx = T_{[m,b]}+C_3\left(\frac{h}{2}\right)^3.$$

The terms C_1h^3, $C_2(h/2)^3$, and $C_3(h/2)^3$ are the error terms. Each constant is different, but we can make the approximation $C_1\approx C_2\approx C_3$ and replace all three with some constant C. Then the difference that we compute in the method is

$$T_{[a,b]}-(T_{[a,m]}+T_{[m,b]})=\frac{3}{4}Ch^3=3\left(C\left(\frac{h}{2}\right)^3+C\left(\frac{h}{2}\right)^3\right).$$

That is, the difference is about three times the error of $T_{[a,m]}+T_{[m,b]}$, which is our approximation to $\int_a^b f(x)\,dx$, so that's where the 3ϵ comes from.

Here is an example of how things might go. Suppose we are estimating $\int_0^{32} f(x)\,dx$ to within a tolerance of ϵ. The figure below shows a hypothetical result of adaptive quadrature on this integral:



In the example above, we start by computing $\left|T_{[0,32]}-(T_{[0,16]}+T_{[16,32]})\right|$. In this hypothetical example, it turns out to be greater than 3ϵ, so we apply the procedure to $[0,16]$ and $[16,32]$ with ϵ replaced with $\epsilon/2$.

Then suppose it turns out that $\left|T_{[0,16]}-(T_{[0,8]}+T_{[8,16]})\right|<3\epsilon/2$, so that section is good and we are done there. However, the right section has $\left|T_{[16,32]}-(T_{[16,24]}+T_{[24,32]})\right|>3\epsilon/2$, so we have to break that up into halves and apply the procedure to each half with tolerance $\epsilon/4$.

We then continue the process a few more times. Notice that if we add up the ϵ values everywhere that we stop (where the check marks are), we get $\frac{3}{2}\epsilon+\frac{3}{4}\epsilon+\frac{3}{8}\epsilon+\frac{3}{16}\epsilon+\frac{3}{16}\epsilon$, which sums up to 3ϵ.

Note that we can also use Simpson's rule or Gaussian quadrature in place of the trapezoid rule here. If we do that, instead of checking if the difference is less than 3ϵ, we check if it is less than $\frac{2}{3}(2^n-1)\epsilon$, where n is the order of the method (4 for Simpson's rule and 2-point Gaussian quadrature, 6 for Boole's rule and 3-point Gaussian quadrature).[1]

[1]The value 2^n-1 comes from the same calculation as we did above for the trapezoid rule. The factor of $\frac{2}{3}$ is recommended in practice to be conservative since the derivation involves making a few approximations.

A program

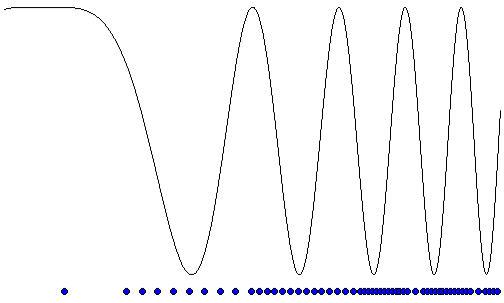
The natural way to implement adaptive quadrature is recursively. Here is a Python implementation:

```
def adaptive_quad(f, a, b, toler):
    m = (a+b) / 2
    T1 = (b-a) * (f(a) + f(b)) / 2
    T2 = (b-a) * (f(a) + f(m)) / 4
    T3 = (b-a) * (f(m) + f(b)) / 4

    if abs(T1-T2-T3)< 3*toler:
        return T2+T3
    else:
        return adaptive_quad(f, a, m, toler/2) + adaptive_quad(f, m, b, toler/2)
```

A problem with this method is that it is a little inefficient in that with each set of recursive calls, we find ourselves recalculating many values that we calculated earlier. For instance, $f(a)$ as part of T1 and T2 and then it may be computed again in the first recursive call to `adaptive_quad`. We can speed things up by saving the values of the function evaluations (which can often be slow if the function is complicated) and passing them around as parameters. In addition, the values of T2 and T3 are recomputed as T1 in the recursive calls. If we save these as parameters, we can get a further speed-up. The improved code is below:

```
def adaptive_quad(f, a, b, toler):
    return adaptive_quad_helper(f, a, b, f(a), f(b), (b-a)*(f(a)+f(b))/2, toler)

def adaptive_quad_helper(f, a, b, fa, fb, T1, toler):
    m = (a+b) / 2
    fm = f(m)
    T2 = (b-a) * (fa + fm) / 4
    T3 = (b-a) * (fm + fb) / 4

    if abs(T1-T2-T3)< 3*toler:
        return T2+T3
    else:
        return adaptive_quad_helper(f, a, m, fa, fm, T2, toler/2) +\
               adaptive_quad_helper(f, m, b, fm, fb, T3, toler/2)
```

Shown below is the graph of $f(x) = \sin(4x^2 - 10x + 3/2)$ from x =1 to 3, and in blue are all the midpoints that are computed by the algorithm above with a tolerance of $\epsilon = .01$.



5.6 Improper and multidimensional integrals

Improper integrals are those whose that have infinite ranges of integration or discontinuities somewhere in the range. Some examples include

$$\int_0^\infty e^{-x^2}\,dx \qquad \text{infinite range of integration}$$

$$\int_0^1 \frac{\sin x}{x}\,dx \qquad \text{undefined at left endpoint of range}$$

$$\int_{-1}^1 \frac{1}{\sqrt{x}}\,dx \qquad \text{undefined in middle of range at } x=0.$$

Integrals with an infinite range of integration

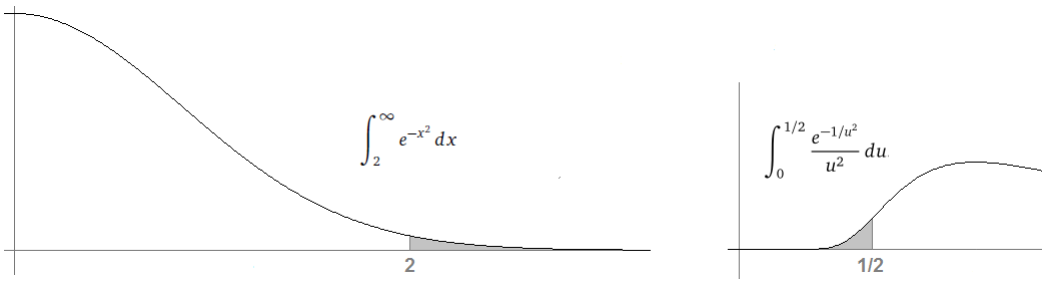
For integrals with an infinite range, the trick is often to use a substitution of the form $u = 1/x$ to turn the infinite range into a finite one. For example, using $u = 1/x$ on $\int_2^\infty e^{-x^2}\,dx$, we have

$$\begin{aligned} u &= \frac{1}{x} \\ du &= -\frac{1}{x^2}\,dx \quad \rightarrow \quad dx = -\frac{1}{u^2}\,du \\ x &= 2 \quad \rightarrow \quad u = 1/2 \\ x &= \infty \quad \rightarrow \quad u = 0 \end{aligned}$$

This gives

$$\int_2^\infty e^{-x^2}\,dx = \int_{1/2}^0 \frac{e^{-1/u^2}}{-u^2}\,du = \int_0^{1/2} \frac{e^{-1/u^2}}{u^2}\,du.$$

Shown below are the two functions. The shaded areas have the same value.



We can then use a numerical method to approximate the value of the integral.

If we were to change the integral to run from $x = 0$ to ∞ instead of from $x = 1$ to ∞, then $u = 1/x$ would give us a problem at 0. A solution to this is to break the integral into two parts:

$$\int_1^\infty e^{-x^2}\,dx = \int_0^1 e^{-x^2}\,dx + \int_1^\infty e^{-x^2}\,dx.$$

In general, for integrals with infinite ranges, a transformation of the form $u = 1/x$ usually will work, but there are other substitutions that are sometimes useful.

Integrals with discontinuities

We have to be careful which method we use when integrating functions with discontinuities. If a discontinuity occurs at an endpoint, then the trapezoid method and those derived from it won't work since they need to know

the values at the endpoints. However, some of the other methods will work. The midpoint method, for example, would be okay, since it does not require points at the ends of the interval. It is sometimes called an *open* method for this reason. Gaussian quadrature would also be okay.

One thing we can do is create an extended midpoint method, analogous to the iterative trapezoid method and extend it via extrapolation into a Romberg-like method. The one trick to doing this is that in order for our extended midpoint method to make use of previous steps, it is necessary to cut the step into one third of itself, instead of in half.

If the discontinuity is removable, one option is to remove it. For instance, $\sin(x)/x$ has a removable discontinuity at $x = 0$, since $\lim_{x\to 0} \sin(x)/x = 1$. We could just pretend $\sin(0)/0$ is 1 when we do our numerical integration. If we know the discontinuity is removable, but don't know the limit, we can use something like the extended midpoint method, Gaussian quadrature, or adaptive quadrature.

If the discontinuity is not removable, like in $\int_0^1 \frac{1}{\sqrt{x}}\,dx$, then the function tends towards infinity at the discontinuity. We could still use the extended midpoint method, Gaussian quadrature, or adaptive quadrature, but there are certain substitutions that can be made that will make things more accurate.

If the discontinuity happens in the middle of the interval, and where know where it occurs, then we can break the integral up at the discontinuity, like below, and then use a numerical method on each integral separately:

$$\int_{-1}^{0} \frac{1}{\sqrt{x}}\,dx + \int_{0}^{1} \frac{1}{\sqrt{x}}\,dx.$$

If we don't know where a discontinuity happens, then there are things one can do, but we won't cover them here.

It is important to note that we could end up with an integral that is infinite, like $\int_0^1 \frac{1}{x}\,dx$. For something like this, the numerical methods we have won't work at all, since there is no finite answer. A good way to recognize that the integral is infinite is if the methods seem to be misbehaving and giving widely varying answers.

Multiple integrals

Multidimensional integrals are somewhat tougher to handle than single integrals. In general, we have an integral of the form

$$\iint \cdots \int_D f(x_1, x_2, \ldots, x_n)\,dV,$$

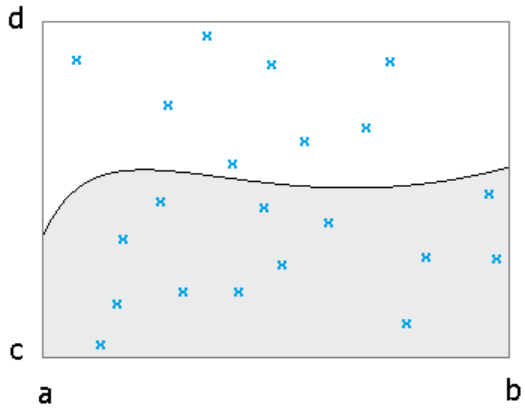
where D is some region. If D is not too complicated and f is relatively smooth, then the integral can be evaluated higher-dimensional analogs of some of the methods we talked about, like Gaussian quadrature.

One problem with this is that the number of computations explodes as the dimension n grows. For instance, if we were to use 10 rectangles to approximate a region in one dimension, for the same level of accuracy in two dimensions, we would have a 10×10 grid of 100 squares. In three dimensions, this would be a $10 \times 10 \times 10$ grid of 1000 cubes, and in n dimensions, it would consist of 10^n (hyper)cubes. So to even get one digit of accuracy for a 20-dimensional integral would be just about impossible using these methods (and 20-dimensional integrals really do show up in practice).

Another option that sometimes works is to reduce a multiple integral into a series of one-dimensional integrals and using a one-dimensional technique on each one.

For high dimensions and really complicated regions of integration, a completely different approach, called Monte Carlo integration, is used. It involves evaluating the integral at random points and combining the information from that to get an approximation of the integral. It is easy to use and widely applicable, but it converges slowly, taking a long time to get high accuracy.

As a simple example of it, suppose we want to estimate $\int_a^b f(x)\,dx$. Imagine drawing a box around the region of integration and throwing darts into the box. Count how many darts land in region and how many land outside it. That percentage, multiplied by the box's area, gives an estimate of the region's area. See the figure below (the x's represent where the darts landed):



Here is some Python code that implements this procedure:

```
from random import uniform

def monte_carlo_integrate(f, a, b, c, d, num_points):
    inside_count = 0
    for i in range(num_points):
        x = uniform(a,b)
        y = uniform(c,d)
        if y <= f(x):
            inside_count += 1
    return inside_count/num_points * (b-a)*(d-c)
```

This estimates $\int_a^b f(x)\,dx$. The values c and d are y-coordinates of the box drawn around the region.

Note that this code will only work if $f(x) \geq 0$. We can fix that by changing the counting code to the following:

```
if 0 <= y <= f(x):
        inside_count += 1
else if f(x) <= y <= 0:
        inside_count -= 1
```

5.7 Summary of integration techniques

The preceding sections are not a how-to guide for what algorithms to use when, nor are they are a rigorous development of methods and error bounds.They are just meant to be an overview of some of the methods and key ideas behind numerical integration. For practical information, see *Numerical Recipes*, and for rigorous derivations, see a numerical analysis textbook.

In practice, *Numerical Recipes* recommends the iterative trapezoid rule, and an iterative Simpson's rule (which is actually just the second column of Romberg integration) as reliable methods, especially for integrals of non-smooth functions (functions with sudden changes in slope). They recommend Romberg integration for smooth functions with no discontinuities.

Chapter 6

Numerical methods for differential equations

A differential equation is, roughly speaking, an equation with derivatives in it. A very simple example of one is $y' = 2x$. Translated to English, it is asking us to find a function, $y(x)$, whose derivative is $2x$. The answer, of course, is $y = x^2$, or more generally, $y = x^2 + C$ for any constant C. Sometimes, we are given an *initial condition* that would allow us to solve for the constant. For instance, if we were told that $y(2) = 7$, then we could plug in to get $C = 3$.

As another example, the integral $y = \int f(x)\,dx$ can be thought of as the differential equation $y' = f(x)$.

Another differential equation is $y' = y$. This is asking for a function which is its own derivative. The solution to this is $y = e^x$, or more generally $y = Ce^x$ for any constant C. There are also differential equations involving higher derivatives, like $y'' - 2y + y = \sin x$.

In a class on differential equations, you learn various techniques for solving particular kinds of differential equations, but you can very quickly get to ones that those techniques can't handle. One of the simplest such equations is $\theta'' = k \sin\theta$, where k is a constant. This equation describes the motion of a pendulum swinging back and forth under the influence of gravity. Considering that a pendulum is a particularly simple physical system, it is not surprising that many of the differential equations describing real-world systems can't be solved with the techniques learned in a differential equations class.

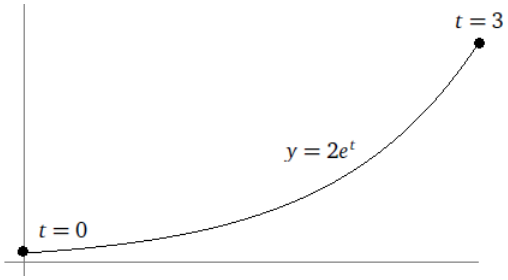
So we need some way of numerically approximating the solutions to differential equations.

Initial value problems

Most of the problems we will be looking at are *initial value problems* (IVPs), specified like in the example below:

$$\begin{cases} y' = y \\ y(0) = 2 \\ t \in [0, 3]. \end{cases}$$

We specify the differential equation, the initial condition(s), and the range of values that we are interested in. We use t for time, thinking of the equation as giving the evolution of something over time. The solution to the IVP of this example is $y = 2e^t$. It is shown below. Notice that the solution curve begins at $(0, 2)$ and runs until $t = 3$.



To start, we will focus on first-order differential equations, those whose highest derivative is y'. We will see later how higher-order differential equations can be reduced to systems of first-order differential equations.

We will also only focus on *ordinary differential equations* (ODEs) as opposed to *partial differential equations* (PDEs), which involve derivatives with respect to multiple variables, such as $\frac{\partial y}{\partial t} + \frac{\partial y}{\partial x} = e^{tx}$.

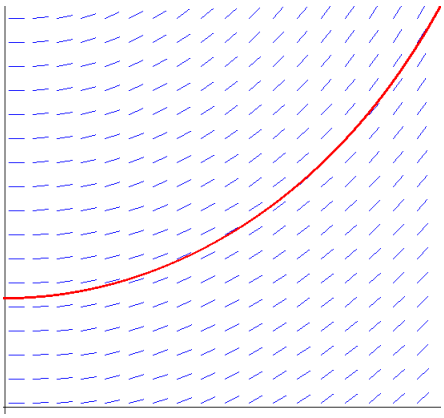
6.1 Euler's method

We will first investigate *Euler's method*. It is a simple method, not always accurate, but understanding it serves as the basis for understanding all of the methods that follow.

Take for example, the following IVP:

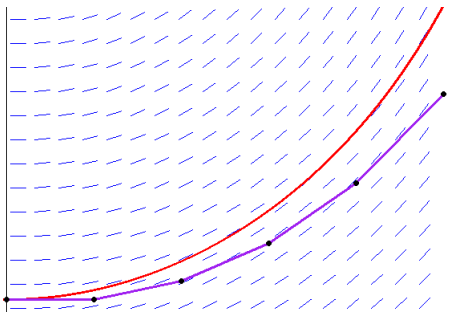
$$\begin{cases} y' = ty^2 + t \\ y(0) = .25 \\ t \in [0, 1]. \end{cases}$$

The equation $y' = ty^2 + t$ can be thought of as giving a slope at each point (t, y). If we plot the slopes at every point, we have a *slope field*, like the one shown below:



Shown in red is the solution to the IVP. Notice how it fits neatly into the slope field. One way to estimate the solution is to start at $(0, .25)$ and step along, continually following the slopes. We first pick a step size, say $h = .2$. We then look at the slope at our starting point $(0, .25)$ and follow a line with that slope for .2 units. The slope is 0 and we end up at $(.2, .25)$. We then compute the slope at this point to get $y' = .41$ and follow a line with that slope for .2 units taking us to the point $(.4, .33)$. We continue this process until we get to $t = 1$, the end of the given interval. Here is the table of values we get and a plot of them along with the actual solution.

t	y
0.0	0.25
0.2	$0.25 + .2(0 \cdot .25^2 + 0) = .25$
0.4	$0.25 + .2(.2 \cdot .25^2 + .2) = .29$
0.6	$0.29 + .2(.4 \cdot .29^2 + .4) = .38$
0.8	$0.38 + .2(.6 \cdot .38^2 + .6) = .51$
1.0	$0.51 + .2(.8 \cdot .51^2 + .8) = .72$



Here is how the method works in general. Assume we have an IVP of the form given below:

$$\begin{cases} y' = f(t,y) \\ y(a) = c \\ t \in [a,b]. \end{cases}$$

Choose a step size h. Set $t_0 = a$, $y_0 = c$, and start at the point (t_0, y_0). We get the next point (t_{n+1}, y_{n+1}) from the previous point by the following:

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + f(t_n, y_n)h.$$

The step size determines how much we move in the t-direction. The change in the y-direction comes from following the slope for h units from the previous point. That slope is given by plugging in the previous point into the differential equation. We continue generating points until we reach the end of the interval.

A short program

Here is a short Python program implementing Euler's method. The method collects all of the points (t_0, y_0), (t_1, y_1), etc. into a list.

```
def euler(f, y_start, t_start, t_end, h):
    t, y = t_start, y_start
    ans = [(t, y)]
    while t < t_end:
        y += h * f(t,y)
        t += h
        ans.append((t,y))
    return ans
```

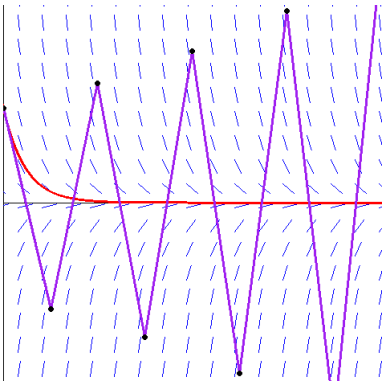
To numerically solve $y' = ty^2 + t$, $y(0) = .25$, $t \in [0,1]$ with $h = .0001$, we could use the following:

```
L = euler(lambda t,y: t*y*y+t, .0001, 0, 1, .25)
```

This stores a rather long list of values in `L`. If we just want the last tuple (t_n, y_n), it is in `L[-1]`.We would use `L[-1][1]` to get just the final y value.[1]

Problems with Euler's method

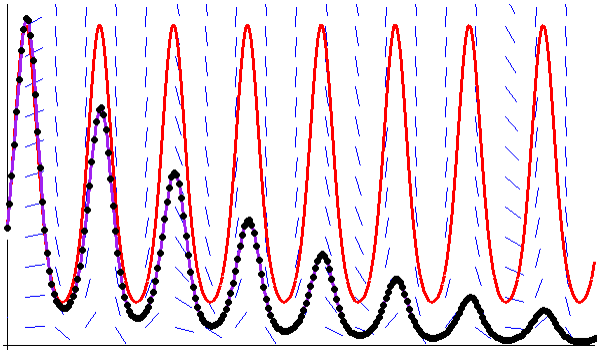
Consider the IVP $y' = -4.25y$, $y(0) = 1$, $t \in [0,4]$. The exact solution to it is $y = e^{-4.25t}$, which seems pretty simple. However, look at what happens with Euler's method with a step size of .5:



[1]Note that for simplicity, the condition on our while loop is `while t < t_end`. However, because of roundoff error, we could end up looping too long. For instance, if our interval is $[0,1]$ and $h = .1$, then because of roundoff error, our last step will take us to .9999..., which will come out less than 1 and we will end up computing one step too many. We can fix this problem by either asking the caller for the number of steps they want to compute instead of h or by computing it ourselves as `n=int(round((t_end-t_start)/h))` and replacing the while loop with `for i in range(len(n))`.

We can see that it goes really wrong. One way to fix this is to use a smaller step size, but there are other differential equations where even a very small step size (like say 10^{-20}) would not be small enough. Using a very small step size can take too long and can lead to error propagation.

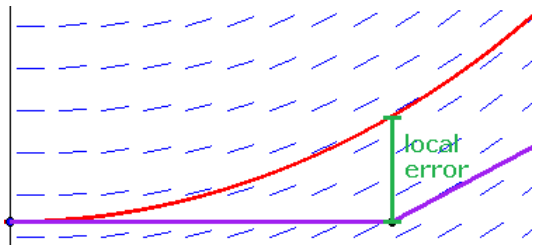
Here is another thing that can go wrong. Consider the IVP $y' = y\cos t$, $y(0) = 10$, $t \in [0, 50]$. Its exact solution is $10e^{\sin t}$. It is shown in red below along with the Euler's method approximation with $h = .2$.



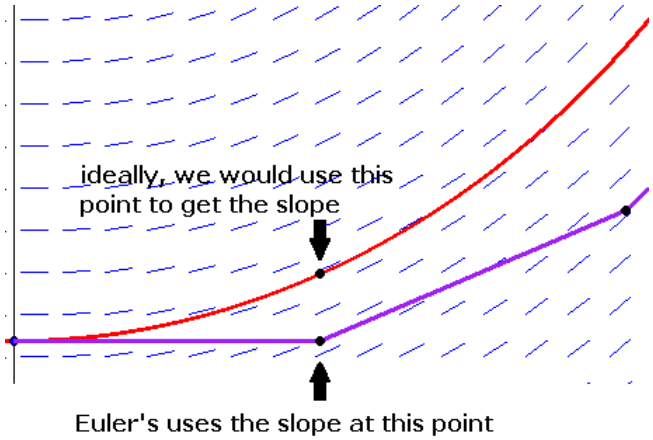
We can see that slowly, but surely, the errors from Euler's method accumulate to cause Euler's method to drift away from the correct solution.

Error in Euler's method

There are two types of error in Euler's method: *local error* and *global error*. Local error is the error that comes from performing one step of Euler's method. From the point (t_0, y_0), we follow the slope at that point for h units to get to (t_1, y_1). But the problem with that is that the slope changes all throughout the range from t_0 to t_1, and Euler's method ignores all of that. See the figure below:



The second type of error is global error. Global error is the error we have after performing several steps of Euler's method. It is the sum of the individual local errors from each step plus another factor: The accumulated local error from the previous steps means that, at the current step, we are some distance away from the actual solution. The slope there is different from the slope at the actual solution, which is the slope we should ideally be following to get to the next point. Following the wrong slope can lead us further from the actual solution. See the figure below:



We can work out formulas for the local and global error by using Taylor series. We start at the point (t_0, y_0) and advance to (t_1, y_1). We have the following, where $y(t)$ is the Euler's method approximation and $\tilde{y}(t)$ is the exact solution:

$$\begin{aligned} t_1 &= t_0 + h \\ y_1 &= y_0 + hf(t_0, w_0) \\ \tilde{y}_1 &= \tilde{y}(t_0 + h) = \tilde{y}(t_0) + h\tilde{y}'(t_0) + \frac{h^2}{2!}\tilde{y}''(t_0) + \ldots. \end{aligned}$$

But $\tilde{y}(t_0)$, the initial condition, is exactly equal to y_0, and $\tilde{y}'(t_0)$ is exactly equal to $f(y_0, t_0)$. So from this we get

$$\tilde{y}_1 - y_1 = \frac{h^2}{2!}\tilde{y}''(t_0) + \ldots.$$

So the local error is on the order of h^2.

However, we perform $(b-a)/h$ total steps, and summing up the local errors from each of these steps (and doing some analysis for the other part of global error, which we will omit), we end up with a global error term on the order of h. As usual, the power of h is the most important part; it is the *order* of the method. Euler's method is a first order method.

It can be shown that the global error at (t_i, y_i) is bounded by the following

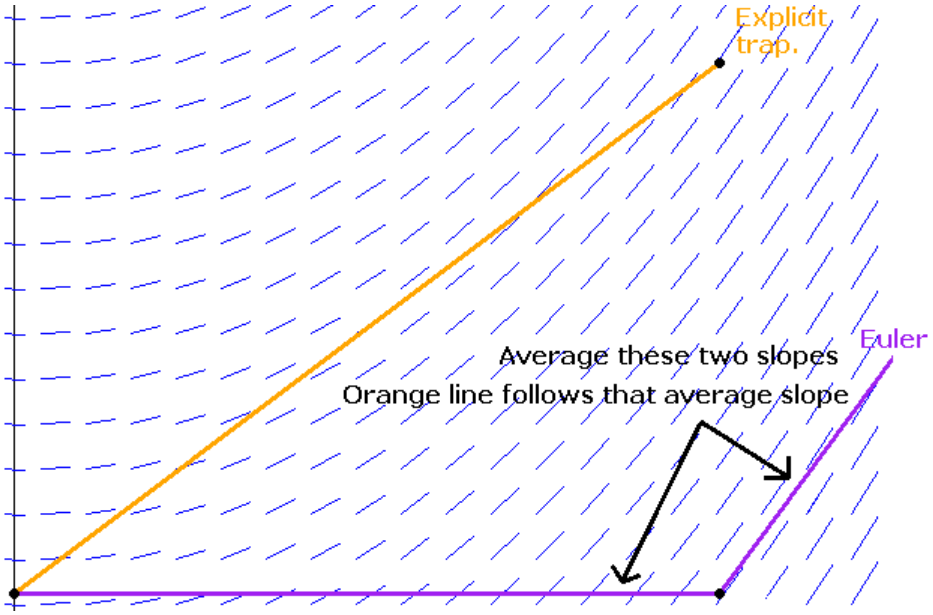
$$\frac{Ch}{L}\left(e^{L(t_i - a)} - 1\right),$$

where C and L are constants.[1] We notice, in particular, the exponential term, which indicates that the error can eventually grow quite large.

6.2 Explicit trapezoid method

One problem with Euler's method is that when going from one point to the next, it uses only the slope at the start to determine where to go and disregards all the slopes along the way. An improvement we can make to this is, when we get to the next point, we take the slope there and average it with the original slope. We then follow that average slope from the first point instead of the original slope. See the figure below:

[1] L is called a Lipschitz constant. It satisfies $|f(t,u) - f(t,v)| \leq L|u - v|$ for all points (t,u) and (t,v) in the domain.



Here is a formal description of the method: Assume we have an IVP of the form given below:

$$\begin{cases} y' = f(t,y) \\ y(a) = c \\ t \in [a,b]. \end{cases}$$

Choose a step size h. Set $t_0 = a$, $y_0 = c$ and start at the point (t_0, y_0). We get each point (t_{n+1}, y_{n+1}) from the previous point by the following:

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_n, y_n) \\ s_2 &= f(t_{n+1}, y_n + hs_1) \\ y_{n+1} &= y_n + h\frac{s_1+s_2}{2} \end{aligned}$$

In the formulas above, s_1 is the slope that Euler's method uses, $(t_n, y_n + hs_1)$ is the point that Euler's method wants to take us to, and s_2 is the slope at that point. We get y_{n+1} by following the average of the slopes s_1 and s_2 for h units from y_n.

This method is called either the *explicit trapezoid method*, *Heun's method*, or the *improved Euler method*. Here is how we might code it in Python.

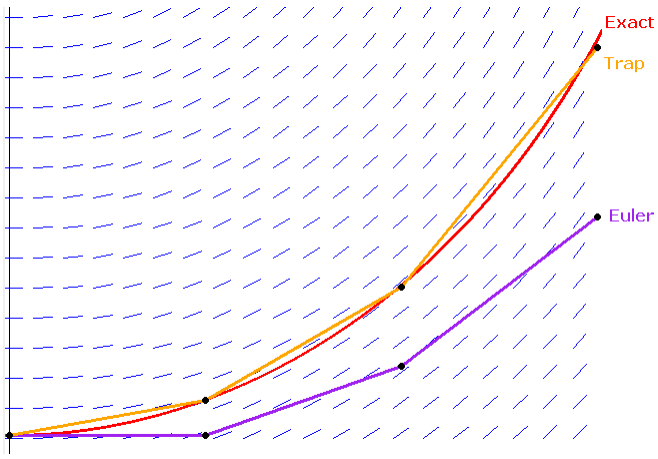
```
def explicit_trap(f, y_start, t_start, t_end, h):
    t, y = t_start, y_start
    ans = [(t, y)]
    while t < t_end:
        s1 = f(t, y)
        s2 = f(t+h, y+h*s1)
        y += h*(s1+s2)/2
        t += h
        ans.append((t,y))
    return ans
```

Example

Consider the IVP $y' = ty^2 + t$, $y(0) = .25$, $t \in [0,1]$. Let's try the explicit trapezoid method with $h = .2$. Here is a table showing the calculations (all rounded to two decimal places):

t_n	s_1	Euler's prediction	s_2	y_n
0	—	—	—	.25
.2	$f(0,1)=0$	$.25+(.2)(0)=.25$	$f(.2,.25)=.21$	$.25+.2\frac{0+.21}{2}=.27$
.4	$f(.2,.27)=.21$	$.27+(.2)(.21)=.31$	$f(.4,.31)=.44$	$.27+.2\frac{.21+.44}{2}=.34$
.6	$f(.4,.34)=.45$	$.34+(.2)(.45)=.43$	$f(.6,.43)=.71$	$.34+.2\frac{.45+.71}{2}=.45$
.8	$f(.6,.45)=.72$	$.45+(.2)(.72)=.60$	$f(.8,.60)=1.08$	$.45+.2\frac{.72+1.08}{2}=.63$
1	$f(.8,.63)=1.12$	$.63+(.2)(1.12)=.86$	$f(1,.86)=1.73$	$.63+.2\frac{1.12+1.73}{2}=.92$

Here are the above points graphed along with Euler's method and the exact solution:



The explicit trapezoid method can be shown using Taylor series to be a second order method.

The explicit trapezoid rule gets its name from its relation to the trapezoid rule for integrals. If our differential equation $y' = f(t,y)$ has no dependence on y, i.e., $y' = f(t)$, then our differential equation reduces to an ordinary integration problem. In this case, the slopes in the explicit trapezoid method reduce to $s_1 = f(t_n)$, $s_2 = f(t_{n+1})$, and the trapezoid rule formula becomes

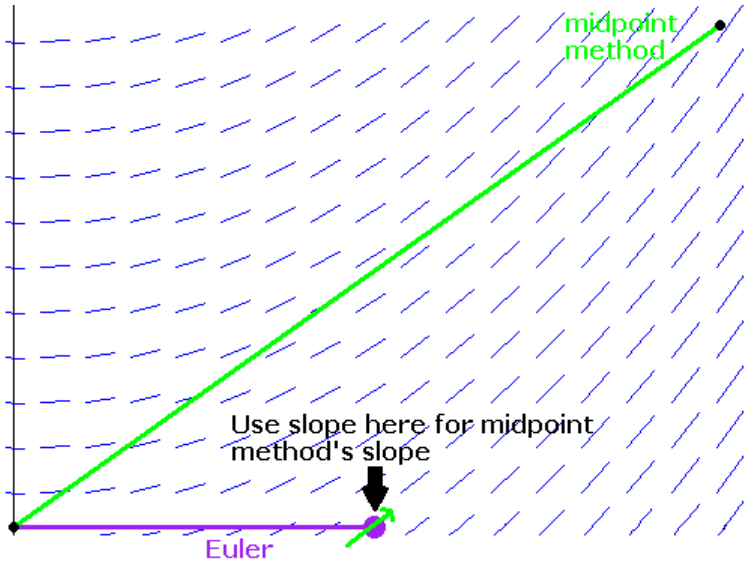
$$y_{n+1} = y_n + h\frac{f(t_n)+f(t_{n+1})}{2}.$$

This is exactly equivalent to the summing of areas of trapezoids, which is what the trapezoid rule for integration does. The result of running the method is essentially a table of approximate values of the antiderivative of f at the values t_0, t_0+h, t_0+2h, etc. achieved from running the trapezoid rule, stopping at each of those points.

Note that a similar calculation shows that Euler's method on $y' = f(t)$ reduces to numerical integration using left-endpoint rectangles.

6.3 The midpoint method

The midpoint method builds off of Euler's method in a similar way to the explicit trapezoid method. From the starting point, we go halfway to the next point that Euler's method would generate, find the slope there, and use that as the slope over the whole interval. See the figure below:



Here is a formal description of the method: Assume we have an IVP of the form given below:

$$\begin{cases} y' = f(t,y) \\ y(a) = c \\ t \in [a,b]. \end{cases}$$

Choose a step size h. Set $t_0 = a$, $y_0 = c$ and start at the point (t_0, y_0). We get each next point (t_{n+1}, y_{n+1}) from the previous point by the following:

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_n, y_n) \\ s_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}s_1) \\ y_{n+1} &= y_n + hs_2 \end{aligned}$$

In the formulas above, s_1 is the slope that Euler's method wants us to follow. We do temporarily follow that slope for $h/2$ units and compute the slope at that point, which we call s_2. Then we start back at (t_n, y_n) and slope s_2 for h units to get our new point (t_{n+1}, y_{n+1}). Note that we could also write y_{n+1} more compactly as follows

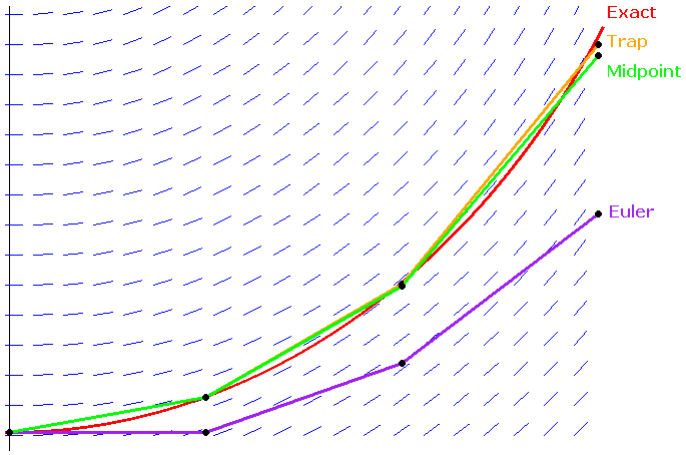
$$y_{n+1} = y_n + hf(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}f(t_n, y_n)).$$

Example

Suppose we have the IVP $y' = ty^2 + t$, $y(0) = .25$, $t \in [0,1]$. Here are the calculations for the midpoint method using $h = .2$ (all rounded to two decimal places):

t_n	s_1	$y + \frac{h}{2}s_1$	s_2	y_n
0	—	—	—	.25
.2	$f(0,.25) = 0$	$.25 + (.1)(0) = .25$	$f(.1,.25) = .11$	$.25 + (.2)(.10) = .27$
.4	$f(.2,.27) = .21$	$.27 + (.1)(.21) = .29$	$f(.3,.29) = .33$	$.27 + (.2)(.33) = .34$
.6	$f(.4,.34) = .45$	$.34 + (.1)(.45) = .38$	$f(.5,.38) = .57$	$.34 + (.2)(.57) = .45$
.8	$f(.6,.45) = .72$	$.45 + (.1)(.72) = .52$	$f(.7,.52) = .89$	$.45 + (.2)(.89) = .63$
1.0	$f(.8,.63) = 1.12$	$.63 + (.1)(1.12) = .74$	$f(.9,.74) = 1.39$	$.63 + (.2)(1.39) = .91$

Here are the three methods we have learned thus far plotted together with the exact solution:



The midpoint method is an order 2 method.

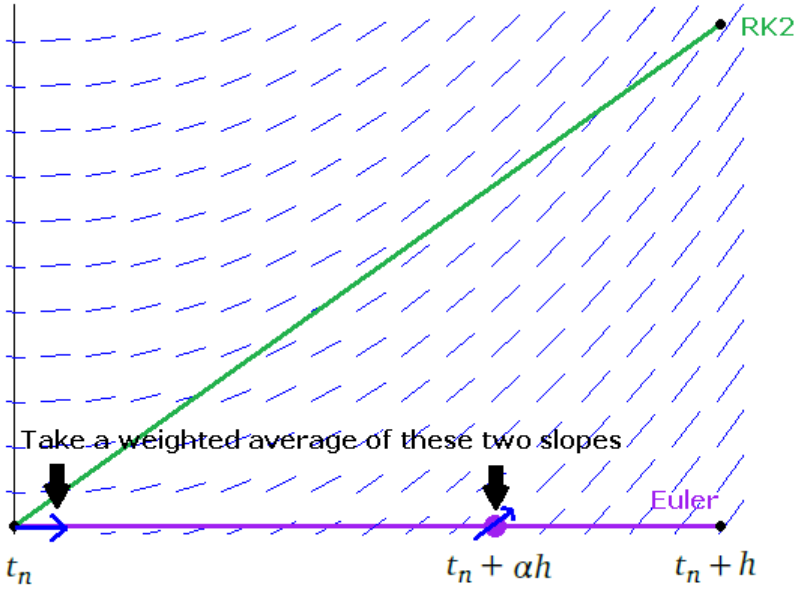
6.4 Runge-Kutta methods

Euler's method, the explicit trapezoid method, and the midpoint method are all part of a family of methods called *Runge-Kutta* methods. There are Runge-Kutta methods of all orders.

The order 1 method Runge-Kutta method is Euler's method. There is an infinite family of order 2 methods of the following form:

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_n, y_n) \\ s_2 &= f(t_n + \alpha h, y_n + \alpha h s_1) \\ y_{n+1} &= y_n + h\frac{(2\alpha - 1)s_1 + s_2}{2\alpha} \end{aligned}$$

Geometrically, these methods can be described as following the Euler's method prediction α percent of the way from t_n to t_{n+1}, computing the slope there, taking a weighted average of that slope and the slope at (t_n, y_n), and following that average slope from (t_n, y_n) for h units. See the figure below:



Each value of α gives a different method. For instance, $\alpha = 1$ leads to the explicit trapezoid method and $\alpha = 1/2$ leads to the midpoint method. The most efficient method (in terms of the coefficient on the error term) turns out to have $\alpha = 2/3$. It's formula works out to the following:

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_n, y_n) \\ s_2 &= f(t_n + \tfrac{2}{3}h, y_n + \tfrac{2}{3}hs_1) \\ y_{n+1} &= y_n + h\frac{s_1 + 3s_2}{4} \end{aligned}$$

RK4

By far the most used Runge-Kutta method (and possibly the most used numerical method for differential equations) is the so-called RK4 method (the 4 is for fourth order). It is relatively accurate and easy to code.

Here is a formal description of the method: Assume we have an IVP of the form given below:

$$\begin{cases} y' = f(t, y) \\ y(a) = c \\ t \in [a, b]. \end{cases}$$

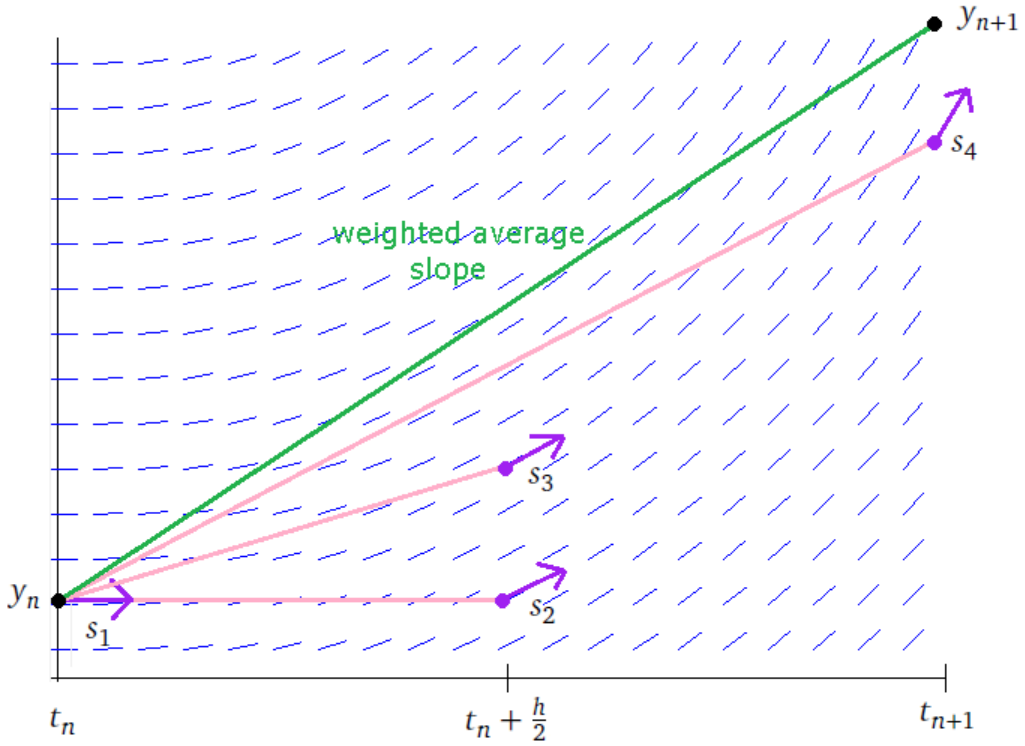
Choose a step size h. Set $t_0 = a$, $y_0 = c$ and start at the point (t_0, y_0). We get each next point (t_{n+1}, y_{n+1}) from the previous point by the following:

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_n, y_n) \\ s_2 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}s_1) \\ s_3 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}s_2) \\ s_4 &= f(t_n + h, y_n + hs_3) \\ y_{n+1} &= y_n + h\frac{s_1 + 2s_2 + 2s_3 + s_4}{6} \end{aligned}$$

The RK4 method builds off of the RK2 methods (trapezoid, midpoint, etc.) by using four slopes instead of two.

- s_1 — Euler's slope at the left endpoint
- s_2 — Slope at midpoint from following Euler's slope
- s_3 — Improved slope at midpoint, from following s_2 instead of s_1
- s_4 — Slope at right endpoint, from following s_3

The new y_{n+1} is computed as a weighted average those four slopes. See the figure below:



To summarize, we start by following the slope at the left endpoint, s_1, over to the midpoint, getting the slope s_2 there. This is just like the midpoint method. But then we follow a slope of s_2 from the left endpoint to the midpoint and compute the slope, s_3, there. This is like adding an additional correction on the original midpoint slope. We then follow a slope of s_3 all the way over to the right endpoint, and compute the slope, s_4, at that point. Finally, we take a weighted average of the four slopes, weighting the midpoint slopes more heavily than the endpoint slopes, and follow that average slope from the left endpoint to the right endpoint to get the new y-value, y_{n+1}.

Here is a Python program implementing the RK4 method:

```
def rk4(f, y_start, t_start, t_end, h):
    t, y = t_start, y_start
    ans = [(t, y)]
    while t < t_end:
        s1 = f(t, y)
        s2 = f(t+h/2, y+h/2*s1)
        s3 = f(t+h/2, y+h/2*s2)
        s4 = f(t+h, y+h*s3)
        y += h * (s1+2*s2+2*s3+s4)/6
        t += h
        ans.append((t,y))
    return ans
```

The RK4 method is developed from Taylor series by choosing the various coefficients in such a way as to cause all the terms of order 4 or less to drop out. The algebra involved, however, is tedious.

There are higher order Runge-Kutta methods, but RK4 is the most efficient in terms of order versus number of function evaluations. It requires 4 function evaluations and its order is 4. The number of function evaluations needed for Runge-Kutta methods grows more quickly than does the order. For instance, an order 8 Runge-Kutta method requires at least 11 function evaluations. As always, the speed of a method depends on both its order (how the error term behaves with respect to the step size) and the number of function evaluations (which can be very slow for some complicated functions).

When applied to the ODE $y' = f(t)$ (an integration problem), RK4 reduces to Simpson's rule.

6.5 Systems of ODEs

Many times we will have a system of several differential equations. For example, a double pendulum (pendulum attached to a pendulum) is a simple example of a physical system whose motion is described by a system. Each pendulum has its own differential equation, and the equations are *coupled*; that is, each pendulum's motion depends on what the other one is doing. As another example, we might have two populations, rabbits and foxes. Each population is governed by a differential equation, and the equations depend on each other; that is, the number of foxes depends on how many rabbits there are, and vice-versa.

To numerically solve a system, we just apply the methods we've learned to each equation separately. Below is a simple system of ODEs.

$$\begin{cases} x' = 2 - xy + y^2 \\ y' = 3 - xy + x^2 \\ x(0) = 1 \\ y(0) = 6 \\ t \in [0,1]. \end{cases}$$

Just to see how things work, let's numerically solve it using Euler's method with $h = .5$:

t_n	x_n	y_n
0.0	1	6
0.5	$1 + .5\left(2-(1)(6)+6^2\right) = 17$	$4 + .5\left(3-(1)(6)+1^2\right) = 3$
1.0	$8 + .5\left(2-(17)(3)+3^2\right) = -12$	$4 + .5\left(3-(17)(3)+17^2\right) = 124.5$

Here is a concise way to implement Euler's method for a system of equations in Python:

```
def euler_system(F, Y_start, t_start, t_end, h):
    t, Y = t_start, Y_start
    ans = [(t_start, Y)]
    while t < t_end:
        Y = [y+h*f(t,*Y) for y, f in zip(Y,F)]
        t+=h
        ans.append((t,Y))
    return ans
```

Here `F` is a list of functions and `Y_start` is a list of initial conditions. Notice the similarity to the earlier Euler's method program.[1]

Here is how we might call this function for the example above:

```
L = system([lambda  t,x,y: 2-x*y+y*y, lambda t,x,y:3-x*y+x*x], [1,6], 0, 1, .5))
```

We need a little more care to apply some of the other methods. For instance, suppose we want to apply the midpoint rule to a system of two ODEs, $x' = f(t,x,y)$ and $y' = g(t,x,y)$. Here is what the midpoint rule would be for this system:

$$\begin{array}{ll} t_{n+1} = t_n + h & \\ r_1 = f(t_n, x_n, y_n) & s_1 = g(t_n, x_n, y_n) \\ r_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_1, y_n + \frac{h}{2}s_1) & s_2 = g(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_1, y_n + \frac{h}{2}s_1) \\ x_{n+1} = x_n + hr_2 & y_{n+1} = y_n + hs_2 \end{array}$$

The key is that not only do we have two sets of things to compute, but since the equations depend on each other, each computation needs to make use of the intermediate steps from the other. For instance, r_2 relies on both r_1 and s_1. Let's try this with the IVP $x' = tx + y^2$, $y' = xy$, $x(0) = 3$, $y(0) = 4$, $t \in [0,1]$. For simplicity, we'll take just one step with $h = 1$. Here are the computations:

[1] The program uses a few somewhat advanced features, including the $*$ operator for turning a list into function arguments and `zip` to tie together two lists.

x	y
$s_1 = (0 \cdot 3 + 4^2) = 16$	$r_1 = (3 \cdot 4) = 12$
$s_2 = f(.5, 3 + 16 \cdot .5, 4 + 12 \cdot .5) = .5 \cdot 11 + 10^2 = 105.5$	$g(.5, 3 + 16 * .5, 4 + 12 * .5) = 11 \cdot 10 = 110$
$x_1 = 3 + 105.5 \cdot 1 = 108.5$	$y_1 = 4 + 110 \cdot 1 = 114$

Here is how we could code this in Python.

```
def system_mid(F, Y_start, t_start, t_end, h):
    t, Y = t_start, Y_start
    ans = [(t_start, Y)]
    while t < t_end:
        S1 = [f(t, *Y) for f in F]
        Z = [y+h/2*s1 for y,s1 in zip(Y,S1)]
        S2 = [f(t+h/2, *Z) for f in F]
        Y = [y+h*s2 for y,s2 in zip(Y,S2)]
        t+=h
        ans.append((t,Y))
    return ans
```

This code can easily be extended to implement the RK4 method.

Higher order equations

Higher order equations are handled by converting them to a system of first-order equations. For instance, suppose we have the system $y'' = y + t$. Let's introduce a variable $v = y'$. We can write the ODE as the system $y' = v$, $v' = y + t$. We can then use the methods from the previous section to estimate a solution. Note that since our system is second-order, we will have two initial conditions, one for y and one for y'.

Here is another example. Suppose we have the IVP $y''' = 3y'' + 4ty - t^2$, $y(0) = 1$, $y'(0) = 3$, $y''(0) = 6$, $t \in [0, 4]$. We can write it as a system of first order equations using the $v = y'$ and $a = y''$. We have chosen the symbols v and a to stand for velocity and acceleration.[1] This gives us the system $y' = v$, $v' = a$, $a' = 3a + 4ty - t^2$. Let's use Euler's method with $h = 2$ to approximate the solution:

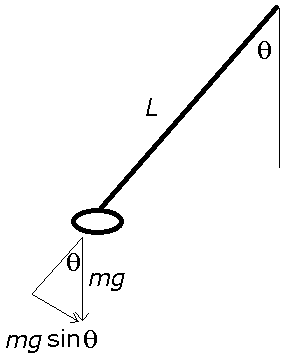
t_n	y	v	a
0.0	1	3	6
2.0	$1 + 2(3) = 7$	$3 + 2(6) = 15$	$6 + 2\big((3)(6) + (4)(1) - 0^2\big) = 46$
4.0	$7 + 2(15) = 37$	$15 + 2(46) = 107$	$46 + 2\big((3)(46) + (4)(7) - 2^2\big) = 368$

Simulating physical systems

The motion of an object can be described by Newton's second law, $F = ma$, where F is the sum of forces on the object, m is its mass, and a is the object's acceleration. Acceleration is the second derivative of position, so Newton's second law is actually a differential equation. The key is to identify all the forces on the object. Once we've done this, we can write Newton's second law as the system $y' = v$, $v' = F/m$.

For example, consider a simple pendulum:

[1] Note also that there are other occasionally useful ways to write a higher order equation as a system, but this way is the most straightforward.



We have to decide on some variables to represent the system. The simplest approach is to use θ, the angle of the pendulum, with $\theta = 0$ corresponding to the pendulum hanging straight down.

Gravity pulls downward on the pendulum with a force of mg, where $g \approx 9.8$. With a little trigonometry, we can work out that the component of this force in the direction of the pendulum's motion is $-m\frac{g}{L}\sin\theta$. This gives us

$$\theta'' = -\frac{g}{L}\sin\theta.$$

This equation cannot easily be solved by standard analytic means. We can make it even harder to analytically solve by adding a bit of friction to the system as follows:

$$\theta'' = -\frac{g}{L}\sin\theta - c\theta',$$

where c is some constant. Despite the difficulty we have in solving it analytically, we can easily find an approximate solution. Here is a Python program that uses Euler's method to numerically estimate a solution and sketches the resulting motion of the pendulum:

```
from math import sin, cos
from tkinter import *

def plot():
    y = 3
    v = 1
    h = .0005
    while True:
        v, y = v + h*f(y,v), y + h*v
        a = 100*sin(y)
        b = 100*cos(y)
        canvas.coords(line, 200, 200, 200+a, 200+b)
        canvas.coords(bob, 200+a-10, 200+b-10, 200+a+10, 200+b+10)
        canvas.update()

f = lambda y, v: -9.8/1*sin(y)-v/10
root = Tk()
canvas = Canvas(width=400, height=400, bg='white')
canvas.grid()
line = canvas.create_line(0, 0, 0, 0, fill='black')
bob = canvas.create_oval(0, 0, 0, 0, fill='black')
plot()
```

This program is very minimalistic, with absolutely no bells and whistles, but it fairly accurately simulates the motion of a pendulum. One improvement would be to only plot the pendulum when it has moved a noticeable amount, such as when `a` or `b` changes by at least one pixel. This would give a considerable speedup. We could also allow the user to change some of the constants, like the length of the pendulum or the damping factor. And a more accurate result can be obtained by using the RK4 method. We can do this by replacing the first line of the while loop with the following code:

```
s1 = v
t1 = f(y,v)
```

```
s2 = v+h/2*s1
t2 = f(y+h/2*s1, v+h/2*t1)

s3 = v+h/2*s2
t3 = f(y+h/2*s2, v+h/2*t2)

s4 = v+h*s3
t4 = f(y+h*s3, v+h*t3)

v = v + h*(t1+2*t2+2*t3+t4)/6
y = y + h*(s1+2*s2+2*s3+s4)/6
```

Many other physical systems can be simulated in this way, by identifying all the forces on an object to find the equation of motion, then numerically solving that equation and plotting the results.

6.6 Multistep and implicit methods

A multistep method is one that, instead of just working off of (t_n, y_n), uses earlier points, like (t_{n-1}, y_{n-1}), (t_{n-2}, y_{n-2}), etc. For example, here is a second-order method called the Adams-Bashforth two-step method (AB2):

$$\begin{aligned} t_{n+1} &= t_n + h \\ s_1 &= f(t_{n-1}, y_{n-1}) \\ s_2 &= f(t_n, y_n) \\ y_{n+1} &= y_n + h\frac{3s_2 - s_1}{2} \end{aligned}$$

The method uses a kind of weighted average of the slopes at (t_n, y_n) and (t_{n-1}, y_{n-1}). *Average* isn't really the best term to use here. It's more like we are primarily relying on the slope at (t_n, y_n) and correcting it slightly with the slope at (t_{n-1}, y_{n-1}).[1]

AB2 is an order 2 method. The good thing about this method is that we compute $f(t_n, y_n)$ to get y_{n+1} and we use it again when we compute y_{n+2}. So at each step, we only have one new function evaluation. This is in contrast two the other order 2 methods we've seen thus far, like the midpoint and explicit trapezoid methods, that require two new function evaluations at each step. On the other hand, AB2 is relying on a slope that is farther from the current point than those that the midpoint and trapezoid methods use, so it is not quite as accurate.[2]

One thing to note about AB2 is that it requires multiple values to get started, namely (t_0, y_0), and (t_1, y_1). The first point is the initial condition, and we can get the second point using something else, like the midpoint method. Here is the Adams-Bashforth two-step method coded in Python:

```
def AB2(f, y_start, t_start, t_end, h):
    t, y = t_start, y_start
    s2 = f(t, y)
    y += h*f(t+h/2, y+h/2*f(t,y))  # midpoint method for y_1
    t += h
    ans = [(t_start,y_start), (t,y)]

    while t < t_end:
        s1 = s2
        s2 = f(t, y)
        y += h * (3*s2-s1)/2
        t += h
        ans.append((t,y))
    return ans
```

[1] We could use other weightings, like $\frac{5s_2-2s_1}{3}$ or $\frac{3s_2+s_1}{4}$, but the Adams-Bashforth method's weightings have been chosen to give the best possible error term using the two slopes.

[2] However, because it is faster, we can use a smaller time step and get a comparable accuracy to those other methods.

Another example of a multistep method is the fourth-order Adams-Bashforth four-step method:

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + h\frac{55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}}{24}.$$

We have used the shorthand $f_i = f(t_i, y_i)$. Note that only one new function evaluation is required at each step.

6.7 Implicit methods and stiff equations

An interesting alternative way to derive Euler's method is to approximate y' with the forward difference formula $y'(t) = \frac{y(t+h)-y(t)}{h}$. Solving this for $y(t+h)$ gives $y(t+h) = y(t) + hy'(t)$ or

$$y_{n+1} = y_n + hf(t, y).$$

Suppose instead we use the backward difference formula $y'(t) = \frac{y(t)-y(t-h)}{h}$. Solve it to get $y(t) = y(t-h)+hy'(t)$ and take $t = t_{n+1}$. This leads to the following method:

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

This is called the *backward Euler method*, and it might seem a little bizarre. We are using the slope at very point that we are trying to find, even though we haven't found that point yet. A better way to look at this is that it gives us an equation that we can solve for y_{n+1}. Chances are that the equation will be difficult to solve numerically, so a numerical root-finding method would be necessary. For instance, if we have $f(t, y) = (t+1)e^y$, $y(0) = 3$, and $h = .1$, to get y_1 we would have to solve the equation $y_1 = 3 + .1(0+1)e^{y_1}$, something we can't solve algebraically.

Here is how we might code the method in Python, relying on the secant method to numerically solve the equation that comes up:

```
def backward_euler(f, y_start, t_start, t_end, h):
    t, y = t_start, y_start
    ans = [(t, y)]
    while t < t_end:
        g = lambda x:y + h*f(t+h,x)-x
        a, b = y,  y + h*f(t+h/2, y+h/2*f(t,y))
        while g(a)!=g(b) and abs(b-a)>.00000001:
            a, b = b, b - g(b)*(b-a)/(g(b)-g(a))
        y = b
        t += h
        ans.append((t, y))
    return ans
```

The bulk of the while loop is spent numerically solving for y_{n+1}. We start by setting g equal to the function we need to solve for y_{n+1}, with x standing for y_{n+1}. Our two initial guesses to get the secant method started are y_0 and the midpoint method's guess for y_1. After the inner while loop finishes, b holds the secant method's estimate for y_{n+1}.

Methods like this, where the expression for y_{n+1} involves y_{n+1} itself, are called *implicit methods*.[1] The backward Euler method is an order 1 implicit method. The *implicit trapezoid method*, given below is an order 2 method:

$$t_{n+1} = t_n + h$$
$$s_1 = f(t_n, y_n)$$
$$s_2 = f(t_{n+1}, y_{n+1})$$
$$y_{n+1} = y_n + h\frac{s_1 + s_2}{2}.$$

[1] In math, if we have a formula for a variable y in terms of other variables, we say that it is given explicitly. But if that variable is given by an equation like $x^2 + y^2 = \sin y$, we say that y is given implicitly. There is still an expression (even if we can't find it) for y in terms of x that is determined by the equation, but that expression is implied rather than explicitly given. Recall also the term *implicit differentiation* from calculus, where we are given an expression like $x^2 + y^2 = \sin y$ that we cannot solve for y, yet we can still find its derivative.

Notice that s_2 depends on y_{n+1} itself.

There are a variety of other implicit methods. Many of them can be generated by numerical integration rules. For instance, integrating

$$\int_{t_n}^{t_{n+1}} f(t,y)\,dt$$

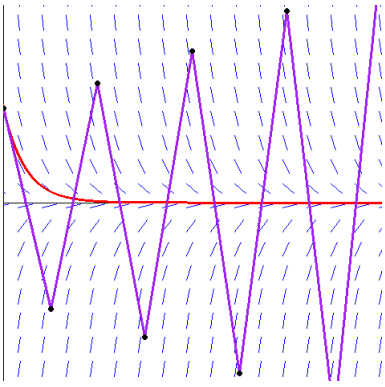
using the trapezoid rule for integration gives the implicit trapezoid method. If we use Simpson's rule to do the integration, we get $y_{n+1} = y_n + \frac{f_{n+1}+4f_n+f_{n-1}}{3}$, a fourth-order method called the Milne-Simpson method. There are also some useful implicit methods in the Runge-Kutta family, though we won't cover them here.

Stiff equations

Implicit methods are particularly useful for solving what are called *stiff equations*. It is difficult to give a precise definition of stiff equations, but we have seen an example of one before:

$$\begin{cases} y' = -4.25y \\ y(0) = 1 \\ t \in [0,4]. \end{cases}$$

Its exact solution is $y = e^{-4.25t}$. Shown below is the result of applying Euler's method with a step size of $h = .5$:



The equation has a nice, simple solution approaching 0, but the large slopes near the solution are enough to throw off Euler's method unless a small h is used. If we jump too far from the solution, we see that we end up in regions of high slope which have the effect of pushing us farther away from the solution. There are much worse examples, where the only way Euler's method and the other Runge-Kutta methods can work is with a very small value of h. The problem with using such a small value of h is the that the method may take too long to complete, and with so many steps, error propagation becomes a problem.

Stiff equations show up in a variety of real-life problems, so it is important to have methods that can handle them. Generally, implicit methods tend to perform better than explicit methods on stiff equations. An implicit method has a lot more work to do at each step since it has to numerically find a root, but it can usually get away with a much larger time step than an explicit method of the same order.

One way to think of a stiff equation is to imagine that we are walking on a precipice with sheer drops on either side of us. If we are an explicit method, we need to tip-toe (that is use really small time-steps) to avoid losing our balance and falling off. An implicit method, however, is like walking along the precipice with a guide rope. It takes some time to set that rope up, but it allows us to take much bigger steps.

Taking a look at the slope field above, we see the steep slopes on either side of the equation, sitting there, just waiting to lead an unwary explicit method off into the middle of nowhere.

Implicit methods tend to be more *stable* than explicit methods. Stability is the property that a small change in the initial conditions leads to a small change in the resulting solution (with an unstable method, a small change in

the initial conditions might cause a wild change in the solution). Numerical analysis books devote a lot of space to studying the stability of various methods.

Implicit methods can certainly be used to numerically solve non-stiff equations, but explicit methods are usually sufficiently accurate for non-stiff equations and run more quickly since they don't have the added burden of having to perform a root-finding method at each step.

6.8 Predictor-corrector methods

Recall that in the backwards Euler method, we compute y_{n+1} by

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

We then find y_{n+1} with a root-finding method. Here is a different approach: Suppose we use the regular Euler method to generate an approximate value for y_{n+1} and use that in the backwards Euler formula. We would have the following method (where $\tilde{y}_{n+1}$ represents the Euler's method prediction):

$$\begin{aligned} t_{n+1} &= t_n + h \\ \tilde{y}_{n+1} &= y_n + hf(t_n, y_n) \\ y_{n+1} &= y_n + hf(t_{n+1}, \tilde{y}_{n+1}). \end{aligned}$$

The idea is that Euler's method *predicts* y_{n+1} and the backward Euler method *corrects* the prediction. This is a simple example of a *predictor-corrector* method. Predictor-corrector methods are explicit methods. Note the similarity of this method to the explicit trapezoid rule.

A popular predictor-corrector method uses the Adams-Bashforth four-step method along with the following order 4 implicit method, called the Adams-Moulton three-step method:

$$\begin{aligned} t_{n+1} &= t_n + h \\ y_{n+1} &= y_n + h\frac{9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}}{24}. \end{aligned}$$

We have used the shorthand $f_i = f(t_i, y_i)$. Here is the combined method:

$$\begin{aligned} t_{n+1} &= t_n + h \\ \tilde{y}_{n+1} &= y_n + h\frac{55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}}{24} \\ y_{n+1} &= y_n + h\frac{9\tilde{f}_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}}{24}. \end{aligned}$$

Here $\tilde{f}_{n+1} = f(t_{n+1}, \tilde{y}_{n+1})$. This method requires four values to get started. The first is the initial value, and the next three can be generated by RK4 or another explicit method.

6.9 Variable step-size methods

Recall how adaptive quadrature works: We compute a trapezoid rule estimate over the whole interval, then break up the interval into halves and compute the trapezoid estimates on those halves. The difference between the big trapezoid estimate and the sum of the two halves gives a decent estimate of the error. If that error is less than some desired tolerance, then we are good on that interval, and if not, then we apply the same procedure to both halves of the interval.

This gives us a process that allows the step size (trapezoid width) to vary, using smaller step sizes in tricky regions and larger step sizes in simple regions. We can develop a similar adaptive procedure for differential equations. The analogous approach would be to try a method with step size h, then with step size $h/2$ in two parts, and compare the results. However, a slightly different approach is usually taken.

We run two methods simultaneously, one of a higher order than the other, say Euler's method and the midpoint method. Suppose Euler's method gives us an estimate E_{n+1} and the midpoint rule gives us an estimate M_{n+1}. We can use $e = |M_{n+1} - E_{n+1}|$ as an estimate of our error, namely, how far apart E_{n+1} is from the true solution. We have some predetermined tolerance, like maybe $T = .0001$, that we want the error to be less than. We do the following:

- If $e > T$, then our error is unacceptably large, so we cut h in half and try again on the same interval.
- If the error is very small, say $e < \frac{T}{10}$, then our estimate is very good, so we can probably get away with twice as large a step size on the next interval.
- Otherwise (if $\frac{T}{10} < e < T$), then our estimate is okay, and we keep the same step size on the next interval.

Here's how we might code this:

```
def simple_adaptive(f, y_start, t_start, t_end, h, T=.0001):
    t, y = t_start, y_start
    ans = [(t, y)]

    while t < t_end:
        s1 = f(t, y)
        s2 = f(t+h/2, y+h/2*s1)

        euler = y + s1*h
        midpt = y + s2*h
        e = abs(midpt-euler)

        # estimate is no good, decrease step size and try this step over
        if e > T:
            h /= 2

        # estimate is very good, increase step size
        elif e < T/10:
            y = midpt
            t += h
            ans.append((t,y))
            h *= 2

        # estimate is acceptable, keep same step size for next step
        else:
            y = midpt
            t += h
            ans.append((t,y))

    return ans
```

In practice, there are several improvements that people make to this method. First, higher order methods are used. Since two methods are required, in order to keep the number of function evaluations down, it is desirable to have two methods that share as many slopes as possible. The most popular[1] pair of methods is a fourth-order and fifth-order pair whose members share the same set of six slopes. This is known as the Runge-Kutta-Fehlberg

[1] Though *Numerical Methods* recommends a slightly different pair called Cash-Karp.

method (RKF45 for short). Here is the method:

$$
\begin{aligned}
s_1 &= hf(t_n, y_n) \\
s_2 &= hf(t + \tfrac{1}{4}h, y_n + \tfrac{1}{4}s_1) \\
s_3 &= hf(t_n + \tfrac{3}{8}h, y_n + \tfrac{3}{32}s_1 + \tfrac{9}{32}s_2) \\
s_4 &= hf(t_n + \tfrac{12}{13}h, y_n + \tfrac{1932}{2197}s_1 - \tfrac{7200}{2197}s_2 + \tfrac{7296}{2197}s_3) \\
s_5 &= hf(t_n + h, y_n + \tfrac{439}{216}s_1 - 8s_2 + \tfrac{3680}{513}s_3 - \tfrac{845}{4104}s_4) \\
s_6 &= hf(t_n + \tfrac{1}{2}h, y_n - \tfrac{8}{27}s_1 + 2s_2 - \tfrac{3544}{2565}s_3 + \tfrac{1859}{4104}s_4 - \tfrac{11}{40}s_5) \\
w_{n+1} &= y_n + h\left(\tfrac{25}{216}s_1 + \tfrac{1408}{2565}s_3 + \tfrac{2197}{4104}s_4 - \tfrac{1}{5}s_5\right) \\
z_{n+1} &= y_n + h\left(\tfrac{16}{135}s_1 + \tfrac{6656}{12825}s_3 + \tfrac{28561}{56430}s_4 - \tfrac{9}{50}s_5 + \tfrac{2}{55}s_6\right)
\end{aligned}
$$

The fourth-order approximation is w_n and the fifth-order approximation is z_n. Our error estimate is

$$e = \frac{|z_{n+1} - w_{n+1}|}{h} = \frac{1}{360}s_1 - \frac{128}{4275}s_3 - \frac{2197}{75240}s_4 + \frac{1}{50}s_5 + \frac{2}{55}s_6.$$

We divide by h here to get the relative error.

Another improvement that is made is that instead of doubling or halving the step size, it is increased by the following factor (where T, the tolerance, is the desired error):

$$\Delta = .84\left|\frac{T}{w_{n+1}}\right|^{1/5}.$$

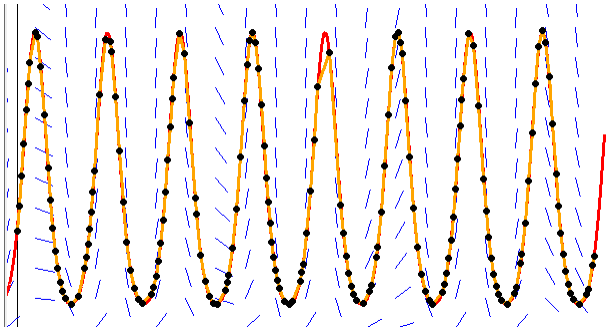
This scaling factor can be shown analytically to be a good choice. In practice, if $\Delta < .1$, we set $\Delta = .1$ and if $\Delta > 4$, we set $\Delta = 4$, both in order to avoid too drastic of a change in step size. Also in practice, the user is allowed to specify a minimum and maximum value for h.

Here is the code for RKF45:

```
def rkf45(f, y_start, t_start, t_end, h, T=.0001, max_h=1):
    t, y = t_start, y_start
    ans = [(t, y)]

    while t < t_end:
        s1 = h*f(t, y)
        s2 = h*f(t+(1/4)*h, y+(1/4)*s1)
        s3 = h*f(t+(3/8)*h, y+(3/32)*s1+(9/32)*s2)
        s4 = h*f(t+(12/13)*h, y+(1932/2197)*s1-(7200/2197)*s2+(7296/2197)*s3)
        s5 = h*f(t+h, y+(439/216)*s1-8*s2+(3680/513)*s3-(845/4104)*s4)
        s6 = h*f(t+(1/2)*h, y-(8/27)*s1+2*s2-(3544/2565)*s2+(1859/4104)*s4-(11/40)*s5)

        w = (25/216)*s1 + (1408/2565)*s3 + (2197/4104)*s4 - (1/5)*s5
        e = abs((1/360)*s1 - (128/4275)*s3 - (2197/75240)*s4 + (1/50)*s5 + (2/55)*s6)/h

        delta = .84 * abs(T/w)**.2
        delta = min(delta, 4)
        delta = max(delta, .1)

        # estimate is very good, increase step size
        if e < T/10:
            y += (16/135)*s1 + (6656/12825)*s3 + (28561/56430)*s4 - (9/50)*s5 + (2/55)*s6
            t += h
            h = min(h / delta, max_h)
            ans.append((t,y))

        # estimate is no good, decrease step size and try this step over
        elif e > T:
            h = h * delta
```

```
            if h < min_h:
                raise Exception("Forced to use step size less than minimum")

        # estimate is acceptable, keep same step size for next step
        else:
            y += (16/135)*s1 + (6656/12825)*s3 + (28561/56430)*s4 - (9/50)*s5 + (2/55)*s6
            t += h
            ans.append((t,y))

    return ans
```

Let's try the method on the IVP $y' = y\cos t$, $y(0) = 10$, $t \in [0, 50]$. We will use $T = .01$ and a maximum h of 1. The approximation is in orange the true solution in red. Notice the uneven spacing of points.



Adaptive step size methods can be extended in a straightforward way to systems of equations. They can also be developed for multistep methods and predictor-corrector methods, though there are some details involved since the variable step size messes with the constant step size required by those methods.

6.10 Extrapolation methods

Recall the process of Romberg integration. We start with the iterative trapezoid method, which uses the trapezoid rule with one trapezoid, then with two, four, eight, etc. trapezoids. We put the results of the method into the first column of a table. We then generate each successive column of the table by applying Richardson extrapolation to the previous column. The order of the iterative trapezoid approximations is 2, the order of the next column is 4, the order of the next column is 6, etc.

We will start with a simple procedure that is a very close analog of Romberg integration. First, in place of the iterative trapezoid rule, we use a second-order method known as the *leapfrog method*. We get it from approximating $y'(t)$ by the centered-difference formula $y'(t) = \frac{y(t+h)-y(t-h)}{h}$. Here is the resulting method:

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_{n-1} + 2hf(t_n, y_n)$$

We see why it is called the leapfrog method: we leap from y_{n-1} to y_{n+1}, jumping over y_n (though using the slope there). We can also see that it is closely related to the midpoint method. A benefit of this method over the usual midpoint method is that this method requires only one new function evaluation at each step, as opposed to the two that the midpoint method requires.

We are interested in the solution of an IVP on some interval of length L. If we were to exactly follow what Romberg integration does, we would run the leapfrog method with $h = L/2$, $L/4$, $L/8$, $L/16$, etc. This would give us a series of estimates and we would then extrapolate and eventually end up with a single value that is the estimate at the right endpoint of the interval.

But we often want to know values of the solution at points inside the interval in order to graph the solution or run a simulation. So what we do is break the interval into subintervals of size h, where h is our time step, and run the entire extrapolation algorithm on each subinterval, generating approximations at the right endpoint of each subinterval.

If we do this, then using $h = h/2$, $h/4$, $h/8$, etc. in the leapfrog method might be overkill, especially if h is relatively small. Instead, we can get away with using the sequence $h/2$, $h/4$, $h/6$, …, $h/16$. It has been shown to work well in practice.

However, when we change the sequence, we have to use a different extrapolation formula. For instance, Romberg-style extrapolation would use $R_{22} = (4R_{21} - R_{11})/3$. However, with this new sequence, the formula turns out to be $R_{22} = (4^2R_{21} - 2^2R_{11})/(4^2 - 2^2)$. Here are the first few rows of the new extrapolation table. Note the similarity to Newton's divided differences. The subsection below explains where the all the terms come from.

R_{11}				
R_{21}	$R_{22} = \frac{4^2R_{21} - 2^2R_{11}}{4^2 - 2^2}$			
R_{31}	$R_{32} = \frac{6^2R_{31} - 4^2R_{21}}{6^2 - 4^2}$	$R_{33} = \frac{6^2R_{32} - 2^2R_{22}}{6^2 - 2^2}$		
R_{41}	$R_{42} = \frac{8^2R_{41} - 6^2R_{31}}{8^2 - 6^2}$	$R_{43} = \frac{8^2R_{42} - 4^2R_{32}}{8^2 - 4^2}$	$R_{44} = \frac{8^2R_{43} - 2^2R_{33}}{8^2 - 2^2}$	
R_{51}	$R_{52} = \frac{10^2R_{51} - 8^2R_{41}}{10^2 - 8^2}$	$R_{43} = \frac{10^2R_{52} - 6^2R_{42}}{10^2 - 6^2}$	$R_{54} = \frac{10^2R_{53} - 4^2R_{43}}{10^2 - 4^2}$	$R_{55} = \frac{10^2R_{54} - 2^2R_{44}}{10^2 - 2^2}$

For example, if we want to estimate the solution to an IVP on the interval $[0, 3]$ using a time step of $h = 1$. The we would first run the leapfrog method on $[0, 1]$ with steps of size $1/2$, $1/4$, $1/6$, …, $1/16$. This would give us the first column of the table and we would extrapolate all the way to R_{88}. This would be our estimate of $y(1)$. Then we would do the same process on $[1, 2]$ to get our estimate of $y(2)$. Finally, we would do the same process on $[2, 3]$ to get our estimate of $y(3)$.

A little about why extrapolation works

The reason we use the leapfrog method as the base step for our extrapolation algorithm (as well as the trapezoid method for Romberg integration) is because the full power series of the error term is of the form

$$c_1h^2 + c_2h^4 + c_3h^6 + \ldots,$$

where the series includes only even terms. If we use a step size of h to get R_{11} and a step size of $h/2$ to get R_{21}, then when we compute $R_{22} = \frac{4R_{21} - R_{11}}{3}$, the h^2 terms cancel out:

$$\frac{4(c_1(h/2)^2 + c_2(h/2)^4 + c_3(h/2)^6 + \ldots) - (c_1h^2 + c_2h^4 + c_3h^6)}{3} = -\frac{1}{4}c_2h^4 - \frac{6}{16}c_3h^6 - \ldots.$$

This means that R_{22} is a fourth order approximation. When we compute $R_{33} = \frac{16R_{22} - R_{32}}{15}$, the h^4 terms cancel out, leaving a 6th order approximation. In general, the order goes up by 2 with each successive level of extrapolation.

More generally, if we have steps of size h/n_1 and h/n_2 to compute R_{11} and R_{22}, we can use the power series to figure out how to eliminate the h^2 term. We end up getting the formula

$$R_{22} = \frac{n_2^2R_{21} - n_1^2R_{11}}{n_2^2 - n_1^2}.$$

The denominator is not necessary to zero out the h^2 term, but it is needed to keep everything scaled properly. The entire expression acts as a sort of odd weighted average with negative weights.

One can also work out a general formula:

$$R_{i,j} = \frac{n_{i+j-1}^2R_{i,j-1} - n_{i-1}^2R_{i-1,j-1}}{n_{i+j-1}^2 - n_{i-1}^2}.$$

This is precisely the formula used by Neville's formula for polynomial interpolation, evaluated at $h = 0$. You may recall that Neville's formula is a close relative of Newton's divided differences. And, indeed, when we do this extrapolation process, we make a table that is reminiscent of the ones we make when doing polynomial interpolation. This because this extrapolation process is really the same thing. We are trying to fit a polynomial to our data and extrapolate to get an estimate of the value at a step size of $h = 0$.

Here is an example of extrapolation. Suppose we run a numerical method with step sizes $1/2$, $1/4$, $1/6$, ... $1/16$ and get the following sequence of values (which is approaching π).[1]

$$3.149041, 3.144575, 3.143303, 3.142737, 3.142428, 3.142237, 3.142110, 3.142020$$

If we put these values into a table and run extrapolation, here is what we get:

1/2	3.149041							
1/4	3.144575	3.143087						
1/6	3.143303	3.142285	3.142185					
1/8	3.142737	3.142010	3.141918	3.141900				
1/10	3.142428	3.141879	3.141805	3.141783	3.141778			
1/12	3.142237	3.141804	3.141745	3.141725	3.141717	3.141716		
1/14	3.142110	3.141757	3.141708	3.141691	3.141683	3.141680	3.141679	
1/16	3.142020	3.141725	3.141684	3.141669	3.141662	3.141659	3.141657	3.141657

We see that the extrapolated values are much closer to π than the original values. Extrapolation takes a sequence of values and makes an educated guess as to what the values are converging to. It bases its guess on both the values and the step sizes.

In addition to polynomial extrapolation, there is also rational extrapolation, which uses rational functions instead of polynomials. It can be more accurate for certain types of problems.

Code for the extrapolation algorithm

Here is some code that computes the entire extrapolation table.

```
def neville(X, Y):
    R = [Y]
    for i in range(1,len(Y)):
        L = []
        for j in range(1,len(R[i-1])):
            L.append((X[i+j-1]**2*R[i-1][j]-X[j-1]**2*R[i-1][j-1])/(X[i+j-1]**2-X[j-1]**2))
        R.append(L)
    return R
```

The list R is a list of the columns of the extrapolation table. Here is how we can code the entire extrapolation algorithm:

```
def extrapolate(f, y_start, t_start, t_end, H):
    ans = [(t_start, y_start)]
    start = t_start
    end = t_start + H
    while end < t_end:
        h = end - start
        n = 8
        X = []
        Y = []

        for i in range(n):
            h  = (end-start)/(2*(i+1))
            t, y, oldy = start, y_start, y_start
            y += h*f(t+h/2, y+h/2*f(t,y))
```

[1] These values come from using the leapfrog method on $y' = 4\sqrt{1-x^2}$, $y(0) = 0$, $t \in [0,1]$. This IVP is equivalent to approximating $4\int_0^1 \sqrt{1-x^2}\,dx$, whose value is π.

```
            t += h
            for j in range(2*(i+1)):
                oldy, oldy2 = y, oldy
                y = oldy2 + 2*h*f(t, y)
                t += h
            X.append(2*(i+1))
            Y.append(oldy)

        N = neville(X,Y)
        ans.append((end, N[n-1][0]))
        start, end = end, end + H
        y_start = N[n-1][0]
    return ans
```

One further improvement we could make here would be to turn this into an adaptive step size algorithm. The way to do that would be to use the difference between the last two extrapolated results (R_{88} and R_{77}) as our error estimate, and adjust h accordingly. It might also not be necessary to do all 8 ministeps at each stage.

6.11 Summary and Other Topics

Euler's method is nice because it is extremely simple to implement, but because of its limited accuracy, it should not be used for anything serious. It is useful, however, if you just need to get something up and running quickly. For more serious work, Runge-Kutta methods with adaptive step size control, like RKF45 and similar methods, are recommended as all-purpose methods. They are reasonably fast and usually work. They are the methods of choice for non-smooth systems. For smooth systems, predictor-corrector and extrapolation methods are recommended. They can provide high accuracy and work quickly. Stiff equations are best dealt with using implicit methods. See *Numerical Recipes* for more particulars on practical matters.

We have only looked at initial value problems. There is another class of ODE problems, called boundary value problems, that are more difficult. A typical example would be an equation of the form $y'' = f(t, y)$, with $t \in [a, b]$, where instead of specifying the initial conditions $y(a)$ and $y'(a)$, we instead specify $y(a)$ and $y(b)$. That is, we specify the values of the function at the boundary of $[a, b]$. An example of this might be a pendulum whose position at time $t = a$ and $t = b$ we know (but not its initial velocity), and we want to know what the motion of the pendulum would be between times a and b.

These types of problems can be generalized to higher dimensions, like a three-dimensional object whose temperature is governed by a differential equation, where we know the temperature at the surface of the object (its boundary), and we want to know the temperature inside the object. One method for numerically solving boundary value problems is the shooting method, which, for a two-point boundary value problem, involves trying to take "shots" from the initial point to hit the end point, using intelligent trial-and-error to eventually reach it. Those "shots" come from making a guess for the initial velocity. Another class of methods for boundary-value problems are finite difference methods, which use numerical derivative approximations, like $\frac{f(t+h)-f(t-h)}{2h}$, to turn a system of ODEs into a system of algebraic equations.

Besides ODEs, there are PDEs, which are used to model many real-life problems. Finite difference methods are one important class of methods for numerically solving PDEs, but there are many other methods. Numerical solution of PDEs is a vast field of active research.

Chapter 7

Exercises

7.1 Exercises for Chapter 1

1. Briefly explain what error propagation is, and give an example demonstrating it.
2. People sometimes mix up the terms "roundoff error" and "error propagation." Explain how they differ.
3. When evaluated on a computer using double-precision floating point (from the IEEE 754 standard), what are the results of $1000000000 + .5$ and $1000000000 + .00000005$?
4. The result of $.2 + .1$ on an IEEE 754 floating-point system is

   (a) exactly .3 (b) a number within around 4×10^{-17} of .3
5. The calculation $123456789.0 + .000000001$ comes out to 123456789.0 and not 123456789.000000001 when using 64-bit floating point arithmetic. Why?
6. Explain why the result of $.1 + .2$ is 0.30000000000000004 in double-precision floating point.
7. Briefly, what is the main difference between the `float` and `double` data types of many programming languages?
8. The radio station WMTB 89.9 FM is running a special promotion where if you buy two items totaling exactly $89.90 using their app, then you get both for free. They programmed the app using the Python code below:

   ```
   if price1 + price2 == 89.9:
       print("Everything's free!!!")
   ```

   But someone could buy something for $57.91 and something else for $31.99 and the program will not correctly catch that as totaling $89.90. Why not? How could the problem be fixed?
9. Consider the expressions $\sqrt{x+h} - \sqrt{x}$ and $\dfrac{h}{\sqrt{x+h} + \sqrt{x}}$.

   (a) Use algebra to show that the expressions are equivalent.

   (b) Compute both expressions with $x = 1$ and $h = .0000000001$. Which one is more accurate? Why?
10. We saw that the number 0.2 is not represented exactly using floating point. In fact, it is represented by 0.200000000000000011102230246251565404236316680908203125. This is actually the closest 64-bit floating point number to .2.

    (a) Explain in terms of binary decimal expansions why this is.

    (b) Give two examples of numbers in the interval $(0, 1)$ that can be represented *exactly* using floating point. Explain briefly.

    (c) It seems annoying that we use a system that can't represent things exactly. Why don't we use such a system?

11. The IEEE 754 double precision standard allows 53 bits for the mantissa, which guarantees roughly 15 decimal digits of accuracy.

    (a) Where does the 15 come from? [Hint: 53 (binary) bits corresponds to 2^{53} possible values — 2^{53} is equal to 10 raised to what power?]

    (b) How many bits would we need for the mantissa if we wanted 100 decimal digits of accuracy?

12. What is the base 10 number whose binary expansion is .010101010101...? [Hint: the geometric series formula may be helpful.]

7.2 Exercises for Chapter 2

1. Consider the bisection method on $x - \cos x = 0$ with the starting interval $[0, 1]$.

    (a) Explain why $[0, 1]$ is a valid starting interval.

    (b) Do three steps of the bisection method and be sure to say what your final approximation is.

2. Use the bisection method to approximate a root of $x^3 - 3x + 1 = 0$. Do four steps of the bisection method and be sure to say what your final approximation is.

3. (a) Express $e^x - x^3 = 0$ as a fixed point problem in three different ways.

    (b) Pick one of those and use it to approximate a solution to $e^x - x^3 = 0$ to within at least 10^{-8}.

    (c) For each of the ways in part (a), determine whether or not the fixed point is attracting.

4. Consider the equation $\cos x = x^3$.

    (a) Rewrite the equation as a fixed point problem, $g(x) = x$, and use fixed point iteration to find a solution correct to within .000001.

    (b) What is the approximate error relationship $e_{i+1} \approx Se_i^p$ in part (a)? Find S and p.

    (c) Rewrite the equation as another fixed point problem in such a way that fixed point iteration will *not* find a solution. Explain why fixed point iteration fails to find the solution.

5. Use Newton's method to estimate a root of $e^{-x} - x$ correct to at least 15 decimal places.

6. Use the secant method to estimate the solution to $\cos x = x$ correct to 15 decimal places.

7. Why do people care about solving equations?

8. Why are numerical methods for solving equations needed?

9. Give an example scenario where a researcher might need to use Newton's method (or one of its more sophisticated relatives).

10. The bisection method, FPI, and Newton's method are used to find roots—that is, solutions to equations of the form $f(x) = 0$. Suppose we want to find the solution to an equation of the form $f(x) = 5$. How would we use these methods to do that?

11. Suppose we are using the bisection method to estimate a root. After 6 steps, starting with initial interval $[1, 4]$, regardless of the equation being considered, the error between the approximation and an actual root will be at most how much?

12. True or False: A fixed point of $f(x)$ corresponds to a place in which $y = f(x)$ intersects $y = x$.

13. Find the fixed point of $f(x) = 2x + 3$.

14. Give an example of a function that has no fixed points.

15. Fill in the blank: If we rewrite $f(x) = 0$ as a fixed point problem $g(x) = x$, then a fixed point r of g corresponds to a _____ of f.

16. In your own words, explain the geometric derivation of Newton's method.

17. If we rewrite $f(x) = 0$ as a fixed-point problem $g(x) = x$, and if we find a value r such that $g(r) = r$, then what must $f(r)$ equal?

18. Describe how fixed point iteration can be explained in terms of cobweb diagrams.

19. Sketch the first few steps of a cobweb diagram for $f(x) = \cos(x)$ with starting value $x = 2$.

20. Suppose you are solving $x^5 - x - 1 = 0$ using fixed-point iteration. You rewrite it as a fixed point problem, iterate, and the iterates settle down around $x = 2.516$. How can you tell that 2.516 cannot be possibly anywhere near correct?

21. When searching for a fixed point, the derivative, $f'(r)$, evaluated at the fixed point r, gives some information. What does it tell you?

22. Which converges faster to a root — the bisection method or fixed point iteration? Or does it depend?

23. Fixed point iteration of $g(x) = x^3 + e^x$ fails to find the fixed point located at approximately -1.133. Why?

24. If e_i denotes the error at step i of a numerical root-finding method, then we have the approximate error relationship, $e_{i+1} \approx Se_i^p$.

    (a) What can be said about S and p for the bisection method?

    (b) What can be said about S and p for fixed point iteration?

    (c) What can be said about p for Newton's Method?

    (d) Does a larger or smaller S mean faster convergence?

    (e) Does a larger or smaller p mean faster convergence?

    (f) A certain numerical root-finding method adds about one new correct decimal place every 4 iterations. In the approximate error relationship $e_{i+1} \approx Se_i^p$, what must S and p be?

    (g) A certain numerical root-finding method triples the number of correct decimal places with each iteration. In the approximate error relationship $e_{i+1} \approx Se_i^p$, if $S = 1$, what must p be?

25. Suppose we want to solve $x^3 - 10 = 0$ via fixed point iteration. One way to rewrite this as a fixed point problem is as $\frac{9}{x^2+x+1} + 1 = x$. If we iterate this starting at 2 and look at the error ratios e_{i+1}/e_i, we get a sequence of values .8501, .7380, .8252, .7562, .8102, .7676, ..., eventually settling down near 0.7861370554624069. That number is a decimal approximation to a particular irrational number that is the limit of the sequence of error ratios just given. What is the exact value of that irrational number? [Hint: Use what you know about error ratios and fixed point iteration to find it.]

26. If numerical root-finding methods A and B have rates of convergence of 1.8 and 2.2, respectively, which converges more quickly? Why? (Assume the constant C in the error relation $e_{i+1} \approx Ce_i^p$ is roughly the same for both methods.)

27. If a numerical root-finding method converges linearly, what does that mean, roughly speaking about the number of new correct decimal places as opposed to a method that converges quadratically?

28. Halley's Method converges cubically. This means that if we have 12 correct decimal places after a certain number of iterations, roughly how many will we have correct after the next iteration?

29. The key geometric insight of Newton's Method involves what about tangent lines?

30. Which numerical method is more reliable — Newton's method or the bisection method?

31. In what sense is the Secant Method an improvement on Newton's Method? In what sense is Newton's Method better?

32. Give some examples of ways in which Newton's method can fail.

33. We have seen some examples of where Newton's method can fail. Is it possible for the secant method to fail? Explain.

34. Recall that Newton's Method is quadratic, while the secant method's power is about 1.618. Compare these: Specifically, if the error starts out at .1, and the error formula is $e_{i+1} \approx e_i^p$, then after 8 iterations, about how many correct decimal places will each method produce?

35. Suppose we take a sequence x_n of values converging to some value x and create the new sequence $y_n = x_n - (x_{n+1} - x_n)^2/(x_n - 2x_{n+1} + x_{n+2}$. This is the sequence given by Aitken's Δ^2 method. What is the main purpose of doing this?

36. Geometrically, what is the relationship between the secant method and Muller's method?

37. We have seen that Brent's method combines the safety of the bisection method with the speed of IQI. Explain briefly how it accomplishes that.

38. Why do we have numerical methods that are specifically designed to find roots of polynomials (and only polynomials)? Why not just use more general methods that work on any function?

39. Explain the importance of the Wilkinson polynomial.

40. Let $p(x) = x^3 + 4x + 9$. Use the long division method described in the section covering Laguerre's method to find $f(2)$ and $f'(2)$.

41. Talk about the roles of Muller's method, IQI, Brent's Method, Aitken's Δ^2 Method, and Laguerre's Method in root-finding.

42. Geometrically speaking, explain why the root-finding methods discussed in class do poorly around a multiple root.

43. Expand $(x-1)^5$ using the binomial formula. Using the expanded formula, compute $f(1.0011417013971329)$. What consequence does this have for root finding?

44. How well would you expect the rootfinding methods we have learned to perform when finding a root of the expanded form of the polynomial below? [Hint: the factored form of this polynomial is $(x-1)^9(x-2)$.]

$$x^{10} - 11x^9 + 54x^8 - 156x^7 + 294x^6 - 378x^5 + 336x^4 - 204x^3 + 81x^2 - 19x + 2$$

45. Why are multiple roots so troublesome for root-finding methods?

46. Consider $f(x) = (x-2)^6 = x^6 - 12x^5 + 60x^4 - 160x^3 + 240x^2 - 192x + 64$.

    (a) If we try to find the root at $x = 2$ for this function using a numerical method like the ones covered in these notes, how accurate of an answer would you expect?

    (b) Would the forward error or backward error be a more appropriate measure of how close an approximation is to the root of this function?

47. Consider the polynomial $(x-1)^9(x-2)$. If we expand it out, we get the polynomial below:

$$x^{10} - 11x^9 + 54x^8 - 156x^7 + 294x^6 - 378x^5 + 336x^4 - 204x^3 + 81x^2 - 19x + 2.$$

    (a) Use Newton's Method in Python on this with a starting value of $x = 1.1$ and perform about 30 iterations. What value do the iterates converge on?

    (b) The real root is of course 1, but the answer from (a) is as close as we can get. Compute the forward and backward errors of the approximation in part (a).

    (c) Explain in terms of the graph of the function why the forward and backward errors are the way they are for this problem.

    (d) If we try a starting value of 1.01, we actually get a division by zero error. In terms of floating point numbers and the functions involved, explain why we get this error.

    Here are the function and its derivative in Python to save you the trouble of entering them in:

```
f = lambda x:x**10-11*x**9+54*x**8-156*x**7+294*x**6-378*x**5+336*x**4
-204*x**3+81*x**2-19*x+2

df = lambda x:10*x**9-99*x**8+432*x**7-1092*x**6+1764*x**5-1890*x**4
+1344*x**3-612*x**2+162*x-19
```

48. (a) Use the secant method to derive the following iteration for estimating $\sqrt{a}$:

$$x_n = \frac{x_{n-1}x_{n-2} + a}{x_{n-1} + x_{n-2}}.$$

(b) Use it to estimate $\sqrt{2}$ correct to 10 decimal places. Please show the starting value and sequence of iterates that you get.

49. Consider the polynomial $(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)(x-8)(x-9)(x-10)$.

(a) What are its roots?

(b) Use Wolfram Alpha or another computer algebra system to expand it. Then increase the coefficient of the x^9 term by .0001. Use Wolfram or a computer algebra system to find the roots of the new polynomial.

50. The Babylonian method for approximating $\sqrt{a}$ involves starting with an approximation x and creating a better approximation by taking the average of x and a/x. To approximate $\sqrt[3]{a}$, you can take the average of x and a/x^2. However, you can make this faster by taking a weighted average of the two. Find the weighting that gives the fastest convergence. Specifically, find w between 0 and 1 such that iterating $wx+(1-w)a/x^2$ will converge as quickly as possible.

51. Use Newton's method on $f(x) = \frac{1}{x} - a$ along with some algebra to derive a way to approximate $\frac{1}{a}$ without any division required. Try the method out to compute $\frac{1}{7}$.

52. One of the solutions of $x^2 - a = 0$ is $\sqrt{a}$. We have already seen a way to apply Newton's method to this to derive a way to approximate $\sqrt{a}$. In this problem you will apply other methods to generate new ways to approximate $\sqrt{a}$. Use each of the following methods to do so. Simplify your expressions as much as is reasonably possible.

(a) Secant method

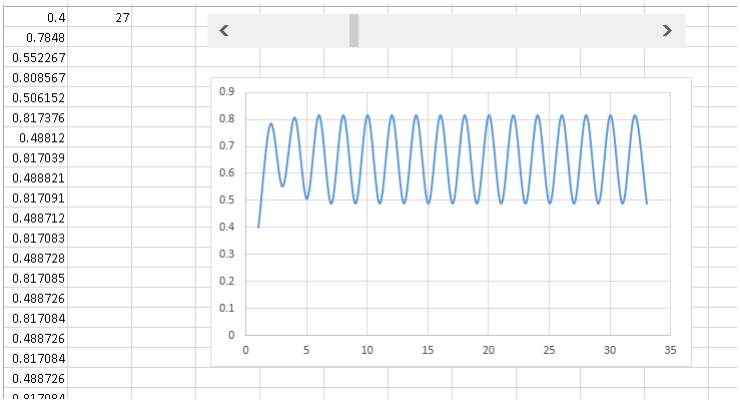
(b) Halley's method

53. Let $p(x)$ be a polynomial of the form $(x-a)^n q(x)$, where $q(x)$ is a polynomial with $q(a) \neq 0$. That is, p has a multiple root of degree n at a. Show that the kth derivative $p^(k)(a) = 0$ for $k = 1, 2, \ldots n-1$ but that $p^(n)(a) \neq 0$.

54. Sometimes Newton's Method can behave chaotically. As mentioned earlier, one instance of Newton's Method reduces to iterating $4x(1-x)$, which behaves chaotically. To show how this, create a spreadsheet that iterates $4x(1-x)$ with starting value .4 for 50-100 iterations and plot the results with a scatter plot (and connect the dots).

Then add a scrollbar to your program to allow the user to be able to vary the 4 in $4x(1-x)$ to values of a between 2 and 4 at intervals of .01. To add a scrollbar, you will need to enable the Developer tab and you'll probably need to google how to insert the scrollbar.

Here is a screenshot of what it looks like on my machine:



55. Create a spreadsheet about ratios of errors for fixed-point iteration. Column A contains the result of iterating $1/(1+x^2)$ 60 times starting at $x = .5$. Column B contains the errors $e_i = |x_i - r|$, where the x_i are the iterates in the first row and r is an estimate of the root (here it will be the entry in cell `A61`. Column C contains the error ratios e_{i+1}/e_i. Recreate this spreadsheet. An example of the first few lines is shown below.

	A	B	C
1	**iterates**	**error**	**error ratio**
2	0.5	0.18233	0.6453881
3	0.8	0.11767	0.6167277
4	0.609756098	0.07257	0.6426762
5	0.72896791	0.04664	0.6288167

56. (a) Implement Aitken's Δ^2 method in a spreadsheet. The first column should be the result of iterating $f(x) = \cos(x)$ and the second column should be the improved sequence from applying Aitken's Δ^2 method to the first column.

    (b) Implement Steffensen's FPI in a spreadsheet to iterate $f(x) = \cos x$. Steffensen's FPI does the following: It does 3 iterations of FPI. Then it applies Aitken's Δ^2 to that to get a value a. Then it does 2 more iterations of FPI, using a as the value to feed into the function. Then it applies Aitken's Δ^2 again, followed by 2 more iterations of FPI. This process of two iterations of FPI followed by one of Aitken's Δ^2 continues. You should do this with a general formula and not by editing each line separately.
    Hint: Look up how to do `if()` expressions in Excel and also how to use the `MOD()` function.

	A	B	C	D	E
1	Part (a)			Part (b)	
2	1	0.72801		1	1
3	0.540302	0.733665		2	0.540302
4	0.857553	0.736906		3	0.857553
5	0.65429	0.73805		4	0.72801
6	0.79348	0.738636		5	0.7465
7	0.701369	0.738877		6	0.73407
8	0.76396	0.738992		7	0.739067
9	0.722102	0.739043		8	0.739097
10	0.750418	0.739066		9	0.739077
11	0.731404	0.739076		10	0.739085
12	0.744237	0.739081		11	0.739085

57. Use the Python `Decimal` class, the Java `BigDecimal` class, or another programming language's decimal class to estimate the solution of $1 - 2x - x^5 = 0$ correct to 50 decimal places.

58. Complex (imaginary) numbers are built into Python. The equation $x^4 + 3x^2 - 2x + 2$ has no real roots, but it does have four complex roots. Use Newton's Method and Python to estimate one of those roots correct to 4 decimal places.

59. Use one of the numerical methods covered in these notes to write a method in your favorite programming language called `my_sqrt` that computes $\sqrt{n}$ as accurately as the programming language's own `sqrt` function (but without using the language's `sqrt` or power functions).

60. (a) Write a function that takes a sequence (a list), and returns a new sequence gotten by applying Aitken's Δ^2 method to it.

    (b) Try it out on the first 30 terms of the sequence gotten from iterating $1.25\cos x$ starting with $x = 0$.

61. Use Python and Newton's Method to estimate a complex root of $x^3 - 1 = 0$.

62. (a) Implement Ridder's method in Python. The method should take as parameters a function f, values a and b, where $f(a) < 0$ and $f(b) > 0$, and a tolerance t such that the result returned by your code should be within t units of a root.

    (b) Try it on $x^2 - 4$ and try it on $(x-1)^5$.

63. (a) Write a function that takes a sequence (a list), and returns a new sequence gotten by applying Aitken's Δ^2 method to it.

    (b) Try it out on the first 30 terms of the sequence gotten from iterating $1.25\cos x$ starting with $x = 0$.

7.3 Exercises for Chapter 3

1. For this problem, we want to find the interpolating polynomial that passes through the points $(-1, 0)$, $(2, 1)$, $(3, 1)$, and $(5, 2)$.

   (a) Use the Lagrange formula. Show all work.

   (b) Use Newton's divided differences. Show all work.

2. The following table gives the percentage of students on a school's Dean's List in the fall of several years. Find the interpolating polynomial through this data and use it to estimate the percentage in 2003.

year	percent
1986	16.3%
1992	18.4%
1998	22.5%
2008	29.8%
2014	35.1%

3. Find the Cheybshev interpolation x-values for the interval $[3, 7]$ with $n = 5$.

4. Find the Chebyshev interpolation x-values points the interval $[4, 6]$ with $n = 7$.

5. Find the natural cubic spline through the points $(1, 3)$, $(2, 4)$, $(4, 7)$, $(5, 12)$, $(7, 19)$, and $(8, 23)$. Be sure to show the matrix you use to find the values of the c_i coefficients. Also, graph the spline.

6. Suppose we have found an interpolating polynomial for a set of 10 data points of the form (x, y). all of whose x values lie between 1 and 10. Would it be reasonable to use the polynomial to estimate the y value for $x = 15$? Why or why not?

7. For a given set of points, let $L(x)$ be the Lagrange interpolating polynomial through those points and let $p(x)$ be the polynomial through these points that we get from Newton's divided differences. Which of the following is true? *Please explain.*

   (1) $L(x)$ and $p(x)$ are actually always the same
   (2) $L(x)$ and $p(x)$ usually agree except for a few very unusual examples
   (3) $L(x)$ and $p(x)$ most of the time are different.

8. What is interpolation used for?

9. What is the Runge phenomenon? What consequences does it have for interpolation?

10. What is the relationship between the roots of the Chebyshev polynomials and interpolation?

11. Give one example situation where cubic spline interpolation would be preferable to polynomial interpolation with NDD/Lagrange's formula, and give one example situation where polynomial interpolation with NDD/Lagrange's formula is preferable to cubic spline interpolation.

12. A cubic Bézier curve is defined by four control points. Geometrically, how do those points determine what the curve will look like?

13. Suppose we have a table of values giving a school's enrollment in 1990, 1993, 1998, 2003, and 2010. Suppose we want an estimate of enrollment in 2006.

    (a) What process would you use?

    (b) Would that process give you a reliable estimate of 2017 enrollment?

14. We have seen several examples of interpolating known functions like $\sin x$ and $\ln x$. Why would that ever be a useful thing to do?

15. If you didn't have a scientific calculator handy, how would it be possible to estimate $\sin(1/3)$ accurate to several decimal places?

16. In cubic spline interpolation, is it okay for two splines to meet at a sharp turn?

17. Give an advantage of cubic spline interpolation over polynomial interpolation (i.e. Lagrange and NDD).

18. Use the technique mentioned in Section 3.5 to estimate $\ln(104)$.

19. Using the sum and double-angle identities for cosine and sine, find an expression for $\cos(3x)$ in terms of $\cos(x)$. Use this to find the Chebyshev polynomial $T_3(x)$.

20. Use five Chebyshev points on the interval $[-2,1]$ to interpolate the function xe^x. Use Newton's divided differences to find the interpolating polynomial and plot both the polynomial and xe^x on the same graph.

21. (a) Find the first 5 terms of the Taylor series of $f(x)=\sqrt{x}$ centered at $x=1.5$.
    (b) Find the 5 Cheybshev points on $[0,10]$, compute their square roots, and use NDD on these to get an interpolating polynomial for $\sqrt{x}$.
    (c) Use the Taylor polynomial as well as the Chebyshev approximation to estimate $\sqrt{.5}$, $\sqrt{1}$, $\sqrt{1.25}$, $\sqrt{2}$, $\sqrt{5}$, and $\sqrt{10}$. Find the errors in each case, and compare their accuracies.
    (d) Neither polynomial is appropriate to estimate $\sqrt{100}$, but if we write $\sqrt{100}$ as $\sqrt{2^4\cdot(100/2^4)}$, we can use properties of square roots and the Cheybshev approximation to estimate $\sqrt{100}$. Explain how this works. How accurate is it for $\sqrt{100}$?

22. Suppose we have the polynomial $p(x)=\frac{\sqrt{2}}{2}(x+1-\pi/4)$ that approximates $\sin x$ in the interval $[0,\pi/2]$. Using only that polynomial and trig identities, compute the following. Explain precisely what trig identities you are using, and remember that you can only use the identities and the polynomial.
    (a) $\sin(3\pi/4)$
    (b) $\sin(5\pi/4)$
    (c) $\sin(27\pi/4)$
    (d) $\sin(-43\pi/4)$

23. Recall the Taylor series for e^x centered at $x=0$ is $1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots$. Use this series to find the Taylor series for e^{2x}. [Hint: the solution should be very short.]

24. The Taylor series for $\frac{1}{1+x}$ centered at $x=0$ is $1-x+x^2-x^3+\ldots$. Integrate both sides to get the Taylor series for $\ln(1+x)$.

25. This problem is about interpolating the function $\frac{1}{1+x^2}$. For most of the parts of this problem, doing things by hand would be painful. I recommend using a spreadsheet or a program to help you out.
    (a) Generate 21 evenly-spaced points in the range from -5 to 5 of the form $(x,\frac{1}{1+x^2})$. These points will start with $(-5,\frac{1}{26})$ and end with $(5,\frac{1}{26})$.
    (b) Then generate the 21 points from the Chebyshev formula.
    (c) Find the two interpolating polynomials through the points from parts (a) and (b). Graph the polynomials and $\frac{1}{1+x^2}$ on the same graph and include a screenshot of your graph.
    (d) Use the error bounds given for evenly-spaced and Chebyshev points to compute the maximum possible interpolation errors at $x=4.8$.
    (e) Compute the exact errors between the two polynomials and $\frac{1}{1+x^2}$ at $x=4.8$.

26. Find data online for the world population starting around 1750 or 1800 up until the present day. Use polynomial interpolation to find a formula for that data. Use the data to approximate the world population in the following years: 1500, 1700, 1885, 1997, 2020, 2080. Please include all your work in a spreadsheet or computer program.

27. Find a polynomial $p(x)$ giving the number of days of each month of the year. In particular, $p(1)=31$, $p(2)=28,\ldots,p(12)=31$.

28. Create a spreadsheet that generates the Chebyshev x-values on $[-1,1]$ for $n=1$ to 20.

29. Create a spreadsheet that generates the coefficients of the Chebyshev polynomials from 1 to 20. Do this using a formula that works for everything, rather than line-by-line editing.

30. Write a function in a programming language that is given an integer n and an interval $[a, b]$ and returns a list of the n Chebyshev x-values on that interval.

31. Write a function in a programming language that is given an integer n and returns the nth Chebyshev polynomial, nicely formatted as a string. For instance, `cheb(5)` should return `16x^5-20x^3+5x`.

32. Write a function in a programming language that is given a list of data points, an x-value, and uses Newton's divided differences to compute the value of the interpolating polynomial at x. It's up to you how to specify how the data points are passed to your function, but make sure that it works for any number of data points.

7.4 Exercises for Chapter 4

1. Use the forward, backward, and centered difference formulas to approximate the derivative of $\sin(x^2)$ at $x = 3$, using $h = .000001$. Compare your results with the exact value.

2. Taking derivatives is generally pretty easy. The rules you learn in Calculus I are enough to take the derivative of any elementary function. So why then are numerical differentiation techniques needed at all? Give at least two reasons.

3. For numerical differentiation, there is a reason why smaller h values are desirable, and there is a reason why smaller h values can be problematic. What are the two reasons?

4. Explain geometrically, in terms of tangent and secant lines, where each of the forward difference, backward difference, and centered difference formulas come from.

5. Suppose we try to estimate the derivative of $f(x) = \sqrt{x}$ at $x = .005$ using the centered difference formula with $h = .01$. What would go wrong? What could we do instead? Be specific.

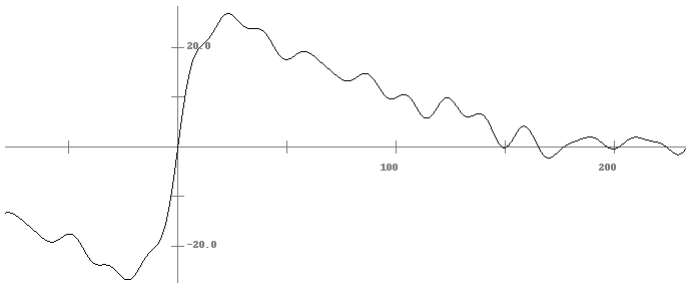
6. If Richardson extrapolation is applied to a numerical differential method of order 2, what can you say about the order of the resulting method?

7. If you are creating a new numerical differentiation technique, and you want to know if it works, what could you do?

8. Recall that the derivative of a function $f(x)$ is defined as $\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$. How does this definition relate to numerical differentiation?

9. Show that the centered difference rule gives the exact derivative (not just an approximation) for any quadratic (i.e. $f(x) = ax^2 + bx + c$).

10. Shown below is the graph of a function defined by the Python code below it. It would be painful to evaluate the derivative of this function using basic calculus. However, it is quick to use a numerical method to estimate its derivative. Use the forward difference formula with $h = 10^{-8}$ to estimate $f'(2)$.




```
from math import sin

def f(x):
    n = 27
    t = 0
    for i in range(50):
```

```
        t += sin(10*x/n)
        n = (n//2) if n%2==0 else 3*n+1
    return t
```

11. Use Taylor series to develop a second-order method for approximating $f'(x)$ that uses the data $f(x+2h)$, $f(x)$, and $f(x-4h)$ only. Find the error term. Then check your work by using it to approximate the derivative of $\sin x$ at $x=3$. You should get something pretty close to $\cos(3)$.

12. Use Taylor series to find a second-order numerical differentiation rule that uses $f(x)$, $f(x-2h)$ and $f(x+5h)$.

13. Use Taylor series to find a first-order numerical differentiation rule that uses $f(x)$, $f(x-h)$ and $f(x+3h)$ to approximate $f''(x)$.

14. Apply one level of Richardson Extrapolation to the values below that are obtained from using the forward difference approximation to estimate $f'(2)$ for $f(x)=\sin(x)$.

h	Approx
0.1	.4609
0.05	.4387
0.025	.4275
0.0125	.4218

15. The second-order formula $\dfrac{f(x-h)-2f(x)+f(x+h)}{h^2}$ can be used to approximate $f''(x)$. Apply Richardson extrapolation to the formula. Please simplify your answer as much as possible.

16. Automatic differentiation and dual numbers:

    (a) Compute $(2+\epsilon)(5-3\epsilon)$

    (b) Explain where we get the identity $f(x+\epsilon)=f(x)+f'(x)\epsilon$ from.

    (c) Compute $\ln(2+3\epsilon)$.

    (d) Analogously to the derivations of the chain rule and product rule given, derive the quotient rule on dual numbers.

    (e) Use the automatic differentiation Python code provided to take the derivative of $\sin(x^2+\cos x)$ at $x=2$ and verify that the answer given is the same as the exact answer. For your answer, please include the command you use, the formula you use for the derivative, and the common value they both give.

17. Consider the following samples of a function $f(x)$ at several values of x.

x	y
0	0
2	1.414
4	2
6	2.449
8	2.828
10	3.162

    The function is actually $f(x)=\sqrt{x}$ and its exact derivative at 3 is 0.2887, to four decimal places.

    (a) Use the centered difference formula with $h=1$ to estimate $f'(3)$. Compare this with the exact derivative.

    (b) If we do a cubic spline interpolation on this data, we get the following piece of the spline on the region $[2,4]$: $s(x)=1.414+0.4938(x-2)-0.1600(x-2)^2+0.0298(x-2)^3$. Evaluate $s'(3)$. Compare this with the exact derivative.

    (c) If we do a Chebyshev approximation on the range $[0,10]$, we get the following polynomial: $g(u)=2.236+1.048u-0.2207u^2+0.3692u^3-0.2779u^4$. Taking $u=(x-5)/5$, evaluate the derivative of this polynomial at $x=3$ and compare with the exact derivative.

18. (a) Write down the equation of the Lagrange interpolating polynomial through the points $(a, f(a))$ and $(a+h, f(a+h))$.
    (b) Take the derivative of your expression from part (a). What numerical differentiation rule does the result look like?
    (c) Write down the equation of the Lagrange interpolating polynomial through the points $(a-h, f(a-h))$, $(a, f(a)$, and $(a+h, f(a+h))$.
    (d) Take the derivative of your expression from part (c), and then plug in $x = a$. What numerical differentiation rule does the result look like?

19. Create a table comparing the errors between the forward and centered difference formulas when used to approximate the derivative of $\sin(x^2)$ at $x = 2$. The table should have entries for $h = .1, .01, .001, \ldots, 10^{-14}$. I recommend using Excel or writing a short program for this.

20. Create a table comparing the forward difference, centered difference, and exact derivatives of $f(x) = \sin(x^2)$ at $x = 2$. The table should have entries for $h = .1, .01, .001, \ldots, 10^{-14}$. It should also have entries for errors (absolute values of the differences between the approximations and the exact answer). I'd recommend using Excel or writing a short program for this.

21. Create a spreadsheet that does the tabular form of Richardson extrapolation to estimate the derivative of $\cos x$ at $x = 1$. Your spreadsheet should use values of h starting at $h = 1$ and going down to $h = 1/64$ by halves. Create an area in your spreadsheet that computes the errors between the estimation and the exact value. The errors should look like the ones below:

0.133398						
0.034626	0.001702					
0.008738	0.000109	2.56E-06				
0.00219	6.84E-06	4.06E-08	5.59E-10			
0.000548	4.28E-07	6.36E-10	2.2E-12	2E-14		
0.000137	2.67E-08	9.95E-12	8.99E-15	3.33E-16	3.33E-16	
3.42E-05	1.67E-09	1.6E-13	4.22E-15	4.22E-15	4.22E-15	4.22E-15

7.5 Exercises for Chapter 5

1. This problem will be looking at $\int_2^4 \sin(x^2)\,dx$. Use each of the methods below to approximate the integral.
   (a) Simpson's rule with 4 rectangles
   (b) Midpoint rule with 4 rectangles
   (c) Three steps of the iterative trapezoid rule
   (d) Romberg integration (find R_{33})
   (e) $n = 2$ Gaussian quadrature
   (f) $n = 3$ Gaussian quadrature

2. What is the degree of precision of the following: trapezoid rule, Simpson's rule, the third column of Romberg integration?

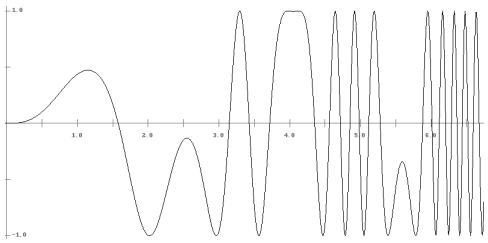
3. Suppose $B(f, a, b, n)$ is the Boole's rule estimation of $\int_a^b f(x)\,dx$ with n rectangles. To within .01, what is $\sum_{i=1}^{\infty}\left(\int_0^i x^3\,dx - B(x^3, 0, i, 2i+5)\right)$?

4. Arrange left rectangle endpoint approximations, midpoint rule, Simpson's rule, 3-point Gaussian quadrature, and the trapezoid rule in order from least to most accurate.

5. What does it mean to say a numerical integration method has a degree of precision of 5?

6. In your own words, describe how adaptive quadrature works.

7. Explain why adaptive quadrature would be useful to numerically integrate this function.



8. What is the role that extrapolation plays in Romberg integration?

9. If the trapezoid rule is used with $n = 20$ trapezoids and then with $n = 40$ trapezoids, is it possible to reuse some of the work from the $n = 20$ case when computing the $n = 40$ case? Explain.

10. Is it possible to integrate any quintic polynomial simply by evaluating the function at no more than three values?

11. What substitution is typically used to numerically estimate an integral like $\int_1^\infty e^{-x^2}\,dx$?

12. Why do higher dimensional analogs of the integration methods we covered have trouble in very high dimensions?

13. Explain how Monte Carlo integration works.

14. How many new function evaluations are required in the 10th iteration of the iterative trapezoid rule?

15. What is true about the order of each successive column in Romberg approximation in relation to the previous column?

16. The adaptive quadrature method we covered uses two estimates in its decision for whether to continue in a region or not. What are those estimates and how are they used?

17. True or False. Monte Carlo integration is especially useful for high dimensional integrals where other methods are ineffective due to the "curse of dimensionality".

18. Which of the following are valid ways to numerically integrate $\int_1^\infty e^{-x^2}$? (Choose all that apply.)

    (a) Use the substitution $u = 1/x$ and apply the midpoint method to the resulting integral.

    (b) Use the substitution $u = 1/\sqrt{x}$ and apply the midpoint method to the resulting integral.

    (c) Use the substitution $u = 1/\sqrt{x}$ and apply the trapezoid method to the resulting integral.

    (d) Use the substitution $u = 1/x^2$ and apply the trapezoid method to the resulting integral.

19. (a) What does it mean to say that an integral does not have an *elementary* antiderivative?

    (b) What is the relevance of part (a) to this subject?

20. The degree of precision of Boole's rule is 5. If we use it with $n = 100$ rectangles to integrate $\int_0^3 (x^4 - x + 1)\,dx$, which of the following is true about the error between the exact integral and the Boole's rule approximation? (a) The error is 0. (b) The error is on the order of $1/n^6$ but not 0. (c) The error term has not been theoretically determined (it is an open problem).

21. What is the degree of precision of the Romberg approximation R_{96}?

22. We have covered two-point and three-point Gaussian quadrature. What rule (that we had also talked about) is one-point Gaussian quadrature equivalent to? Explain.

23. If you use the midpoint rule to estimate $\int_1^4 \sin(x^3)\,dx$ with $n = 40$ rectangles, to four decimal places you will get .1973, which is off by .007 from the exact answer .2043.

    (a) What is the error bound value given by the formula on page 59 of the notes)? [Hint: you will probably want to use Wolfram Alpha to find $|\max f''|$.]

    (b) Why is this not equal to .007?

24. Recall that the second column of Romberg integration is actually equivalent to Simpson's rule. Show that. In particular, look a the first two entries in the first column, T_0 and T_1 and show that the corresponding Romberg entry in the second column matches Simpson's rule's $S_0 = \frac{h}{6}(f(a) + 4f(m) + f(b))$ (where $m = (b+a)/2$).

25. Use one of the integration methods we have covered to estimate the value of π correct to several decimal places. To do this, you will need to find a definite integral whose exact value is π or some multiple of π.

26. Recall the "curse of dimensionality". Suppose we need to approximate $\int\int\cdots\int f(x_1, x_2, \ldots, x_{27})\,dV$, where there are 27 integral signs. Assume the region of integration is the unit 27-dimensional hypercube, and we want to divide it up using a mesh of size .01 in all dimensions. If the numerical method we use requires one function evaluation for each piece of the mesh, how many function evaluations will be needed?

27. Use a numerical method to estimate $\int_2^\infty e^{\sqrt{x}-x}\,dx$ correct to within .001 by first using a change of variables. The exact answer is .77976 to 5 significant figures.

28. The function $\text{Li}(x) = \int_2^x \frac{1}{\ln t}\,dt$ provides a remarkably good estimate for the number of primes less than x. Write a spreadsheet that uses Simpson's rule with 100 parabolic rectangles to estimate Li(10000). [Note that there are exactly 1229 primes less than 10,000.]

29. Included below is Python code to plot a function parametrically. Currently the code plots $x = \cos t$ and $y = \sin t$, which gives a circle. Change the code so that it plots $x = \int_0^t \cos(u^2)\,du$ and $y = \int_0^t \sin(u^2)\,du$. Use the trapezoid method code provided to do this. You should get a pretty interesting shape.

```
from tkinter import *
from math import *

def trap(f, a, b, n):
    dx = (b-a)/n
    return dx/2*(f(a)+f(b) + 2*sum(f(a+i*dx) for i in range(1,n)))

def plot():
    t = -10
    oldx = func1(t) * 100 + 200
    oldy = func2(t) * -100 + 200
    while t < 10:
        t += .01
        x = func1(t) * 100 + 200
        y = func2(t) * -100 + 200
        canvas.create_line(oldx, oldy, x, y)
        oldx = x
        oldy = y

func1 = lambda t:cos(t)
func2 = lambda t:sin(t)

root = Tk()
canvas = Canvas(width=400, height=400, bg='white')
canvas.grid(row=0, column=0)
plot()
```

30. Consider the non-elementary function $f(x) = \int_0^x \sqrt[3]{t^2-1}\,dt$. Use this definition along with the Python trapezoid method code below to estimate a nonzero root of $f(x)$ using Newton's method. You will want to use the `cbrt` function below to do cube roots as raising things to the 1/3 power won't work quite right. [Hints: The nonzero roots are near ± 1.87. Use FTC Part 1 to find $f'(x)$.]

```
def cbrt(x):
    return x**(1/3) if x>=0 else -(-x)**(1/3)

def trap(f, a, b, n):
    dx = (b-a)/n
    return dx/2*(f(a)+f(b) + 2*sum(f(a+i*dx) for i in range(1,n)))
```

31. Write a Python program that implements Simpson's rule in an a manner analogous to code for the trapezoid rule in these notes. Test it out on $\int_2^5 \sin x$ with $n = 100$ and compare with the exact answer.

32. (a) Modify the provided Python Monte Carlo program provided earlier to write a Python function that estimates $\int_a^b \int_c^d f(x,y)\,dy\,dx$.

    (b) Use it to estimate $\int_2^5 \int_3^7 (x^2+y^2)\,dy\,dx$. The exact answer is 472.

33. Recall that the big idea of the iterative trapezoid rule is that we cut the step size h in half at each step and that allows us to make use of the estimate from the previous step.

    There is also an iterative midpoint rule that is developed from the midpoint rule in a similar way that the iterative trapezoid rule is developed from the trapezoid rule. The major difference is that where the iterative trapezoid rule cuts the step size h in half at every step, the iterative midpoint rule has to cut the step size into a third (i.e. $h_{n+1} = h_n/3$).

    (a) Using an example, explain why cutting the step size in half won't work for the extended midpoint rule, but cutting it into a third does work.

    (b) Modify the included Python code provided earlier for the iterative trapezoid rule to implement the iterative midpoint rule. Be careful — several things have to change.

7.6 Exercises for Chapter 6

1. Consider the IVP $y' = t\sin y$, $y(0) = 1$, $t \in [0,4]$. Estimate a solution to it by hand with the following methods (using a calculator to help with the computations):

    (a) Euler's Method by hand with $h = 2$

    (b) Explicit trapezoid method by hand with $h = 2$

    (c) Midpoint Method by hand with $h = 2$

    (d) RK4 by hand with $h = 4$

2. Consider the IVP $y' = -5y$, $y(1) = .25$, $t \in [1,2]$. Estimate a solution to it by hand with the following methods (using a calculator to help with the computations):

    (a) Use the backward (implicit) Euler method with $h = 1$ to approximate $y(2)$.

    (b) Use the predictor-corrector method using Euler's method as the predictor and the backward Euler method as the corrector with $h = 1$ to approximate $y(2)$.

    (c) Use the Adams Bashforth 2-step method with $h = 1$ to approximate $y(3)$, given $y(1) = .25$ and $y(2) = .002$.

3. Consider the IVP $y' = 2ty$, $y(2) = 1$, $t \in [2,6]$. Estimate a solution to it by hand with the following methods (using a calculator to help with the computations):

    (a) Euler's method with $h = 2$

    (b) The midpoint method with $h = 4$

    (c) RK4 with $h = 4$

    (d) AB2 with $h = 2$ using $y(4)$ as the Euler's method estimate of $y(4)$ (so you know $y(2)$ and $y(4)$ and are using them to find $y(6)$ with the AB2 formula)

    (e) Backward (implicit) Euler with $h = 4$

    (f) The Euler/Backward Euler predictor corrector with $h = 4$

4. Consider the system $x' = 2y + t$, $y' = xy$, $x(0) = 2$, $y(0) = -1$, $t \in [0,4]$. Estimate a solution to it by hand with the following methods (using a calculator to help with the computations):

    (a) Euler's method

    (b) Explicit trapezoid method (be careful)

5. Rewrite $y''' + 2y'' + \sin(y') + y = e^x$ as a system of first order equations.

6. Write $y^{(4)} = ty'' + 3y' - e^t y$ as a system of first order ODEs.

7. Use Euler's method with step size $h = 2$ to estimate the solution to $x' = 2y + t$, $y' = xy$, $x(0) = 5$, $y(0) = 3$, $t \in [0, 2]$.

8. Use Euler's method with step size $h = 2$ to estimate the solution to $x' = 2yt$, $y' = y - x - 2t$, $x(0) = 3$, $y(0) = 4$, $t \in [0, 2]$.

9. Use Euler's Method with step size $h = 2$ to estimate the solution to $y'' = ty^2$, $y(4) = 3$, $y'(4) = 5$, $t \in [4, 6]$.

10. Use Euler's method with $h = 1$ to solve $y'' - t(y')^2 + \sin(y) = 0$, $y(2) = 3$, $y'(2) = 5$, $t \in [2, 3]$.

11. Below shows the result of running the leapfrog method with step sizes $h/2$ through $h/16$. Complete the next two columns of the table that would be produced by extrapolating these values.

Step size	Estimate	First Extrapolation	Second Extrapolation
$h/2$	3.4641	—	—
$h/4$	3.2594		—
$h/6$	3.2064		
$h/8$	3.1839		
$h/10$	3.1720		
$h/12$	3.1648		
$h/14$	3.1600		
$h/16$	3.1567		

12. For the ODE extrapolation method, suppose we are solving the IVP $y' = f(t, y)$, $y(2) = y_0$, $t \in [2, 3]$ with $h = .1$.

    (a) For the first subinterval, the 5th leapfrog step will evaluate the function at which values of t?

    (b) Show below are the first few entries in the extrapolation table. Fill in the missing entries.

$h/2$	.8233			
$h/4$	.8820			
$h/6$	.9015			
$h/8$	.9098			

13. What is a differential equation? Why do we need all these methods for numerically solving them?

14. What is an IVP?

15. Rewrite the integration problem $\int_2^5 e^{x^2}\,dx$ as an initial value (differential equation) problem.

16. Arrange the following numerical ODE methods in increasing order of accuracy: Euler's method, RK4, Midpoint method

17. Explain, using the ideas of slope fields, how the midpoint method is an improvement on Euler's method.

18. Explain, using the ideas of slope fields, where all the parts of the RK4 method come from.

19. Explain geometrically how Euler's method works and describe how the formula for Euler's method relates to your geometrical explanation.

20. Explain, using the ideas of slope fields, how the midpoint method is an improvement on Euler's method.

21. Explain why global error for a numerical ODE method is more than just a sum of local errors.

22. How a does predictor-corrector method avoid the root-finding step in the implicit method that is part of the predictor-corrector pair?

23. Explain the ideas behind how the implicit (backward) Euler method works and how we can develop a predictor-corrector method based on it.

24. Give one benefit that the Adams Bashforth 2-step method has over the midpoint method.

25. Give one advantage multistep methods have over single-step methods and one advantage that single-step methods have over multistep methods.

26. Implicit methods are mostly used for stiff equations, but they can be used for non-stiff equations. But usually explicit methods are preferred in that case. Why?

27. Roughly, what makes "stiff" ODEs difficult to solve numerically, and what type of numerical methods are recommended for solving them?

28. One adaptive step-size ODE method we talked about starts by computing the Euler and Midpoint estimates for the same h value.

    (a) What value does it compute from there?

    (b) How does it use that value to decide on a new h?

29. True or False (give a reason) — The RK4 method can be used to estimate $\int_0^1 \sqrt[3]{1+x^2}\,dx$.

30. In your own words, describe the basic ideas of the extrapolation method for solving differential equations that we covered in these notes. Be sure to discuss the similarities to and differences from Romberg integration.

31. Suppose we want to use the extrapolation method to estimate the solution to an IVP on the interval $[0,3]$ with a step size of $h = 1$. To do this, we will end up calling the leapfrog method precisely how many times?

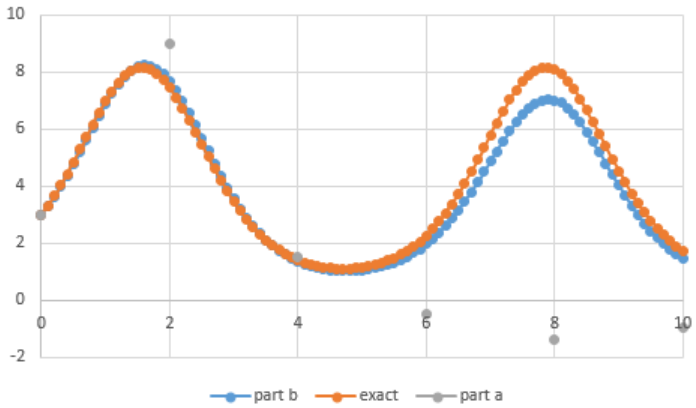
32. Consider the following IVP:

$$\begin{cases} y' = y\cos t \\ y(0) = 3 \\ t \in [0,10]. \end{cases}$$

    (a) Use Euler's method with $h = 2$. Do this by hand (with a calculator to help with the computations). Show all your work.

    (b) Use a spreadsheet to implement Euler's method to numerically solve the IVP using $h = .1$.

    (c) The exact solution is $y = 3e^{\sin t}$. Use a spreadsheet to plot your solutions from parts (a) and (b) on the same graph as the exact solution. Shown below is what your plot should look like.



33. Consider the IVP $y' = t\sin y$, $y(0) = 1$, $t \in [0,4]$. Estimate a solution to it using the following:

    (a) Euler's Method by spreadsheet with $h = .01$

    (b) Explicit trapezoid method by spreadsheet with $h = .01$

    (c) Midpoint Method by spreadsheet with $h = .01$

    (d) RK4 by spreadsheet with $h = .01$

34. For the system $x' = 2y + t$, $y' = xy$, $x(0) = 5$, $y(0) = 3$, $t \in [0,2]$, use the following to estimate a solution:

    (a) Euler's method by spreadsheet with $h = .01$

(b) Midpoint method by spreadsheet with $h = .01$

35. For the IVP $y'' = ty^2$, $y(4) = 3$, $y'(4) = 5$, $t \in [4,6]$, use the following to estimate a solution:

(a) Euler's method by spreadsheet with $h = .01$

(b) Midpoint method by spreadsheet with $h = .01$

36. Here is the Adams-Bashforth four-step method:

$$t_{n+1} = t_n + h$$
$$y_{n+1} = y_n + h\frac{55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}}{24}.$$

Here $f_i = f(t_i, y_i)$. It requires four initial points to get started. You can use the midpoint rule or RK4 to generate them. Modify the Adams-Bashforth two-step program provided earlier to implement the four-step method.

37. In Section 6.8 there is a predictor-correct method using the Adams-Bashforth four-step method along with the Adams-Moulton three-step method. Implement this method in Python.

38. Modify the backward Euler program given earlier to implement the implicit trapezoid method.

Index