Computational Physics and Numerical Analysis
Class Notes

Leon Hostetler

July 12, 2019
Version 0.51

Contents

Preface

About These Notes

These are primarily my class notes from two courses on numerical analysis (MAD3703 and MAD4704) taught by Professor Mark Sussman at Florida State University. The primary textbook used was *Numerical Analysis*, 8th edition by R. Burden and J.D. Faires. I have also included some notes from a course on computational physics (PHZ4151C) taught by Professor Paul Eugenio also at Florida State University. The textbook used in that class was *Computational Physics* by Mark Newman.

Numerical analysis is primarily focused on the study of numerical algorithms—their efficiency and complexity, and computational physics is primarily focused on the application of those algorithms to physical processes.

My class notes can be found at `www.leonhostetler.com/classnotes`

Please bear in mind that these notes will contain errors. Any errors are certainly my own. If you find one, please email me at leonhostetler@gmail.com with the name of the class notes, the page on which the error is found, and the nature of the error.

Updates

Last Updated: July 12, 2019
Version 0.51 (July 11, 2019): Updated layout
Version 0.50 (May 16, 2017): Initial upload

Chapter 1

Series Approximations

1.1 Taylor Series

For any smooth function $f(x)$, **Taylor's theorem** states that we can write it as

$$\boxed{f(x) = P_n(x) + R_n(x),}$$

where

$$\boxed{P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i,}$$

is the polynomial expansion of $f(x)$ about x_0, and

$$\boxed{R_n(x) = \frac{f^{(n+1)}\left(\xi(x)\right)}{(n+1)!}(x - x_0)^{n+1},}$$

is the remainder term. This is just the $n+1$ term of the Taylor series with x replaced by $\xi(x)$ in the derivative. Any smooth function can be written as a polynomial function plus a remainder term. Note that $\xi(x)$ is a number (that depends on x) between x and x_0. In general, we don't know what $\xi(x)$ is, but we can often still use $R_n(x)$ to get a bound on the error.

Taylor's theorem requires that the function be smooth near x_0. That is, $f(x)$ and all of its derivatives must be defined between x_0 and the point x at which the function is to be approximated.

For example, to find the nth order Taylor series approximation of $\ln x$ about $x_0 = 1$, we start by making the following table:

i	$f^{(i)}(x)$	$f^{(i)}(x_0)$
0	$\ln x$	0
1	$\frac{1}{x}$	1
2	$-\frac{1}{x^2}$	-1
3	$\frac{2}{x^3}$	2
4	$-\frac{6}{x^6}$	-6
5	$\frac{24}{x^5}$	24

Next, we determine the pattern. In this case, we see that

$$f^{(i)}(x) = \frac{(-1)^{i+1}(i-1)!}{x^i}, \quad \text{for } i > 0,$$

so

$$f^{(i)}(1) = (-1)^{i+1}(i-1)!, \quad \text{for } i > 0.$$

Taylor's theorem gives us

$$P_n(x) = 0 + \sum_{i=1}^{n} \frac{(-1)^{i+1}(i-1)!}{i!}(x-1)^i = \sum_{i=1}^{n} \frac{(-1)^{i+1}}{i}(x-1)^i,$$

for the polynomial expansion of $\ln x$ about $x_0 = 1$, and a remainder term

$$R_n(x) = \frac{(-1)^{n+2}n!}{[\xi(x)]^{n+1}} \frac{1}{(n+1)!}(x-1)^{n+1} = \frac{(-1)^{n+2}}{[\xi(x)]^{n+1}\,(n+1)}(x-1)^{n+1}.$$

Notice that for small n, the error is quite large.

The zeroth order Taylor approximation for $\ln x$ at $x_0 = 1$ is just

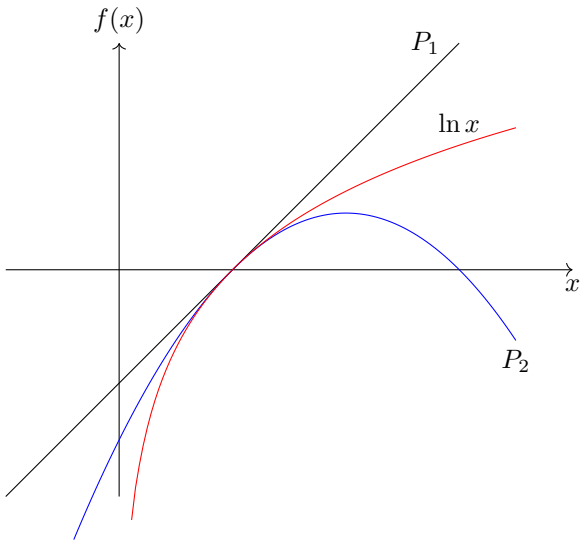
$$P_0 = 0.$$

The first order Taylor approximation is the tangent linear approximation

$$P_1 = x - 1.$$

The second order Taylor approximation is

$$P_2 = x - 1 - \frac{1}{2}(x-1)^2.$$



The **ratio test** tells us that a series converges if

$$\lim_{n\to\infty} \left| \frac{a_{n+1}}{a_n} \right| < 1,$$

and diverges if

$$\lim_{n\to\infty} \left| \frac{a_{n+1}}{a_n} \right| > 1.$$

In our case,

$$\lim_{n\to\infty}\left|\frac{a_{n+1}}{a_n}\right| = \lim_{n\to\infty}\left|\frac{(-1)^{n+2}n(x-1)^{n+1}}{(n+1)(-1)^{n+1}(x-1)^n}\right| = \lim_{n\to\infty}\left|\frac{(-1)^{n+2}}{(-1)^{n+1}}\right|\left|\frac{n(x-1)}{(n+1)}\right|$$
$$= |x-1|\lim_{n\to\infty}\left|\frac{n}{(n+1)}\right| = |x-1|.$$

By the ratio test,

$$\ln x = \sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{i}(x-1)^i,$$

converges for $|x-1|<1$ or for $0<x<2$, and diverges for $|x-1|>1$ or $x<0$ and $x>2$. The ratio test doesn't tell us what happens at $x=0$ or $x=2$, but we know that $\ln 0$ is undefined, so $x=0$ is excluded from convergence. By the alternating series test, we know that

$$\ln 2 = \sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{i},$$

is convergent. In summary, our series converges for $0<x\leq 2$.

As n is increased, the approximation gets better and better provided that the series converges. That is, for the values of x for which the series converges to $f(x)=\ln x$, Taylor's theorem implies that

$$\lim_{n\to\infty} R_n(x) = 0.$$

How fast does the approximation approach the actual solution? What is the bound on the error when $P_n(x)$ is used to approximate $\ln x$? To answer these questions, we need to analyze the error term

$$R_n(x) = \frac{(-1)^{n+2}}{[\xi(x)]^{n+1}\,(n+1)}(x-1)^{n+1}.$$

By Taylor's theorem, we know that $\lim_{n\to\infty} R_n(x)=0$ for $0<x\leq 2$. Note that at $x=1$, the error is zero.

Suppose that $1\leq x\leq 2$. Then the bound on the error is given by

$$\max|R_n(x)| = \max\left|\frac{(x-1)^{n+1}}{[\xi(x)]^{n+1}\,(n+1)}\right|.$$

Ordinarily, to find the maximum of a function, we might obtain the critical points by finding where the first derivative is zero. We can't use that here since we don't know what $\xi(x)$ is. However, by using the property

$$\boxed{\max|f(x)g(x)| \leq \max|f(x)|\max|g(x)|,}$$

we can bound the error term as

$$\max|R_n(x)| \leq \max\left|\frac{1}{[\xi(x)]^{n+1}}\right|\max\left|\frac{(x-1)^{n+1}}{n+1}\right|.$$

We know that $x_0<\xi(x)<x$, so the maximum value of $1/[\xi(x)]^{n+1}$ is when $\xi(x)$ is as small as possible $\xi(x)=1$, then $\max\left|1/[\xi(x)]^{n+1}\right|=1$. The second function, $(x-1)^{n+1}/(n+1)$ is maximized when x is as large as possible, so $x=2$. Therefore,

$$\max|R_n(x)| \leq \frac{1}{n+1},$$

Tip

A direct application of the ratio test fails if every other term is zero as with cosine and sine. In such cases, instead of using consecutive terms, you just use consecutive nonzero terms.

Tip

If you're given some error tolerance ε and you have an upper bound B on the error $R_n(x)$, but you cannot solve $B \leq \varepsilon$ explicitly for n, you can always try different values of n until you find one that is just large enough to satisfy the error tolerance.

or

$$R_n(x) \leq \frac{1}{n+1}, \quad \text{for } 1 \leq x \leq 2.$$

This is a bound on our error, but not a very good one.

For example, how large does n have to be so that the error is less than or equal to 0.01?

$$\frac{1}{n+1} \leq 0.01 \implies n \geq 99.$$

So to approximate $\ln x$ in the interval $1 \leq x \leq 2$, we need $n = 99$ terms in $P_n(x)$ if our error tolerance is 0.01.

Example 1.1.1

Compare the actual error to the error bound when $n = 2$ and $x = 3/2$.

The condition $n = 2$ means our approximation contains the first two terms of the Taylor series

$$P_2 = x - 1 - \frac{1}{2}(x-1)^2.$$

Our approximation of $\ln\left(\frac{3}{2}\right)$ is

$$P_2\left(\frac{3}{2}\right) = \frac{3}{8}.$$

Our error bound is

$$\frac{1}{n+1} = \frac{1}{3}.$$

That is, we know by the error bound that the true value is

$$\frac{3}{8} - \frac{1}{3} \leq \ln\left(\frac{3}{2}\right) \leq \frac{3}{8} + \frac{1}{3}.$$

The actual value is

$$\ln\left(\frac{3}{2}\right) = 0.405465,$$

to six decimal places, so the actual error in our approximation is

$$\left|0.405465 - \frac{3}{8}\right| = 0.030465.$$

Notice that the actual error in our approximation is much smaller than our error bound of 1/3. That is, our error bound is not very good.

What if we want to approximate $\ln x$ in the interval $0 < x < 1$? We know that our series is convergent in this interval, but we only bounded the error for the interval $1 \leq x \leq 2$. We have a problem when we try to bound $R_n(x)$ for $0 < x < 1$. We know that $0 < x < \xi(x) < 1$, so $[\xi(x)]^{n+1}$ grows small as $n \to \infty$, so $R_n(x)$ grows larger.

The procedure for approximating the value of a function $f(x)$ near x_0 within some given error tolerance is as follows.

1. Make a table of the derivatives of $f(x)$ evaluated at x_0.
2. Use these to determine the Taylor expansion $P_n(x)$ and the error term $R_n(x)$.
3. Determine for what values of x that $P_n(x)$ converges to $f(x)$. The Taylor expansion can only be used to approximate $f(x)$ for these values of x.
4. Given some interval for x, obtain an upper bound on the error using $R_n(x)$.
5. Given an error tolerance, determine the number of terms n needed to approximate $f(x)$ to within the error tolerance.

For a Python program that calculates the partial sum of a Taylor series, see page 104.

For a Python program that plots a function along with its nth-order Taylor approximation, see page 104.

1.2 Geometric Series

A geometric series has the general form

$$\boxed{\frac{a}{1-x} = \sum_{i=0}^{\infty} ax^i = a + ax + ax^2 + \cdots .}$$

The Taylor expansion of $a/(x-1)$ about $x_0 = 0$, gives us the exact same series, as expected. The ratio test gives us

$$\lim_{n\to\infty} \left| \frac{ax^{n+1}}{ax^n} \right| = |x|,$$

so the series converges if $-1 < x < 1$.

For $f(x) = a/(1-x)$, if we make our table of $f^{(i)}(x)$ and $f^{(i)}(x_0)$, we find that

$$f^{(i)}(x) = \frac{i!\, a}{(1-x)^{i+1}}, \qquad f^{(i)}(0) = i!\, a,$$

so the Taylor expansion for $x_0 = 0$ is

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(0)}{i!} x^i = \sum_{i=0}^{n} ax^i.$$

Note that is the same as the geometric series but truncated at n terms. The remainder term by the Taylor theorem is

$$R_n(x) = \frac{f^{(n+1)}\left(\xi(x)\right) x^{n+1}}{(n+1)!} = \frac{ax^{n+1}}{\left(1-\xi(x)\right)^{n+2}}.$$

We know that our series is convergent for $-1 < x < 1$, however, the remainder term cannot be bounded for all values of x for which the series converges. Suppose $x = 0.9$. Then we know that $0 < \xi(x) < 0.9$. Then

$$\max |R_n(x)| = \max \left| \frac{ax^{n+1}}{\left(1-\xi(x)\right)^{n+2}} \right|.$$

The maximum value for the expression on the right occurs when the denominator is minimized, and that occurs when $\xi(x)$ is maximized or $\xi(x) = 0.9$.

$$\max |R_n(x)| = \left| \frac{a(0.9)^{n+1}}{(0.1)^{n+2}} \right| = \frac{a(0.9)^{n+1}}{(0.1)^{n+2}} = \frac{a(0.9)^{n+1}}{(0.1)\,(0.1)^{n+1}} = 10a9^{n+1}.$$

So

$$R_n(x) \leq 10a9^{n+1}.$$

However, this error bound is not going to zero. That is, the more terms that are added to our approximation, the larger the error bound grows. But we know that the actual error approaches zero as more terms are added provided that $-1 < x < 1$. Therefore, this remainder term is useless.

Fortunately, for geometric series we can construct a better error term. If $f(x) = a/(1-x)$, then the error term is the difference between the full geometric series and the polynomial truncation. That is,

$$\begin{aligned} R_n(x) = f(x) - P_n(x) &= \sum_{i=0}^{\infty} ax^i - \sum_{i=0}^{n} ax^i = \sum_{i=n+1}^{\infty} ax^i \\ &= ax^{n+1} + ax^{n+2} + ax^{n+3} + \cdots = x^{n+1}\left[a + ax + ax^2 + \cdots\right]. \end{aligned}$$

Simplifying, we get

$$\boxed{R_n(x) = x^{n+1}\frac{a}{1-x}.}$$

For this remainder term we see that $R_n(x) \to 0$ as $n \to \infty$ provided that $-1 < x < 1$. The beauty of this remainder term is that we don't have to deal with an unknown quantity $\xi(x)$. Notice that as $x \to 1$, the error gets larger. When $x = 1$, the error is infinite.

Suppose our error tolerance is some small number ε. That is, we want

$$f(x) = P_n(x) \pm \varepsilon,$$

for some interval $|x| < x_1$ within the interval of convergence. How large a value of n do we need?

$$\begin{aligned} \left| x_1^{n+1}\frac{a}{1-x_1} \right| &\leq \varepsilon \\ x_1^{n+1} &\leq \frac{\varepsilon(1-x_1)}{a} \\ n &\geq \frac{\ln\left(\frac{\varepsilon(1-x_1)}{a}\right)}{\ln x_1} - 1. \end{aligned}$$

Chapter 2

Computational Errors

- Logic error
- Random error
- Approximation errors (e.g. Taylor series)
- Range error
- Round-off error

What is the machine accuracy?

2.1 Roundoff Error

Try the following pseudocode on your computer.

```
x = 1.0
eps = 1.0
for i = 1, ..., 30
   print i
   print x
   print eps

   if x == x + eps
      print ‘‘Failure’’
   else
      print ‘‘Success’’

   eps = eps/10.0
```

You should get a "failure" by about $i = 18$. You start off with $x = \varepsilon = 1$ then you repeatedly divide ε by 10.0. At some point, the quantity ε will have more zeros behind the decimal place than your computer can handle and it will calculate $x + \varepsilon$ as being the same thing as x. At that point, the program stops and prints "failure".

We can expect a computer to perform 16-digit rounding.

Roundoff error occurs frequently when we subtract very nearly equal numbers.

Chapter 3

Floating Point Arithmetic

Floating point arithmetic is in contrast to symbolic arithmetic.

In **decimal arithmetic**, a number is written in the form

$$d_1.d_2d_3\cdots d_k \times 10^n = \left(d_1 \times 10^0 + d_2 \times 10^{-1} + d_3 \times 10^{-2} + \cdots + d_k \times 10^{-k+1}\right) \times 10^n,$$

where the digits are $0 \leq d_j \leq 9$. For example, 6.73×10^2 means $\left(6 \times 10^0 + 7 \times 10^{-1} + 3 \times 10^{-2}\right)\times 10^2$.

In **binary arithmetic**, a number is written in the form

$$b_1.b_2b_3\cdots b_k \times 2^n = \left(b_1 \times 2^0 + b_2 \times 2^{-1} + b_3 \times 2^{-2} + \cdots + b_k \times 2^{-k+1}\right) \times 2^n,$$

where the digits are 0 and 1. For example, 1.01×2^3 means $\left(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}\right)\times 2^3$. Notice that if you perform these calculations, you get 11, the value of the number in decimal notation.

To convert a decimal number to the equivalent binary representation, we subtract the largest power of two that we can. Then we subtract the largest power of 2 from the remainder and so on, until we have no more remainder left. Then our powers of 2 that we used give us the placements of the 1's in our binary representation.

Example 3.0.1

Convert 37 (decimal) to its binary representation.

The largest power of 2 less than or equal to 37 is 5, and that gives us a remainder of 5. The largest power of two less than or equal to 5 is 2. Now we have a remainder of 1. The largest power of two less than or equal to 1 is 0. Now we have no more remainder left. In other words, we can say that

$$37 = 2^5 + 2^2 + 2^0.$$

The 5, 2, and 0 give us the placements of the 1's in our binary representation, so

$$37\,(\text{decimal}) = 100101\,(\text{binary}).$$

Example 3.0.2

Convert 0.1 from decimal to its binary representation.

The power of 2 that is less than or equal to 0.1 is -4 with a remainder of $0.1 - 0.0625 = 0.0375$. That is,

$$0.1 = 2^{-4} + 0.0375.$$

Now we repeat the process with the remainder 0.0375, and we find that the largest power of two less than or equal to this is -5. We repeat this process again, and we find that

$$0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 0.00234375.$$

We could continue this process indefinitely, because 0.1 (decimal) does not terminate in binary representation.

$$0.1 \text{ (decimal)} = 0.00011001\ldots \text{ (binary)}.$$

We can also write this as

$$0.1 \text{ (decimal)} = 1.1001\ldots \times 2^{-4} \text{ (binary)}.$$

In the example above, we found that we cannot fully represent 0.1 on a computer since it is a repeating "decimal" in binary representation. How are floating point numbers stored in a computer? The double precision floating point format uses 64 bits (8 bytes) to store a floating point number. Given a floating point x, we decompose it as

$$\boxed{x = (-1)^S 2^{C-1023}(1+f),}$$

where s is the sign bit, c is the "exponent", and f is the mantissa. In this format, the sign bit occupies a single bit. It is 0 if x is positive and 1 if x is negative. The exponent c occupies the next 11 bits, and the mantissa fills the remaining 52 bits.

Single precision numbers are stored using 32 bits. They use the sign bit, 8 bits for the exponent, and 23 bits for the mantissa.

To convert a decimal number to the form that is stored by a computer, we follow this procedure:

1. Convert the decimal to binary form.
2. Normalize the binary representation in the form $b_1.b_2b_3\cdots b_k \times 2^n$ such that it starts with a 1.
3. Find the mantissa f, which is the first 52 digits (for doubles) following the decimal point.
4. Determine the decimal value of C. If your normalized binary number is multiplied by 2^n, then $C - 1023 = n$.
5. Convert C to binary representation.
6. Write the number as stored by a computer, starting with the sign bit, followed by the 11 bits for the exponent, followed by the 52 bits for the mantissa.

Example 3.0.3

Write the decimal 0.1 as it is stored in a computer.
Recall that the binary representation is

$$0.1 \text{ (decimal)} = 0.0001\overline{1001} \text{ (binary)}.$$

Normalizing it gives us

$$0.1 \text{ (decimal)} = 1.\overline{1001} \times 2^{-4} \text{ (binary)}.$$

Since the number is positive, the sign bit is 0. The mantissa is the first 52 digits following the decimal point: 10011001...1001. The exponent is $C - 1023 = -4 \implies C = 1019$. Converting 1019 to binary gives us

$$1019 \text{ (decimal)} = 1111111011 \text{ (binary)},$$

since

$$1019 = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0.$$

This is only ten digits, and a computer stores this part in eleven bits, so we have

$$1019 \text{ (decimal)} = 01111111011 \text{ (binary)}.$$

Given a binary number like 1111111011 which has mostly ones, we can do a quick check on its value. This number has ten digits. The largest binary number that can be formed with ten digits is $2^{10} - 1 = 1023$–if all the digits are ones. From this, we just subtract the numbers corresponding to the digits that are not ones. In our case, only the third digit is not 1, and that digit corresponds to $2^2 = 4$, so our binary number 1111111011 is $1023 - 4 = 1019$ in decimal.

So the number stored in the computer is

$$\underline{0\overline{01111111011}}\underline{10011001...1001},$$

where the underline and overline distinguish the sign bit, the exponent, and the mantissa.

If p is the exact value and p^* is the approximation, then the **absolute error** is

$$|p - p^*|,$$

and the **relative error** is

$$\frac{|p - p^*|}{|p|}.$$

The relative error is typically preferred unless $p = 0$.

What is the largest possible error in a 64-bit floating point representation if using binary chopping? The largest possible error would occur if all digits after the chopping point are ones. The last digit in a 64-bit representation is at 2^{-52}. All the digits after that, that is, those corresponding to $2^{-53}, 2^{-54}, \ldots$ are chopped off. If they are all ones, then their total value is

$$\sum_{i=0}^{\infty} 2^{-53-i} = 2^{-53} \sum_{i=0}^{\infty} \frac{1}{2^i} = 2^{-53}(2) = 2^{-52}.$$

Suppose our normalized binary number has the form $1.b_1b_2b_3 \ldots \times 2^{-n}$. Then the largest absolute error due to chopping, which occurs when all the chopped digits are ones, is $2^{-52} \times 2^{-n} \approx (2^{-n}) \cdot 1.39 \times 10^{-17}$.

Example 3.0.4

Write the 64-bit computer representation of the decimal number 1.0.

The normalized binary representation is just

$$1.0 \text{ (decimal)} = 1.0 \times 2^0 \text{ (binary)}.$$

The number is positive, so the sign bit is zero. The exponent is $0 = C - 1023$, which tells us $C = 1023$. Converting this to decimal gives us

$$1023 \text{ (decimal)} = 01111111111 \text{ (binary)}.$$

Since all digits after the decimal point are zero, f is just a 52-bit string of zeros. So the binary representation as it is stored on a computer is

$$\underline{0\overline{01111111111}}\underline{000\ldots000}.$$

Note that the error is zero. This is the exact value of 1.0.

We now switch back to decimal numbers. The normalized form of a decimal number is

$$\pm 0.d_1 d_2 d_3 \dots d_k \times 10^n.$$

Notice that $1 \leq d_1 \leq 9$ but the rest are $0 \leq d_i \leq 9$. That is, the first digit cannot be a zero or it is not normalized. For example, the normalized version of 0.0045 is 0.45×10^{-2}.

There are two ways to restrict a number to the precision supported by a computer. Those two ways are **chopping** and **rounding**. Suppose our given number is

$$y = \pm 0.d_1 d_2 d_3 \dots d_k d_{k+1} d_{k+2} \dots \times 10^n,$$

but our computer system only supports k-digit arithmetic. Then k-digit chopping gives us

$$\text{chop}_k(y) = \pm 0.d_1 d_2 d_3 \dots d_k \times 10^n.$$

For k-digit rounding, there are two algorithms we can use

The first algorithm is

$$\text{round}_k(y) = \text{chop}_k\left(y + 5.0 \times 10^{-k+n-1}\right).$$

That is, given the number $y = \pm 0.d_1 d_2 d_3 \dots d_k d_{k+1} d_{k+2} \dots \times 10^n$, we first add $0.000 \dots 05 \times 10^n$ to y, where there are k zeros between the decimal point and the 5. Then we chop the resulting number at the kth digit.

The other method is the more intuitive method that we all know. If digit $d_{k+1} < 5$ then perform k-digit chopping on y. Otherwise, if $d_{k+1} \geq 5$, then add "1" to the d_k decimal place and then do the chopping.

3.1 Evaluating Polynomials

When evaluating a polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n,$$

you may have a loss of precision, especially if it's an alternating series, since you may be subtracting nearly equal numbers from each other. The straightforward "brute force" method for evaluating a polynomial is

```
p = 0.0
for i = 0, ... , n
   p = p + a_i * pow(x, i)
```

The complexity of this algorithm, if we only count the multiplies, is of order n^2.

Consider the polynomial

$$P_2(x) = -x^2 + 1.0001x + 0.0005,$$

and suppose we want to evaluate it at $x = 1$. Note that the exact solution is $P_2(1) = 0.0006$. The brute force algorithm will go as follows:

$$\begin{aligned} p &= 0.0 \\ p &= p + a_0 \\ p &= p + a_1 x \\ p &= p + a_2 x^2. \end{aligned}$$

If we use 4-digit chopping arithmetic, it will look like this:

$$\begin{aligned} p &= \text{chop}_4\left(0.0000\right) \\ p &= \text{chop}_4\left(p + \text{chop}_4\left(a_0\right)\right) \\ p &= \text{chop}_4\left(p + \text{chop}_4\left(a_1\right) \times \text{chop}_4\left(x\right)\right) \\ p &= \text{chop}_4\left(p + \text{chop}_4\left(a_2\right) \times \text{chop}_4\left(x^2\right)\right). \end{aligned}$$

When we plug in our numbers and evaluate, we get

$$\begin{aligned}
p &= \text{chop}_4\left(0.0000\right) = 0.0000 \\
p &= \text{chop}_4\left(0.0000 + \text{chop}_4\left(0.0005\right)\right) = 0.0005 \\
p &= \text{chop}_4\left(0.0005 + \text{chop}_4\left(1.0001\right) \times \text{chop}_4\left(1.000\right)\right) = 1.000 \\
p &= \text{chop}_4\left(1.000 + \text{chop}_4\left(-1.000\right) \times \text{chop}_4\left(1.000\right)\right) = 0.0000.
\end{aligned}$$

Our approximation is $P_2(1) \approx 0.0000$. The true answer is $P_2(1) = 0.0006$, so our relative error is

$$\frac{|0.0006 - 0.0000|}{|0.0006|} = 1.$$

Because of the 4-digit chopping, we lost all precision.

Horner's method is a better algorithm for evaluating polynomials since there is less loss of precision. Given a polynomial of the form

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

Horner's method is the equivalent of

$$P_n(x) = x(x(x(\cdots x(a_n x + a_{n-1}) + a_{n-2}) + a_{n-3}) + \cdots + a_0,$$

where there are a total of $n - 1$ x's in $x(x(x(\cdots x($. Note, Horner's method is also sometimes called **nested polynomial evaluation**.

For a polynomial of the form given above, the algorithm for Horner's method is as follows:

```
p = a_n
for i = n, n-1, n-2, ..., 1
   p = p*x + a_{i-1}
```

The complexity of this algorithm is only of order n.

Let's try Horner's method with our polynomial

$$P_2(x) = -x^2 + 1.0001x + 0.0005,$$

and evaluate it at $x = 1$. The algorithm will go as follows:

$$\begin{aligned}
p &= a_2 \\
p &= px + a_1 \\
p &= px + a_0.
\end{aligned}$$

If we use 4-digit chopping arithmetic, it will look like this:

$$\begin{aligned}
p &= \text{chop}_4\left(a_2\right) \\
p &= \text{chop}_4\left(\text{chop}_4\left(p\right) \times \text{chop}_4\left(x\right) + a_1\right) \\
p &= \text{chop}_4\left(\text{chop}_4\left(p\right) \times \text{chop}_4\left(x\right) + a_0\right).
\end{aligned}$$

When we plug in our numbers and evaluate, we get

$$\begin{aligned}
p &= \text{chop}_4\left(-1.000\right) = -1.000 \\
p &= \text{chop}_4\left(\text{chop}_4\left(-1.000\right) \times \text{chop}_4\left(1.000\right) + 1.0001\right) = 0.0000 \\
p &= \text{chop}_4\left(\text{chop}_4\left(0.0000\right) \times \text{chop}_4\left(1.0000\right) + 0.0005\right) = 0.0005.
\end{aligned}$$

This time our relative error is only

$$\frac{|0.0006 - 0.0005|}{|0.0006|} = 0.17.$$

3.2 Quadratic Formula

The well-known quadratic formula for a quadratic equation of the form $ax^2 + bx + c = 0$ is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

However, consider an equation like

$$0.001x^2 + 1000x + 0.001 = 0.$$

The actual solutions are approximately

$$x_1 = -1.000000000001000000000002 \cdots \times 10^{-6}$$
$$x_2 = -999999.999998999999999998999999 \cdots$$

which is essentially,

$$x_1 = -0.000001$$
$$x_2 = -1,000,000.$$

If we try a straightforward application of the quadratic formula with a calculator, we might get something like

$$x_1 = 0$$
$$x_2 = -1,000,000.$$

This is incorrect, and it is incorrect because the calculator rounds $1,000,000 - 0.0000004$ to $1,000,000$ before taking the square root. This rounding occurs, because the calculator doesn't use enough digits to handle a number like 999,999.999996. If you don't believe me, just type this number into a calculator like the TI-84 and hit enter. It will return the rounded number 1,000,000. Normally, this is okay, but in this case, it gives us an incorrect answer for our first root.

What if we try this on a computer. The python implementation of the quadratic formula might look like

```
def quadratic_formula(a, b, c):
    """
    This function computes the solutions to a quadratic equation using
    the regular form of the quadratic formula that everyone knows.
    """
    x1 = (-b + np.sqrt(b*b - 4*a*c))/(2*a)
    x2 = (-b - np.sqrt(b*b - 4*a*c))/(2*a)
    return x1, x2
```

This function gives us

```
(-9.999894245993346e-07, -999999.99999899999)
```

Notice that the absolute error of the first solution is about 10^{-5} while that of the second solution is effectively zero. Why the relatively large error in the first solution? The error occurs because for the first solution, we are subtracting two large and very nearly equal numbers.

Consider the following subtraction of two large and nearly equal numbers

$$1,000,000.000123456789 - 1,000,000 = 0.000123456789.$$

If you try this in your calculator, you will probably get 0.0001234. The relative error is 4.6×10^{-4}. But what happens when you add?

$$1,000,000.000123456789 + 1,000,000.$$

Now, the actual answer is

$$2,000,000.000123456789,$$

but if you try it in your calculator, you will probably get 2,000,000. Here, the relative error is seven orders of magnitude smaller

$$\frac{0.000123456789}{2,000,000.000123456789} \approx \frac{0.0001234}{2,000,000} = 6 \times 10^{-11}.$$

In the quadratic formula, if $b^2 >> 4ac$ then one of the solutions will occur with the subtraction of two large nearly equal numbers. However, by multiplying the numerator and the denominator by $-b \mp \sqrt{b^2 - 4ac}$ we get an alternative formulation of the quadratic formula

$$x = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}.$$

The python implementation of this might look like:

```
def quadratic_formula_modified(a, b, c):
    """
    This function computes the solutions to a quadratic equation using
    a modified form of the quadratic formula.
    """
    x1 = (-2*c)/(b + np.sqrt(b*b - 4*a*c))
    x2 = (-2*c)/(b - np.sqrt(b*b - 4*a*c))
    return x1, x2
```

This function gives us

```
(-1.0000000000010001e-06, -1000010.5755125057)
```

Now, the first solution is essentially exact, and the second solution has a relatively large error. This is because the rearranged form of the quadratic equation associates the malicious subtraction with the other root.

The proper implementation of the quadratic formula on a computer system is a combination of both of these.

Chapter 4

Root Finding

We are often interested in minimizing or maximizing some function. This involves finding the roots of the first derivative of the function, and finding the roots is often the hard part of the problem. For example, if your function is fifth degree polynomial, then to optimize it, you need to find the roots of a fourth degree polynomial.

Another application of root-finding is to approximate irrational numbers. For example, the approximation of the roots of $f(x) = x^2 - 2$ give us an approximation of $\sqrt{2}$.

4.1 Bisection Method

A **bracketing interval** of a function $f(x)$, is an interval $[a, b]$ such that $f(a) \cdot f(b) < 0$. If you have a bracketing interval $[a, b]$ of $f(x)$ then $f(x)$ has at least one root in the bracketing interval.

For example, $[0, 10]$ is a bracketing interval for the function $f(x) = x^2 - 2x - 3$ because $f(0) \cdot f(10) = (-3) \cdot (77) < 0$. Since $f(x)$ is continuous, the intermediate value theorem implies that $f(x)$ has at least one root in the interval $[0, 10]$.

To use the bisection method to find the root of a function, you start with a bracketing interval of the function. Then to narrow in on the root, the bisection method chooses the midpoint of the bracketing interval and checks the sign of the function there. This bisects the interval into two more intervals–one of which is another bracketing interval. Choose that sub-interval as your new bracketing interval and repeat. At each iteration, the bracketing interval gets smaller and the bisection method narrows down to the root. The error is basically the length of your final bracketing interval.

Suppose you are given a root-finding problem $f(x) = 0$ and a bracketing interval $[a, b]$ where $f(a)f(b) < 0$. If your error tolerance is ε, then your algorithm is as follows:

```
flag = False
while(|b - a| > epsilon) and (flag == False)
   fa = f(a)
   fb = f(b)
   c = (a + b)/2
   fc = f(c)

   if fc == 0.0
      flag = True
   else if (fa * fc < 0.0)
      b = c
   else if (fb * fc < 0.0)
      a = c
   else
      abort (‘‘Something went wrong’’)
```

After the algorithm has run, c will be your approximation of the root.

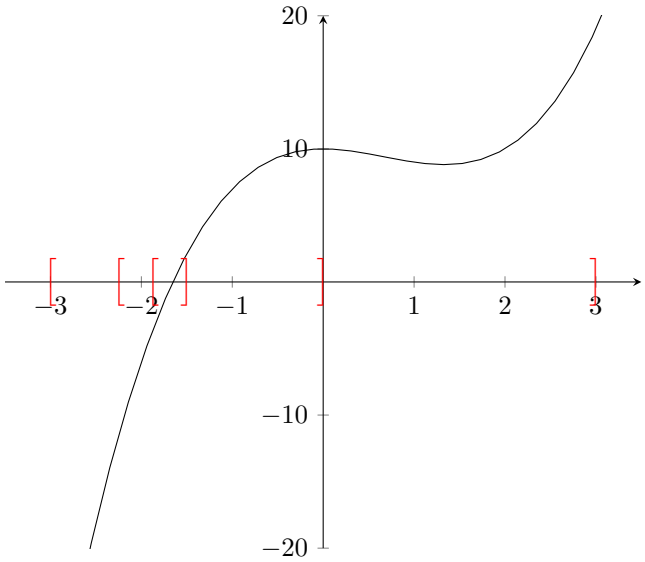
Suppose we are given the polynomial

$$f(x) = x^3 - 2x^2 + 10,$$

and we want to find the root(s). That is, we want to solve $f(x) = 0$. We can start with the bracketing interval $[-3, 3]$. This is a bracketing interval since $f(-3)f(3) = (-35)(19) < 0$.

1. Next, we compute the midpoint, which is $c = (3 + (-3))/2 = 0$. Now, we check the value of $f(c) = f(0) = 10$. Since this value is positive, our new bracketing interval is the left subinterval $[-3, 0]$.
2. Again, we compute the midpoint $c = (-3 + 0)/2 = -1.5$ and evaluate the function there. Since $f(-1.5) > 0$, our new subinterval is again the left half, $[-3, -1.5]$.
3. Again, we compute the midpoint $c = (-3-1.5)/2 = -2.25$ and evaluate the function there. Since $f(-2.25) < 0$, our new subinterval is now the right half, $[-2.25, -1.5]$.
4. Again, we compute the midpoint $c = (-2.25 - 1.5)/2 = -1.875$ and evaluate the function there. Since $f(-1.875) < 0$, our new subinterval is again the right half, $[-1.875, -1.5]$.

In the graph below, we can see how successive bracketing intervals are narrowing in on the root of $f(x)$.



Note: For a given bracketing interval, the bisection method will converge to a single root in the interval, even if there are multiple roots in the interval. To find multiple roots using the bisection method, you have to run the algorithm with different bracketing intervals.

The bisection method converges slowly. The fastest it converges is a factor of two since the error is halved every iteration. If p is a root and $[a_1, b_1]$ is your starting bracketing interval, then

$$|p - c_n| \leq \frac{|b_1 - a_1|}{2^n}.$$

Note that c_n is the midpoint of your nth bracketing interval. That is, it is your nth iteration approximation of the root p.

Suppose we want to reduce the error by a factor of 10^{15}. How many iterations of the bisection method are required? At $n = 1$, our error is

$$|p - c_1| \leq \frac{|b_1 - a_1|}{2}.$$

We want to find N such that

$$\frac{\frac{|b_1-a_1|}{2^N}}{\frac{|b_1-a_1|}{2}} \leq 10^{-15}.$$

This gives us

$$\frac{1}{2^{N-1}} \leq \frac{1}{10^{15}}.$$

Solving for N gives us

$$N \geq 51.$$

So it requires 51 iterations of the bisection method to reduce the error by a factor of 10^{15}. This is very slow.

The bisection method is not ideal for finding roots. It converges too slowly, and it is not applicable in multiple dimensions.

4.2 Newton's Method

Newton's method is an iterative root-finding method to solve $f(x) = 0$. We start with some guess $x^{(0)}$ of the root. Then assuming the function is approximately linear at $x^{(0)}$, you find the tangent linear approximation

$$y = f'(x^{(0)}) \left(x - x^{(0)}\right) + f(x^{(0)}).$$

Notice that the tangent linear approximation is also the first two terms of the Taylor expansion of $f(x)$. Our next guess of the root of $f(x)$, $x^{(1)}$, is the root of this tangent linear approximation. That is, we set it equal to zero

$$0 = f'(x^{(0)}) \left(x^{(1)} - x^{(0)}\right) + f(x^{(0)}),$$

then solve for $x^{(1)}$ to get

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

The iteration for Newton's method is then

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

The algorithm for Newton's method, given a function $f(x)$ and $f'(x)$, an initial guess of the root $x^{(0)}$, and an error tolerance ε, is as follows

```
k = 0
oldx = x^(0)
status = False
while status == False
   newx = oldx - f(oldx)/f'(oldx)
   oldx = newx

   if |f(oldx)| < epsilon
      status = True

   k = k + 1
```

The variable k is just a counter to keep track of the number of iterations done.

Let's try Newton's method with the polynomial

$$f(x) = x^3 - 2x^2 + 10,$$

and we want to find the root(s). That is, we want to solve $f(x) = 0$. Note that the derivative is

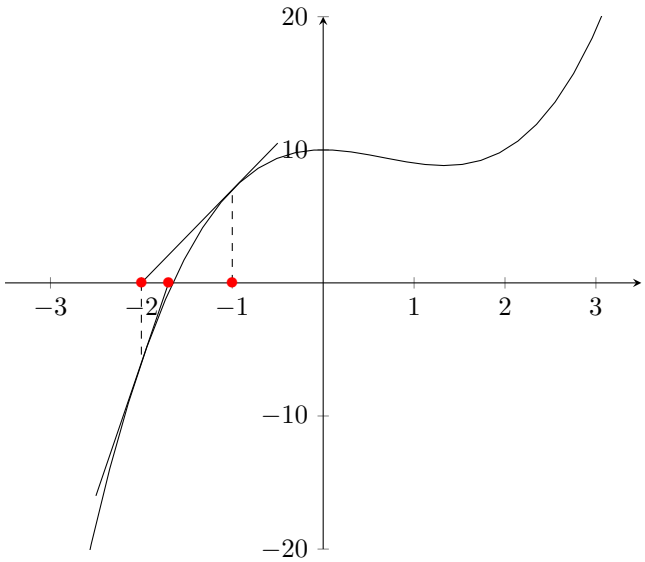
$$f'(x) = 3x^2 - 4x.$$

Our initial guess of the root is $x^{(0)} = -1$.

1. Our next guess is
$$x^{(1)} = -1 - \frac{f(-1)}{f'(-1)} = -2.$$
2. Our next guess is
$$x^{(2)} = -2 - \frac{f(-2)}{f'(-2)} = -1.7.$$

After only two iterations, we are already very close to the actual root at $x = -1.654$.

In the graph below, we can see how our successive approximations are narrowing in on the root of $f(x)$.



Note that Newton's method fails if the derivative at our initial point is zero. If the derivative is zero at our guess value, then the linear tangent approximation will be horizontal, and it will not have a root. In terms of the computer algorithm, this results in a division by zero error. A good strategy is to try Newton's method, but switch to the bisection method if it fails.

In general, Newton's method is preferred over the bisection method because it converges much faster.

A python implementation of Newton's method is given here:

```
def f(x):
    """
    Define the polynomial.
    """
    return 15*x - 70*x**3 + 63*x**5


def f_prime(x):
    """
    Define the derivative of the polynomial.
    """
    return 15 - 210*x**2 + 315*x**4
```

```
def newtons_method(x, tolerance):
    """
    This function returns a single root for the polynomial defined in f(x). Takes
        two parameters--a guess x and the error tolerance.
    """
    errormet = False

    while errormet is False:
        x_new = x - f(x)/f_prime(x)

        if abs(x_new - x) < tolerance:
            errormet = True

        x = x_new

    return x_new


root = newtons_method(1.0, 1.0e-10)
print(root)
```

A better version of this program can be found on page 105. That program finds all the roots of a polynomial by using Newton's method in combination with deflation.

4.3 Fixed-point Methods

Newton's method is a member of a whole class of root-finding methods called "fixed-point" methods. The bisection method, for example, is not generalizable to multiple dimensions, but the fixed-point methods are.

A number p is a **fixed-point** of a function $g(x)$ if $g(p) = p$.

A fixed point problem is a problem of the form

$$x = g(x).$$

Example 4.3.1

What are the fixed points of

$$g(x) = \frac{(x-2)^2}{10} + 2?$$

To find the fixed points, we can set the function equal to x and then factor it

$$\begin{aligned} x &= \frac{(x-2)^2}{10} + 2 \\ 0 &= x - 2 - \frac{(x-2)^2}{10} = (x-2)\left(1 - \frac{x-2}{10}\right). \end{aligned}$$

Solving each of these factors, we find that the fixed points are $p = 2, 12$. In other words, there are two places where $y = x$ intersects $g(x)$.

A root-finding problem is a problem of the form

$$f(x) = 0.$$

A fixed-point problem is a problem of the form

$$x = g(x).$$

Any root-finding problem can be cast as a fixed-point problem.

For example, $x^2 - 2 = 0$ is a root-finding problem, but we can recast it as a fixed-point problem in several different ways including

$$x = x + x^2 - 2$$
$$x = x - \frac{x^2 - 2}{2x}$$
$$x = \frac{\left(x + x^2 - 2\right) + \left(x - \frac{x^2-2}{2x}\right)}{2}.$$

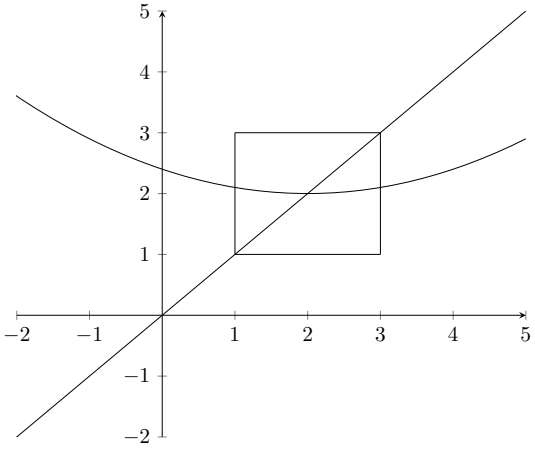
Notice that the first one is formed trivially by adding x to both sides of $x^2 - 2 = 0$. All of these have the fixed point $p = \pm\sqrt{2}$, but the last one, for example, also has the fixed point $x = 1/2$.

Theorem: Given the fixed point problem $x = g(x)$ where $a \leq x \leq b$, if $g(x) \in [a, b]$ whenever $x \in [a, b]$, and if $|g'(x)| \leq k < 1$ whenever $x \in [a, b]$, then there exists a unique fixed point p, such that $p \in [a, b]$.

What does this theorem mean? Consider the function

$$g(x) = \frac{(x-2)^2}{10} + 2,$$

that is graphed below. We have also graphed $y = x$ since a fixed point problem has the form $x = g(x)$. Notice that $1 \leq g(x) \leq 3$ whenever $1 \leq x \leq 3$. The fixed point $p = 2$ corresponds to the point where x intersects with $g(x)$ in the box. There is another fixed point $12 = g(12)$ that occurs outside the box (i.e. in another box).



The condition $|g'(x)| \leq k < 1$ is called the **contraction mapping** condition. It ensures that there is exactly one fixed point inside the box.

If we know there is a square box containing a fixed point p of $g(x)$, then the iterative method to find that fixed point is

$$\boxed{p_{k+1} = g(p_k),}$$

where p_n is our approximation of the fixed point after n iterations. For example, if we pick an x-value in the box, such as $p_0 = 1.3$, then

$$p_1 = g(1.3) = 2.049$$
$$p_2 = g(2.049) = 2.0002401.$$

Notice that the fixed point iteration converges quickly to the fixed point.

We can write the error as

$$|p_n - p| = |g(p_{n-1}) - g(p)|,$$

where the approximation of the fixed point p_n is equal to $g(p_{n-1})$ by definition of fixed point iteration, and $p = g(p)$ by definition of a fixed point. We can write the right side as

$$|g(p_{n-1}) - g(p)| = |g'(\xi)||p_{n-1} - p|,$$

where $g'(\xi)$ is between p_{n-1} and p by the squeeze theorem. Then

$$\begin{aligned} |p_n - p| &= |g'(\xi)||p_{n-1} - p| \\ &\leq k|p_{n-1} - p| \\ &\leq kk|p_{n-2} - p| \\ &\leq kkk|p_{n-3} - p| \\ &\leq k^n \max\{|p_0 - a|, |b - p_0|\}. \end{aligned}$$

So after n iterations of the fixed point method, our bound on the error is

$$|p_n - p| \leq k^n \max\{p_0 - a, b - p_0\}.$$

For Newton's method, which is a specific case of the fixed point method, to solve the fixed point problem cast as the root-finding problem $f(x) = 0$, the iteration is

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}.$$

This is equivalent to

$$p_{n+1} = g(p_n),$$

where

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

Then a useful theorem is: For a root p of $f(x)$ with multiplicity 1, there exists an interval $[a, b] = [p - \delta, p + \delta]$ about the root such that $g(x) \in [a, b]$ whenever $x \in [a, b]$ and $|g'(x)| \leq k < 1$ for any $x \in [a, b]$ and $g'(p) = 0$.

Note that for a root finding problem $f(x) = 0$,

$$g'(x) = 1 - \frac{(f'(x))^2 - f''(x)f(x)}{(f'(x))^2} = \frac{f''(x)f(x)}{(f'(x))^2} = 0,$$

because $f(x) = 0$.

Example 4.3.2

Solve the fixed point problem

$$g(x) = \frac{(x-2)^2}{10} + 2,$$

for $[a, b] = [1, 3]$.

We start by verifying that the fixed point conditions are satisfied. To show that $g(x) \in [1, 3]$ whenever $x \in [1, 3]$, we check that the endpoints $g(1) = 2.1$ and $g(3) = 2.1$ are both in $[1, 3]$. Then, we check that the local extrema of $g(x)$ are inside the box. Differentiating gives us $g'(x) = (x-2)/5$ which gives us the critical point $x = 2$. Checking this, we see that $g(2) = 2$ is also in the box.

In general, we need to show that $g(x) \in [a, b]$ for the endpoints and all the critical numbers in $x \in [a, b]$.

Next, we verify the contraction mapping. We check the endpoints of the derivative $g'(1) = -1/5$ and $g'(3) = 1/5$ as well as the critical points by setting $g''(x) = 0$. From that, we get the critical point $x = 1/5$. Since $k = 1/5 < 1$, the contraction mapping condition is satisfied.

Now, we are ready to perform the fixed point iteration. Let $p_0 = 1$, then

$$\begin{aligned} p_1 &= g(1.0) = 2.1 \\ p_2 &= g(2.1) = 2.001 \\ p_3 &= g(2.001) = 2.0000001. \end{aligned}$$

Consider the three functions

$$\begin{aligned} g_1(x) &= x - \frac{x^3 - 27}{3x^2} \\ g_2(x) &= x - \frac{x^3 - 27}{6x^2} \\ g_3(x) &= x - \frac{x^3 - 27}{12x^2}. \end{aligned}$$

These functions all have the same fixed point $p = 3$. If we apply the fixed point algorithm, with which of these will our algorithm converge to $p = 3$ the fastest? Let the interval be $[a, b] = [2.5, 4]$. After checking, we confirm that each of these satisfy the conditions for fixed point iteration. Our k for each function is the largest (absolute magnitude) of $g'(a)$, $g'(b)$ and $g'(c)$ where c are the solutions of $g''(x)$. In our case, we find that

$$k_1 = 0.48, \qquad k_2 = 0.69, \qquad k_3 = 0.85.$$

From the error estimate

$$|p_n - p| \leq k^n \max\{p_0 - a, b - p_0\}.$$

we know that our algorithm will converge fastest for $g_1(x)$ since k_1 is the smallest. And since $0 < k < 1$, we know that k^n is very small, and for a given n, $k_1^n < k_2^n < k_3^n$. Therefore, the error will be smallest in n iterations of the fixed point method if we use $g_1(x)$.

4.4 Applications

In general, to find roots of some function, we follow this procedure:

1. Perform N iterations of the bisection method given some bracketing interval. After N iterations, the interval has shrunk to

$$|b_N - a_N| = \frac{|b - a|}{2^N}.$$

   We want N to be large enough so that the interval is within $[p - \delta, p + \delta]$, so the theorem about fixed point methods can be applied.
2. Switch to Newton's method to finish the process.

There are methods other than just Newton's method. There also include the Nelder-Mead optimization and Brent's method.

In multiple dimensions, root-finding gets more complicated and the algorithm often has to be tailored to the problem.

Chapter 5

Interpolation and Approximation

Interpolation is the process of filling in the gaps when we have missing data. For example, if we have a table of experimental data, we might want to find the polynomial that goes through all those points.

If we have two points (x_1, y_1) and (x_2, y_2), the linear interpolation, that is, the line going through both points is

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1.$$

In general, given $n + 1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, we want to fit them to a polynomial of the form

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

In general, we need to find the coefficients $a_0, \ldots, a_n$ such that the polynomial goes through all of the given points.

Given $n+1$ points, there is a unique nth degree polynomial that goes through all of the points. For example, given four points, there is a unique cubic polynomial that goes through all four points.

5.1 Brute Force Method

Given $n + 1$ points $(x_0, y_0), \ldots, (x_n, y_n)$, there is a unique polynomial of the form

$$P_n(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

that goes through those n points.

With our $n+1$ points, we have a linear system of $n+1$ equations and $n+1$ unknowns.

$$\begin{aligned}
a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n &= y_0 \\
a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n &= y_1 \\
\vdots \qquad\qquad\qquad\qquad &\quad \vdots \\
a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n &= y_n.
\end{aligned}$$

We can write this as the matrix system

$$C\vec{\boldsymbol{a}} = \vec{\boldsymbol{y}},$$

where

$$C = \begin{bmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^n \\
1 & x_1 & x_1^2 & \cdots & x_1^n \\
\vdots & \vdots & \vdots & & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^n
\end{bmatrix}$$

is an $n+1$ by $n+1$ matrix. Provided that none of the input points are repeated, this matrix is always non-singular, however, it is very ill-conditioned. The vectors $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{y}}$ are

$$\vec{\boldsymbol{a}} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \qquad \vec{\boldsymbol{y}} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Example 5.1.1

Suppose we are given the three points $(0,0)$, $(1,1)$, and $(2,8)$, and we want to find the unique quadratic interpolant of the form

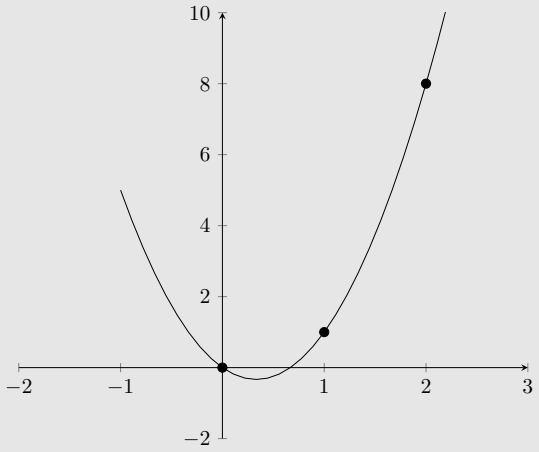
$$P_2(x) = a_0 + a_1 x + a_2 x^2.$$

Our matrix equation is

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 8 \end{bmatrix}.$$

Solving this matrix equation gives us $a_0 = 0$, $a_1 = -2$, and $a_2 = 3$, so our quadratic interpolant is

$$P_2(x) = -2x + 3x^2.$$



Notice that the extremum of the interpolant is outside the range of our data points.

To check our results, we plug the x-values of our data points into $P_2(x)$ to confirm that it returns the y-values of those points.

The C matrix is notoriously ill-conditioned, and with more than two data points, the brute force method is not ideal, so we use other methods.

At the very least, the brute force method has a computational complexity of $O(n^3)$.

5.2 Lagrange Interpolation

Lagrange interpolation is more reliable than the brute force method. Given $n+1$ data points $(x_0, y_0), \ldots, (x_n, y_n)$, our interpolating polynomial is

$$\boxed{P_n(x) = y_0 L_{n,0}(x) + y_1 L_{n,1}(x) + \cdots + y_n L_{n,n}(x),}$$

where

$$\boxed{L_{n,i}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j},}$$

are the Lagrange interpolating polynomial basis functions

For example, given two points (x_0, y_0) and (x_1, y_1), our linear interpolating function is

$$P_1(x) = y_0 L_{1,0}(x) + y_1 L_{1,1}(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0}(x - x_1) + y_1.$$

Example 5.2.1

Consider again the three data points $(0, 0)$, $(1, 1)$, and $(2, 8)$. The interpolating polynomial is

$$P_2(x) = (0)L_{2,0}(x) + (1)L_{2,1}(x) + (8)L_{2,2}(x).$$

The basis functions are

$$L_{2,0}(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{1}{2}(x - 1)(x - 2)$$
$$L_{2,1}(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = -x(x - 2)$$
$$L_{2,2}(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{1}{2}x(x - 1).$$

Putting it all together, we get the same result as before

$$P_2(x) = 3x^2 - 2x.$$

Consider the generating function

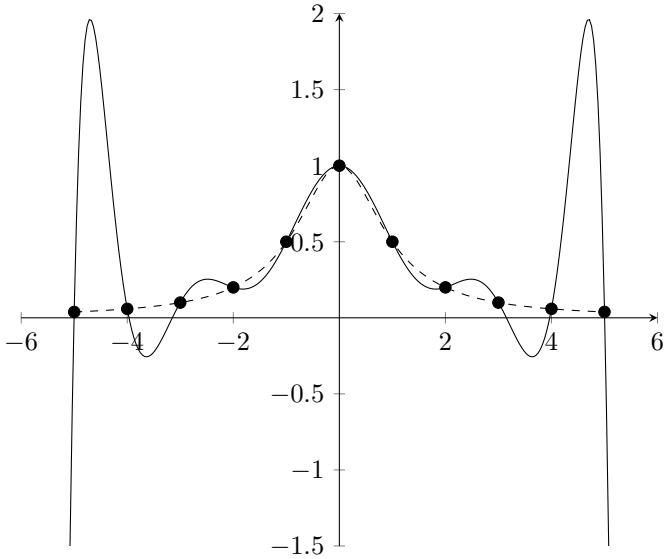
$$f(x) = \frac{1}{1 + x^2},$$

on the interval $[-5, 5]$. If we take some number of points on this curve and then interpolate them using polynomial interpolation, how well will our interpolation match with $f(x)$?

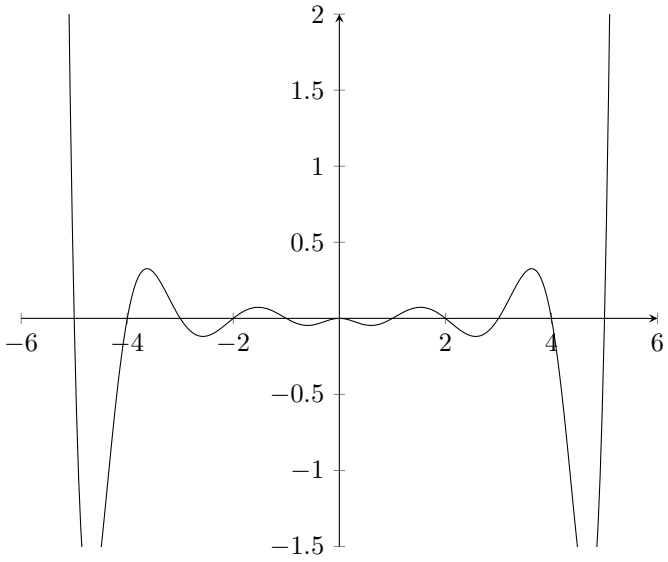
Suppose we take eleven evenly spaced points $(-5, 1/26), (-4, 1/17), \ldots, (4, 1/17), (5, 1/26)$. Then our interpolating polynomial has the form

$$P_{10}(x) = y_0 L_{10,0}(x) + y_1 L_{10,1}(x) + \cdots + y_{10} L_{10,10}(x).$$

Below is a plot of $f(x)$ (the dashed curve) and the Lagrange interpolating polynomial. Notice that near the endpoints there is unstable behavior. A small perturbation of a data point near one of these instabilities will cause a large change in the interpolant. This unstable behavior is called the **Runge phenomenon**.



Below is a plot of the error–the difference between $f(x)$ and the interpolating function. Notice that the error is large near the endpoints and small near the middle.



Suppose we have a generating function $f(x)$ that generates the $n+1$ points

$$(x_0, f(x_0)), \ldots, (x_n, f(x_n)),$$

our interpolating polynomial is

$$P_n(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x),$$

such that $P_n(x_k) = f(x_k)$ for $k = 0, 1, \ldots, n$. That is, the interpolating function goes through each of the generated points. Then the generating function and the interpolating function are related as

$$f(x) = P_n(x) + R_n(x),$$

where

$$\boxed{R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n),}$$

is the remainder term. Notice the similarity with the Taylor series error term.

Example 5.2.2

Given the generating function $f(x) = x^4$ and the points $x_0 = -1$, $x_1 = 1$, and $x_2 = 2$, find the interpolating quadratic and find a bound on the error on the interval $[-1, 2]$.

We find that the interpolating quadratic is

$$P_2(x) = 5x^2 - 4,$$

and the remainder term is

$$R_2(x) = \frac{f'''(\xi(x))}{6}(x+1)(x-1)(x-2).$$

The third derivative is $f'''(x) = 24x$ so $f'''(\xi(x)) = 24\xi(x)$. Then

$$R_2(x) = 4[\xi(x)](x^2-1)(x-2).$$

Now we want to find an upper bound on $R_2(x)$ in the interval $[-1, 2]$. We know that

$$\max|R_2(x)| \leq \max|\xi(x)| \max\left|4(x^2-1)(x-2)\right|.$$

Since $-1 \leq x \leq 2$ we know that $-1 \leq \xi(x) \leq 2$, so the maximum value of $\xi(x)$ on this region is 2. Therefore,

$$\max|R_2(x)| \leq 8 \max\left|(x^2-1)(x-2)\right|.$$

Next, we find the maximum of $g(x) = (x^2-1)(x-2)$ on the interval $[-1, 2]$. To find the maximum, we find the zeros of the first derivative and check the endpoints $g(-1)$ and $g(2)$. If we take the first derivative $g'(x)$ and set it equal to zero, we find that the maximum occurs at $x = 2/3 - \sqrt{7}/3$, and the maximum is

$$g\left(\frac{2}{3} - \frac{\sqrt{7}}{3}\right) = \frac{20 + 14\sqrt{7}}{27} \approx 2.113.$$

Therefore, our error on the given interval is bounded by

$$\max|R_2(x)| \leq 8\frac{20 + 14\sqrt{7}}{27} \approx 16.90.$$

What is the computational complexity of

$$P_n(x) = \sum_{i=0}^{n} y_i L_{n,i}(x),$$

where

$$L_{n,i}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}?$$

The pseudocode to compute $L_{n,i}(x)$ is

```
L = 1.0
for j = 0, ..., n
   if j != i, then
      L = L * (x - x[j])/(x[i] - x[j])
```

Here there are $2n$ multiplies and divides, so to compute $P_n(x)$ requires $(2n)(n+1)+n+1 = 2n^2+3n+1$ multiplies and divides.

The full algorithm given the inputs $x_0, \ldots, x_n, y_0, \ldots, y_n$ is shown below. Then x is the point at which we want the value predicted and y, the output, is the approximated value of the function at x.

```
p = 0.0
for i = 0, ..., n
   L = 1.0
   for j = 0, ..., n
      if j != i, then
         L = L * (x - x[j])/(x[i] - x[j])
   p = p + L * y[i]
```

5.3 Newton's Divided Difference Table

Newton's divided differences is another method for finding the unique interpolating polynomial given $n+1$ points $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$. The interpolating polynomial is

$$\begin{aligned} P_n(x) = f[x_i] + f[x_i, x_{i+1}](x - x_i) + f[x_i, x_{i+1}, x_{i+2}](x - x_i)(x - x_{i+1}) \\ + \cdots + f[x_i, x_{i+1}, \ldots, x_{i+n}](x - x_i)(x - x_{i+1}) \cdots (x - x_{i+n-1}), \end{aligned}$$

where the zeroth order Newton's divided difference (NDD) is defined as

$$f[x_i] = f(x_i),$$

the first order NDD is defined in terms of the zeroth order NDD as

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i},$$

the second order NDD is

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i},$$

and in general, the NDD is defined recursively as

$$f[x_i, x_{i+1}, \ldots, x_{i+p}] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_{i+p}] - f[x_i, x_{i+1}, \ldots, x_{i+p-1}]}{x_{i+p} - x_i}.$$

The most efficient way to use Newton's divided differences to construct the interpolating polynomial is to construct a Newton's divided differences table as shown below.

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, \ldots, x_{i+p}]$
x_0	y_0	$\frac{y_1 - y_0}{x_1 - x_0}$	$\frac{f[x_1,x_2] - f[x_0,x_1]}{x_2 - x_0}$	$\frac{f[x_1,\ldots,x_n] - f[x_0,\ldots,x_{n-1}]}{x_n - x_0}$
x_1	y_1	$\frac{y_2 - y_1}{x_2 - x_1}$	$\frac{f[x_2,x_3] - f[x_1,x_2]}{x_3 - x_1}$	–
x_2	y_2	$\frac{y_3 - y_2}{x_3 - x_2}$	$\frac{f[x_3,x_4] - f[x_2,x_3]}{x_4 - x_2}$	–
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{n-2}	y_{n-2}	$\frac{y_{n-1} - y_{n-2}}{x_{n-1} - x_{n-2}}$	$\frac{f[x_{n-1},x_n] - f[x_{n-2},x_{n-1}]}{x_n - x_{n-2}}$	–
x_{n-1}	y_{n-1}	$\frac{y_n - y_{n-1}}{x_n - x_{n-1}}$	–	–
x_n	y_n	–	–	–

Notice that almost half of the table is empty.

The computational complexity of the NDD method if we use Horner's method to evaluate the polynomial is $O(n)$.

Using Newton's divided differences, how do we find the interpolating polynomial if our generating function is $f(x) = x^4$ and our points are $x_0 = -1$, $x_1 = 1$, and $x_2 = 2$? Our table in this case is

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$
-1	$\boxed{1}$	$\frac{1-1}{1-(-1)} = \boxed{0}$	$\frac{15-0}{2-(-1)} = \boxed{5}$
1	1	$\frac{16-1}{2-1} = 15$	–
2	16	–	–

Next, we construct the interpolating polynomial using the elements in the first row

$$P_2(x) = (1) + (0)(x+1) + (5)(x+1)(x-1) = 5x^2 - 4.$$

The order of the data points doesn't matter. If we reordered the points, we would use a different set of coefficients from the table, but we would end up with the same interpolating polynomial.

With the NDD table method, it is straightforward to extend it with additional data points. For example, if we want to add the point $x_3 = 0$, we just add an additional row and column to the table.

x_i	$f[x_i]$			
-1	1	0	$\boxed{5}$	$\boxed{2}$
1	$\boxed{1}$	$\boxed{15}$	7	–
2	16	8	–	–
0	0	–	–	–

We can choose different coefficients, which will only affect the order of our x's. In this cases, we choose the ones that are boxed in the table above. That is, our points are $x_0 = 1$, $x_1 = 2$, $x_2 = -1$, and $x_3 = 0$. Then our interpolating polynomial is

$$P_3(x) = 1 + 15(x-1) + 5(x-1)(x-2) + 2(x-1)(x-2)(x+1).$$

For Newton's divided difference method, we have the inputs $x_0, \ldots, x_n, y_0, \ldots, y_n$. Then x is the point at which we want the value predicted and y, the output, is the approximated value of the function at x. When implementing the NDD method, we start by filling the NDD table. The pseudocode for doing that is shown here:

```
for i = 0, ..., n
F[i, 0] = y[i] # Zeroth order NDD in first column
for i = 1, ..., n
   for j = 1, ..., i
      F[i, j] = (F[i, j-1] - F[i-1,j-1]) / (x[i] - x[i-j])
```

Here, the NDD table is the 2D array F. The coefficients for $P_n(x)$ are on the diagonal of F. The interpolating polynomial is then

$$P_n(x) = F[0,0] + F[1,1](x - x_0) + \cdots + F[n,n](x - x_0)\cdots(x - x_{n-1}).$$

This can be evaluated using Horner's method as follows:

```
p = F[n, n]
for i = n, ..., 1
   p = p * (x - x[i-1]) + F[i-1, i-1]
```

5.4 'Near Minimax' Approximation

A major problem with polynomial interpolation is the Runge phenomenon, the overshoot, that occurs near the endpoints. The remainder term

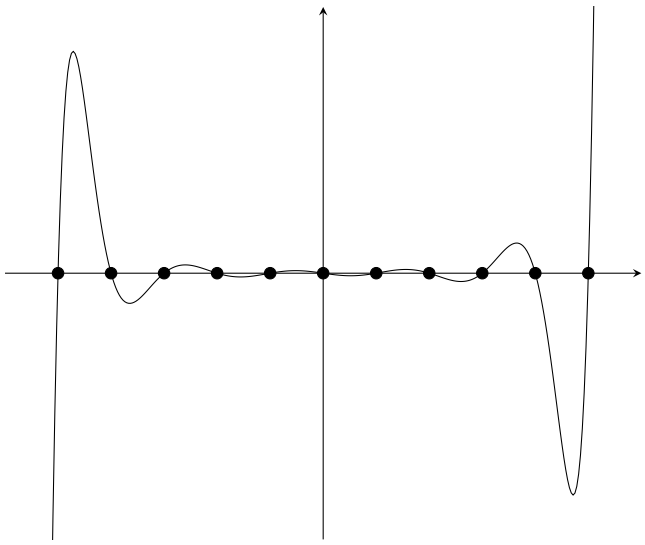
$$\boxed{R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)(x-x_1)\cdots(x-x_n),}$$

gets very large near the endpoints.

The Runge phenomenon occurs when the points $x_0, \ldots, x_n$ are evenly spaced. When they are evenly spaced, then the points are given by

$$x_i = a + \frac{b-a}{n}i,$$

if our interval is $[a, b]$. The remainder term, as shown below, gets large near the endpoints.



One way to combat this error is to strategically choose the points $x_0, \ldots, x_n$ so the error is more evenly distributed. The problem of finding this optimal spacing is called a **minimax** problem.

Given our input points in the interval $[a, b]$ ordered such that $a \le x_0 \le x_1 \le \cdots \le x_n \le b$, the minimax problem is

$$\min_{x_0,\ldots,x_n}\left(\max_{a\le x\le b}|f(x)-P_n(x)|\right),$$

which can be written as

$$\min_{x_0,\ldots,x_n}\max_{a\le x\le b}\left|\frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-x_0)\cdots(x-x_n)\right|.$$

However, this is a hard problem. If we assume that the derivative $f^{(n+1)}(x)$ is constant, we get the **near minimax** problem

$$\min_{x_0,\ldots,x_n}\max_{a\le x\le b}|(x-x_0)(x-x_1)\cdots(x-x_n)|,$$

which is easier to solve.

It turns out that the optimal spacing of the input points is the roots of the **Chebyshev polynomials**.

The Chebyshev polynomials are given by

$$T_n(x) = \cos\left(n \cos^{-1} x\right).$$

The first several ones are

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1. \end{aligned}$$

The Chebyshev polynomials can also be defined recursively as

$$T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x).$$

They are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = 1/\sqrt{1-x^2}$. That is,

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = \begin{cases} 0 & \text{if } n \neq m \\ \text{not } 0 & \text{if } n = m. \end{cases}.$$

The roots of the nth Chebyshev polynomial are given by

$$\tilde{x}_k = \cos\left(\frac{2k-1}{2n}\pi\right), \quad k = 1, 2, \ldots, n.$$

These points x_k are in the range $[-1, 1]$, and we want them to be in the range $[a, b]$, so we have to do a transformation. We need $n+1$ interpolating points, so we can write this as

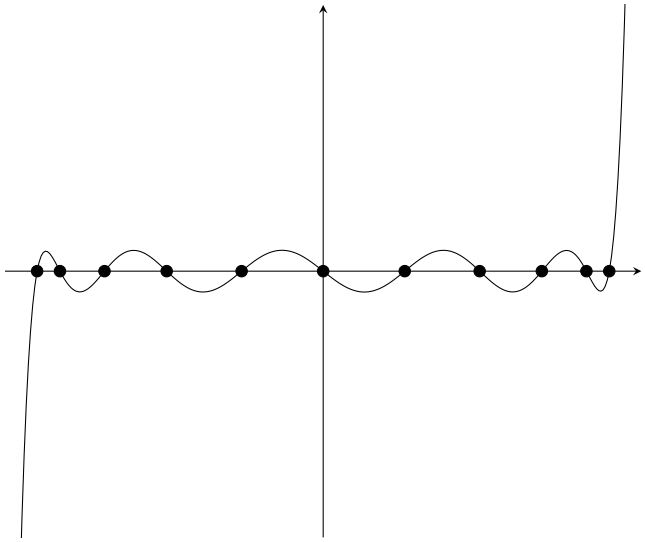
$$\tilde{x}_k = \cos\left(\frac{k+\frac{1}{2}}{n+1}\pi\right), \quad k = 0, 1, 2, \ldots, n.$$

Next, we need to transform from $-1 \leq \tilde{x}_k \leq 1$ to $a \leq x_k \leq b$. For a linear transformation, we know that $x_k = \alpha + \beta\tilde{x}_k$. This gives us $a = \alpha - \beta$ and $b = \alpha + \beta$ or in terms of a and b, it gives us $\alpha = (a+b)/2$ and $\beta = (b-a)/2$. Therefore, the near minimax optimal interpolating point distribution is

$$\boxed{x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{k+\frac{1}{2}}{n+1}\pi\right), \quad k = 0, 1, 2, \ldots, n.}$$

Using these points will take care of the Runge phenomenon.

Suppose we are doing a polynomial interpolation with $n+1$ points. If we let the x-values of those points, $x_0, \ldots, x_n$ be roots of the nth-order Chebyshev polynomial, then the polynomial that interpolates these points will still overshoot, but now the peaks are all the same height. That is, there is no Runge phenomenon like that which occurs when the input points are evenly spaced. Also, notice that the roots of the Chebyshev polynomials are clustered toward the endpoints of our interval.



There are two potential problems with this approach:

1. The *near minimax* approach only works if the data at the optimal points is available.
2. This approach relies on the derivative $f^{(n+1)}(x)$ being nearly constant. This is not true for discontinuous functions. If you interpolate discontinuous functions, you will have to deal with Gibbs phenomenon.

Example 5.4.1

Given the generating function $f(x) = x(x^2 - 1)$ on the interval $[0, 2]$ compare the accuracy of the interpolating polynomials if you use evenly spaced points $x_0 = 0$, $x_1 = 1$, and $x_2 = 2$ and if you use optimally spaced points. The generating function is shown as the solid curve in the plot below.

Using the evenly spaced points, our interpolating polynomial is

$$P_2(x) = 3x^2 - 3x.$$

This interpolating polynomial is shown as the dashed curve in the plot below.

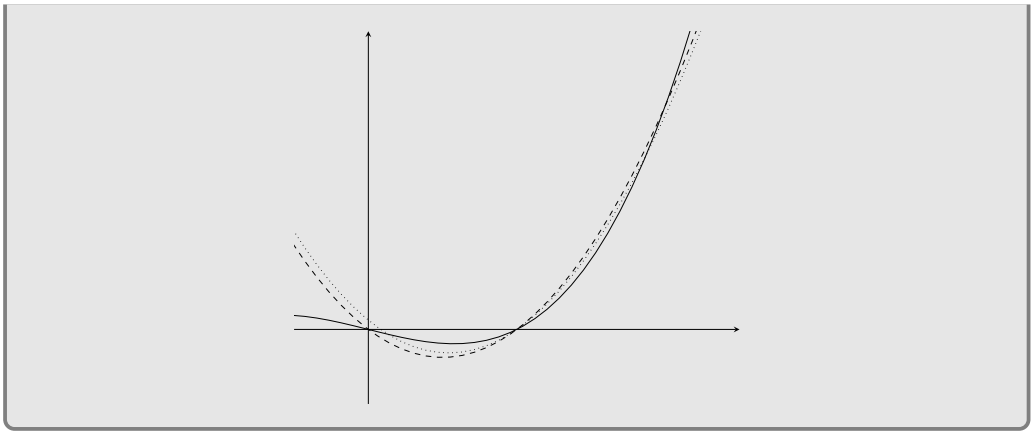
The optimal interpolating points are

$$x_0 = 1 + \cos\frac{\pi}{6} = 1 + \frac{\sqrt{3}}{2}$$
$$x_1 = 1 + \cos\frac{3\pi}{6} = 1$$
$$x_2 = 1 + \cos\frac{5\pi}{6} = 1 - \frac{\sqrt{3}}{2}.$$

Using the optimally-spaced points, our interpolating polynomial is approximately

$$P_2(x) = 3.000x^2 - 3.250x + 0.2501.$$

It is the dotted curve on the graph below.



Chapter 6

Numerical Differentiation

6.1 First Derivative

Forward Difference

Recall the limit definition of a derivative. Given a function $f(x)$, its derivative can be written as

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

From this, we can approximate a derivative by using a small h in what is called the **forward difference** method

$$D_1^+ \left(f(x)\right) = \frac{f(x+h) - f(x)}{h} \approx f'(x),$$

provided that $h << 1$. We might expect our approximation to approach $f'(x)$ as $h \to 0$, but this is not true on a finite precision computer. We will end up with a loss of precision due to the subtraction of two nearly equal numbers at some point, and this means that our approximation of $f'(x)$ approaches the true value only to a certain small h. Using a smaller h results in a loss of precision.

Suppose $f(x) = \sqrt{1+x}$ and we want to approximate $f'(0) = 1/2$. Our approximate derivative is

$$f'(0) \approx \frac{\sqrt{1+h} - \sqrt{1}}{h}.$$

Suppose we use 3-digit rounding, then we get the results shown in the table below. Notice that as h decreases, the approximation gets better up to a point. Past that point, the approximation quickly becomes worthless.

h	$\frac{\sqrt{1+h}-\sqrt{1}}{h}$	$f'(x)$	$\frac{\lvert p-p^* \rvert}{\lvert p \rvert}$
1.0	0.410	$\frac{1}{2}$	0.18
0.5	0.440	$\frac{1}{2}$	0.12
0.1	0.500	$\frac{1}{2}$	0
0.01	0.000	$\frac{1}{2}$	1
0.001	0.000	$\frac{1}{2}$	1

We can think of the forward difference in terms of the Taylor expansion of $f(x+h)$ for small h

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

Solving this for $f'(x)$ gives us

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

This tells us that the *approximation* error, which comes from chopping off part of the Taylor series, of the forward difference method is

$$\varepsilon_a = \frac{h}{2}\left|f''(x)\right|.$$

Like it was mentioned above, this suggests that making h smaller will reduce the error, but this is not completely true. The total error of the forward difference method is

$$\varepsilon = \frac{h}{2}\left|f''(x)\right| + \frac{2c|f(x)|}{h},$$

where the second term gives the rounding error, where $c \approx 10^{-16}$ in Python is the **error constant**. Notice that if we decrease h, the error ε will decrease only to a certain point before it increases again. To find the optimal value h, we set the derivative $\frac{d\varepsilon}{dh}$ equal to zero and solve for h. When we do that, we find that the optimal value of h for the forward difference method is

$$h_{optimal} = \sqrt{4c\left|\frac{f(x)}{f''(x)}\right|}.$$

If we use this optimal h, then our error is

$$\varepsilon = \sqrt{4c|f(x)f''(x)|}.$$

If $f(x)$ and $f''(x)$ are on the order of 1, then $h_{optimal} \sim 10^{-8}$.

Backward Difference

Similarly, we can define the **backward difference** method as

$$D_1^-\left(f(x)\right) = \frac{f(x) - f(x-h)}{h} \approx f'(x).$$

The forward and backward difference methods give similar accuracy, but one or the other might be preferred at the boundary of bounded functions.

For these derivative approximations, reducing h reduces the approximation error, but increases the roundoff error. There is typically roundoff error associated with the numerator, and when you then divide that by a very small h, the roundoff error is amplified. For example, suppose your numerator approximates 1.01 as 1.0. The absolute error is 0.01. However, if you now divide by 0.00001, you get 100,000 versus the actual value of 101,000. The absolute error is now 1000.

Central Difference

Another method of approximating the first derivative is the **central difference** method

$$D_1^c\left(f(x)\right) = \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} \approx f'(x).$$

This method is overall better than the forward or backward difference methods. By subtracting the Taylor expansion of $f(x-h/2)$ from that of $f(x+h/2)$, we find that

$$\frac{f\left(x+\frac{h}{2}\right)-f\left(x-\frac{h}{2}\right)}{h}=f'(x)+\frac{h^2}{24}f'''(x)+O(h^3),$$

so for the central difference method, our approximation error is

$$\varepsilon_a=\frac{h^2}{24}\left|f'''(x)\right|,$$

and the total error is

$$\varepsilon=\frac{h^2}{24}\left|f'''(x)\right|+\frac{2c|f(x)|}{h}.$$

Now, our optimal h is

$$h_{optimal}=\left(24c\left|\frac{f(x)}{f'''(x)}\right|\right)^{\frac{1}{3}}.$$

If $f(x)$ and $f'''(x)$ are on the order of 1, then $h_{optimal}\sim 10^{-5}$, then the error is on the order of 10^{-10}.

Other first derivative approximations of $f(x)$ include:

$$M_3(h)=\frac{-f(x+2h)+4f(x+h)-3f(x)}{2h}$$
$$M_4(h)=\frac{f(x-2h)-4f(x-h)+3f(x)}{2h}.$$

As h is decreased, $M_i(h)$ more and more closely approximates $f'(x)$. That is, as h approaches zero, the approximation error also goes to zero. However, as h decreases, the roundoff error increases. This means for each $M_i(h)$ and for a given $f(x)$, there is an optimal value of h that is small, but not too small.

6.2 Second Derivative

By applying the forward and backward derivative formulas, we can construct the central difference for the second derivative. If we replace x by $x+h/2$ and $x-h/2$ in the central difference formula for the first derivative, we get

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

Therefore, the central difference formula for the second derivative is

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

This simplifies to

$$D_2^c\left(f(x)\right)=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}\approx f''(x).$$

By adding the Taylor expansion of $f(x+h)$ to that of $f(x-h)$, we find that

$$\frac{f(x+h)-2f(x)+f(x-h)}{h^2}=f''(x)-\frac{h^2}{12}f^{(4)}(x)+O(h^6).$$

This implies that the approximation error is

$$\varepsilon_a = \frac{h^2}{12}\left|f^{(4)}(x)\right|,$$

and the total error is

$$\varepsilon = \frac{h^2}{12}\left|f^{(4)}(x)\right| + \frac{4c|f(x)|}{h}.$$

Now, our optimal h is

$$h_{optimal} = \left(48c\left|\frac{f(x)}{f^{(4)}(x)}\right|\right)^{\frac{1}{4}}.$$

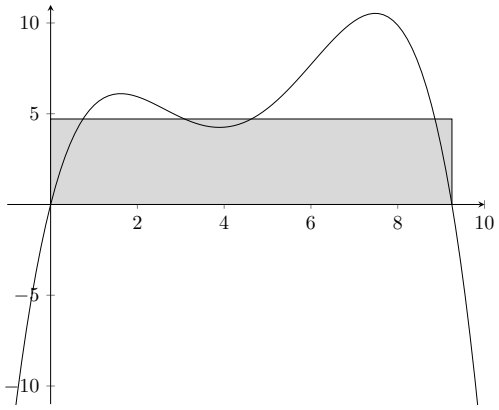
If $f(x)$ and $f^{(4)}(x)$ are on the order of 1, then $h_{optimal} \sim 10^{-4}$, then the error is on the order of 10^{-8}.

Chapter 7

Numerical Integration

7.1 Rectangle Rule

Perhaps the simplest approximation of an integral is to multiply the value of the function at the midpoint of the integration interval with the width of the integration interval. As you can see from the graph below, this is not a very good approximation.

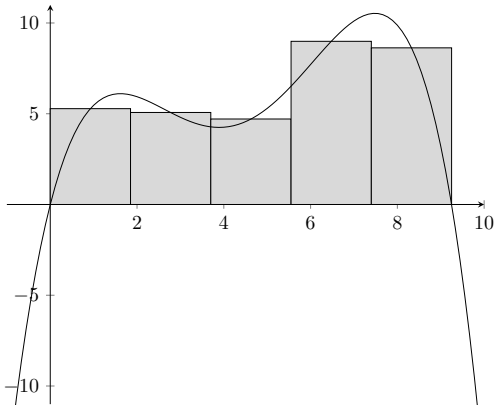


With the rectangle rule, our approximation is

$$\int_a^b f(x)\,dx \approx (b-a) f\left(\frac{b-a}{2}\right).$$

7.2 Composite Rectangle Rule

The area under a curve can be approximated by splitting the area into some number of rectangles and adding the areas of the rectangles.



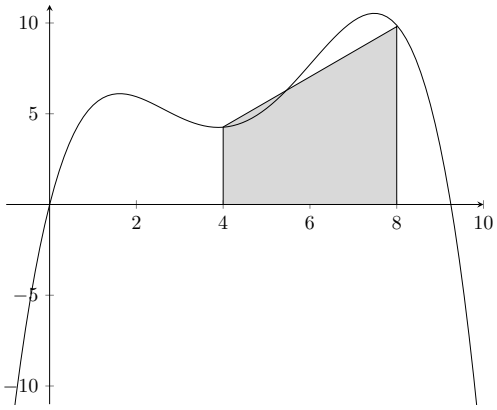
As the number of rectangles goes to infinity, the value given by the rectangular method approaches the exact value of the integral. That is, as $N \to \infty$, the approximation error goes to zero. However, as N becomes large, the computational difficulty grows large too. If we need our approximation to have a minimum error, we might be better off reducing the approximation error from the beginning by using a better algorithm than to use the rectangular method with a very large N.

Below is a simple python algorithm that approximates the area under a curve using the rectangles method.

```
N = 10            # Number of rectangles to use
a = 0.0           # Lower integration limit
b = 10.0          # Upper integration limit

def f(x):
    """
    Define the function to be integrated.
    """
    return x**4

w = (b-a)/N       # Width of each rectangle
s = 0.0           # Sum the area of all the rectangles

for i in range(N):
    s += f(a + w/2 + i*w)*w

print(s)
```

7.3 Trapezoid Rule

One of the simplest integral approximations is to replace the the function by a straight line. This is illustrated in the graph below where we approximate the integral of the function between $x = 4$ and $x = 8$ with a straight line. This is the trapezoid rule.



With the trapezoid rule,

$$\int_a^b f(x)\,dx \approx (b-a)\frac{f(a)+f(b)}{2}.$$

The trapezoid rule can be derived using Lagrange interpolation. Recall that the linear Lagrange interpolant is

$$f(x) \approx f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}.$$

Integrating this gives us

$$\int_a^b f(x)\,dx = \frac{f(a)}{a-b}\int_a^b (x-b)\,dx + \frac{f(b)}{b-a}\int_a^b (x-a)\,dx = \frac{b-a}{2}\left(f(a)+f(b)\right).$$

What about the error of the trapezoid rule? Recall that $f(x) = P_1(x) + R_1(x)$ where

$$R_1(x) = \frac{f''\left(\xi(x)\right)}{2!}(x-a)(x-b).$$

Integrating this gives us

$$\int_a^b R_1(x)\,dx = \int_a^b \frac{f''\left(\xi(x)\right)}{2!}(x-a)(x-b)\,dx = \frac{f''\left(\eta\right)}{2}\int_a^b (x-a)(x-b)\,dx,$$

where $a \le \eta \le b$. Here we have applied the mean value theorem for integrals. If we evaluate this integral, we find that

$$\int_a^b R_1(x)\,dx = -\frac{f''\left(\eta\right)}{2}\frac{(b-a)^3}{6}.$$

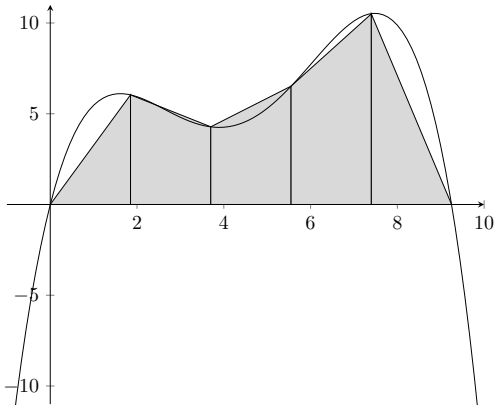
Therefore, for the trapezoid rule,

$$\boxed{\int_a^b f(x)\,dx = \frac{b-a}{2}\left[f(a)+f(b)\right] - \frac{f''\left(\eta\right)}{12}(b-a)^3.}$$

This tells us that the larger the interval, the larger the error.

7.4 Composite Trapezoid Rule

With the composite trapezoid rule, we break our integration interval into many pieces then apply the trapezoid rule to each piece. Like with the composite rectangle rule, the sum of the areas of the trapezoids approaches the true value of the integral as the number of trapezoids is increased to a large number. However, the computation takes longer for increased trapezoids.



For the composite trapezoid rule, we break the interval into n evenly-spaced intervals of width

$$h = \frac{b-a}{n}.$$

The points at which we evaluate the function are given by

$$x_i = a + ih = a + \frac{b-a}{n} i.$$

Since integration is a linear operator, we break the interval into many slices and then apply the trapezoid rule to each slice.

$$\begin{aligned}
\int_a^b f(x)\,dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,dx \\
&\simeq \sum_{i=0}^{n-1} (x_{i+1} - x_i) \left[\frac{f(x_i) + f(x_{i+1})}{2} \right] \\
&= \frac{h}{2} \left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + f(x_n) \right] \\
&= \frac{h}{2} \left[f(x_0) + f(x_n) + 2 \sum_{i=1}^{n-1} f(x_i) \right] \\
&= \frac{h}{2} \left[f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right].
\end{aligned}$$

For the remainder term of the composite trapezoid rule, we get

$$\begin{aligned}
R(x) &= -\frac{1}{12} \sum_{i=0}^{n-1} f''(\eta_i) \left(\frac{b-a}{n} \right)^3 \\
&= -\frac{1}{12} f''(c) \sum_{i=0}^{n-1} \left(\frac{b-a}{n} \right)^3 \\
&= -\frac{1}{12} f''(c) \left(\frac{b-a}{n} \right)^2 \frac{b-a}{n} n \\
&= -\frac{f''(c)}{12} \frac{(b-a)^3}{n^2}.
\end{aligned}$$

In the first step, we used the mean value theorem for the second derivative. Note that $a \leq c \leq b$. This remainder term implies that as we increase n, the error goes down. It does not go down very fast though.

Tip

The number of "slices" N must be even for Simpson's Rule.

In summary, the composite trapezoid rule gives us the approximation

$$\boxed{\int_a^b f(x)\,dx \simeq \frac{h}{2}\left[f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_i)\right],}$$

with an error of

$$\boxed{R(x) = -\frac{b-a}{12}h^2 f''(c)\,.}$$

The composite trapezoid rule is a "second-order" method since if h is reduced by half, the error is reduced by a quarter.

Note, by construction, the composite trapezoid rule is exact for linear functions.

A quadrature (i.e. numerical integration) method has a **degree of precision** (DOP) of p if the method is exact for all polynomials of degree less than or equal to p. The trapezoidal method, then, has a DOP of 1, since it is exact for linear functions but not for quadratic functions.

Below is a simple python algorithm that approximates the area under a curve using the composite trapezoid method.

```
N = 10                  # Number of trapezoids to use
a = 0.0                 # Lower integration limit
b = 10.0                # Upper integration limit


def f(x):
    """
    Define the function to be integrated.
    """
    return x**4

w = (b-a)/N                   # Width of each trapezoid
s = 0.5*f(a)*w + 0.5*f(b)*w   # Area of first and last trapezoids

for i in range(1, N):         # Area of rest of trapezoids
    s += f(a + i*w)*w

print(s)
```

7.5 Simpson's Rule

Where the composite trapezoid rule uses the linear interpolant, the composite Simpson's rule uses the quadratic interpolant. Again, we slice the integral into equally-spaced intervals

$$h = \frac{b-a}{n},$$

but now we take a pair of them and approximate the function in that pair of intervals with the quadratic interpolant. By construction, Simpson's rule is exact for quadratic functions. It can also be shown that it is exact for cubic functions. Simpson's method, therefore, has a DOP of 3.

If we have only a single pair of intervals (i.e. $n = 2$), then $h = (b-a)/2$ and Simpson's rule gives us

$$\boxed{\int_a^b f(x)\,dx \simeq \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right].}$$

With n intervals (where n is an even number), Simpson's rule gives us

$$\int_a^b f(x)\,dx \simeq \frac{h}{3}f(a) + \frac{h}{3}f(b) + \frac{4h}{3}\sum_{i=0}^{\frac{n}{2}-1} f(x_{2i+1}) + \frac{2h}{3}\sum_{i=1}^{\frac{n}{2}-1} f(x_{2i}).$$

The remainder term is

$$R = -\frac{b-a}{180}h^4 f^{(4)}(\mu).$$

Notice that Simpson's rule is a fourth-order method.

Below is a simple python algorithm that approximates the area under a curve using Simpson's Rule.

```
N = 10  # Number of trapezoids to use
a = 0.0  # Lower integration limit
b = 10.0   # Upper integration limit


def f(x):
    """
    Define the function to be integrated.
    """
    return x**4

# The number of slices must be even for Simpson's Rule to be accurate
if N % 2 != 0:
    N += 1
    print("Number of slices was odd so 1 was added to N.")


w = (b-a)/N                      # Width of each slice
s = (1/3)*f(a)*w + (1/3)*f(b)*w  # Area of first and last slices

for i in range(1, N, 2):         # Odd terms
    s += f(a + i*w)*w*(4/3)

for i in range(2, N, 2):         # Even terms
    s += f(a + i*w)*w*(2/3)

print(s)
```

7.6 Gaussian Quadrature

For the study of Gaussian quadrature, it is convenient to use the interval $[-1, 1]$. So in this section, we will be interested in the integral

$$\int_{-1}^{1} f(x)\,dx.$$

This is a good integral with which to study the different methods. The interval $[-1, 1]$ works because we can get any limits given an appropriate change of variables.

For this integral, the trapezoid method gives us

$$\int_{-1}^{1} f(x)\,dx \simeq \frac{b-a}{2}\left[f(a) + f(b)\right] = f(1) + f(-1).$$

Simpson's rule gives us

$$\int_{-1}^{1} f(x)\,dx \simeq \frac{1}{3}\left[f(-1) + 4f(0) + f(1)\right].$$

In general,

$$\int_{-1}^{1} f(x)\,dx \simeq c_1 f(x_1) + c_2 f(x_2) + \cdots + c_n f(x_n).$$

What are the c_j for $j = 1, \cdots, n$ and x_j which optimize the degree of precision? What is the maximum DOP that can be achieved? If $n = 2$, we have four unknowns, c_1, c_2, x_1, x_2, which gives us a maximum DOP of 3. Given $2n$ unknowns, c_j and x_j, the maximum DOP that can be obtained is $2n - 1$. How do we find these optimal nodes (x_j) and weights (c_j)?

For example, if we use the near-minimax points

$$x = \cos\left(\frac{\left[m + \frac{1}{2}\right]\pi}{n}\right),$$

to evaluate our function at, we will get very good results for quadrature, but these are still not the optimal points.

First, we will look at the brute force approach. Our quadrature rules for integrating $f(x)$ on $[-1, 1]$ for different degrees of x are

$$\begin{aligned}
2 &= c_1 + c_2 + \cdots + c_n, \quad \text{if } f(x) = x^0 \\
0 &= c_1 x_1 + c_2 x_2 + \cdots + c_n x_n, \quad \text{if } f(x) = x^1 \\
\frac{2}{3} &= c_1 x_1^2 + c_2 x_2^2 + \cdots + c_n x_n^2, \quad \text{if } f(x) = x^2 \\
0 &= c_1 x_1^3 + c_2 x_2^3 + \cdots + c_n x_n^3, \quad \text{if } f(x) = x^3 \\
&\vdots \\
0 &= c_1 x_1^{2n-1} + c_2 x_2^{2n-1} + \cdots + c_n x_n^{2n-1}, \\
&\qquad \text{if } f(x) = x^{2n-1}.
\end{aligned}$$

So we have a nonlinear system that is not easy to solve.

If $n = 1$, then our quadrature rule is

$$\int_{-1}^{1} f(x)\,dx \simeq c_1 f(x_1).$$

We know that

$$\begin{aligned}
2 &= c_1 \\
0 &= c_1 x_1,
\end{aligned}$$

so $c_1 = 2$ and $x_1 = 0$. That is,

$$\int_{-1}^{1} f(x)\,dx \simeq 2f(0).$$

This is just the midpoint rule, and it has a DOP of 1.

If $n = 2$, then our quadrature rule is

$$\int_{-1}^{1} f(x)\,dx \simeq c_1 f(x_1) + c_2 f(x_2).$$

We have four unknowns, so we need four equations. We know that

$$\begin{aligned}
2 &= c_1 + c_2 \\
0 &= c_1 x_1 + c_2 x_2 \\
\frac{2}{3} &= c_1 x_1^2 + c_2 x_2^2 \\
0 &= c_1 x_1^3 + c_2 x_2^3.
\end{aligned}$$

If we assume that the two points x_1 and x_2 are equidistant from the origin, that is, $x_1 = -x_2$, then our solution to this system of equations gives us

$$\int_{-1}^{1} f(x)\,dx \simeq f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right).$$

This is the method for $n = 2$, so it has a DOP of 3. This is the same DOP as Simpson's rule, but it is more efficient since we only have to evaluate $f(x)$ twice as opposed to three times for Simpson's rule.

What about for general n? Given the quadrature rule,

$$\int_{-1}^{1} f(x)\,dx \simeq c_1 f(x_1) + c_2 f(x_2) + \cdots + c_n f(x_n),$$

if $x_1, \ldots, x_n$ are the roots of $P_n(x)$ where $P_n(x)$ is the Legendre polynomial of nth degree, and if $c_1, \ldots, c_n$ are defined accordingly, then the DOP is $2n - 1$.

The first several **Legendre polynomials** are

$$\begin{aligned}
P_0(x) &= 1 \\
P_1(x) &= x \\
P_2(x) &= x^2 - \frac{1}{3} \\
P_3(x) &= x^3 - \frac{3}{5}x \\
P_4(x) &= x^4 - \frac{6}{7}x^2 + \frac{3}{35}.
\end{aligned}$$

Notice that they alternate odd and even. The Legendre polynomials are orthogonal. That is,

$$\int_{-1}^{1} P_i(x)P_j(x)\,dx = \begin{cases} 0 & \text{if } i \neq j \\ \text{not } 0 & \text{if } i = j \end{cases}.$$

The Legendre polynomials also form a complete set, so given any polynomial $Q(x)$ of degree n, there exist $a_0, \ldots, a_n$ such that

$$Q(x) = a_0 P_0(x) + a_1 P_1(x) + \cdots + a_n P_n(x).$$

The coefficients are given by

$$a_k = \frac{\int_{-1}^{1} Q(x)P_k(x)\,dx}{\int_{-1}^{1} P_k^2(x)\,dx}.$$

Recall from Lagrange interpolation that

$$f(x) \simeq \sum_{i=1}^{n} f(x_i) \prod_{\substack{j=1 \\ j\neq i}} \frac{x - x_j}{x_i - x_j}.$$

If we integrate this, we get

$$\int_{-1}^{1} f(x)\,dx \simeq \sum_{i=1}^{n} f(x_i) \int_{-1}^{1} \prod_{\substack{j=1 \\ j\neq i}} \frac{x - x_j}{x_i - x_j}\,dx.$$

The maximum DOP for Gaussian quadrature is $2n - 1$ when the $x_1, \ldots, x_n$ are the roots of the Legendre polynomials. Then

$$\boxed{\int_{-1}^{1} f(x)\,dx = \sum_{i=1}^{n} c_i f(x_i) + \int_{-1}^{1} R_{n-1}(x)\,dx,}$$

where

$$c_i = \int_{-1}^{1} \prod_{\substack{j=1 \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \, dx$$

$$R_{n-1}(x) = \frac{f^{(n)}\left(\xi(x)\right)}{n!}(x - x_1) \cdots (x - x_n).$$

Note that the minimum DOP is $n - 1$. This quadrature rule is exact for a polynomial of order $n - 1$ regardless of the choice of x_i since the remainder term is then zero due to the $f^{(n)}\left(\xi(x)\right)$ derivative. The coefficients c_i are not easy to compute, and they are generally pre-computed.

Chapter 8

Numerical Linear Algebra

In numerical linear algebra, we are interested in solving

$$A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}},$$

where A is a square $n \times n$ matrix, and $\vec{\boldsymbol{x}}$ and $\vec{\boldsymbol{b}}$ are $n \times 1$ vectors.

There are

- Direct methods
- Iterative methods (used for sparse matrix systems)
  - Conjugate gradient method
  - Multi-grid method

Then we will look at eigenvalues and eigenvectors. The QR method is an important technique for finding singular value decompositions.

8.1 Gaussian Elimination with Pivoting

Gaussian Elimination

Gaussian elimination is a technique for solving matrix equations of the form

$$A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}.$$

The general procedure is as follows.

1. Write the matrix equation as an augmented matrix.
2. Perform elementary row operations to transform the left side of the augmented matrix into upper triangular form. The allowed row operations are:
   - Exchange one row with another row.
   - Scale an entire row by a constant.
   - Replace a row with a itself plus another (scaled) row.
3. Finally, solve the system using **back substitution**. Since the left side of the augmented matrix is now in upper triangular form, the bottom row allows you to solve for x_n. Once this is found, you can substitute it into the next higher row and solve for x_{n-1} and so on until you have all x_i.

A python program implementing the Gaussian elimination algorithm can be found on page 107.

Partial Pivoting

Suppose we are given the following matrix and vector

$$A = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 0 & 3 \\ 0 & 1 & 1 \end{bmatrix}, \quad \vec{\boldsymbol{b}} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

To perform Gaussian elimination with partial pivoting by hand, we start by forming the augmented matrix.

$$\left[\begin{array}{ccc|c} 0 & 2 & 1 & 1 \\ 1 & 0 & 3 & 0 \\ 0 & 1 & 1 & 1 \end{array}\right]$$

In the first pivot position, a_{11}, we have a zero. With partial pivoting, we define

$$p = \text{argmax}_{1 \le i \le n} |a_{i1}| .$$

That is, we find the row containing the largest (absolute value) element in the first column. The largest element $a_{21} = 1$ is found in row $p = 2$. We swap the first row with the pth row.

$$\left[\begin{array}{ccc|c} 1 & 0 & 3 & 0 \\ 0 & 2 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right]$$

The next step would be to eliminate all the elements below the pivot position in column 1, but since they are already 0, we are done with column 1.

Now, we go to the next column (column two) and repeat the process, but this time, we examine all the elements in the second column except for the one in the first row. We already took care of the first row.

$$p = \text{argmax}_{2 \le i \le n} |a_{i2}| = 2.$$

This time, the largest element is already in the second row, so we don't have to perform any swapping. Next, we eliminate the 1 in the column below the pivot position. We do this by performing the row operation $R_3 = R_3 - R_2/2$.

$$\left[\begin{array}{ccc|c} 1 & 0 & 3 & 0 \\ 0 & 2 & 1 & 1 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{array}\right]$$

The final step is to do the **back substitution** to get our solutions.

$$\begin{aligned} x_3 &= \frac{\frac{1}{2}}{\frac{1}{2}} = 1 \\ x_2 &= \frac{1 - 1x_3}{2} = 0 \\ x_1 &= \frac{0 - 3x_3 - 0x_2}{1} = -3. \end{aligned}$$

Using partial pivoting is better than no pivoting at all, but it can still give large roundoff errors when dealing with floating point computation. An example is the **Hilbert matrix**, which is an $n \times n$ matrix whose elements are the unit fractions

$$a_{ij} = \frac{1}{i + j - 1}.$$

Suppose we are given

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}, \quad \vec{\boldsymbol{b}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Notice that A is the Hilbert matrix for $n = 3$. Suppose we want to solve $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ and we are limited to doing 2-digit arithmetic. Any digits after two are chopped off without rounding. Then our initial augmented matrix is

$$\left[\begin{array}{ccc|c} 1.0 & .50 & .33 & 0.0 \\ .50 & .33 & .25 & 0.0 \\ .33 & .25 & .20 & 1.0 \end{array}\right]$$

Looking at the first column, we see that the largest element is already in the first row, so we don't have to do any pivoting. Next, we eliminate the bottom two elements in the first column by performing the row operations $R_2 = R_2 - .50R_1$ and $R_3 = R_3 - .33R_1$.

$$\left[\begin{array}{ccc|c} 1.0 & .50 & .33 & 0.0 \\ 0.0 & .08 & .09 & 0.0 \\ 0.0 & .09 & .10 & 1.0 \end{array}\right]$$

Since $|a_{32}| > |a_{22}|$ we swap rows two and three.

$$\left[\begin{array}{ccc|c} 1.0 & .50 & .33 & 0.0 \\ 0.0 & .09 & .10 & 1.0 \\ 0.0 & .08 & .09 & 0.0 \end{array}\right]$$

Next, we eliminate a_{32} with the row operation $R_3 = R_3 - .88R_2$.

$$\left[\begin{array}{ccc|c} 1.0 & .50 & .33 & 0.0 \\ 0.0 & .09 & .10 & 1.0 \\ 0.0 & 0.0 & .01 & -.88 \end{array}\right]$$

Now, we are ready to perform the back substitution. We get

$$\begin{aligned} x_3 &= -88 \\ x_2 &= 9800 \\ x_1 &= -4900 \end{aligned}$$

The exact solution is

$$\begin{aligned} x_3 &= 180 \\ x_2 &= -180 \\ x_1 &= 30. \end{aligned}$$

So our approximation using only partial pivoting and two-digit arithmetic is way off. The Hilbert matrix is **ill conditioned**.

If $\tilde{\vec{\boldsymbol{x}}}$ is our approximation for the solution to $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$, then the **residual vector** is given by

$$\vec{\boldsymbol{r}} = \vec{\boldsymbol{b}} - A\tilde{\vec{\boldsymbol{x}}}.$$

This vector can be used to judge the error of our approximation. It is zero if our approximation is exact.

Suppose we want to write a computer program to solve a matrix using Gaussian elimination with partial pivoting.

To swap the pth row with the ith row in matrix A with elements a_{ij}, we would use something like

```
for j = i, ..., n
   temp = a[p,j]
   a[p,j] = a[i,j]
   a[i,j] = temp
```

Note, don't forget to swap the appropriate elements in $\vec{\boldsymbol{b}}$ if you swap rows in A.

A python program implementing the partial pivoting algorithm can be found on page 108.

Scaled Partial Pivoting

Scaled partial pivoting is similar to partial pivoting, but rows are scaled prior to pivoting.

We start by constructing the vector

$$\vec{\boldsymbol{s}} = \max_{1\leq \ell \leq n} |a_{k\ell}|\,,$$

where the elements of the vector are s_k. That is

$$\vec{\boldsymbol{s}}_k = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} = \begin{bmatrix} \max_{1\leq \ell \leq n} |a_{1\ell}| \\ \max_{1\leq \ell \leq n} |a_{2\ell}| \\ \vdots \\ \max_{1\leq \ell \leq n} |a_{n\ell}| \end{bmatrix}.$$

Note that s_1 is the largest (absolute value) element in row 1 of A, s_2 is the largest (absolute value) element in row 2 of A, and so on.

Then, when we calculate p (the number of the row with the largest value in the subcolumn containing the pivot and elements under it), we don't just look for the largest value in the subcolumn. We look for the largest *scaled* value. That is, for $i = 1, \ldots, n-1$ (i.e. for all the pivot points), we calculate

$$p = \text{argmax}_{i\leq j\leq n} \frac{|a_{ji}|}{s_j}.$$

The s_j in the denominator is the only difference between the scaled partial pivoting method and the regular partial pivoting method.

Let's look at the algorithm for solving $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ using Gaussian elimination with scaled partial pivoting.

The first step is to compute the vector containing our scaled factors. The pseudocode for this is

```
for k = 1,...,n
   s[k] = 0
   for j = 1,...,n
      if |A[k,j]| > |s[k]| then
         s[k] = |A[k,j]|
```

The next step is to perform the Gaussian elimination with partial pivoting. The pseudocode for this is

```
for i = 1, ...,n

   p = i
   for j = i,...,n
      if |A[j,i]|/s[j] > |A[p,i]|/s[p] then
         p = j

   if A[p,p] == 0
      ABORT

   for j = i,...n
      temp = A[p,j]
      A[p,j] = A[i,j]
      A[i,j] = temp

   temp = b[p]
   b[p] = b[i]
   b[i] = temp

   temp = s[p]
   s[p] = s[i]
   s[i] = temp

   for j = i+1,...,n
      alpha = -A[j,i]/A[i,i]
      for k = i+1,...,n
         A[j,k] = A[j,k] + alpha*A[i,k]
      b[j] = b[j] + alpha*b[i]
```

In the first second-level for loop above, we find the largest scaled value in each pivot subcolumn. In the following if condition, we abort the program if a_{pp} is zero because that would imply a singular matrix. In the next for loop, we perform the row swap, being careful to also swap the corresponding elements in $\vec{\boldsymbol{b}}$ and $\vec{\boldsymbol{s}}$. In the final for loop, we eliminate all the entries below the pivot.

The final step is to do the back-substitution to get our solution $\vec{\boldsymbol{x}}$.

```
x[n] = b[n]/A[n,n]
for i = n-1 down to 1
   x[i] = b[i]
   for j = i+1,...,n
      x[i] = x[i] - A[ij]* x[j]
   x[i] = x[i]/A[i,i]
```

What is the complexity of the scaled partial pivoting algorithm?

In the back-substitution step, we have 1 operation on the first line. The inner for loop requires $j = i+1, \ldots, n = n-(i+1)+1 = n-i$ multiplication/division operations for each i in $i = n-1, \ldots, 1$. This is a total of $1+2+3+\cdots+n-1 = n(n-1)/2$ operations for this innermost for loop. However, on the final line, we have an additional operation for each i. So the total operations performed in the back-substitution step is

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

That is, the operations performed in the back-substitution step alone is proportional to n^2.

If we calculate the total operations performed in the scaled partial pivoting algorithm, we find that it is proportional to n^3. That means this is a very slow algorithm. How can we improve on it?

If the matrix is **dense** (i.e. most values are nonzero), we are out of luck, although

we might be able to parallelize it so that multiple computers can be used. For **sparse matrices**, we can speed up the process.

For a sparse matrix system, the cost can be reduced.

- The conjugate gradient method can be used when the matrix is symmetric.
- Preconditioned conjugate gradient method
  - Jacobi preconditioning
  - Multi-grid preconditioning

A python program implementing the scaled partial pivoting algorithm can be found on page 109.

Complete Pivoting

Recall that for partial pivoting, we identified the largest element in the subcolumn containing the pivot. Then we swapped rows to bring that element to the pivot position.

With complete pivoting, we identify the largest element in the entire submatrix containing the pivot. Then we swap both rows and columns in order to bring that element to the pivot position. The tricky thing with swapping columns is that it changes the order of your solutions, so you have to keep track of those swaps and then do the reverse to the solution to get the elements of the solution vector in the right order in the end.

A python program implementing the complete pivoting algorithm can be found on page 111.

8.2 Matrix Norms

The L^2 norm of a vector $\vec{\boldsymbol{x}}$ can be written in the following ways

$$||\vec{\boldsymbol{x}}||_2 = \sqrt{(\vec{\boldsymbol{x}}, \vec{\boldsymbol{x}})} = \sqrt{\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{x}}} = \sqrt{\vec{\boldsymbol{x}}^T \vec{\boldsymbol{x}}} = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

This is the regular Euclidean norm of a vector.

The L^1 norm is

$$||\vec{\boldsymbol{x}}||_1 = \sqrt{\sum_{i=1}^{n} |x_i|}.$$

This is also known as the **taxicab norm**.

The L^∞ norm is

$$||\vec{\boldsymbol{x}}||_\infty = \max_i |x_i|.$$

The L^p norm of an $n \times n$ matrix is

$$||A||_p = \max_{||\vec{\boldsymbol{x}}||_p \neq 0} \frac{||A\vec{\boldsymbol{x}}||_p}{||\vec{\boldsymbol{x}}||_p} = \max_{||\vec{\boldsymbol{x}}||_p = 1} ||A\vec{\boldsymbol{x}}||_p.$$

Here, $\max_{||\vec{\boldsymbol{x}}||_p=1}$ means the maximum over all vectors $\vec{\boldsymbol{x}}$ of length n with p-norm equal to 1. Suppose $n = 1$. That is, the matrix A is just a scalar $A = a$. Then the L^p norm is

$$||A||_p = \max_{||\vec{\boldsymbol{x}}||_p=1} ||A\vec{\boldsymbol{x}}||_p = \max_{||\vec{\boldsymbol{x}}||_p=1} ||a\vec{\boldsymbol{x}}||_p = |a|.$$

As another example, suppose A is a diagonal matrix with the constant α on the diagonal. Then

$$||A||_p = \max_{||\vec{\boldsymbol{x}}||_p=1} ||A\vec{\boldsymbol{x}}||_p = ||\alpha\vec{\boldsymbol{x}}||_p = |\alpha|\, ||\vec{\boldsymbol{x}}||_p = |\alpha|.$$

In general, the infinity-norm of a matrix A can be calculated as

$$||A||_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|.$$

That is, for each row, you sum the absolute values of the elements in the row. Then the infinity-norm of the matrix is the largest of these row sums.

If A is an $n \times n$ matrix with a complete set of eigenvectors, then the L^2-norm is

$$||A||_2 = \max_{1\leq i\leq n} |\lambda_i|.$$

That is, it is the largest absolute value eigenvalue of A.

8.3 Jacobi Method

Given an $n \times n$ matrix A, the Jacobi method is an iterative way of approximating the solution $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$. It is only useful if A is a sparse matrix. That is, if most elements in A are zero. The Jacobi method is the most basic iterative method, but it is not very efficient.

The matrix equation $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ is equivalent to the linear system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad & \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned}$$

To use the Jacobi method, we make an initial guess, often

$$\vec{\boldsymbol{x}}^{(0)} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Then you plug this guess into the system as

$$\begin{aligned} a_{11}x_1^{(1)} + a_{12}x_2^{(0)} + \cdots + a_{1n}x_n^{(0)} &= b_1 \\ a_{21}x_1^{(0)} + a_{22}x_2^{(1)} + \cdots + a_{2n}x_n^{(0)} &= b_2 \\ \vdots \qquad\qquad & \qquad \vdots \\ a_{n1}x_1^{(0)} + a_{n2}x_2^{(0)} + \cdots + a_{nn}x_n^{(1)} &= b_n. \end{aligned}$$

Then you solve each row for the element on the diagonal to get your next guess

$$\vec{\boldsymbol{x}}^{(1)} = \begin{bmatrix} x_1^{(1)} \\ \vdots \\ x_n^{(1)} \end{bmatrix}.$$

Then you plug this into the system of equations and repeat.

Example 8.3.1

Approximate the solution to the following matrix equation using the Jacobi method.

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ -1 & 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Our first guess is $\vec{\boldsymbol{x}}^{(0)} = \vec{\mathbf{0}}$. Plugging this into the system of equations gives us

$$\begin{aligned} 2x_1^{(1)} + 1(0) + 0(0) &= 1 \\ 1(0) + 4x_2^{(1)} + 1(0) &= 2 \\ -1(0) + 2(0) + 5x_3^{(1)} &= 3. \end{aligned}$$

Solving for the second guess gives us

$$\vec{\boldsymbol{x}}^{(1)} = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{3}{5} \end{bmatrix}.$$

Repeating the process gives us

$$\begin{aligned} 2x_1^{(2)} + 1\left(\frac{1}{2}\right) + 0\left(\frac{3}{5}\right) &= 1 \\ 1\left(\frac{1}{2}\right) + 4x_2^{(2)} + 1\left(\frac{3}{5}\right) &= 2 \\ -1\left(\frac{1}{2}\right) + 2\left(\frac{1}{2}\right) + 5x_3^{(2)} &= 3. \end{aligned}$$

Solving for the next guess gives us

$$\vec{\boldsymbol{x}}^{(2)} = \begin{bmatrix} \frac{1}{4} \\ \frac{9}{40} \\ \frac{1}{2} \end{bmatrix}.$$

Repeating the process gives us

$$\begin{aligned} 2x_1^{(3)} + 1\left(\frac{9}{40}\right) + 0\left(\frac{1}{2}\right) &= 1 \\ 1\left(\frac{1}{4}\right) + 4x_2^{(3)} + 1\left(\frac{1}{2}\right) &= 2 \\ -1\left(\frac{1}{4}\right) + 2\left(\frac{9}{40}\right) + 5x_3^{(3)} &= 3. \end{aligned}$$

Solving for the next guess gives us

$$\vec{\boldsymbol{x}}^{(3)} = \begin{bmatrix} \frac{31}{80} \\ \frac{5}{16} \\ \frac{14}{25} \end{bmatrix} \approx \begin{bmatrix} 0.39 \\ 0.31 \\ 0.56 \end{bmatrix}.$$

Already, with only a few iterations, we are getting close to the actual answer

$$\vec{\boldsymbol{x}} = \begin{bmatrix} \frac{11}{30} \\ \frac{4}{15} \\ \frac{17}{30} \end{bmatrix} \approx \begin{bmatrix} 0.37 \\ 0.27 \\ 0.57 \end{bmatrix}.$$

In general, the "guess" for the next iteration can be computed as

$$\begin{aligned} x_1^{(k+1)} &= b_1 - \frac{a_{12}x_2^{(k)} + a_{13}x_3^{(k)} + \cdots + a_{1n}x_n^{(k)}}{a_{11}} \\ x_2^{(k+1)} &= b_2 - \frac{a_{21}x_1^{(k)} + a_{23}x_3^{(k)} + \cdots + a_{2n}x_n^{(k)}}{a_{11}} \\ &\vdots \qquad\qquad \vdots \\ x_n^{(k+1)} &= b_n - \frac{a_{n1}x_1^{(k)} + \cdots + a_{n,n-1}x_{n-1}^{(k)}}{a_{nn}} \end{aligned}$$

We can also write our matrix as

$$A = D - L - U,$$

where D is the diagonal matrix

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & \cdots & & a_{nn} \end{bmatrix},$$

L is the lower triangular matrix

$$L = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & 0 & \cdots & 0 \\ -a_{31} & -a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \\ -a_{n1} & -a_{n2} & \cdots & -a_{n,n-1} & 0 \end{bmatrix},$$

and U is the upper triangular matrix

$$U = \begin{bmatrix} 0 & -a_{12} & -a_{13} & \cdots & -a_{1n} \\ 0 & 0 & -a_{23} & \cdots & -a_{2n} \\ 0 & 0 & 0 & \cdots & -a_{3n} \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix},$$

Given $A = D - L - U$, we can write

$$\begin{aligned} A\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ (D - L - U)\,\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ D\vec{\boldsymbol{x}} &= (L + U)\,\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}} \\ \vec{\boldsymbol{x}} &= D^{-1}\,(L + U)\,\vec{\boldsymbol{x}} + D^{-1}\vec{\boldsymbol{b}}. \end{aligned}$$

This is now a multi-dimensional fixed-point problem. Any matrix system $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ can be written as a fixed point problem and can be approximated by Jacobi iteration.

Then the steps of the Jacobi method can be expressed as

$$\vec{\boldsymbol{x}}^{(k+1)} = D^{-1}\left[\vec{\boldsymbol{b}} + (L + U)\,\vec{\boldsymbol{x}}^{(k)}\right],$$

where the inverse of the previous diagonal matrix is

$$D^{-1} = \begin{bmatrix} -\frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & -\frac{1}{a_{22}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & \cdots & & -\frac{1}{a_{nn}} \end{bmatrix}.$$

The Jacobi algorithm can be written as

1. Give a guess $\vec{\boldsymbol{x}}^{(0)}$ and set $k = 0$.
2. While the residual $||\vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(k)}||$ is greater than some tolerance ε, compute

$$\vec{\boldsymbol{x}}^{(k+1)} = D^{-1}\left[\vec{\boldsymbol{b}} + (L + U)\,\vec{\boldsymbol{x}}^{(k)}\right].$$

In practice, the Jacobi method is not implemented in this manner. Since the Jacobi method is only beneficial for sparse matrices, any algorithm employing the method would take advantage of the sparsity of A.

For the multi-dimensional analogue of the fixed-point method, we write

$$\vec{\boldsymbol{x}}^{(k+1)} = \vec{\boldsymbol{g}}(\vec{\boldsymbol{x}}^{(k)}).$$

The condition for convergence is

$$\left|\left|\frac{\partial g}{\partial x}\right|\right| \leq k < 1.$$

For the Jacobi method,

$$\vec{\boldsymbol{g}}(\vec{\boldsymbol{x}}^{(k)}) = D^{-1}\,(L + U)\,\vec{\boldsymbol{x}}^{(k)} + D^{-1}\vec{\boldsymbol{b}}.$$

Differentiating gives us

$$\frac{\partial g}{\partial x} = D^{-1}\,(L + U)\,.$$

The condition for convergence is that

$$\left|\left|D^{-1}\,(L + U)\right|\right| \leq k < 1.$$

Example

Suppose you have

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}.$$

Will the Jacobi method converge for this matrix?

We can write

$$\vec{\boldsymbol{x}}^{(k+1)} = D^{-1}\left(L+U\right)\vec{\boldsymbol{x}}^{(k)} + D^{-1}\vec{\boldsymbol{b}} = T_J\vec{\boldsymbol{x}}^{(k)} + \vec{\boldsymbol{C}},$$

where

$$T_J = D^{-1}\left(L+U\right),$$

and

$$\vec{\boldsymbol{C}} = D^{-1}\vec{\boldsymbol{b}}.$$

Notice that $L+U$ is just A without its diagonal elements and D^{-1} contains just the negative reciprocals of the diagonal elements of A, so both are easy to calculate.

$$T_J = \begin{bmatrix} -\frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & 0 \\ 0 & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix}.$$

The infinity-norm of this matrix is

$$||T_J||_\infty = 1,$$

but what we really want is the L^2-norm. To find the L^2-norm, we need to find the eigenvalues of T_J, which turn out to be $\lambda = 0, \pm\frac{1}{\sqrt{2}}$. So the L^2-norm is

$$||T_J||_2 = \max_i|\lambda_i| = \frac{1}{\sqrt{2}}.$$

Since

$$||T_J||_2 = \left|\left|D^{-1}\left(L+U\right)\right|\right| = \frac{1}{\sqrt{2}} < 1,$$

we know that the Jacobi method will converge for this matrix.

Example

We will now analyze the Jacobi method for a sparse tridiagonal matrix of the specific form

$$A = \begin{bmatrix} \alpha & -\beta & 0 & \cdots & 0 \\ -\beta & \alpha & -\beta & & 0 \\ 0 & -\beta & \alpha & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\beta \\ 0 & 0 & \cdots & -\beta & \alpha \end{bmatrix},$$

where α and β are constants. For this matrix, the equivalent system of linear equations has the form

$$\begin{aligned} \alpha x_1 - \beta x_2 &= b_1 \\ -\beta x_1 + \alpha x_2 - \beta x_3 &= b_2 \\ \vdots \qquad & \qquad \vdots \\ -\beta x_{n-1} + \alpha x_n &= b_n. \end{aligned}$$

An algorithm that exploits the specific sparsity of this matrix is as follows:

1. Give a guess $\vec{\boldsymbol{x}}^{(0)}$ and set $k = 0$.

2. While the residual $||\vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(k)}||$ is greater than some tolerance ε, do all of the following:
   a) Set the first term
   $$x_1^{(k+1)} = \frac{b_1 + \beta x_2^{(k)}}{\alpha}.$$
   b) For all $i = 2, 3, \ldots, n-1$, compute
   $$x_i^{(k+1)} = \frac{b_i + \beta x_{i-1}^{(k)} + \beta x_{i+1}^{(k)}}{\alpha}.$$
   c) Set the last term
   $$x_n^{(k+1)} = \frac{b_n + \beta x_n^{(k)}}{\alpha}.$$

What's the complexity of the Jacobi method here? The operations done per iteration here is $3n-2$. How fast does this method converge to an acceptable approximation? That is, how many iterations have to be done? If it converges in n^2 iterations, for example, then the total operations are $(3n-2)n^2$, which is the same as for partial pivoting. If that is the case, then Jacobi iteration gives us no advantage.

An algorithm that exploits the specific sparsity of a general tridiagonal matrix is as follows:

1. Give a guess $\vec{\boldsymbol{x}}^{(0)}$ and set $k = 0$.
2. While the residual $||\vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(k)}||$ is greater than some tolerance ε, do all of the following:
   a) Set the first term
   $$x_1^{(k+1)} = \frac{b_1 - U_1 x_2^{(k)}}{D_1}.$$
   b) For all $i = 2, 3, \ldots, n-1$, compute
   $$x_i^{(k+1)} = \frac{b_i - L_{i-1} x_{i-1}^{(k)} - U_i x_{i+1}^{(k)}}{D_i}.$$
   c) Set the last term
   $$x_n^{(k+1)} = \frac{b_n - L_{n-1} x_n^{(k)}}{D_n}.$$
   d) Set $k+ = 1$ to keep track of the number of iterations that were done.

Here, $\vec{\boldsymbol{D}}$ is the vector containing the diagonal elements of A, $\vec{\boldsymbol{U}}$ is the vector containing the line of elements above the diagonal, and $\vec{\boldsymbol{L}}$ is the vector containing the line of elements below the diagonal.

Spectral Radius

Recall that
$$\vec{\boldsymbol{x}}^{(k+1)} = D^{-1}(L+U)\vec{\boldsymbol{x}}^{(k)} + D^{-1}\vec{\boldsymbol{b}} = T_J\vec{\boldsymbol{x}}^{(k)} + \vec{\boldsymbol{C}}.$$
where
$$T_J = \frac{\partial \vec{\boldsymbol{g}}}{\partial \vec{\boldsymbol{x}}}.$$
Computing $\vec{\boldsymbol{x}}^{(k+1)}$ for the first several k
$$\begin{aligned}
\vec{\boldsymbol{x}}^{(1)} &= T_J\vec{\boldsymbol{x}}^{(0)} + \vec{\boldsymbol{C}} \\
\vec{\boldsymbol{x}}^{(2)} &= T_J^2\vec{\boldsymbol{x}}^{(0)} + T_J\vec{\boldsymbol{C}} + \vec{\boldsymbol{C}} \\
\vec{\boldsymbol{x}}^{(3)} &= T_J^3\vec{\boldsymbol{x}}^{(0)} + T_J^2\vec{\boldsymbol{C}} + T_J\vec{\boldsymbol{C}} + \vec{\boldsymbol{C}},
\end{aligned}$$

shows that we can write

$$\vec{\boldsymbol{x}}^{(k)} = T_J^k \vec{\boldsymbol{x}}^{(0)} + \sum_{i=0}^{k-1} T_J^i \vec{\boldsymbol{C}}.$$

For an $n \times n$ matrix A, the **spectral radius** of A is the largest (absolute value) eigenvalue of A

$$\rho(A) = \max_i |\lambda_i|.$$

The following are all equivalent. If any of them are true, then the Jacobi method will converge for A.

$$\begin{aligned} A^k &\to 0 \\ \lim_{n\to\infty} ||A^n||_2 &= 0 \\ \lim_{n\to\infty} ||A^n||_p &= 0, \quad \text{for any valid } p\text{-norm} \\ \rho(A) &< 1 \\ \lim_{n\to\infty} A^n \vec{\boldsymbol{x}} &= 0, \quad \text{for any } \vec{\boldsymbol{x}}. \end{aligned}$$

We can write the residual between the solution $\vec{\boldsymbol{x}}$ and our approximation after k Jacobi iterations, $\vec{\boldsymbol{x}}^k$, as

$$||\vec{\boldsymbol{x}}^{(k)} - \vec{\boldsymbol{x}}||_2 = ||T_J^k \vec{\boldsymbol{x}}^{(0)} + \sum_{i=0}^{k-1} T_J^i \vec{\boldsymbol{c}} - \vec{\boldsymbol{x}}||_2.$$

If $\rho(T_J) < 1$, we can treat $\sum_{i=0}^{k-1} T^i$ as a geometric series. The partial sum of such a series is

$$\sum_{i=0}^{k-1} T_J^i = \frac{1 - T_J^k}{1 - T_J}.$$

Earlier, we found that $\vec{\boldsymbol{x}} = T_J \vec{\boldsymbol{x}} + \vec{\boldsymbol{c}}$, which gives us $\vec{\boldsymbol{c}} = \vec{\boldsymbol{x}}(1 - T_J)$. Making these substitutions, gives us

$$\begin{aligned} ||\vec{\boldsymbol{x}}^{(k)} - \vec{\boldsymbol{x}}||_2 &= || \left(T_J^k \vec{\boldsymbol{x}}^{(0)} + \sum_{i=0}^{k-1} T_J^i \vec{\boldsymbol{c}} \right) - \vec{\boldsymbol{x}}||_2 = ||T_J^k \vec{\boldsymbol{x}}^{(0)} + \frac{1 - T_J^k}{1 - T} \vec{\boldsymbol{x}}(1 - T_J) - \vec{\boldsymbol{x}}||_2 \\ &= ||T_J^k \vec{\boldsymbol{x}}^{(0)} - T_J^k \vec{\boldsymbol{x}}||_2 = ||T_J^k (\vec{\boldsymbol{x}}^{(0)} - \vec{\boldsymbol{x}})||_2. \end{aligned}$$

If we take the limit of the residual as the number of iterations goes to infinity, then if $\rho(T_J) < 1$, then by the equivalencies listed earlier, we know that $\lim_{k\to\infty} T_J^k (\vec{\boldsymbol{x}}^{(0)} - \vec{\boldsymbol{x}}) = 0$. Therefore, if $\rho(T_J) < 1$, then the Jacobi method will converge since

$$\lim_{k\to\infty} ||\vec{\boldsymbol{x}}^{(k)} - \vec{\boldsymbol{x}}||_2 = 0.$$

Example

Consider the **heat equation** as a one-dimensional boundary problem

$$\begin{aligned} U_{xx} &= f \\ U(0) &= \alpha \\ U(1) &= \beta. \end{aligned}$$

We can discretize it as $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$, where A is a tridiagonal matrix of the form

$$A = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & 0 \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & & \ddots & \ddots & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}.$$

The more accurate we want to approximate the solution, the more rows and columns we use.

The cost for the direct method that uses partial pivoting is $O(n^3)$ for this system. Then $T_J = D^{-1}(L+U)$ is

$$T_J = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \cdots & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & & 0 \\ 0 & \frac{1}{2} & 0 & \ddots & 0 \\ \vdots & & \ddots & \ddots & \frac{1}{2} \\ 0 & 0 & \cdots & \frac{1}{2} & 0 \end{bmatrix}.$$

If we have an $n \times n$ **tridiagonal matrix** with constants along the diagonals as in

$$\begin{bmatrix} \delta & \tau & 0 & \cdots & 0 \\ \sigma & \delta & \tau & & 0 \\ 0 & \sigma & \delta & \ddots & 0 \\ \vdots & & \ddots & \ddots & \tau \\ 0 & 0 & \cdots & \sigma & \delta \end{bmatrix},$$

then the eigenvalues are

$$\lambda_\ell = \delta + 2\sqrt{\sigma\tau}\cos\left(\frac{\ell\pi}{n+1}\right),$$

where $\ell = 1, 2, \ldots, n$. A matrix in which each diagonal is constant is a **Toeplitz matrix**.

In our case, the eigenvalues of T_J are

$$\lambda_\ell = \cos\left(\frac{\ell\pi}{n+1}\right),$$

so the spectral radius of T_J is

$$||T_J||_2 = \rho(T_J) = \cos\left(\frac{\ell\pi}{n+1}\right).$$

Suppose $n >> 1$, that is, we are dealing with a very large Toeplitz matrix, then for $\ell = 1$, we have

$$\lambda_1 = \cos\left(\frac{\pi}{n+1}\right) \approx 1 - \frac{\left(\frac{\pi}{n+1}\right)^2}{2}.$$

The residual is

$$||\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^{(m)}|| = ||T_J^m(\vec{\boldsymbol{x}} - \vec{\boldsymbol{x}}^{(0)})||$$

where m is the number of iterations.

8.4 Conjugate Gradient Method

When dealing with sparse matrices, we don't store the complete matrix A in the computer. When solving the heat equation in 1D, for example, A will be a tridiagonal matrix, so instead of storing the 2D matrix A, we store the three bands as 1D vectors. For the 2D heat equation, we would have 5 bands instead of 3.

Recall that with the Jacobi method, we decompose the matrix A as $A = D - L - U$. The more general approach for solving $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ is to split A into one part that you can find the inverse for much easier than for A. For example, we want to write $A = M - N$ where M^{-1} is easy to find. Then we can write

$$\begin{aligned} A\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ (M-N)\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ M\vec{\boldsymbol{x}} &= N\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}} \\ \vec{\boldsymbol{x}} &= M^{-1}N\vec{\boldsymbol{x}} + M^{-1}\vec{\boldsymbol{b}}. \end{aligned}$$

This is the equivalent fixed-point problem

$$\vec{\boldsymbol{x}}^{(k+1)} = M^{-1}N\vec{\boldsymbol{x}}^{(k)} + M^{-1}\vec{\boldsymbol{b}},$$

which we can write as

$$\vec{\boldsymbol{x}}^{(k+1)} = T\vec{\boldsymbol{x}}^{(k)} + \vec{\boldsymbol{c}},$$

where $T = M^{-1}N$ and $\vec{\boldsymbol{c}} = M^{-1}\vec{\boldsymbol{b}}$. This defines a general class of algorithms of which the Jacobi method is a member.

If M is nonsingular and $\rho(T) < 1$, then you can prove that the error is

$$||\vec{\boldsymbol{x}}^{(k)} - \vec{\boldsymbol{x}}||_2 = ||T^k(\vec{\boldsymbol{x}}^{(0)} - \vec{\boldsymbol{x}})||_2.$$

If T is symmetric and has a complete set of eigenvalues and eigenvectors, then

$$||\vec{\boldsymbol{x}}^{(k)} - \vec{\boldsymbol{x}}||_2 \leq ||T||_2^k ||\vec{\boldsymbol{x}}^{(0)} - \vec{\boldsymbol{x}})||_2,$$

where $||T||_2^k = \rho(T)^k$. Unfortunately, for PDEs, $\rho(T)^k \sim 1$, so this method converges very slowly.

For the **Gauss-Seidel method**, we solve $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ by writing $A = D - L - U$ like with the Jacobi method. But then we write $M = D - L$ and $N = U$.

$$\begin{aligned} A\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ (D-L-U)\vec{\boldsymbol{x}} &= \vec{\boldsymbol{b}} \\ (D-L)\vec{\boldsymbol{x}} &= U\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}} \\ \vec{\boldsymbol{x}}^{(k+1)} &= (D-L)^{-1}U\vec{\boldsymbol{x}}^{(k)} + (D-L)^{-1}\vec{\boldsymbol{b}}. \end{aligned}$$

Unfortunately, T has a poor spectral radius for the Gauss-Seidel method, so it converges slowly.

If A is symmetric (i.e. $A^T = A$) and has real values, then solving $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ is equivalent to solving the minimization problem

$$\vec{\boldsymbol{x}} = \text{argmin}_{\vec{\boldsymbol{x}}} \left(g(\vec{\boldsymbol{x}}) \right),$$

where

$$g(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} - 2\vec{\boldsymbol{b}}^T\vec{\boldsymbol{x}},$$

provided that A is symmetric and positive definite. The motivation for the conjugate gradient method, is that it is much faster to minimize $g(\vec{\boldsymbol{x}})$ than it is to solve $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$.

This can also be written as

$$g(\vec{\boldsymbol{x}}) = \langle \vec{\boldsymbol{x}}, A\vec{\boldsymbol{x}} \rangle - 2\langle \vec{\boldsymbol{x}}, \vec{\boldsymbol{b}} \rangle,$$

where

$$\langle \vec{\boldsymbol{x}}, \vec{\boldsymbol{y}} \rangle = \vec{\boldsymbol{x}}^T \vec{\boldsymbol{y}} = \sum_{i=1}^{n} x_i y_i,$$

is the dot product.

A is **positive definite** if $\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} > 0$ for any $\vec{\boldsymbol{x}} \neq \vec{\boldsymbol{0}}$. If A is symmetric and positive definite, then A has only positive eigenvalues.

To minimize a multivariable function $g(x_1, \ldots, x_n)$, we find where the gradient is zero. Recall that the gradient is defined as

$$\nabla g(x_1, \ldots, x_n) = \begin{bmatrix} \frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial g}{\partial x_n} \end{bmatrix}.$$

Suppose our problem $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ is

$$\begin{aligned} 2x_1 + x_2 &= 5 \\ x_1 + 2x_2 &= 1. \end{aligned}$$

Then

$$\begin{aligned} g(x_1, x_2) &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 2\begin{bmatrix} 5 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix} - 10x_1 - 2x_2 \\ &= (2x_1 + x_2)x_1 + (x_2 + 2x_2)x_2 - 10x_1 - 2x_2. \end{aligned}$$

Taking the gradient gives us

$$\nabla g(x_1, x_2) = \begin{bmatrix} 4x_1 + 2x_2 - 10 \\ x_1 + 4x_2 - 2 \end{bmatrix}.$$

Setting this equal to zero gives us

$$\begin{aligned} 4x_1 + 2x_2 &= 10 \\ 2x_1 + 4x_2 &= 2. \end{aligned}$$

This is the same as our original A, demonstrating that solving $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$ for a positive definite and symmetric A is equivalent to finding $\operatorname{argmin}_{\vec{\boldsymbol{x}}} \left(\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} - 2\vec{\boldsymbol{b}}^T \vec{\boldsymbol{x}} \right)$.

To motivate the conjugate gradient method, we will first look at the simpler **steepest descent method**. In general, we have

$$g(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} - 2\vec{\boldsymbol{x}}^T \vec{\boldsymbol{b}},$$

and

$$\nabla g = 2A\vec{\boldsymbol{x}} - 2\vec{\boldsymbol{b}}.$$

Then given $\vec{\boldsymbol{x}}^{(0)})$, we compute the next iteration as

$$\vec{\boldsymbol{x}}^{(1)} = \vec{\boldsymbol{x}}^{(0)} - t\,(\nabla g) = \vec{\boldsymbol{x}}^{(0)} + t\vec{\boldsymbol{v}},$$

where

$$\vec{\boldsymbol{v}} = -\vec{\boldsymbol{\nabla}} g = 2\vec{\boldsymbol{b}} - 2A\vec{\boldsymbol{x}} = 2\vec{\boldsymbol{r}},$$

where $\vec{\boldsymbol{r}}$ is the residual vector.

Note that $g(\vec{\boldsymbol{x}})$ forms an n-dimensional surface, and our goal is to locate the minimum of this surface. The initial guess $\vec{\boldsymbol{x}}^{(0)}$ is a point on this surface, and in every iteration, we want to move toward the minimum of $g(\vec{\boldsymbol{x}})$. With the steepest descent method, we just look at the gradient at $\vec{\boldsymbol{x}}^{(0)}$ and move in the opposite direction since the gradient points "uphill".

Our basic algorithm for the steepest descent method is as follows:

1. Given $\vec{\boldsymbol{x}}^{(0)}$ and set the counter $k = 0$.
2. While $||\nabla g(x^{(k)})||_2 > \varepsilon$,

$$\vec{\boldsymbol{x}}^{(k+1)} = \vec{\boldsymbol{x}}^{(k)} + t_k \vec{\boldsymbol{v}}_k$$
$$k+ = 1$$

where $\vec{\boldsymbol{v}}_k = -\vec{\boldsymbol{\nabla}} g(x^{(k)})$. Note that t is just a number and changes at each iteration. To find t_k, we solve a simpler minimization problem $h(t) = g(\vec{\boldsymbol{x}} + t\vec{\boldsymbol{v}})$ by finding where $\frac{\partial h}{\partial t} = 0$. This is now a minimization problem in 1 variable instead of in n variables.

$$\begin{aligned} h(t) &= g(\vec{\boldsymbol{x}} + t\vec{\boldsymbol{v}}) \\ &= (\vec{\boldsymbol{x}} + t\vec{\boldsymbol{v}})^T A (\vec{\boldsymbol{x}} + t\vec{\boldsymbol{v}}) - 2(\vec{\boldsymbol{x}} + t\vec{\boldsymbol{v}})^T \vec{\boldsymbol{b}} \\ &= \vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} + t\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} + t\vec{\boldsymbol{v}}^T A\vec{\boldsymbol{x}} + t^2 \vec{\boldsymbol{v}}^T A\vec{\boldsymbol{v}} - 2\vec{\boldsymbol{x}}^T \vec{\boldsymbol{b}} - 2t\vec{\boldsymbol{v}}^T \vec{\boldsymbol{b}} \\ &= \vec{\boldsymbol{x}}^T A\vec{\boldsymbol{x}} + 2t\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} + t^2 \vec{\boldsymbol{v}}^T A\vec{\boldsymbol{v}} - 2\vec{\boldsymbol{x}}^T \vec{\boldsymbol{b}} - 2t\vec{\boldsymbol{v}}^T \vec{\boldsymbol{b}}. \end{aligned}$$

Here we used the fact that $\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} = \vec{\boldsymbol{v}}^T A\vec{\boldsymbol{x}}$. We can do this because the order of the two vectors used in a dot product does not matter and $A^T = A$ since A is symmetric.

$$\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} = \langle \vec{\boldsymbol{x}}, A\vec{\boldsymbol{v}} \rangle = \langle A\vec{\boldsymbol{v}}, \vec{\boldsymbol{x}} \rangle = (A\vec{\boldsymbol{v}})^T \vec{\boldsymbol{x}} = \vec{\boldsymbol{v}}^T A^T \vec{\boldsymbol{x}} = \vec{\boldsymbol{v}}^T A\vec{\boldsymbol{x}}.$$

The derivative we want is

$$\frac{\partial h}{\partial t} = 2\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} + 2t\vec{\boldsymbol{v}}^T A\vec{\boldsymbol{v}} - 2\vec{\boldsymbol{v}}^T \vec{\boldsymbol{b}}.$$

Setting it equal to zero and solving for t gives us

$$t = \frac{-\vec{\boldsymbol{x}}^T A\vec{\boldsymbol{v}} + \vec{\boldsymbol{v}}^T \vec{\boldsymbol{b}}}{\vec{\boldsymbol{v}}^T A\vec{\boldsymbol{v}}} = \frac{\vec{\boldsymbol{v}}^T \left(-A\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}}\right)}{\vec{\boldsymbol{v}}^T A\vec{\boldsymbol{v}}}.$$

We can now write our steepest descent algorithm as follows:

1. Given $\vec{\boldsymbol{x}}^{(0)}$
2. Set the counter $k = 0$
3. Set $\vec{\boldsymbol{v}}^{(0)} = 2\vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(0)}$. Note that $\vec{\boldsymbol{v}}$ is like the residual.
4. While $||\vec{\boldsymbol{v}}^{(k)}|| > \varepsilon$,

$$\begin{aligned} t_k &= \frac{\vec{\boldsymbol{v}}^{(k)} \cdot \vec{\boldsymbol{v}}^{(k)}}{2\vec{\boldsymbol{v}}^{(k)} \cdot A\vec{\boldsymbol{v}}^{(k)}} \\ \vec{\boldsymbol{x}}^{(k+1)} &= \vec{\boldsymbol{x}}^{(k)} + t_k \vec{\boldsymbol{v}}_k \\ \vec{\boldsymbol{v}}^{(k+1)} &= 2\left(\vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(k+1)}\right) \\ k &= k + 1. \end{aligned}$$

This is very similar to the conjugate gradient method, but with that method, we use a different vector $\vec{\boldsymbol{v}}$.

8.5 Sparse Matrix Techniques

In this section, we looked at several algorithms for solving sparse matrix systems. In general, to solve $A\vec{x} = \vec{b}$ using **fixed point methods**, we write $A = M - N$, such that M is easy to invert. Then we have

$$\begin{aligned} A\vec{x} &= \vec{b} \\ (M-N)\vec{x} &= \vec{b} \\ M\vec{x} &= N\vec{x} + \vec{b} \\ \vec{x} &= M^{-1}N\vec{x} + M^{-1}\vec{b}. \end{aligned}$$

This is the equivalent fixed-point problem

$$\vec{x}^{(k+1)} = M^{-1}N\vec{x}^{(k)} + M^{-1}\vec{b}.$$

Jacobi Method

For the **Jacobi method**, $M = D$ is diagonal (so it is trivial to invert), and $N = L + U$. The fixed point problem is then

$$\vec{x}^{(k+1)} = D^{-1}(L+U)\vec{x}^{(k)} + D^{-1}\vec{b}.$$

If A is $n \times n$, then the complexity for a tridiagonal system is n per iteration, and the number of iterations needed is n^2, so the total complexity is $O(n^3)$.

Steepest Descent Method

For the **steepest descent method**, for $A\vec{x} = \vec{b}$, find $\min(g(x))$, where $g(x) = \vec{x}^T A\vec{x} - 2\vec{x}^T\vec{b}$. Then you compute $\nabla g = 2(A\vec{x} - \vec{b}) = -2\vec{r}$ where $\vec{r} = \vec{b} - A\vec{x}$ is the residual vector.

The algorithm for the steepest descent method is as follows:

1. Given $\vec{x}^{(0)}$, set $\vec{v}^{(1)} = \vec{r}^{(0)}$, where $\vec{r}^{(k)} = \vec{b} - A\vec{x}^{(k)}$.
2. For each $k = 1, 2, 3, \ldots$, do the following:

$$\begin{aligned} T_k &= \frac{\vec{v}^{(k)} \cdot \vec{r}^{(k-1)}}{\vec{v}^{(k)} \cdot A\vec{v}^{(k)}} \\ \vec{x}^{(k)} &= \vec{x}^{k-1} + T_k\vec{v}^{(k)} \\ \vec{v}^{(k+1)} &= \vec{r}^{(k)}. \end{aligned}$$

If A is a $n \times n$ tridiagonal matrix, what is the complexity of $\vec{v}^{(k)} \cdot \vec{r}^{(k-1)}$? Note that $\vec{v}^{(k)}$ and $\vec{r}^{(k-1)}$ are just vectors of length n, so their dot product requires n multiplications and n additions, for a total of $2n$ operations. For $\vec{v}^{(k)} \cdot A\vec{v}^{(k)}$, there are $3n+2n$ operations. The total complexity of each iteration is proportional to n, but how many iterations does it take to converge? The steepest descent algorithm is slow, and it actually takes more iterations to converge than the Jacobi method.

Conjugate Gradient Method

The **conjugate gradient method** uses a better choice for the search direction $\vec{v}$. However, for this method, A must be symmetric and positive definite.

The algorithm for the conjugate gradient is as follows:

1. Given $\vec{x}^{(0)}$, set $\vec{v}^{(1)} = \vec{r}^{(0)}$, where $\vec{r}^{(k)} = \vec{b} - A\vec{x}^{(k)}$.

2. For each $k = 1, 2, 3, \ldots$, do the following:

$$\begin{aligned} T_k &= \frac{\vec{\boldsymbol{v}}^{(k)} \cdot \vec{\boldsymbol{r}}^{(k-1)}}{\vec{\boldsymbol{v}}^{(k)} \cdot A\vec{\boldsymbol{v}}^{(k)}} \\ \vec{\boldsymbol{x}}^{(k)} &= \vec{\boldsymbol{x}}^{k-1} + T_k \vec{\boldsymbol{v}}^{(k)} \\ \vec{\boldsymbol{r}}^{(k)} &= \vec{\boldsymbol{b}} - A\vec{\boldsymbol{x}}^{(k)} \\ s_k &= \frac{\vec{\boldsymbol{r}}^{(k)} \cdot \vec{\boldsymbol{r}}^{(k)}}{\vec{\boldsymbol{r}}^{(k-1)} \cdot \vec{\boldsymbol{r}}^{(k-1)}} \\ \vec{\boldsymbol{v}}^{(k+1)} &= \vec{\boldsymbol{r}}^{(k)} + s_k v^{(k)}. \end{aligned}$$

Notice that if you set $s_k = 0$, you get the steepest descent method.

The properties of the conjugate gradient method include the following:

- Orthogonality of A.
$$\vec{\boldsymbol{v}}^{(i)} \cdot A\vec{\boldsymbol{v}}^{(j)} = 0, \quad \text{for } i \neq j.$$
This is a good sanity test to use to check your conjugate gradient algorithm.
- Instead of using $\vec{\boldsymbol{v}}^{(k)} \cdot \vec{\boldsymbol{r}}^{(k-1)}$ in the conjugate gradient algorithm, it is common to use $\vec{\boldsymbol{r}}^{(k-1)} \cdot \vec{\boldsymbol{r}}^{(k-1)}$ to slightly speed of the algorithm since these inner products are the same. That is,
$$\vec{\boldsymbol{r}}^{(k-1)} \cdot \vec{\boldsymbol{v}}^{(i)} = 0, \quad \text{for } i = 1, 2, \ldots, k-1.$$
- The conjugate gradient method converges in at most n iterations. This is another sanity check for your algorithm. The residual vector needs to be approximately 0 after n iterations.

Overall, the complexity of the conjugate gradient method is $O(n^2)$ operations for our sparse matrix. However, most people used a **preconditioned** conjugate gradient method. The main ones include

- Jacobi preconditioner
- Multigrid preconditioner
- Incomplete LU preconditioner

By using a preconditioner, you can often get the complexity down to $O(n \ln n)$. For tridiagonal matrices, you can actually solve them exactly in $O(n)$ operations.

A python program implementing several iterative methods to solve a particular kind of sparse tridiagonal matrix can be found on page 112.

Chapter 9

Eigenvalues

Eigenvalues are often used to analyze the stability of systems.

The vectors $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_k$ are **linearly independent** if $a_1 = a_2 = \cdots = a_k = 0$ is the only solution to

$$a_1\vec{\boldsymbol{v}}_1 + a_2\vec{\boldsymbol{v}}_2 + \cdots + a_k\vec{\boldsymbol{v}}_k = \vec{\mathbf{0}}.$$

Given $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$ linearly independent vectors in $\mathbb{R}^n$, then they form a **basis** for $\mathbb{R}^n$. That is, given any vector $\vec{\boldsymbol{x}}$ in $\mathbb{R}^n$, there exist coefficients $b_1, b_2, \ldots, b_n$, such that

$$\vec{\boldsymbol{x}} = b_1\vec{\boldsymbol{v}}_1 + b_2\vec{\boldsymbol{v}}_2 + \cdots + b_n\vec{\boldsymbol{v}}_n.$$

Here, the set $\{b_i\}$ are the coordinates of $\vec{\boldsymbol{x}}$ relative to the basis $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$. We can also write the equation above as the matrix equation

$$\begin{bmatrix} \vec{\boldsymbol{v}}_1 & \vec{\boldsymbol{v}}_2 & \cdots & \vec{\boldsymbol{v}}_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \vec{\boldsymbol{x}},$$

or more succinctly as

$$A\vec{\boldsymbol{b}} = \vec{\boldsymbol{x}}.$$

Since $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$ are independent, we know that A^{-1} exists.

The vectors $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$ form an **orthogonal** set if

$$\vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{v}}_j = \begin{cases} 0 & \text{if } i \neq j \\ \text{not } 0 & \text{if } i = j \end{cases}.$$

They form an **orthonormal basis** if

$$\vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{v}}_j = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}.$$

If the vectors $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$ form an orthonormal basis, then they are linearly independent.

Any orthogonal matrix Q satisfies the relation

$$Q^{-1} = Q^T.$$

If $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_n$ form an orthonormal basis in $\mathbb{R}^n$, then

$$Q = \begin{bmatrix} \vec{\boldsymbol{v}}_1 & \vec{\boldsymbol{v}}_2 & \cdots & \vec{\boldsymbol{v}}_n \end{bmatrix},$$

is an orthogonal matrix.

A **permutation matrix** is a matrix formed by permuting the rows of an identity matrix. For example,

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

is an example of a permutation matrix. Permutation matrices are orthogonal. To see that P is orthogonal, just check that $P^T P = I$.

The matrices A and B are **similar** if

$$A = S^{-1}BS.$$

If A is similar to B, then they share the same eigenvalues, and if $\vec{\boldsymbol{x}}$ is an eigenvector of A, then $S\vec{\boldsymbol{x}}$ is an eigenvector of B. This can be shown as follows. Assuming $A\vec{\boldsymbol{x}} = \lambda\vec{\boldsymbol{x}}$, then

$$\begin{aligned} A &= S^{-1}BS \\ A\vec{\boldsymbol{x}} &= S^{-1}BS\vec{\boldsymbol{x}} \\ \lambda\vec{\boldsymbol{x}} &= S^{-1}BS\vec{\boldsymbol{x}} \\ S\lambda\vec{\boldsymbol{x}} &= BS\vec{\boldsymbol{x}} \\ B\left(S\vec{\boldsymbol{x}}\right) &= \lambda\left(S\vec{\boldsymbol{x}}\right). \end{aligned}$$

Given any $n \times n$ matrix A, there exists a T and U such that

$$T = U^{-1}AU,$$

where T is an upper triangular matrix with the eigenvalues of A on the diagonal, and U is a unitary matrix which satisfies

$$||U\vec{\boldsymbol{x}}||_2 = ||\vec{\boldsymbol{x}}||_2.$$

The eigenvectors of A are also easy to find. Suppose $\vec{\boldsymbol{y}}$ is an eigenvector of T, then $U\vec{\boldsymbol{y}}$ is an eigenvector of A since

$$\begin{aligned} T\vec{\boldsymbol{y}} &= \lambda\vec{\boldsymbol{y}} \\ U^{-1}AU\vec{\boldsymbol{y}} &= \lambda\vec{\boldsymbol{y}} \\ A\left(U\vec{\boldsymbol{y}}\right) &= \lambda\left(U\vec{\boldsymbol{y}}\right). \end{aligned}$$

The tricky part is finding a matrix U that works.

If A is symmetric, then there exists an orthogonal matrix Q such that

$$D = Q^{-1}AQ,$$

where D is a diagonal matrix with the eigenvalues of A along the diagonal of D.

9.1 The Power Method

The power method gives the eigenvalue with the largest magnitude in a matrix A.

Let $\vec{\boldsymbol{x}}$ be an initial guess for the eigenvector. A good choice for $\vec{\boldsymbol{x}}$ is the row of A in which the infinity norm $|A_{i*}|_\infty$ is the largest. The power method will not work if $\vec{\boldsymbol{x}}$ is orthogonal to the eigenvector you're trying to find. Let $\vec{\boldsymbol{a}}_i$ be the ith row of A, then let

$$\vec{\boldsymbol{x}} = \text{argmax}_{\vec{\boldsymbol{a}}_i}|\vec{\boldsymbol{a}}_i|_\infty.$$

The pseudocode of the power method is as follows:

1. Set $k = 1$
2. Set p equal to the index of the largest (absolute magnitude) component of $\vec{\boldsymbol{x}}$.
3. Next, normalize $\vec{\boldsymbol{x}}$ so that $|\vec{\boldsymbol{x}}|_\infty = 1$ by
$$\vec{\boldsymbol{x}} = \frac{\vec{\boldsymbol{x}}}{x_p}.$$
4. While $k \leq N$, do the following:
   a) Set
   $$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}}.$$
   b) Set
   $$\mu = y_p.$$
   c) Set p equal to the index of the largest (absolute magnitude) component of $\vec{\boldsymbol{y}}$.
   d) If $y_p = 0$, then stop because A has a zero eigenvalue with $\vec{\boldsymbol{x}}$ as the corresponding eigenvector. Otherwise, continue.
   e) Compute the error as
   $$err = \left|\left|\vec{\boldsymbol{x}} - \frac{\vec{\boldsymbol{y}}}{y_p}\right|\right|_\infty.$$
   f) Set
   $$\vec{\boldsymbol{x}} = \frac{\vec{\boldsymbol{y}}}{y_p}.$$
   g) If the error is less than a given error tolerance, stop. Your eigenvalue is μ and $\vec{\boldsymbol{x}}$ is the corresponding eigenvector. Otherwise continue.
   h) Set $k = k + 1$.

Example 9.1.1

Use the power method to approximate the dominant eigenvalue and its associated eigenvector of
$$A = \frac{1}{2}\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}.$$
Choose your initial guess to be the first row of A, that is,
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}.$$
We set $p = 1$, the index of the largest component of $\vec{\boldsymbol{x}}$. Then we normalize $\vec{\boldsymbol{x}}$ such that the infinity-norm is 1
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \frac{1}{3} \end{bmatrix}.$$
Now we begin the first iteration of our while loop.
1. Set $\vec{\boldsymbol{y}}$ as
$$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}} = \begin{bmatrix} \frac{5}{3} \\ 1 \end{bmatrix}.$$
2. Set $\mu = 5/3$, the pth component of $\vec{\boldsymbol{y}}$.
3. Set $p = 1$, the index of the largest component of $\vec{\boldsymbol{y}}$.
4. The error is now $err = 4/15$.
5. Set
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \frac{3}{5} \end{bmatrix}.$$

Now we repeat the loop.

1. Set $\vec{\boldsymbol{y}}$ as
$$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}} = \begin{bmatrix} \frac{9}{5} \\ \frac{7}{5} \end{bmatrix}.$$
2. Set $\mu = 9/5$, the pth component of $\vec{\boldsymbol{y}}$.
3. Set $p = 1$, the index of the largest component of $\vec{\boldsymbol{y}}$.
4. Set
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \frac{7}{9} \end{bmatrix}.$$

Let's repeat the loop one more time.

1. Set $\vec{\boldsymbol{y}}$ as
$$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}} = \begin{bmatrix} \frac{17}{9} \\ \frac{5}{3} \end{bmatrix}.$$
2. Set $\mu = 17/9$, the pth component of $\vec{\boldsymbol{y}}$.
3. Set $p = 1$, the index of the largest component of $\vec{\boldsymbol{y}}$.
4. Set
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \frac{45}{51} \end{bmatrix}.$$

So after three iterations, our approximation of the dominant eigenvalue of A and its corresponding eigenvector are

$$\mu \approx 1.89, \qquad \vec{\boldsymbol{x}} \approx \begin{bmatrix} 1 \\ 0.88 \end{bmatrix}.$$

The actual values are

$$\mu = 2, \qquad \vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Why does the power method work?

Assume for simplicity that A is symmetric and has a full set of eigenvectors. That is, its eigenvectors $\vec{\boldsymbol{x}}_1, \ldots, \vec{\boldsymbol{x}}_n$ are linearly independent. Suppose that the ordered eigenvalues of A are

$$|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|.$$

Suppose we are given an initial guess $\vec{\boldsymbol{x}}$ which is not orthogonal to the eigenvector $\vec{\boldsymbol{x}}_1$ that corresponds to the dominant eigenvalue λ_1. Any $\vec{\boldsymbol{x}}$ can be written as some linear combination of the eigenvectors

$$\vec{\boldsymbol{x}} = b_1\vec{\boldsymbol{x}}_1 + \cdots + b_n\vec{\boldsymbol{x}}_n.$$

Since $\vec{\boldsymbol{x}}$ is not orthogonal to the eigenvector $\vec{\boldsymbol{x}}_1$, we know that $b_1 \neq 0$. Let $\alpha_1 = |x_q|$, where x_q is the largest component, in absolute magnitude, of $\vec{\boldsymbol{x}}$. Then we normalize $\vec{\boldsymbol{x}}$ so its infinity-norm is 1

$$\vec{\boldsymbol{x}} = \frac{b_1\vec{\boldsymbol{x}}_1 + \cdots + b_n\vec{\boldsymbol{x}}_n}{\alpha_1}.$$

Next, we compute

$$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}} = \frac{b_1\lambda_1\vec{\boldsymbol{x}}_1 + \cdots + b_n\lambda_n\vec{\boldsymbol{x}}_n}{\alpha_1}.$$

Now we again normalize using α_2 which is the largest in absolute magnitude component of $\vec{\boldsymbol{y}}$, to get

$$\vec{\boldsymbol{x}} = \frac{b_1\lambda_1\vec{\boldsymbol{x}}_1 + \cdots + b_n\lambda_n\vec{\boldsymbol{x}}_n}{\alpha_1\alpha_2}.$$

Doing it again gives us

$$\vec{\boldsymbol{x}} = \frac{b_1\lambda_1^2\vec{\boldsymbol{x}}_1 + \cdots + b_n\lambda_n^2\vec{\boldsymbol{x}}_n}{\alpha_1\alpha_2\alpha_3}.$$

After k iterations, we have

$$\vec{\boldsymbol{x}} = \frac{b_1\lambda_1^k\vec{\boldsymbol{x}}_1 + \cdots + b_n\lambda_n^k\vec{\boldsymbol{x}}_n}{\alpha_1\alpha_2\cdots\alpha_{k+1}} = \frac{\lambda_1^k\left[b_1\vec{\boldsymbol{x}}_1 + b_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\vec{\boldsymbol{x}}_2 + \cdots + b_n\left(\frac{\lambda_n}{\lambda_1}\right)^k\vec{\boldsymbol{x}}_n\right]}{\alpha_1\alpha_2\cdots\alpha_{k+1}}.$$

In each iteration, $\vec{\boldsymbol{x}}$ is infinity-normalized, so its largest component is 1. Since the eigenvalues are all ordered and λ_1 is the largest one, we know that all the rest when divided by λ_1 are fractions less than 1. When they are raised to the kth power, they become even smaller. So after k iterations,

$$\vec{\boldsymbol{x}} \approx \frac{\lambda_1^k b_1}{\alpha_1\alpha_2\cdots\alpha_{k+1}}\vec{\boldsymbol{x}}_1,$$

with an infinity-norm of 1.

A simple way to do the power method is as follows:

1. Set the initial guess to
$$\vec{\boldsymbol{x}} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$
2. Set
$$\vec{\boldsymbol{y}} = A\vec{\boldsymbol{x}}.$$
3. Set
$$\vec{\boldsymbol{x}} = \frac{\vec{\boldsymbol{y}}}{||\vec{\boldsymbol{y}}||_2}.$$
4. If $\vec{\boldsymbol{y}}$ is an eigenvector of A, then $A\vec{\boldsymbol{y}} = \lambda\vec{\boldsymbol{y}}$. Dotting from the right gives $A\vec{\boldsymbol{y}}\cdot\vec{\boldsymbol{y}} = \lambda\vec{\boldsymbol{y}}\cdot\vec{\boldsymbol{y}}$, so we can set
$$\lambda = \frac{(A\vec{\boldsymbol{y}})\cdot\vec{\boldsymbol{y}}}{\vec{\boldsymbol{y}}\cdot\vec{\boldsymbol{y}}}.$$
5. If
$$||A\vec{\boldsymbol{y}} - \lambda\vec{\boldsymbol{y}}|| > \text{error tolerance},$$
then return to step 2.

This won't work if the initial guess is orthogonal to the eigenvector we are trying to find.

After finding the dominant eigenvalue and the corresponding eigenvector using the power method, we can use the **deflation** method to find the rest of the eigenvalues.

The **Wielandt deflation** method is as follows. Assuming the eigenvalues $\lambda_1, \ldots, \lambda_n$ are ordered from largest to smallest and the corresponding eigenvectors are $\vec{\boldsymbol{v}}_1, \ldots, \vec{\boldsymbol{v}}_n$, and assuming all the eigenvalues have multiplicity 1, then after finding the largest eigenvalue λ_1 and its eigenvector $\vec{\boldsymbol{v}}_1$ using the power method, set

$$\vec{\boldsymbol{x}} = \frac{1}{\lambda_1(\vec{\boldsymbol{v}}_1)_i}\begin{pmatrix} a_{i1} & a_{i2} & \ldots & a_{in} \end{pmatrix}^T,$$

where $\vec{\boldsymbol{v}}_1)_i$ is the ith component of the eigenvector, and the quantity in parentheses is the ith row of A. Next, set

$$B = A - \lambda_1 \vec{\boldsymbol{v}}_1 \vec{\boldsymbol{x}}^T.$$

The eigenvalues of B are $0, \lambda_2, \lambda_3, \ldots, \lambda_n$. So B has all the eigenvalues of A except that the dominant eigenvalue of A has been replaced with 0. Therefore, if we now apply the power method to B, we obtain λ_2, the next largest eigenvalue of A. We can repeat this process to obtain all the eigenvalues of A. The eigenvectors B are $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{w}}_2, \ldots, \vec{\boldsymbol{w}}_n$, where the eigenvectors $\vec{\boldsymbol{v}}_i$ of A are related to the eigenvectors $\vec{\boldsymbol{w}}_i$ of B by

$$\vec{\boldsymbol{v}}_j = (\lambda_j - \lambda_1)\vec{\boldsymbol{w}}_j + \lambda_1(\vec{\boldsymbol{x}}^T \vec{\boldsymbol{w}}_j)\vec{\boldsymbol{v}}_1.$$

Note, the power method won't work unless there is a *dominant* eigenvalue in A. Also, if the largest eigenvalue of A has a multiplicity other than 1, the power method will not converge.

9.2 Householder Transformation

If you have a vector $\vec{\boldsymbol{w}}$ in $\mathbb{R}^n$ such that $\vec{\boldsymbol{w}}^T \vec{\boldsymbol{w}} = 1$, then

$$P = I_n - 2\vec{\boldsymbol{w}}\vec{\boldsymbol{w}}^T.$$

is an $n \times n$ matrix called a **Householder transformation**. Note that P is a symmetric and orthogonal matrix, so

$$P = P^{-1} = P^T.$$

This also means

$$P^T AP = P^{-1}AP = PAP.$$

An **upper Hessenberg matrix** is like an upper triangular matrix but with one band below the diagonal. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 7 & 8 \end{bmatrix},$$

is a 3×3 upper Hessenberg matrix. To convert a general 3×3 matrix to upper Hessenberg form, we have to eliminate the bottom left element. To convert a general 4×4 matrix to upper Hessenberg form, we have to eliminate three values (in two columns) from the bottom left. To eliminate such elements, we perform Householder transformations.

To convert a general $n \times n$ matrix A to upper Hessenberg form, we perform Householder transformations on the matrix. For each column that we need to eliminate elements from (one for a 3×3, two for a 4×4, and so on), we have to perform a Householder transformation. For each Householder transformation, we strategically find $\vec{\boldsymbol{w}}$ such that PAP eliminates the appropriate elements from A.

Suppose we have the 3×3 matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

that we want to convert to an upper Hessenberg matrix by eliminating the "7" in the bottom left. We need to find a vector

$$\vec{\boldsymbol{w}} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix},$$

that works. Our Householder transformation $P = I_3 - 2\vec{\boldsymbol{w}}\vec{\boldsymbol{w}}^T$ has the form

$$P = \begin{bmatrix} 1-2w_1^2 & -2w_1w_2 & -2w_1w_3 \\ -2w_1w_2 & 1-2w_2^2 & -2w_2w_3 \\ -2w_3w_1 & -2w_3w_2 & 1-2w_3^2 \end{bmatrix}.$$

Setting $w_1 = 0$ ensures that in $\tilde{A} = PAP$, element $\tilde{a}_{11} = a_{11}$, so our transformation becomes

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1-2w_2^2 & -2w_2w_3 \\ 0 & -2w_3w_2 & 1-2w_3^2 \end{bmatrix}.$$

We want $\tilde{A} = PAP$ such that $\tilde{a}_{31} = 0$. If we write out the matrices PAP, we see that this gives us the condition

$$1(0) + 4(-2w_2w_3 + 7(1-2w_3^2) = 0,$$

or $8w_2w_3 + 14w_3^2 = 7$. But we also have the condition that $\vec{\boldsymbol{w}}^T\vec{\boldsymbol{w}} = w_1^2 + w_2^2 + 2_3^2 = 1$, and since $w_1 = 0$, we have $w_3^2 = 1 - w_2^2$. Plugging this into the earlier condition gives us

$$8w_2\sqrt{1-w_2^2} + 14(1-w_2^2) = 7.$$

Solving this gives us

$$w_2 = \sqrt{\frac{1}{2} + \frac{2}{\sqrt{65}}},$$

then

$$w_3 = \sqrt{\frac{1}{2} - \frac{2}{\sqrt{65}}}.$$

Then our Householder transformation is

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{4}{\sqrt{65}} & -\frac{7}{\sqrt{65}} \\ 0 & -\frac{7}{\sqrt{65}} & \frac{4}{\sqrt{65}} \end{bmatrix}.$$

If we perform the transformation, we get

$$PAP = \frac{1}{65}\begin{bmatrix} 65 & -29\sqrt{65} & -2\sqrt{65} \\ -65\sqrt{65} & 913 & 184 \\ 0 & 54 & -3 \end{bmatrix}.$$

This is now an upper Hessenberg matrix, and since our transformation was a similarity transformation, it has the same eigenvalues as our original matrix A.

9.3 The QR Method

The QR method is used to repeatedly transform a symmetric matrix A in a specific manner until it becomes an upper triangular matrix at which point the eigenvalues of A can be read directly from the diagonal elements.

The QR method is ideal when you want to find the eigenvalues and eigenvectors of a dense matrix.

The basic algorithm for the QR method is as follows:

1. Convert A into an upper Hessenberg matrix $\tilde{A}$ using similarity (Householder) transformations.
2. Set $A_1 = \tilde{A}$
3. Set the counter $k = 1$
4. Decompose A as QR
$$A_k = Q_k R_k,$$
where Q_k is an orthogonal matrix, and R_k is an upper triangular matrix.
5. Set
$$A_{k+1} = R_k Q_k.$$
6. Increment $k = k + 1$.
7. If A_{k+1} is not upper triangular, then go back to step 4.

With this method, we start with an upper Hessenberg matrix, and end with an upper triangular matrix.

Note, we can perform step 5 as

$$A_{k+1} = R_k Q_k = Q_k^{-1} A_k Q_k = Q_k^T A_k Q_k,$$

since Q is an orthogonal matrix.

9.4 Singular Value Decomposition

For a real or complex $m \times n$ matrix A, there exists a factorization called the **singular value decomposition** such that

$$A = U\Sigma V^*,$$

where U is a real or complex $m \times m$ unitary matrix, Σ is an $n \times m$ diagonal matrix, and V is an $n \times n$ real or complex unitary matrix. The * in V^* denotes the conjugate transpose.

The diagonal matrix Σ has the form

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \cdots & 0 \\ 0 & 0 & \cdots & \sigma_r & \cdots & 0 \\ \vdots & & & & 0 & \\ 0 & & & & & \ddots \end{bmatrix}.$$

The σ_i are the **singular values** of A.

Note, a unitary matrix with only real values is an orthogonal matrix.

$$\begin{aligned} AA^* &= U\Sigma V^* V \Sigma^* U^* \\ AA^* &= U\Sigma\Sigma^* U^* \\ B &= U\Sigma^2 U^* \\ BU &= U\Sigma^2, \end{aligned}$$

where $B = AA^*$ is an $m \times m$ matrix.

This implies that

$$U = \begin{bmatrix} \vec{\boldsymbol{x}}_1 & \vec{\boldsymbol{x}}_2 & \cdots & \vec{\boldsymbol{x}}_m \end{bmatrix},$$

where the $\vec{\boldsymbol{x}}_i$ are the eigenvectors of $B = AA^*$.

Then

$$\Sigma^2 = \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \cdots & 0 \\ 0 & 0 & \cdots & \sigma_r^2 & \cdots & 0 \\ \vdots & & & & 0 & \\ 0 & & & & & \ddots \end{bmatrix}.$$

Similarly,

$$\begin{aligned} A^*A &= V\Sigma^*U^*U\Sigma V^* \\ A^*A &= V\Sigma^2V^* \\ C &= V\Sigma^2V^* \\ CV &= V\Sigma^2. \end{aligned}$$

This implies that

$$V = \begin{bmatrix} \vec{\boldsymbol{y}}_1 & \vec{\boldsymbol{y}}_2 & \cdots & \vec{\boldsymbol{y}}_n \end{bmatrix},$$

where the $\vec{\boldsymbol{y}}_i$ are the eigenvectors of $C = A^*A$.

If the singular value decomposition of A is $A = U\Sigma V^*$, then the **pseudoinverse** of A is

$$A^+ = V\Sigma^+U^*,$$

where Σ^+ is the pseudoinverse of Σ. It is formed by replacing the diagonal elements by their reciprocals, and then taking the transpose.

$$\Sigma^+ = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \cdots & 0 \\ 0 & 0 & \cdots & \frac{1}{\sigma_r} & \cdots & 0 \\ \vdots & & & & 0 & \\ 0 & & & & & \ddots \end{bmatrix}.$$

Recall that Σ is an $m \times n$ matrix, so Σ^+ is an $n \times m$ matrix.

Some properties of the pseudoinverse include

$$\begin{aligned} A^+AA^+ &= A^+ \\ AA^+A &= A. \end{aligned}$$

Given a matrix equation of the form

$$A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}},$$

where A is an $m \times n$ matrix, $\vec{\boldsymbol{x}}$ is a vector of length n, and $\vec{\boldsymbol{b}}$ is a vector of length m, it can be shown that if a solution exists, it will have the form

$$\vec{\boldsymbol{x}} = A^+\vec{\boldsymbol{b}} + \left(I - A^+A\right)\vec{\boldsymbol{w}},$$

where $\vec{\boldsymbol{w}}$ is any vector. The best solution is the one in which $\vec{\boldsymbol{w}} = \vec{\boldsymbol{0}}$.

Chapter 10

Approximation Theory and Applications

Approximation techniques are concerned with representing large quantities of data with only a little information.

Advanced approximation techniques include

- Discrete cosine transform (used in JPEG compression)
- Wavelet transform
- Principal component analysis (PCA)

These advanced techniques are used a lot in data mining, image processing, etc.

This chapter will be approached from the perspective of image processing.

10.1 Discrete Least Squares Approximation

If you have a function $f(x)$ and some basis such that

$$f(x) = \sum_{i=1}^{\infty} a_i \phi_i(x),$$

where the functions $\phi_i(x)$ are orthogonal, that is,

$$\int_a^b \phi_i(x)\phi_j(x)\,dx = \begin{cases} 0 & \text{if } i \neq j \\ \text{not } 0 & \text{if } i = j \end{cases},$$

then your coefficients will be given by

$$a_i = \frac{\int_a^b f(x)\,\phi_i(x)\,dx}{\int_a^b \phi_i^2(x)\,dx}.$$

If $\phi_j(x) = \cos(jx)$, for example, with $j = 0, 1, 2, \ldots$, and your interval of definition is $[a, b] = [0, 2\pi]$, then an approximation to $f(x)$ is

$$f_k(x) = \sum_{j=1}^{k} a_j \cos(jx).$$

Then the least squares error $||f - f_k||$ is minimized when the coefficients are

$$a_j = \frac{\int_0^{2\pi} f(x)\,\cos(jx)\,dx}{\int_0^{2\pi} \cos^2(jx)\,dx}.$$

10.2 Discrete Cosine Transform

Suppose we have a function $f(x)$ defined on the interval $[-\pi, \pi]$ and then this segment is repeated to the left and right. If we know the function $f(x)$, then we can write the continuous, periodic function as the **Fourier series**

$$f(x) = a_0 + \sum_{k=1}^{\infty} \Big[a_k \cos(kx) + b_k \sin(kx) \Big],$$

where the coefficients are given by

$$\begin{aligned} a_0 &= \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)\, dx \\ a_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(kx)\, dx \\ b_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(kx)\, dx. \end{aligned}$$

The Fourier series, as it is defined above, is used when the function is continuous and periodic. What if the function is periodic, but not continuous. For example, suppose we have a periodic signal, but we can only sample it every second. In such a case, we need a discrete Fourier series. The **discrete Fourier series** for a data set containing N points $(x_i, f(x_i))$ on the interval $[-\pi, \pi]$ is

$$f(x) = a_0 + \sum_{k=1}^{\infty} \Big[a_k \cos(kx) + b_k \sin(kx) \Big],$$

where the coefficients are given by

$$\begin{aligned} a_0 &= \frac{1}{N-1} \sum_{i=1}^{N-1} f(x_i) \\ a_k &= \frac{2}{N-1} \sum_{i=1}^{N-1} f(x_i) \cos(kx_i) \\ b_k &= \frac{2}{N-1} \sum_{i=1}^{N-1} f(x_i) \sin(kx_i). \end{aligned}$$

Both of these are for periodic functions, which is not appropriate when dealing with images. Instead, we use a **discrete Fourier transform**.

For the **Discrete cosine transform**, our orthogonal basis functions are cosines

$$X_k = \alpha_k \sum_{n=0}^{N-1} x_n \cos\left(k t_n\right), \quad k = 0, 1, 2, \ldots, N-1$$

where

$$\begin{aligned} t_n &= \frac{\pi\left(n + \frac{1}{2}\right)}{N} \\ \alpha_0 &= \sqrt{\frac{1}{N}} \\ \alpha_k &= \sqrt{\frac{2}{N}}, \quad \text{for } k > 0. \end{aligned}$$

Note that x_n is actually $x(t_n)$. The x_n are our input data points, and the X_k are our transformed data points. The discrete cosine transform takes us from the data domain to the frequency domain.

The **inverse discrete cosine transform** is

$$x_k = \sqrt{\frac{2}{N}}\left(\frac{1}{\sqrt{2}}X_0 + \sum_{n=1}^{N-1} X_n \cos\left(nt_k\right)\right),$$

where $k = 0, 1, 2, \ldots, N-1$ and

$$t_n = \frac{\pi\left(k+\frac{1}{2}\right)}{N}.$$

The inverse cosine transform takes us from the frequency domain back to the data domain.

The discrete cosine transform and its inverse can be written as the pair of matrix equations

$$\vec{\boldsymbol{X}} = A\vec{\boldsymbol{x}}$$
$$\vec{\boldsymbol{x}} = B\vec{\boldsymbol{X}}.$$

It can be shown that $B = A^T$ and $AA^T = I$. The elements of A are

$$A_{ij} = \alpha_i \cos\left(\frac{i\pi\left(j+\frac{1}{2}\right)}{N}\right),$$

where $\alpha_0 = 1/\sqrt{N}$ and $\alpha_i = \sqrt{2/N}$ for $i > 0$. After constructing A, B can be found as $B = A^T$.

Note that both of these transforms are for one dimension. To work with a two-dimensional image, we could apply the DCT to each row of pixels, and then apply the DCT to the result to get the 2D version of the DCT.

The 2D version of the DCT is

$$X_{k_1,k_2} = \alpha_{k_1}\alpha_{k_2}\sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} x_{n_1,n_2}\cos\left(k_1 t_{n_1}\right)\cos\left(k_2 t_{n_2}\right).$$

where both k's go over $0, 1, 2, \ldots, N-1$, and

$$\alpha_0 = \sqrt{\frac{1}{N}}, \quad \text{for } k_1, k_2 = 0$$
$$\alpha_{k_i} = \sqrt{\frac{2}{N}}, \quad \text{for } k_1, k_2 > 0.$$

The 2D version of the inverse DCT is

$$x_{k_1,k_2} = \frac{2}{N}\sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} \beta_{n_1}\beta_{n_2}X_{n_1,n_2}\cos\left(n_1 t_{k_1}\right)\cos\left(n_2 t_{k_2}\right).$$

where both k's go over $0, 1, 2, \ldots, N-1$, and

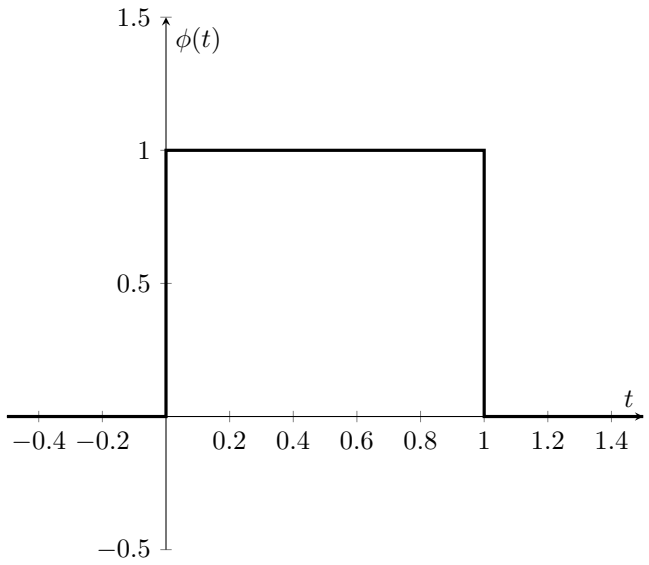
$$\beta_0 = \frac{1}{\sqrt{2}}, \qquad \beta_n = 1, \quad \text{for } n > 0.$$

10.3 Haar Wavelet Transform

The Haar wavelet basis functions are generated from the scaling basis function,

$$\phi(t) = \begin{cases} 1 & 0 \le t \le 1 \\ 0 & \text{otherwise} \end{cases},$$

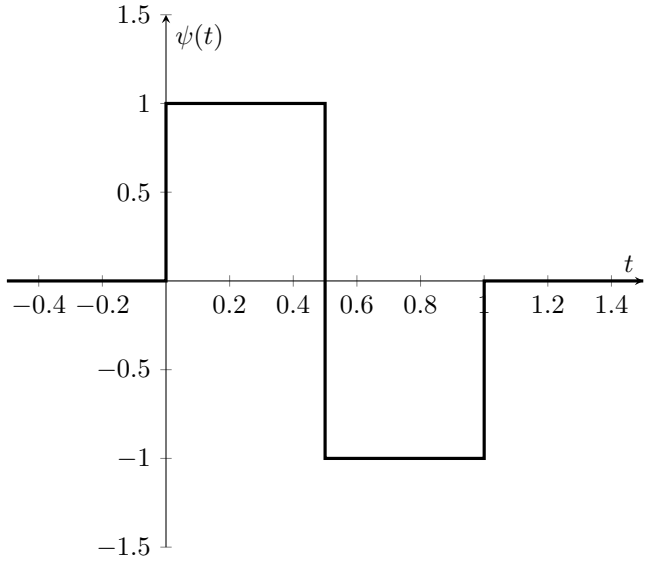
and it is graphed here:



The mother wavelet function is

$$\psi(t) = \begin{cases} 1 & 0 \le t \le \frac{1}{2} \\ -1 & \frac{1}{2} \le t \le 1 \\ 0 & \text{otherwise} \end{cases},$$

and it is graphed here:

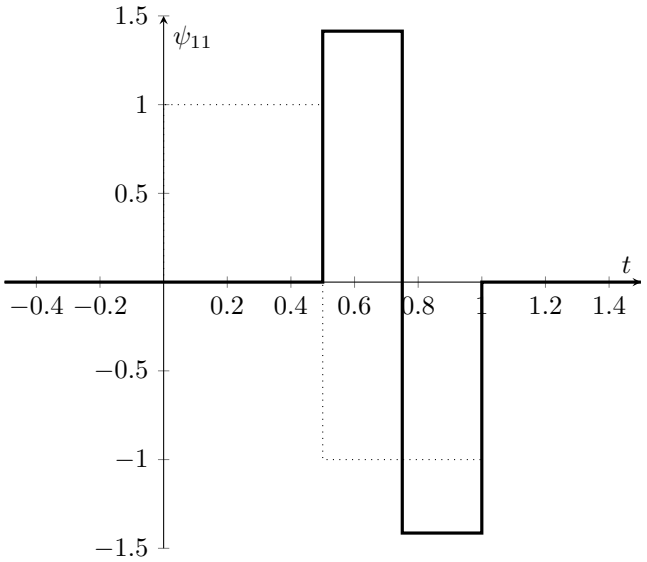


From these, we can derive all kinds of wavelet basis functions, such as

$$\phi_{jk}(t) = 2^{\frac{j}{2}}\phi\left(2^j t - k\right)$$
$$\psi_{jk}(t) = 2^{\frac{j}{2}}\psi\left(2^j t - k\right).$$

The number k is a shift index since it lets us translate $\phi(t)$ to the left or right. We need to be able to translate the functions since they are not periodic like the basis functions of the Fourier transform (i.e. sines and cosines). The number j is a frequency index since $2^{\frac{j}{2}}$ out front is a frequency enhancement.

Below we have graphed the derived basis function $\psi_{11}(t) = \sqrt{2}\psi\left(2t-1\right)$ so that we can compare it to the mother wavelet function (dotted line).



We know from studying PDEs, that for *continuous* transform of a function $f(t)$ with orthogonal basis functions $\phi(t)$ and $\psi(t)$, if the orthogonality is given by

$$X_0 = \frac{\int_0^1 f(t)\phi(t)\,dt}{\int_0^1 \phi^2(t)\,dt}, \qquad X_{jk} = \frac{\int_0^1 f(t)\psi_{jk}(t)\,dt}{\int_0^1 \psi_{jk}^2(t)\,dt}.$$

for $j = 0, 1, 2, \ldots$ and $k = 0, 1, \ldots 2^j - 1$, then the function $f(t)$ is approximated by

$$f(t) \approx X_0\phi(t) + \sum_{j=0}^{j_{max}} \sum_{k=0}^{2^j-1} X_{jk}\psi_{jk}(t).$$

Our equations for *discrete* transform looks similar. If we are given the data points x_n for $n = 0, 1, \ldots, m-1$ where $m \equiv 2^p$ (i.e. m is a power of 2), located at the points

$$t_n = \frac{n + \frac{1}{2}}{m},$$

then the 1D **discrete wavelet transform** is

$$X_0 = \frac{1}{\sqrt{m}} \sum_{n=0}^{m-1} x_n\phi_{00}(t_n), \qquad X_{jk} = \frac{1}{\sqrt{m}} \sum_{n=0}^{m-1} x_n\psi_{jk}(t_n),$$

where $j = 0, 1, \ldots, p-1$ and $k = 0, 1, \ldots 2^j - 1$. The 1D **inverse discrete wavelet transform** is

$$x_n = \frac{1}{\sqrt{m}} \left(X_0 \phi_{00}(t_n) + \sum_{j=0}^{p-1} \sum_{k=0}^{2^j-1} X_{jk} \psi_{jk}(t_n) \right).$$

Note that the transform and the inverse transform are orthogonal, and can be represented by the equations

$$\vec{\boldsymbol{X}} = A\vec{\boldsymbol{x}}$$
$$\vec{\boldsymbol{x}} = B\vec{\boldsymbol{X}},$$

where $B = A^T$.

Suppose we are given the four data points x_0, x_1, x_2, and x_3, then

$$X_0 = \frac{1}{2}(x_0 + x_1 + x_2 + x_3)$$
$$X_{00} = \frac{1}{2}(x_0 + x_1 - x_2 - x_3)$$
$$X_{10} = \frac{\sqrt{2}}{2}(x_0 - x_1)$$
$$X_{11} = \frac{\sqrt{2}}{2}(x_2 - x_3).$$

Then our matrix equation is

$$\begin{bmatrix} X_0 \\ X_{00} \\ X_{10} \\ X_{11} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

Notice that the matrix is orthogonal which makes it easy to find the inverse transform.

10.4 Image Compression

Following are some basic MatLab commands for working with images.

Import an image as a matrix:

```
A = imread('mypic.jpg');
```

To display the image:

```
image(A)
```

To get information about the image/matrix:

```
whos A
```

Export the matrix as an image:

```
imwrite(A, 'mynewpic.jpg');
```

If we import a 900×900 image as an array, the resulting array will be of size $900 \times 900 \times 3$. The third dimension stores the RGB color values of the pixels. Each R, G, and B value can range from 0 to 255.

We can eliminate the one dimension of this array, so it becomes a 900×900 matrix by storing the three RGB values in a single number

$$C = 256^2 R + 256G + B.$$

How can we get the individual RGB values back given the single number C? We start by finding R as the integer part of $C/256^2$

$$R = \left\lfloor \frac{C}{256^2} \right\rfloor.$$

Once we have R, we can subtract 256^2R from C then find the integer part of the remaining number divided by 256 to get G

$$G = \left\lfloor \frac{C - 256R^2}{256} \right\rfloor.$$

Once we have R and G, B is what's left over

$$B = C - 256^2R - 256G.$$

From now on, we will assume that our image is imported as a 2D matrix instead of a 3D array.

Now that we have the image array, how do we compress the image?

Singular Value Decomposition

With SVD or **principal component analysis**, we import the image as an $n \times m$ array A, then perform a singular value decomposition so

$$A = U\Sigma V^T,$$

where U is an $n \times n$ matrix, Σ is $n \times m$, and V is an $m \times m$ matrix. We can write this as

$$A = \sum_{i=1}^{r} \sigma_i \vec{\boldsymbol{u}}_i \vec{\boldsymbol{v}}_i^T,$$

where

$$U = \Big[\vec{\boldsymbol{u}}_1 \ \vec{\boldsymbol{u}}_2 \ \cdots \ \vec{\boldsymbol{u}}_n \Big],$$

and

$$V = \Big[\vec{\boldsymbol{v}}_1 \ \vec{\boldsymbol{v}}_2 \ \cdots \ \vec{\boldsymbol{v}}_m \Big],$$

and we have r non-zero singular values σ_i.

To compress the image, we discard some of the singular values, then

$$A_k = \sum_{i=1}^{k} \sigma_i \vec{\boldsymbol{u}}_i \vec{\boldsymbol{v}}_i^T,$$

where $k < r$. Then $A_k \approx A$. If we do this, then the matrix A_k is the closest rank k matrix to the original matrix A. That is, the Frobenius norm $||A_k - A||_F$ is minimized for rank k matrices. Therefore, this is a method of compression.

The **compression ratio** is

$$\frac{k + mk + nk}{r + mr + nr} = \frac{k}{r}.$$

If you use SVD for *lossless* compression, that is, you keep all the singular values, then by storing the SVD, you'll actually end up storing more data than if you just stored the original matrix. The SVD is not an ideal compression scheme.

Discrete Cosine Transform

The benefit of the discrete cosine transform, when it comes to image compression, is that you won't be storing extra data with the DCT even if you perform lossless compression. This is in contrast to the SVD image compression scheme.

Suppose we have a one-dimensional image that we want to compress using the discrete cosine transform. If we do the discrete cosine transform to get the transformed points X_k from the data points x_k (i.e. pixel values), we can discard some of the X_k to compress our data without losing important information. This is particularly relevant in image compression. That is, once we have our coefficients, we can compress the data by discarding the coefficients with very high k values.

To convert the data back into an image array, we perform the inverse cosine transform

$$\vec{\boldsymbol{x}}_Q = \sqrt{\frac{2}{N}}\left(\frac{1}{\sqrt{2}}X_0 + \sum_{n=1}^{Q-1} X_n \vec{\boldsymbol{\phi}}_n\right),$$

where

$$\vec{\boldsymbol{\phi}}_n = \begin{bmatrix} \cos\left(\frac{\pi n(0+\frac{1}{2})}{N}\right) \\ \cos\left(\frac{\pi n(1+\frac{1}{2})}{N}\right) \\ \vdots \\ \cos\left(\frac{\pi n(N-\frac{1}{2})}{N}\right) \end{bmatrix}.$$

Note that we are storing all the x_k in a single vector $\vec{\boldsymbol{x}}$, but our transform is only going up to the Qth term, so we are compressing our data.

But we're not interested in compressing 1D images, whatever such objects might be. Instead, we want to compress two-dimensional images, and for that, we need the 2D versions of the DCT and inverse DCT.

$N = 8$ turns out to be ideal for JPEG compression, so to compress an image using the discrete cosine transform, you start by breaking the image into 8×8 pixel blocks. It is assumed that the lower left corner of each block is at position $(0, 0)$, and the upper right corner of each block is at position (π, π).

If we truncate with Q X's, we will lose some information, but probably not too much. We could probably set $Q = 4$ without losing a lot of image quality.

What is the decay rate of the coefficients of the Fourier cosine series of a smooth function? For image compression, look at how quickly the coefficients decay. If they decay fast, you can delete the higher order coefficients without much information loss. Then for each 8×8 block, decide how many of the coefficients to store.

The DCT of an image imported as the $M \times N$ matrix A is

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\left(pt_m\right) \cos\left(qt_n\right),$$

where $p = 0, 1, 2, \ldots, M-1$, $q = 0, 1, 2, \ldots, N-1$, and

$$t_m = \frac{\pi\left(m + \frac{1}{2}\right)}{M}$$
$$t_n = \frac{\pi\left(n + \frac{1}{2}\right)}{N}$$
$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}} & \text{for } p = 0 \\ \sqrt{\frac{2}{M}} & \text{for } p > 0 \end{cases}$$
$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}} & \text{for } q = 0 \\ \sqrt{\frac{2}{N}} & \text{for } q > 0 \end{cases}$$

Suppose you've imported an image as a matrix and applied the DCT above to get a transformed array. To convert the transformed array back to the image array, use the inverse DCT

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos\left(pt_m\right) \cos\left(qt_n\right),$$

where $m = 0, 1, 2, \ldots, M-1$, $n = 0, 1, 2, \ldots, N-1$, and t_m, t_n, α_p and α_q are the same as above.

If you want to compress the image, discard some of the higher-order terms by limiting the summation when doing the inverse DCT as in

$$A_{mn} = \sum_{p=0}^{K-1} \sum_{q=0}^{K-1} \alpha_p \alpha_q B_{pq} \cos\left(pt_m\right) \cos\left(qt_n\right),$$

where $K < M, N$.

Another way to apply the DCT to images is to compute the **DCT transform matrix** T. Since we chop a given input image into 8×8 blocks, the DCT transform matrix is usually an 8×8 matrix. The following two equations define the general $M \times M$ DCT transform matrix. The top row is given by

$$T_{0q} = \frac{1}{\sqrt{M}}, \quad q = 0, 1, \ldots, M-1.$$

The rest of the matrix elements are given by

$$T_{pq} = \sqrt{\frac{2}{M}} \cos\left(\frac{\pi(2q+1)p}{2M}\right),$$

where $p = 1, 2, \ldots M-1$ and $q = 0, 1, \ldots M-1$. Then given an input image block A, the matrix product TA is an $M \times M$ matrix whose columns are the 1D DCTs of the columns of A. The matrix product

$$B = TAT^T,$$

is then the 2D DCT of A. The inverse transform is given by

$$A = T^T B T.$$

Wavelets

With wavelets, you also don't store extra data when your compression is lossless, unlike the SVD method.

Unlike with the discrete cosine method, we don't break our image into 8×8 blocks when we do the wavelet transform. Instead, we treat the image as a single block with the origin $(0, 0)$ at the bottom left corner and the point $(1, 1)$ at the top right corner.

One way to approach image compression with the wavelet transform is to just lower the j_{max}. However, this will just blur the image. You really want to use the magnitude to determine if you want to get rid of some modes. There are all kinds of strategies to determine which coefficients to discard.

Chapter 11

Initial Value Problems

Initial value problems are ordinary differential equations in which we are given the initial values. This is as opposed to boundary value problems, which are ordinary differential equations in which we are given the values on the boundaries.

First order initial value problems have the form

$$\frac{dy}{dt} = f\left(y(t), t\right),$$

and we are given $y(0) = y_0$.

11.1 Euler's Method

Consider the differential equation

$$\frac{dN}{dt} = -\frac{1}{\tau}N,$$

governing radioactive decay where $N(t)$ is the amount of atoms remaining at time t. We know the solution to this differential equation is

$$N(t) = N_0 e^{-\frac{t}{\tau}},$$

but we will solve it numerically as an illustration. To do so, we start by replacing the derivative with the forward difference formula

$$\frac{dN}{dt} = \frac{N(t+h) - N(t)}{h}.$$

This gives us

$$\begin{aligned}\frac{N(t+h) - N(t)}{h} &= -\frac{1}{\tau}N(t) \\ N(t+h) &= \left(1 - \frac{h}{\tau}\right)N(t).\end{aligned}$$

This gives us the incremental equation

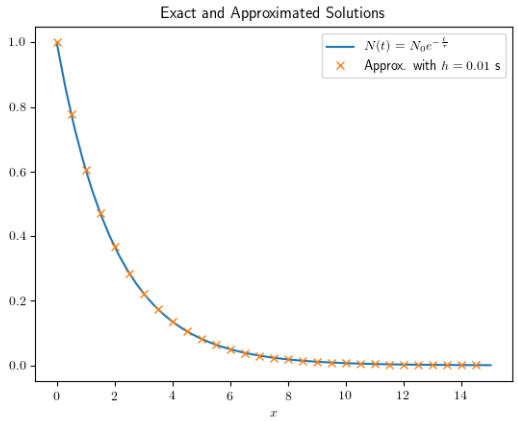
$$N_{i+1} = \left(1 - \frac{h}{\tau}\right)N_i.$$

This method of approximating the solution to an ordinary differential equation is called **Euler's method**.

A simple application of Euler's method in Python for this differential equation with $\tau = 2.0$ is given here:

```
from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np

h, tmax = 0.01, 15.0

# Create lists to store the t and N(t) values for the numerically approximated
    solution. Initialize the lists with your initial values.
t_list = [0.0]
N_list = [1.0]

# Euler's method
i = 0
for t in np.arange(h, tmax, h):
    t_list += [t]
    N_list += [N_list[i] - 0.5*h*N_list[i]]
    i += 1

# Plot the results
plt.title("Numerical Approximation of N(t)")
plt.plot(t_list, N_list)
plt.xlabel("t")
plt.ylabel("N(t)")
plt.show()
```

Below is a plot of the exact solution (blue line) for the case $h = 0.01$s and $\tau = 2.0$s, and the numerically approximated solution (the x's). Only every 50th point of the approximated solution is plotted so that the plot of the exact solution is not completely obscured. The Python code for this can be found on page 116.



One way of deriving Euler's method is to integrate both sides of the ODE as

$$\int_{t_0}^{t_1} \frac{d}{dt} y(t)\, dt = \int_{t_0}^{t_1} f\left(y(t), t\right) dt,$$

which gives us

$$y(t_1) - y(t_0) = \int_{t_0}^{t_1} f\left(y(t), t\right) dt.$$

Suppose the interval $[t_0, t_1]$ is a very small interval, and $t_1 - t_0 = k$ is our "time step". Then with Euler's method, we assume that on this interval

$$f\left(y(t), t\right) = f\left(y(t_0), t_0\right).$$

This should be approximately true provided that $[t_1, t_0]$ is small enough. Then

$$y(t_1) - y(t_0) \simeq \int_{t_0}^{t_1} f\left(y(t_0), t_0\right) dt = kf\left(y(t_0), t_0\right)$$
$$y(t_1) = y(t_0) + kf\left(y(t_0), t_0\right).$$

This gives us an incremental equation that we can use to construct the approximate solution. Given a first-order ODE of the form $y' = f(y,t)$, the initial condition $y(0) = y_0$, the ending time T, and the time step k, we can implement Euler's solution with the following algorithm.

$t = 0.0$
$n = 0$
$y_n = y_0$
while $t < T$:
    $y_{n+1} = y_n + k \cdot f\left(y_n, t\right)$
    $t = t + k$
    $n = n + 1$

When comparing how well ODE algorithms perform, one measure is the **local truncation error** (LTE), which quantifies the amount that the exact solution fails to satisfy the difference approximation. It is the error that the method makes in a single time step. It is the difference between the numerical solution after one step from y_n to y_{n+1} and the exact solution at time $t_{n+1} = t_n + k$. For Euler's method, the numerical solution is

$$y(t_{n+1}) = y)t_n) + kf\left(y(t_n), t\right).$$

For the exact solution, we start with the Taylor expansion about $t = t_n$

$$y(t) = y(t_n) + y'(t_n)(t - t_n) + \frac{1}{2}y''(t_n)(t - t_n)^2 + \cdots$$

Replacing t with $t_n + k$ gives us

$$y(t_n + k) = y(t_n) + ky'(t_n) + \frac{k^2}{2}y''(t_n) + O(k^3).$$

The LTE is the difference between the numerical and exact solutions

$$\begin{aligned}
\text{LTE} &= \frac{1}{k}[y(t_n + k) - y(t_{n+1})] \\
&= \frac{1}{k}\Big[y(t_n) + ky'(t_n) + \frac{k^2}{2}y''(t_n) + O(k^3) - y(t_n) - kf\left(y(t_n), t\right)\Big] \\
&= \frac{1}{k}\Big[ky'(t_n) + \frac{k^2}{2}y''(t_n) + O(k^3) - kf\left(y(t_n), t\right)\Big].
\end{aligned}$$

Since $y(t)$ is the exact solution, we know that $y' = f(y,t)$, so two of the terms cancel each other, and we get

$$\text{LTE} = \frac{1}{k}\left[\frac{k^2}{2}y''(t_n) + O(k^3)\right] = \frac{k}{2}y''(t_n) + O(k^2).$$

Since LTE $\propto k$ for Euler's method, we say that Euler's method is a first-order method.

Another measure of the performance of an ODE algorithm is the **region of stability**. Recall that the incremental equation for Euler's method is

$$y_{n+1} = y_n + kf\left(y_n, t\right).$$

Consider the test ODE

$$y' = \lambda y,$$

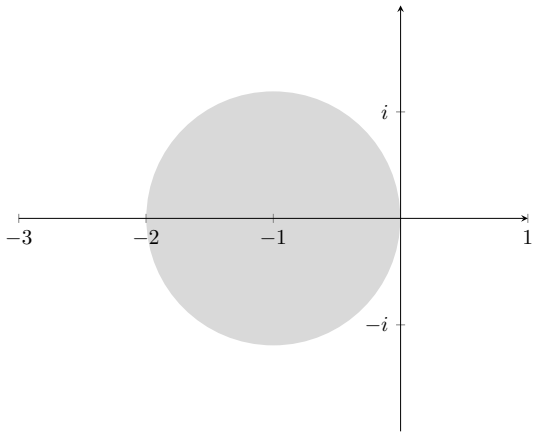
then $f(y,t) \equiv \lambda y$. If we let $z = k\lambda$ be a complex number, where k is the time step, then

$$y_{n+1} = y_n + k\lambda y_n = (1+z)y_n.$$

The region of stability is the values of z in which the solution decays

$$\left|\frac{y_{n+1}}{y_n}\right|^2 = |1+z|^2 \leq 1.$$

This is the region where the magnitude of the numerical approximation moves toward the actual solution at each step. Note that the region $|1+z|^2 \leq 1$ is a disk of radius 1 centered at $z = -1$ in the complex plane. If $z = k\lambda$ is in this region, then the Euler method approaches the actual solution.



11.2 A Second Order Method

If we are given an initial value problem $y' = f(y,t)$ a time interval $[0,T]$, the initial condition $y(0) = y_0$, and N time steps, then a second-order algorithm is as follows:

$$\begin{aligned}
&t = 0.0\\
&n = 0\\
&k = \frac{T}{N}\\
&y_n = y_0\\
&\text{while } n < N:\\
&\quad y^* = y_n + kf\left(y_n, t_n\right)\\
&\quad y_{n+1} = y_n + \frac{k}{2}\left[f\left(y_n, t_n\right) + f\left(y^*, t_{n+1}\right)\right]\\
&\quad t = t + k\\
&\quad n = n + 1
\end{aligned}$$

Suppose we have the differential equation

$$\frac{dy}{dt} = \lambda y,$$

with the initial condition $y(0) = 1$. Now suppose that λ is much less than one, for example, $\lambda = -1000$.

Using our second order method, suppose we divide our stopping time is $T = 1$, and we divide the time interval into $N = 2$ slices. Then using this method, we get

$$y(t_1) = 124,501.$$

For the second step, we get

$$y(t_2) = -1.6 \times 10^{10}.$$

However, the actual solution is $y(t) = e^{-1000t}$, and $y(1) \simeq 0$. The solution decays to zero very quickly. It's nowhere close to -1.6×10^{10} or 124,501. What went wrong? For this ODE, if we perturb the initial condition only a little bit, the result will vary drastically. That is, it is unstable. Such ODE's are called "stiff" problems, and explicit algorithms like this one or the Euler method, do not work well for stiff ODEs.

More precisely, an ODE of the form $y' = f(y,t)$ is **stiff** if

$$\left|\frac{\partial f}{\partial y}\right| >> 1.$$

In our case, we have $\left|\frac{\partial f}{\partial y}\right| = 1000 >> 1$.

How do we analyze a method and determine if it is good for a stiff method?

11.3 Backwards Euler Method

If we are given an initial value problem $y' = f(y,t)$ a time interval $[0,T]$, the initial condition $y(0) = y_0$, and N time steps, then the backwards Euler algorithm is as follows:

$$\begin{aligned}
&t_0 = 0.0\\
&n = 0\\
&k = \frac{T}{N}\\
&y_n = y_0\\
&\text{while } n < N:\\
&\qquad t_{n+1} = t_n + k\\
&\qquad y_{n+1} = y_n + kf\left(y_{n+1}, t_{n+1}\right)\\
&\qquad n = n + 1
\end{aligned}$$

Notice that y_{n+1} appears on both sides of the incremental equation. That makes this an **implicit** method. Implicit methods are good for stiff ODEs.

Since this is an implicit method, we also have to implement a root-finding or fixed-point method. For example, we can write $g(x) = f\left(y_{n+1}, t_{n+1}\right)$ where $g(x) \equiv x - kf(x, t_{n+1}) - y_n$. Then solve $g(x) = 0$ using a root-finding technique.

Like the Euler method, the backwards Euler method is also a first-order method. That is, LTE $= O(k)$.

To find the region of stability for the backward Euler method, we again use the test function $y' = \lambda y$. Then $f(y,t) \equiv \lambda y$. If we let $z = k\lambda$ be a complex number, where k is the time step, then

$$y_{n+1} = y_n + kf\left(y_{n+1}, t_{n+1}\right),$$

becomes

$$\begin{aligned}
y_{n+1} &= y_n + k\lambda y_{n+1} = y_n + zy_{n+1}\\
\left|\frac{y_{n+1}}{y_n}\right|^2 &= \frac{1}{|1-z|^2} \leq 1.
\end{aligned}$$

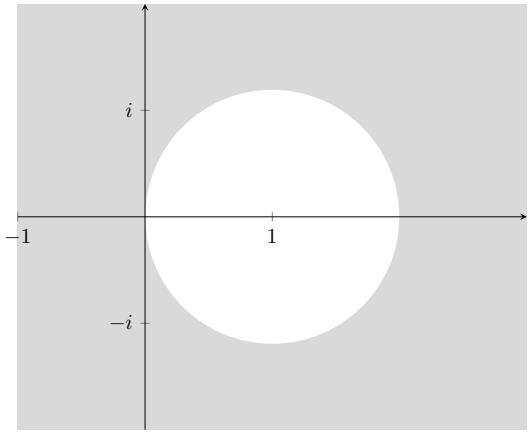
If we replace z with $a + bi$, we get

$$\frac{1}{|1 - a - ib|^2} = \frac{1}{(1-a)^2 + b^2} \leq 1.$$

Rearranging gives us

$$(1-a)^2 + b^2 \geq 1.$$

In other words, the region of stability for the backward Euler method is the area outside of a circle of radius one centered on $z = 1$ in the complex plane.



11.4 Runge-Kutta Method

The fourth-order **Runge-Kutta method** is the most commonly used method to solve ordinary differential equations. For a differential equation of the form

$$\frac{dx}{dt} = f(x,t),$$

The fourth-order Runge-Kutta method is

$$x(t+h) = x(t) + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

where

$$\begin{aligned}
k_1 &= hf(x,t) \\
k_2 &= hf\left(x + \frac{1}{2}k_1,\ t + \frac{1}{2}h\right) \\
k_3 &= hf\left(x + \frac{1}{2}k_2,\ t + \frac{1}{2}h\right) \\
k_4 &= hf\left(x + k_3,\ t + h\right).
\end{aligned}$$

The fourth-order Runge-Kutta method has a truncation error of $O(h^5)$.

The pseudocode for the 4th-order Runge-Kutta method is

$$
\begin{aligned}
&t = 0.0 \\
&n = 0 \\
&h = \frac{T}{N} \\
&x_n = x_0 \\
&\text{while } n < N: \\
&\quad k_1 = hf(x_n, t_n) \\
&\quad k_2 = hf\left(x_n + \frac{1}{2}k_1,\ t_n + \frac{1}{2}h\right) \\
&\quad k_3 = hf\left(x_n + \frac{1}{2}k_2,\ t_n + \frac{1}{2}h\right) \\
&\quad k_4 = hf\left(x_n + k_3,\ t_n + h\right) \\
&\quad x_{n+1} = x_n + \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right] \\
&\quad t_{n+1} = t_n + h \\
&\quad n = n + 1
\end{aligned}
$$

Tip

When implementing the Runge-Kutta method, make sure you don't make a mistake when writing the equations above. It's easy to miss a term or write it incorrectly. The Runge-Kutta method will then put out an incorrect result even though the graph may look plausible.

Here, we use the fourth-order Runge-Kutta method to solve the same differential equation.

```
from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np


def f(N):
    """
        dN/dt = -0.5*N
    """
    return -0.5*N

h, tmax = 0.01, 15.0

# Create lists to store the t and N(t) values for the numerically approximated
    solution. Initialize the lists with your initial values.
t_list = [0.0]
N_list = [1.0]

# Runge-Kutta4 method
i = 0
for t in np.arange(h, tmax, h):
    t_list += [t]
    k1 = h*f(N_list[i])
    k2 = h*f(N_list[i] + 0.5*k1)
    k3 = h*f(N_list[i] + 0.5*k2)
    k4 = h*f(N_list[i] + k3)
    N_list += [N_list[i] + (k1 + 2*k2 + 2*k3 + k4)/6]
    i += 1

# Plot the results
plt.title("Numerical Approximation of N(t)")
plt.plot(t_list, N_list)
plt.xlabel("t")
```

```
plt.ylabel("N(t)")
plt.show()
```

11.5 Coupled Differential Equations

Suppose you have two coupled (i.e. simultaneous) first-order ordinary differential equations of the form

$$\frac{dx}{dt} = f(x, y, t)$$
$$\frac{dy}{dt} = g(x, y, t).$$

The solution to such a set of ODEs is a pair of equations $x(t)$ and $y(t)$.

The key for coupled first-order ODEs is to write them using vectorized notation. Instead of writing the pair of ODEs above, we would write

$$\frac{d\vec{\boldsymbol{r}}}{dt} = \vec{\boldsymbol{f}}(\vec{\boldsymbol{r}}, t),$$

where $\vec{\boldsymbol{r}} = (x, y)$, and $\vec{\boldsymbol{f}}(\vec{\boldsymbol{r}}, t) = (f(\vec{\boldsymbol{r}}, t), g(\vec{\boldsymbol{r}}, t))$. Now instead of passing numbers into the function, we're passing 2-value lists to the function. The beauty of Python is that we can just as easily pass entire arrays into functions as we can pass a single scalar.

The vectorized form of Euler's method is

$$\vec{\boldsymbol{r}}(t + h) = \vec{\boldsymbol{r}}(t) + h\vec{\boldsymbol{f}}(\vec{\boldsymbol{r}}, t).$$

The vectorized form of the fourth-order Runge-Kutta method is The fourth-order Runge-Kutta method is

$$\vec{\boldsymbol{r}}(t + h) = \vec{\boldsymbol{r}}(t) + \frac{1}{6}\left(\vec{\boldsymbol{k}}_1 + 2\vec{\boldsymbol{k}}_2 + 2\vec{\boldsymbol{k}}_3 + \vec{\boldsymbol{k}}_4\right),$$

where

$$\vec{\boldsymbol{k}}_1 = hf(\vec{\boldsymbol{r}}, t)$$
$$\vec{\boldsymbol{k}}_2 = hf\left(\vec{\boldsymbol{r}} + \frac{1}{2}\vec{\boldsymbol{k}}_1,\ t + \frac{1}{2}h\right)$$
$$\vec{\boldsymbol{k}}_3 = hf\left(\vec{\boldsymbol{r}} + \frac{1}{2}\vec{\boldsymbol{k}}_2,\ t + \frac{1}{2}h\right)$$
$$\vec{\boldsymbol{k}}_4 = hf\left(\vec{\boldsymbol{r}} + \vec{\boldsymbol{k}}_3,\ t + h\right).$$

An implementation of the fourth-order Runge-Kutta method in Python might look like the following. Given a function $\frac{dx}{dt} = f(x, t, h)$, this Python function returns the next values of x. Here, x can be a scalar value or a vector if you're dealing with coupled ODEs.

```
def rk4(func, r, t, h):
    k1 = h*func(r, t)
    k2 = h*func(r + 0.5*k1, t + 0.5*h)
    k3 = h*func(r + 0.5*k2, t + 0.5*h)
    k4 = h*func(r+k3, t+h)
    return (k1 + 2*k2 + 2*k3 + k4)/6
```

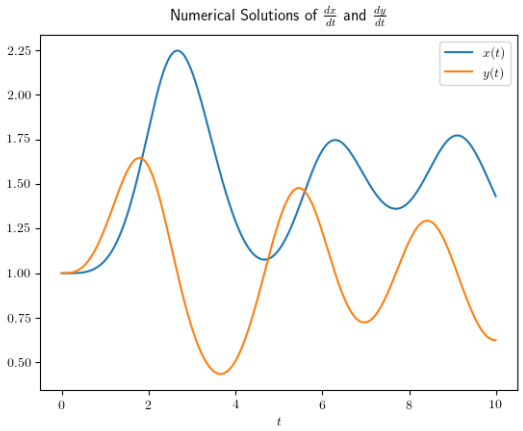
Suppose you have the pair of coupled ODEs

$$\frac{dx}{dt} = xy - x$$
$$\frac{dy}{dt} = y - xy + \sin^2(t).$$

The vectorized form $\vec{\boldsymbol{f}}(\vec{\boldsymbol{r}}, t)$ of this function in Python would look like this:

```
def f(r, t):
    """
    Write your set of differential equations
        dx/dt, dy/dt
    as a single vectorized function of the form
        f(r,t) = dr/dt
    where
        f(x,t) = dx/dt, f(y,t) = dy/dt.
    """

    x = r[0]
    y = r[1]
    fx = x*y - x
    fy = y - x*y + sin(t)**2

    return np.array([fx, fy], float)
```

The solution of this pair of coupled ODEs for the initial conditions $x(0) = y(0) = 1$ is shown below. The full Python code for solving this can be found on page 117. An object-oriented version of the program can be found on page 118.



Chapter 12

Boundary Value Problems

Boundary value problems are ordinary differential equations in which we are given the values on the boundaries.

We can use methods from boundary value ODEs to solve pretty much any PDE, even in multiple dimensions.

Consider the Sturm-Liouville equation

$$-\frac{d}{dx}\left(p(x)\frac{dy}{dx}\right) + q(x)y = f(x),$$

on the interval $0 \leq x \leq 1$ with the Dirichlet boundary condition

$$y(0) = y_0,$$

on the left and the Neumann boundary condition

$$y'(1) = 0,$$

on the right.

If we let $z = \frac{dy}{dx}$, we can write this differential equation as

$$-p'z - z'p + qy = f.$$

Then our second-order ODE becomes the pair of first order ODEs

$$\begin{aligned} z' &= \frac{-p'z + qy - f}{p} \\ y' &= z. \end{aligned}$$

What about the initial conditions? We know that $y(0) = y_0$, but we don't know what $z(0)$ is. We only know that $z = y'$, so we cannot use ordinary methods.

We could use a **shooting method** where we guess the initial slope of y and try to hit $y(1) = 0$. After trying, we modify the guess and shoot again until we hit our target.

12.1 Piecewise Linear Rayleigh-Ritz Method

Consider again the Sturm-Liouville equation

$$-\frac{d}{dx}\left(p(x)\frac{dy}{dx}\right) + q(x)y = f(x),$$

on the interval $0 \leq x \leq 1$ with the boundary conditions

$$\begin{aligned} y(0) &= y_0 \\ y'(1) &= 0. \end{aligned}$$

With the Galerkin method, we start by discretizing our domain $0 \leq x \leq 1$ into N intervals, so that

$$x_i = ih,$$

with $i = 0, \ldots, N$. The width of each interval is $h = 1/N$. Note that in this boundary value problem, x is the independent variable, and y is the independent variable. With the initial value problems of the previous section, t was our independent variable, and y was our independent variable.

After discretizing our domain, we assume a solution of the form

$$y(x) = \sum_{i=0}^{\infty} c_i \phi_i(x).$$

Our discrete approximation to this solution is

$$y_h(x) = \sum_{i=0}^{N} c_i \phi_i(x).$$

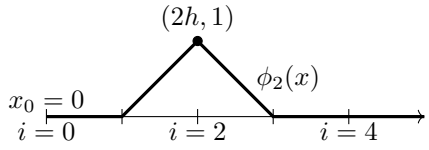
In our case, we also specify the boundary condition $c_0 = y_0$.

Our basis functions are

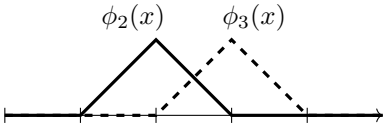
$$\phi_i(x) = \begin{cases} \frac{1}{h}(x - x_{i-1}) & \text{for } x_{i-1} < x < x_i \\ \frac{1}{h}(x_{i+1} - x) & \text{for } x_i < x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

for $i = 0, \ldots, N$.

For example, the basis function $\phi_2(x)$ is shown below.



Notice that these is some region of overlap for consecutive basis functions.



Notice that our boundary condition on the left is satisfied. All the basis functions are zero at $x = 0$ except for $\phi_0(x)$, which gives $\phi_0(0) = 1$, so

$$y_h(0) = \sum_{i=0}^{N} c_i \phi_i(0) = c_0 = y_0.$$

Next, we choose the coefficients c_i so that our approximation $y_h(x)$ satisfies the given differential equation.

$$-\frac{d}{dx}\left(p(x)\frac{d}{dx}y_h(x)\right) + q(x)y_h(x) = f(x).$$

However, the $\phi_i(x)$ within $y_h(x)$ are not continuous, so their derivatives are not defined. So we look for weak solutions by multiplying both sides by a test function $\phi_k(x)$ and then

integrating over the domain. This is called the "weak formulation".

$$\int_0^1 \phi_k \left[-(py_h')' + qy_h\right] dx = \int_0^1 \phi_k f \, dx$$
$$-\int_0^1 \phi_k (py_h')' \, dx + \int_0^1 \phi_k q y_h \, dx = \int_0^1 \phi_k f \, dx$$

For the first integral, we perform integration by parts to get

$$-\int_0^1 \phi_k (py_h')' \, dx = -py'\phi_k \Big|_0^1 + \int_0^1 \phi_k' p y_h' \, dx.$$

Assuming that $0 < k < N$, then the first term on the right is zero, and we have

$$\int_0^1 \phi_k' p y_h' \, dx + \int_0^1 \phi_k q y_h \, dx = \int_0^1 \phi_k f \, dx.$$

Substituting y_h with its series definition gives us:

$$\int_0^1 \left(\phi_k'(x) p(x) \left[\sum_{i=0}^N c_i \phi_i(x)\right]' + \phi_k(x) q(x) \left[\sum_{i=0}^N c_i \phi_i(x)\right] \right) dx = \int_0^1 f(x)\phi_k(x) \, dx$$
$$\sum_{i=0}^N c_i \int_0^1 \Big(p(x)\phi_i'(x)\phi_k'(x) + q(x)\phi_k(x)\phi_i(x) \Big) \, dx = \int_0^1 f(x)\phi_k(x) \, dx.$$

We can write this as the matrix equation

$$A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}},$$

where

$$\vec{\boldsymbol{x}} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix},$$

the elements of A are

$$a_{ij} = \int_0^1 p\phi_i'\phi_j' \, dx + \int_0^1 q\phi_i\phi_j \, dx,$$

for $i = 1, \ldots, N$ and $j = 0, \ldots, N$. The elements of $\vec{\boldsymbol{b}}$ are

$$b_1 = \int_0^1 f\phi_1 \, dx - y_0 a_{10}, \qquad b_i = \int_0^1 f\phi_i \, dx,$$

for $i = 2, \ldots, N$.

Our goal is to find the coefficients c_i by solving the matrix system $A\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$. We can then plot the solution to the ODE by outputting pairs of (x_i, c_i) for $i = 0, \ldots, N$.

The next step is to solve the integrals which appear in the coefficients. We start by making piecewise continuous approximations for $p(x)$, $q(x)$, and $f(x)$. For example, the piecewise continuous approximation $p_h(x)$ is

$$p_h(x) = \begin{cases} p\left(\frac{x_0+x_1}{2}\right) & x_0 < x < x_1 \\ p\left(\frac{x_1+x_2}{2}\right) & x_1 < x < x_2 \\ \vdots & \\ p\left(\frac{x_{N-1}+x_N}{2}\right) & x_{N-1} < x < x_N \end{cases}$$

We can write the derivative of the basis functions as

$$\phi_i'(x) = \begin{cases} \frac{1}{h} & \text{for } x_{i-1} < x < x_i \\ -\frac{1}{h} & \text{for } x_i < x < x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Notice that the only basis functions that have a nonzero overlap with $\phi_i(x)$ are $\phi_{i-1}(x)$ and $\phi_{i+1}(x)$. This means our matrix A will be tridiagonal allowing us to use a sparse matrix solver. That means we only have to consider a few cases. Also, in all cases, the product $\phi_i(x)\phi_j(x)$ and the product $\phi_i'(x)\phi_j'(x)$ are zero except over a small interval. This greatly simplifies our integrals.

For the diagonal elements of A, we have

$$\begin{aligned} a_{ii} &= \int_0^1 p(\phi_i')^2\,dx + \int_0^1 q(\phi_i)^2\,dx \\ &= p\left(\frac{x_{i-1}+x_i}{2}\right)\int_{x_{i-1}}^{x_i}\left(\frac{1}{h}\right)^2 dx + p\left(\frac{x_i+x_{i+1}}{2}\right)\int_{x_i}^{x_{i+1}}\left(-\frac{1}{h}\right)^2 dx \\ &\quad + q\left(\frac{x_{i-1}+x_i}{2}\right)\int_{x_{i-1}}^{x_i}\left(\frac{x-x_{i-1}}{h}\right)^2 dx + q\left(\frac{x_i+x_{i+1}}{2}\right)\int_{x_i}^{x_{i+1}}\left(\frac{x_{i+1}-x}{h}\right)^2 dx \\ &= \frac{1}{h}p\left(\frac{x_{i-1}+x_i}{2}\right) + \frac{1}{h}p\left(\frac{x_i+x_{i+1}}{2}\right) + \frac{h}{3}q\left(\frac{x_{i-1}+x_i}{2}\right) + \frac{h}{3}q\left(\frac{x_i+x_{i+1}}{2}\right). \end{aligned}$$

Here, we used the fact that the intervals are h, so for example, $x_{i+1} - x_i = h$. These are only for $i \neq N$. We have to treat the last element in the diagonal separately, so that our integrals don't extend past the end of our domain at $x = 1$.

$$\begin{aligned} a_{NN} &= \int_0^1 p(\phi_i')^2\,dx + \int_0^1 q(\phi_i)^2\,dx \\ &= p\left(\frac{x_{N-1}+x_N}{2}\right)\int_{x_{N-1}}^{x_N}\left(\frac{1}{h}\right)^2 dx + q\left(\frac{x_{N-1}+x_N}{2}\right)\int_{x_{N-1}}^{x_N}\left(\frac{x-x_{N-1}}{h}\right)^2 dx \\ &= \frac{1}{h}p\left(\frac{x_{N-1}+x_N}{2}\right) + \frac{h}{3}q\left(\frac{x_{N-1}+x_N}{2}\right). \end{aligned}$$

For the upper and lower diagonal bands, we get

$$\begin{aligned} a_{i,i+1} = a_{i+1,i} &= \int_0^1 p\phi_i'\phi_{i+1}'\,dx + \int_0^1 q\phi_i\phi_{i+1}\,dx \\ &= p\left(\frac{x_i+x_{i+1}}{2}\right)\int_{x_i}^{x_{i+1}}\left(-\frac{1}{h}\right)\left(\frac{1}{h}\right)dx \\ &\quad + q\left(\frac{x_i+x_{i+1}}{2}\right)\int_{x_i}^{x_{i+1}}\left(\frac{x_{i+1}-x}{h}\right)\left(\frac{x-x_i}{h}\right)dx \\ &= -\frac{1}{h}p\left(\frac{x_i+x_{i+1}}{2}\right) + \frac{h}{6}q\left(\frac{x_i+x_{i+1}}{2}\right). \end{aligned}$$

For the elements of $\vec{b}$, we get

$$\begin{aligned} b_i &= \int_0^1 f\phi_i\,dx \\ &= f\left(\frac{x_{i-1}+x_i}{2}\right)\int_{x_{i-1}}^{x_i}\frac{x-x_{i-1}}{h}\,dx + f\left(\frac{x_i+x_{i+1}}{2}\right)\int_{x_i}^{x_{i+1}}\frac{x_{i+1}-x}{h}\,dx \\ &= \frac{h}{2}f\left(\frac{x_{i-1}+x_i}{2}\right) + \frac{h}{2}f\left(\frac{x_i+x_{i+1}}{2}\right), \end{aligned}$$

for $i = 2, \ldots, N-1$, and

$$b_1 = \frac{h}{2} f\left(\frac{x_0 + x_1}{2}\right) + \frac{h}{2} f\left(\frac{x_1 + x_2}{2}\right) + y_0 \frac{1}{h} p\left(\frac{x_1 + x_2}{2}\right) - y_0 \frac{h}{6} q\left(\frac{x_1 + x_2}{2}\right)$$
$$b_N = \frac{h}{2} f\left(\frac{x_{N-1} + x_N}{2}\right).$$

Note, the right boundary condition, $y'(1) = 0$ is automatically taken care of here. Python code implementing all of this can be found on page 122.
In the special case that $q = 0$ and $p = f = 1$, our matrix system becomes

$$\begin{bmatrix} \frac{2p}{h} & -\frac{p}{h} & & & & & \\ -\frac{p}{h} & & & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & & \frac{2p}{h} & -\frac{p}{h} & \\ & & & & -\frac{p}{h} & \frac{p}{h} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{N-1} \\ c_N \end{bmatrix} = \begin{bmatrix} fh + \frac{y_0 p}{h} \\ fh \\ \vdots \\ fh \\ \frac{fh}{2} \end{bmatrix}$$

To confirm that the boundary condition on the right, $y'(1) = 0$ is satisfied, we look at the last equation

$$-\frac{p}{h} c_{N-1} + \frac{p}{h} c_N = \frac{fh}{2}.$$

We can write this as

$$p \frac{c_N - c_{N-1}}{h} = \frac{fh}{2}.$$

On the left, we have what looks like a derivative, and the right side goes to zero as h goes to zero. As $h \to 0$, this expression becomes $y'(1) \to 0$.

12.2 1D Schrodinger Equation

The time-independent one-dimensional Schrodinger equation

$$-\frac{\hbar^2}{2m} \frac{d^2\psi}{dx^2} + V(x)\psi = E\psi,$$

is a second-order linear differential equation. In the natural units where $m = \hbar = 1$, we can write this as

$$-\frac{1}{2} \frac{d^2\psi}{dx^2} + V(x)\psi = E\psi.$$

We can write this as the pair of first-order coupled ODEs as

$$\frac{d\phi}{dx} = 2\left[V(x) - E\right]\psi$$
$$\frac{d\psi}{dx} = \phi.$$

If we had the initial conditions $\psi(0)$ and $\psi'(0)$, we could solve this problem as we have done before—as a coupled initial value problem. However, we don't know the initial values. Instead, we are typically given the values of the wavefunction on the boundary. To further complicate things, we don't know the allowed energies E.

To simplify our problem, we will assume that the potential $V(x)$ is even about $x = 0$. From quantum mechanics, we know that a parity even $V(x)$ implies that the energy eigenstates alternate even and odd, with the ground state being an odd function. With a parity even $V(x)$, we can use the symmetry to our advantage. For an even parity solution, we need $\frac{d\psi}{dx} = 0$ at $x = 0$. We are free to choose $\psi(0) = 1$, and then renormalize later.

For a parity odd solution, we need $\psi(0) = 0$, and we are free to choose $\frac{d\psi}{dx} = 1$ at $x = 0$. By exploiting the symmetry of $V(x)$, we have defined initial conditions for our problem, we if we guess a value of E, we can apply solve this as a system of coupled ODEs with initial values given.

We have to be careful, however. If we choose a random E, our wavefunction will most likely not be an energy eigenstate. It will most likely not satisfy our final boundary condition—that $\psi(x) \to 0$ as $x \to \infty$. However, we can try different values of E, using a "shooting method" to narrow in on an eigenvalue E that corresponds to a valid wavefunction. If we choose some E, then for large x, our wavefunction will shoot to positive or negative infinity. If we find a pair of E such that one shoots to positive infinity at large x and the other shoots to negative infinity at large x, then we know there is an eigenvalue between those two E's where the wavefunction tends to zero instead of shooting to positive or negative infinity.

Our general procedure for finding the ground state energy and wavefunction will be as follows:

1. We start by writing the time-independent Schrodinger equation as a pair of coupled first-order ODEs as detailed above.
2. We know the ground state is parity even, so we start with the initial conditions $\psi(0) = 1$ and $\phi(0) = 0$.
3. Next, we need to start with a guess for E. We know that the ground state energy is larger than the minimum potential energy, so we start there. If $V_{min} = 0$, then we start with $E = 0$.
4. Once we have our guess for E, we solve for the wavefunction using the fourth-order Runge-Kutta method.
5. Then we guess a new (larger) value for E, and again solve for the wavefunction. We repeat this process until we have a pair of energy values such that the wavefunction of one goes to positive infinity at large x and the wavefunction of the other goes to negative infinity at large x. We now have a bracketing interval for the ground state energy eigenvalue.
6. Using this bracketing interval, we find the energy eigenvalue to a high precision by using the secant method. We can also now obtain the energy eigenstate associated with this eigenvalue.
7. Finally, we extend the wavefunction to $x < 0$ using the fact that the ground state wavefunction is even.

If we want to find the next energy level and eigenstate (i.e. the first excited state), we repeat this process with some differences. We start the search for the energy eigenvalue by now starting with $E = E_0$ and increment up from there. That is, we know that the next energy level will be higher than the ground state that we just found. We will now be looking for a parity odd eigenstate, so we have to change our initial conditions to $\psi(0) = 0$ and $\phi(0) = 1$. Finally, when we extend the wavefunction to $x < 0$, we use the fact that it is an odd function.

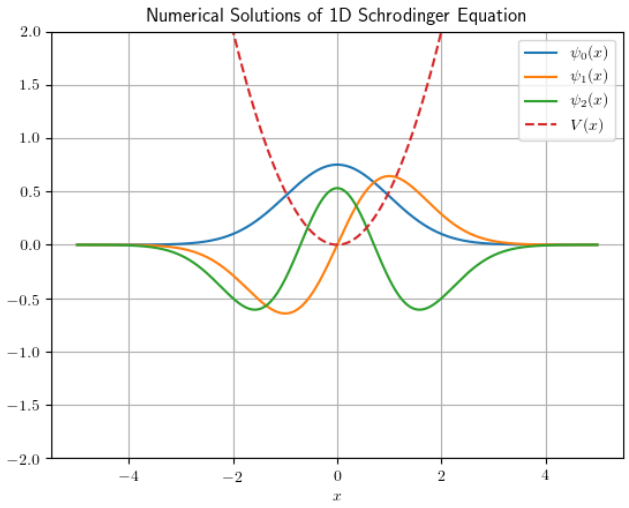
If we want to find many energy eigenvalues and eigenstates, we just keep repeating this process. However, we have to keep track of the parity and alternate the parity for consecutive eigenstates.

NOTE: Any eigenstate found using this process, will shoot to positive or negative infinity if you look at large enough x. You have to play with it a bit, adjusting your domain so that it's large enough that the eigenstate decays to zero but not so large that it shoots to infinity.

A Python program implementing all of this starts on page 125. Running this program for the harmonic oscillator potential $V(x) = x^2/2$ gives us:

```
How many solutions do you want. E.g. enter 3 if you want the ground state and
    first two excited states: 3
Enter a plotting limit (E.g. 4): 5
```

```
State 0
E_0 = 0.50000000008
Normalization constant: 1.33133536356
<x^2>: 0.499999997735

State 1
E_1 = 1.50000000373
Normalization constant: 0.941396255641
<x^2>: 1.49999989707

State 2
E_2 = 2.50000008623
Normalization constant: 1.88279227436
<x^2>: 2.49999787583
```



Chapter 13

Big Data

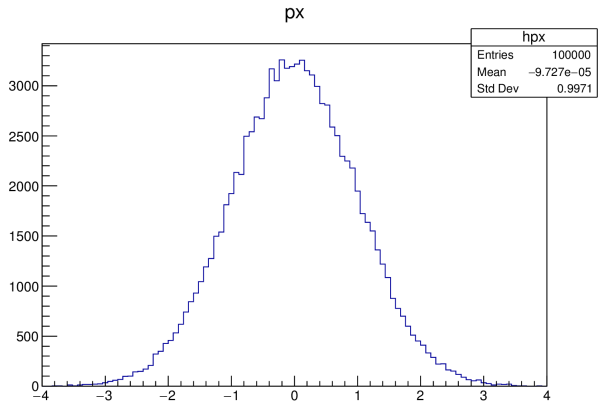
Two tools that can be helpful in determining if a dataset is filled with random noise or has an underlying pattern are **lag plots**[1] and **spectral plots**.[2]

13.1 ROOT

ROOT is a data analysis framework developed by CERN that is well-suited for the analysis of certain large scientific data sets such as particle collision events and astronomical data.

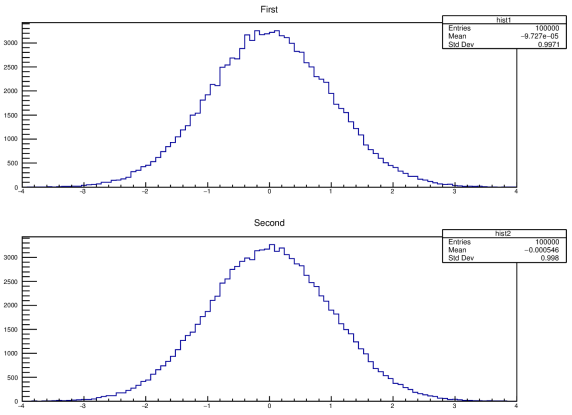
One useful part of ROOT is their TLorentzVector class—objects that store and can work with relativistic four-vectors such as the space-time 4-vector or the energy-momentum 4-vector. A python program exploring these objects can be found on page 119.

Another useful ROOT feature are the histograms. A simple program the histograms 100,000 Gaussian random numbers can be found on page 120. It's output is shown here:



A second program is given on page 120 that shows how the ROOT TCanvas can be split into regions to show multiple plots on the same canvas. It's output is shown here:

[1] http://www.itl.nist.gov/div898/handbook/eda/section3/lagplot.htm
[2] http://www.itl.nist.gov/div898/handbook/eda/section3/spectrum.htm


First
hist1
Entries 100000
Mean -9.727e-05
Std Dev 0.9971
Second
hist2
Entries 100000
Mean −0.000546
Std Dev 0.998

Chapter 14

Computational Optimization

There are several things we can look at when we want to improve the accuracy or decrease the time required to obtain our computed results.

- Language
- Algorithm
- Hardware

14.1 Language

Python, for example, can do arbitrary precision arithmetic with integers. That is, it stores integers with an unlimited number of digits (well, limited by the memory available on your computer) as opposed to many other languages, which only store the first 15 or 16 digits. In python, you can easily calculate the exact value of, $2^{1000000}$, for example.

Some languages are faster than others.[1]

In general, compiled, as opposed to interpreted, implementations run faster. For example, running a program in compiled C++ could make it faster than running it in interpreted Python by an order of magnitude.

14.2 Algorithm

We can often improve the performance of our computation by tweaking our algorithms or by replacing them altogether with better algorithms.

For example, if your algorithm converges slowly, you may want to look for a different algorithm.

14.3 Hardware

We can often greatly speed up our computation by using multiple processor cores if they are available.

[1] http://benchmarksgame.alioth.debian.org/

Appendix A

Appendix

A.1 Python Programs

"Taylor Series Partial Sum"

```
#
# This program gives the partial sum of a Taylor polynomial
# evaluated at some given x.
#

import numpy as np

# Enter your summation limits here and the value of x at which you
# want the Taylor series evaluated.
lower = 0
upper = 7
x = np.pi

# Enter the summand here. Your variable is x and your summation index is k
def Summand(k):
    return
        (1/np.math.factorial(2*k)-(3*x)/np.math.factorial(2*k+1))*x**(2*k)*(-1)**k

# This is the function that sums the series
def PartialSum(a, b):
    f = np.array([Summand(k) for k in range(a, b+1)])
    return f.sum()

# This prints the results to the console
print('Your partial sum is: ' + str(PartialSum(lower,upper)))
```

"Taylor Approximation Plot"

```
# The following Python program plots the results of a Taylor approximation
# for some interval of x and compares it to the true value of the function
# being approximated. The Taylor approximation is shown in a dotted line.
#


import numpy as np
import matplotlib.pyplot as plt

# Enter your summation limits here and the value of x at which you
# want the Taylor series evaluated.
```

```
lower = 0
upper = 2

# Adjust the plotting resolution here
res = 1000

# Define the graph bounds here
left = -np.pi
right = np.pi

# Enter the summand here. Your variable is x and your summation index is k
def Summand(x,k):
    return
        (1/np.math.factorial(2*k)-(3*x)/np.math.factorial(2*k+1))*x**(2*k)*(-1)**k

'''
################################
Compute the graph points for the partial sum. No need to change
anything below here.
################################
'''
x = np.linspace(left,right,res) # Generates your x-values

# This is the function that gives you the partial sum of the series
def PartialSum(x, a, b):
    f = np.array([Summand(x,k) for k in range(a, b+1)])
    return f.sum()

y = np.array([PartialSum(i,lower,upper) for i in x]) # Computes your y-values

'''
################################
Compute the graph points for the true function
################################
'''
r = np.linspace(left,right,res) # Generates the x-values for this function

def fun(x):       # Define the true function here
    return np.cos(x) - 3*np.sin(x)

y2 = np.array([fun(i) for i in r]) # Computes the y-values for this function

'''
################################
Plot both. The dashed plot gives the plot of the Taylor approximation
################################
'''
fig, ax = plt.subplots()
ax.plot(x,y,linestyle='dashed')
ax.plot(r,y2)
ax.grid()
axes = plt.gca()
plt.title(r'$P$' + '$_' + str(upper) + '$' + r'$(x)$')
axes.set_xlim([left,right])
ax.axhline(y=0, color='k')
ax.axvline(x=0, color='k')
```

"Newton's Method with Deflation"

```
import numpy as np


def newtons_method(x, a, tolerance):
    """
    This function returns a single root for a polynomial.
    Takes three parameters--a guess x of the root, the array
    of coefficients, and the error tolerance.
    """
    errormet = False
    notFoundFlag = False
    iterations = 0.0

    while errormet is False:

        fx = 0.0
        for i in range(len(a)): # Calculate f(x)
            fx += a[i]*x**i

        fprimex = 0.0
        for i in range(1, len(a)):
            fprimex += i*a[i]*x**(i-1)

        x_new = x - fx/fprimex

        if abs(x_new - x) < tolerance:
            errormet = True

        x = x_new
        iterations += 1

        # If you do 10,000 Newton iterations and still no root
        # then the remaining roots are probably complex. Stop.
        if iterations > 10000:
            return 0.0, True

    return x_new, notFoundFlag


def deflate_polynomial(a, r):
    """
    Given a polynomial with coefficients in matrix a and a root r of
    the polynomial, deflate the polynomial and return the result.
    """

    k = len(a) - 1
    b = np.zeros([k])

    b[k-1] = a[k]
    for i in range(k-1, 0, -1):
        b[i-1] = a[i] + r*b[i]

    return b


# Enter the ordered coefficients of your polynomial here.
# Order them starting with the zero order term as in
#    f(x) = a_0 + a_1x^1 + a_2x^2 + ...
```

```
coefficients = np.array([0.0, 13.125, 0.0, -78.75, 0.0, 86.625])

k = len(coefficients) - 1
Tol = 1.0e-10

while k >= 1:
    rootAndFlag = newtons_method(1.0, coefficients, Tol)
    root = rootAndFlag[0]
    flag = rootAndFlag[1]

    if flag is True:
        print("The remaining roots were not found. They are probably complex.")
        break

    print(root)
    B = deflate_polynomial(coefficients, root)
    coefficients = B
    k -= 1
```

"Gaussian Elimination Algorithm"

```
#
#  This program solves a matrix equation of the form Ax = b
#  for b, where A is a square matrix. This is a basic method for solving
#  a matrix. Pivoting is only performed when the pivot is zero.
#

import sys
import numpy as np

#
#  Enter your A and b here
#

A = np.array([[3, -2, 5],
              [4, -7, -1],
              [5, -6, 4]], dtype=float) # Enter your matrix

b = np.array([2, 19, 13], dtype=float) # Enter your b-vector

epsilon = 1e-18

#
#  This function solves the linear system
#


def solve_basic(A, b):
    mat = A.copy() # Copy passed arrays since they are mutable
    vec = b.copy()
    n = len(b)
    for i in range(0, n-1):

        # Swap with row beneath if pivot element is zero
        if np.absolute(mat[i, i]) < epsilon:
            for j in range(0, n):
                mat[i, j], mat[i+1, j] = mat[i+1, j], mat[i, j]

            # Swap elements in b-vector
```

```
            vec[i], vec[i+1] = vec[i+1], vec[i]

        # Eliminate all the values under the pivot positions
        for j in range(i+1, n):      # For each element below the pivot position
            m = mat[j, i]/mat[i, i]  # Compute the scale factor
            for q in range(0, n):    # Row op: R_j = R_j - m*R_i
                mat[j, q] = mat[j, q] - m*mat[i, q]
            vec[j] = vec[j] - m*vec[i]

    if np.absolute(mat[n-1, n-1]) < epsilon:
        sys.exit('No unique solution exists!')

    # Here we start backward substitution
    solution = np.zeros(n)
    solution[n-1] = vec[n-1]/mat[n-1, n-1]
    for i in range(n-2, -1, -1):
        total = 0
        for j in range(i+1, n):
            total += mat[i, j]*solution[j]
        solution[i] = (vec[i] - total)/mat[i, i]

    return solution

#
#  Output the results
#

soln = solve_basic(A, b)
print("Your solution is:\n", soln)
print("\nYour residual vector is:")
print(np.absolute(b - np.dot(A, soln)))
```

"Partial Pivoting Algorithm"

```
#
#  This program solves a matrix equation of the form Ax = b
#  for b by using partial pivoting.
#

import sys
import numpy as np

#
#  Enter your A and b here
#

A = np.array([[1, 1/2, 1/3],
              [1/2, 1/3, 1/4],
              [1/3, 1/4, 1/5]], dtype=float) # Enter your matrix

b = np.array([0, 0, 1], dtype=float) # Enter your b-vector

epsilon = 1e-18

#
#  This function solves the linear system
#
```

```
def solve_partial_pivoting(mat, vec):
    mat = A.copy() # Copy passed arrays since they are mutable
    vec = b.copy()
    n = len(b)
    for i in range(0, n-1):

        # Identify the row p containing the largest element in the subcolumn
        p = np.absolute(mat[i:, i]).argmax() + i

        # Swap the rows with indices i and p
        for j in range(0, n):
            mat[i, j], mat[p, j] = mat[p, j], mat[i, j]

        # Swap elements in b-vector as well
        vec[i], vec[p] = vec[p], vec[i]

        # Eliminate all the values under the pivot positions
        for j in range(i+1, n):      # For each element below the pivot position
            m = mat[j, i]/mat[i, i]  # Compute the scale factor
            for q in range(0, n):    # Row op: R_j = R_j - m*R_i
                mat[j, q] = mat[j, q] - m*mat[i, q]
            vec[j] = vec[j] - m*vec[i]

    if np.absolute(mat[n-1, n-1]) < epsilon:
        sys.exit('No unique solution exists!')

    # Here we start backward substitution
    solution = np.zeros(n)
    solution[n-1] = vec[n-1]/mat[n-1, n-1]
    for i in range(n-2, -1, -1):
        total = 0
        for j in range(i+1, n):
            total += mat[i, j]*solution[j]
        solution[i] = (vec[i] - total)/mat[i, i]

    return solution

#
#  Output the results
#
soln = solve_partial_pivoting(A, b)
print("Your solution is:\n", soln)
print("\nYour residual vector is:")
print(np.absolute(b - np.dot(A, soln)))
```

"Scaled Partial Pivoting Algorithm"

```
#
#  This program solves a matrix equation of the form Ax = b
#  for b by using scaled partial pivoting.
#

import sys
import numpy as np

#
#  Enter your A and b here
#
```

```
A = np.array([[0, -2, 5],
              [4, -7, -1],
              [5, -6, 4]], dtype=float) # Enter your matrix

b = np.array([2, 19, 13], dtype=float) # Enter your b-vector

epsilon = 1e-18

#
#  This function solves the linear system
#


def solve_scaled_partial_pivoting(mat, vec):
    mat = A.copy() # Copy passed arrays since they are mutable
    vec = b.copy()
    n = len(vec)

    # Create a vector containing the largest element of each row of A
    scale = np.zeros(n)
    for i in range(n):
        scale[i] = np.absolute(mat[i, :]).max()

    for i in range(0, n-1):

        # Identify the row p containing the largest scaled element in the
            subcolumn
        p = np.absolute(mat[i:, i] / scale[i:]).argmax() + i

        # Swap the rows with indices i and p
        for j in range(n):
            mat[i, j], mat[p, j] = mat[p, j], mat[i, j]

        # Swap elements in b and scale vectors as well
        vec[i], vec[p] = vec[p], vec[i]
        scale[i], scale[p] = scale[p], scale[i]

        # Eliminate all the values under the pivot positions
        for j in range(i+1, n):      # For each element below the pivot position
            m = mat[j, i]/mat[i, i]  # Compute the scale factor
            for q in range(0, n):    # Row op: R_j = R_j - m*R_i
                mat[j, q] = mat[j, q] - m*mat[i, q]
            vec[j] = vec[j] - m*vec[i]

    if np.absolute(mat[n-1, n-1]) < epsilon:
        sys.exit('No unique solution exists!')

    # Here we start backward substitution
    solution = np.zeros(n)
    solution[n-1] = vec[n-1]/mat[n-1, n-1]
    for i in range(n-2, -1, -1):
        total = 0
        for j in range(i+1, n):
            total += mat[i, j]*solution[j]
        solution[i] = (vec[i] - total)/mat[i, i]

    return solution
```

```
#
#  Output the results
#
soln = solve_scaled_partial_pivoting(A, b)
print("Your solution is:\n", soln)
print("\nYour residual vector is:")
print(np.absolute(b - np.dot(A, soln)))
```

"Complete Pivoting Algorithm"

```
#
#  This program solves a matrix equation of the form Ax = b
#  for b by using complete pivoting.
#

import sys
import numpy as np

#
#  Enter your A and b here
#

A = np.array([[1, 1/2, 1/3, 1/4],
              [1/2, 1/3, 1/4, 1/5],
              [1/3, 1/4, 1/5, 1/6],
              [1/4, 1/5, 1/6, 1/7]], dtype=float) # Enter your matrix

b = np.array([0, 0, 0, 1], dtype=float) # Enter your b-vector

epsilon = 1e-18

#
#  This function solves the linear system
#


def solve_complete_pivoting(A, b):
    mat = A.copy() # Copy passed arrays since they are mutable
    vec = b.copy()
    n = len(b)
    solnswap = np.arange(n) # Matrix storing the indices of the solution vector
    for i in range(0, n-1):

        # Return index (as tuple) of largest element in submatrix
        ind = np.unravel_index(np.absolute(mat[i:, i:]).argmax(), mat[i:,
            i:].shape)

        # Swap the rows with indices i and p
        for j in range(0, n):
            mat[i, j], mat[ind[0]+i, j] = mat[ind[0]+i, j], mat[i, j]

        # Swap elements in b-vector as well
        vec[i], vec[ind[0]+i] = vec[ind[0]+i], vec[i]

        # Swap the columns with indices i and c
        for j in range(0, n):
            mat[j, i], mat[j, ind[1]+i] = mat[j, ind[1]+i], mat[j, i]

        # Swap the indices in the solution vector
```

```
        solnswap[i], solnswap[ind[1]+i] = solnswap[ind[1]+i], solnswap[i]

        # Eliminate all the values under the pivot positions
        for j in range(i+1, n):      # For each element below the pivot position
            m = mat[j, i]/mat[i, i]  # Compute the scale factor
            for q in range(0, n):    # Row op: R_j = R_j - m*R_i
                mat[j, q] = mat[j, q] - m*mat[i, q]
            vec[j] = vec[j] - m*vec[i]

    if np.absolute(mat[n-1, n-1]) < epsilon:
        sys.exit('No unique solution exists!')

    # Here we start backward substitution
    solution = np.zeros(n)
    solution[n-1] = vec[n-1]/mat[n-1, n-1]
    for i in range(n-2, -1, -1):
        total = 0
        for j in range(i+1, n):
            total += mat[i, j]*solution[j]
        solution[i] = (vec[i] - total)/mat[i, i]

    # Reorder the solution vector to its original order
    solncopy = solution.copy()
    for i in range(n):
        solution[i] = solncopy[np.argwhere(solnswap == i)[0]]

    return solution

#
#  Output the results
#
soln = solve_complete_pivoting(A, b)
print("Your solution is:\n", soln)
print("\nYour residual vector is:")
print(np.absolute(b - np.dot(A, soln)))
```

"Iterative Methods"

```
#
#  This program uses several iterative methods to solve a sparse tridiagonal
    matrix.
#
#  Leon Hostetler
#  Feb. 5, 2017
#

import numpy as np


n = 10 # The size of the matrix
epsilon = 1
alpha = 1
beta = 100
TOLfactor = 1e-7 # Reduce residual by this factor

# Initialize the vectors
rho = np.zeros(n+1)
D = np.zeros(n)
U = np.zeros(n-1)
```

```
L = np.zeros(n-1)
B = np.zeros(n)


def generate_vectors(n, epsilon, alpha, beta):
    """
    This function generates the D, L, and U vectors corresponding to the
    specific sparse matrix, we are using in this program.
    """
    h = 1/n
    for i in range(n+1):
        ri = (i+1)*h
        if i == 0:
            rho[i] = epsilon
        elif i > 0 and ri <= 0.5:
            rho[i] = alpha
        elif ri > 0.5 and i < n:
            rho[i] = beta
        elif i == n:
            rho[i] = epsilon
    for i in range(n):
        ri = (i+1)*h
        B[i] = (h**2)*((1-ri)**2)*(ri**2)
        D[i] = rho[i] + rho[i+1]
    for i in range(n-1):
        U[i] = -rho[i+1]
        #L[i] = -rho[i+1]
        L[i] = U[i]


def print_matrices():
    """
    This function prints the matrix A for visualization purposes, as well as
    all of the vectors used in this program.
    """
    A = np.zeros([n, n])
    for i in range(n):
        for j in range(n):
            if j == i:
                A[i, j] = D[i]
            elif j == i-1:
                A[i, j] = L[i-1]
            elif j == i+1:
                A[i, j] = U[i]
            else:
                A[i, j] = 0
    print("A = ")
    print(A)
    print("B = ", B)
    print("D = ", D)
    print("U = ", U)
    print("L = ", L)
    print("rho = ", rho)


def Adotx(size, x, diagonal, upper, lower):
    """
    Computes the matrix-vector product Ax, where A is given as three
```

```
    diagonal band vectors.
    """
    z = np.zeros(size)
    z[0] = diagonal[0]*x[0] + upper[0]*x[1]
    for i in range(1, n-1):
        z[i] = diagonal[i]*x[i] + upper[i]*x[i+1] + lower[i-1]*x[i-1]
    z[n-1] = diagonal[n-1]*x[n-1] + lower[n-2]*x[n-2]
    return z


def residual(size, x, diagonal, upper, lower, b):
    """
    Computes the matrix-vector product Ax, where A is given as three
    diagonal band vectors.
    """
    res = np.abs(b - Adotx(size, x, diagonal, upper, lower))
    return res


def jacobi(size, diagonal, upper, lower, b):
    """
    For a tridiagonal matrix, this function takes in the size (i.e. number
    of rows in the square matrix), the vector b, and the three vectors
    corresponding to the diagonal, upper diagonal row, and lower diagonal row
    of the matrix. It returns the approximation x, as well as the number of
    Jacobi iterations that were performed.
    """
    k = 0
    x = np.zeros(n)
    TOL = TOLfactor*np.mean(residual(n, x, D, U, L, b))
    while np.mean(residual(n, x, D, U, L, b)) > TOL:
        X = np.zeros(n)
        X[0] = (b[0] - upper[0] * x[1]) / diagonal[0]
        for i in range(1, n-1):
            X[i] = (b[i] - lower[i-1]*x[i-1] - upper[i]*x[i+1])/diagonal[i]
        X[size-1] = (b[size-1] - lower[size-2]*x[size-2])/diagonal[size-1]
        x = X
        k += 1
    return x, k


def steepest_descent(size, diagonal, upper, lower, b):
    """
    For a tridiagonal matrix, this function takes in the size (i.e. number
    of rows in the square matrix), the vector b, and the three vectors
    corresponding to the diagonal, upper diagonal row, and lower diagonal row
    of the matrix. It returns the approximation x, as well as the number of
    steepest descent iterations that were performed.
    """
    k = 0
    x = np.zeros(size)
    v = b - Adotx(size, x, diagonal, upper, lower)
    TOL = TOLfactor * np.mean(residual(n, x, D, U, L, b))
    while np.mean(np.absolute(v)) > TOL:
        t = (np.dot(v, v))/(np.dot(v, Adotx(size, v, diagonal, upper, lower)))
        x += t*v
        v = b - Adotx(size, x, diagonal, upper, lower)
        k += 1
```

```
    return x, k


def conjugate_gradient(size, diagonal, upper, lower, b):
    """
    For a tridiagonal matrix, this function takes in the size (i.e. number
    of rows in the square matrix), the vector b, and the three vectors
    corresponding to the diagonal, upper diagonal row, and lower diagonal row
    of the matrix. It returns the approximation x, as well as the number of
    conjugate gradient iterations that were performed.
    """
    k = 0
    x = np.zeros(size)
    r1 = b - Adotx(size, x, diagonal, upper, lower)
    v = r1
    TOL = TOLfactor * np.mean(residual(n, x, D, U, L, b))
    while np.mean(np.absolute(v)) > TOL:
        t = np.dot(r1, r1)/np.dot(v, Adotx(size, v, diagonal, upper, lower))
        x += t*v
        r2 = r1 - t*Adotx(size, v, diagonal, upper, lower)
        s = np.dot(r2, r2)/np.dot(r1, r1)
        v = r2 + s*v
        r1 = r2
        k += 1
    return x, k


def BICGSTAB(size, diagonal, upper, lower, b):
    """
    For a tridiagonal matrix, this function takes in the size (i.e. number
    of rows in the square matrix), the vector b, and the three vectors
    corresponding to the diagonal, upper diagonal row, and lower diagonal row
    of the matrix. It returns the approximation x, as well as the number of
    BICGSTAB iterations that were performed.
    """
    k = 0
    x = np.zeros(size)
    r = b - Adotx(size, x, diagonal, upper, lower)
    rhat = r.copy()
    rho0, alpha, omega = 1, 1, 1
    v, p = np.zeros(size), np.zeros(size)
    TOL = TOLfactor * np.mean(residual(n, x, D, U, L, b))
    while np.mean(residual(n, x, D, U, L, b)) > TOL:
        k += 1
        rho1 = np.dot(rhat, r)
        beta = (rho1/rho0)*(alpha/omega)
        p = r + beta*(p - omega*v)
        v = Adotx(size, p, diagonal, upper, lower)
        alpha = rho1/np.dot(rhat, v)
        h = x + alpha*p
        if np.mean(residual(n, h, D, U, L, b)) < TOL:
            return h, k
        s = r - alpha*v
        t = Adotx(size, s, diagonal, upper, lower)
        omega = np.dot(t, s)/np.dot(t, t)
        x = h + omega*s
        if np.mean(residual(n, x, D, U, L, b)) < TOL:
            return x, k
```

```
        r = s - omega*t
        rho0 = rho1
    return x, k


generate_vectors(n, epsilon, alpha, beta)
print_matrices()

jSolution, jruns = jacobi(n, D, U, L, B)
sdSolution, sdruns = steepest_descent(n, D, U, L, B)
cgSolution, cgruns = conjugate_gradient(n, D, U, L, B)
BICGSTABSolution, BICGSTABruns = BICGSTAB(n, D, U, L, B)
print("\nJacobi iterations = ", jruns)
print("x = ", jSolution)
print("\nSteepest Descent iterations = ", sdruns)
print("x = ", sdSolution)
print("\nConjugate Gradient iterations = ", cgruns)
print("x = ", cgSolution)
print("\nBICGSTAB iterations = ", BICGSTABruns)
print("x = ", BICGSTABSolution)
```

"Basic ODE"

```
"""
This program numerically approximates the solution to
    dN/dt = -N/tau
where N is a function of t. The solution is approximated using
the difference equation
    N[i+1] = (1 - h/tau)N[i]
where h is a small number. The exact solution is easily found to be
    N(t) = exp(-t/tau)
so we can compare the exact solution with the numerically
approximated solution by plotting the two.

In this example, we have h = 0.01s, and 0.0 < t < 15.0 seconds.
"""

from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np


# Constants
h, tmax, tau = 0.01, 15.0, 2.0

# The t and N(t) values for the exact solution
tlist = np.linspace(0.0, 15.0)
Nlist = [np.exp(-0.5*t) for t in tlist]

# The t and N(t) values for the numerically approximated solution
ta = [0]
Na = [1] # at t=0, N(t) = 100%

i = 0
for t in np.arange(h, tmax, h):
    ta += [t]
    Na += [Na[i] - (h/tau)*Na[i]]
    i += 1
```

```
# Plot the results. For the approximated solution, only every
# 50th point is plotted so the plot of the approximated solution
# does not obscure the plot of the exact solution.
plt.rc('text', usetex=True)
plt.title("Exact and Approximated Solutions")
plt.plot(tlist, Nlist, label=r"$N(t) = N_0 e^{-\frac{t}{\tau}}$")
plt.plot(ta, Na, linestyle="none", marker="x", \
        markevery=50, label=r"Approx. with $h = 0.01$ s")
plt.legend(loc=1)
plt.xlabel(r"$x$")
plt.show()
```

"Coupled ODEs"

```
from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np


def rk4(func, r, t, h):
    """
    4th order Runge-Kutta method for solving 1st order differential equations

    Given a function f(x, t, h) = dx/dt and initial starting
    conditions for x, rk4() returns the next values of x.

    func: user defined function for the 1st order differential equations
    r: dependent variable
    t: independent variable
    h: independent variable step value
    """
    k1 = h*func(r, t)
    k2 = h*func(r + 0.5*k1, t + 0.5*h)
    k3 = h*func(r + 0.5*k2, t + 0.5*h)
    k4 = h*func(r+k3, t+h)
    return (k1 + 2*k2 + 2*k3 + k4)/6


def f(r, t):
    """
    Write your set of differential equations
        dx/dt, dy/dt
    as a single vectorized function of the form
        f(r,t) = dr/dt
    where
        f(x,t) = dx/dt, f(y,t) = dy/dt.
    """

    x = r[0]
    y = r[1]
    fx = x*y - x
    fy = y - x*y + np.sin(t)**2

    return np.array([fx, fy], float)


# Constants
tMin, tMax, N = 0.0, 10.0, 1000
```

```
h = (tMax - tMin)/N

# The list of points for time, x(t), and y(t)
tPoints = np.arange(tMin, tMax, h)
xPoints, yPoints = [], []

# Initialize the list (of lists) of function values with the initial conditions
r = np.array([1.0, 1.0], float)

# Apply the Runge-Kutta method
for t in tPoints:
    xPoints += [r[0]]
    yPoints += [r[1]]
    r += rk4(f, r, t, h)

# Plot the numerical solutions of x(t) and y(t)
plt.rc('text', usetex=True)
plt.title(r"Numerical Solutions of $\frac{dx}{dt}$ and $\frac{dy}{dt}$")
plt.plot(tPoints, xPoints, label=r"$x(t)$")
plt.plot(tPoints, yPoints, label=r"$y(t)$")
plt.legend(loc=1)
plt.xlabel(r"$t$")
plt.show()
```

"Coupled ODEs (Object-Oriented)"

```
from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np


class RungeKutta:
    """Wrap Runge-Kutta method in a class, so you can access parameters later."""

    def __init__(self, function):
        self.func = function
        self.order = 4 # The order of the Runge-Kutta method

    def __call__(self, r, t, h):
        k1 = h*self.func(r, t)
        k2 = h*self.func(r + 0.5*k1, t + 0.5*h)
        k3 = h*self.func(r + 0.5*k2, t + 0.5*h)
        k4 = h*self.func(r + k3, t + h)
        return (k1 + 2*k2 + 2*k3 + k4)/6


def f(r, t):
    """Write your set of differential equations
        dx/dt, dy/dt
    as a single vectorized function of the form
        f(r,t) = dr/dt
    where
        f(x,t) = dx/dt, f(y,t) = dy/dt.
    """

    x = r[0]
    y = r[1]
    fx = x * y - x
    fy = y - x * y + np.sin(t)**2
```

```
    return np.array([fx, fy], float)


# Constants
tMin, tMax, N = 0.0, 10.0, 1000
h = (tMax - tMin)/N

# Create a Runge-Kutta object, and pass our function to it.
rk = RungeKutta(f)

# The list of points for time, x(t), and y(t)
tPoints = np.arange(tMin, tMax, h)
xPoints, yPoints = [], []

# Initialize the list (of lists) of function values with the initial conditions
r = np.array([1.0, 1.0], float)

# Apply the Runge-Kutta method
for t in tPoints:
    xPoints += [r[0]]
    yPoints += [r[1]]
    r += rk(r, t, h)

# Plot the numerical solutions of x(t) and y(t)
plt.rc('text', usetex=True)
plt.title(r"Numerical Solutions of $\frac{dx}{dt}$ and $\frac{dy}{dt}$")
plt.plot(tPoints, xPoints, label=r"$x(t)$")
plt.plot(tPoints, yPoints, label=r"$y(t)$")
plt.legend(loc=1)
plt.xlabel(r"$t$")
plt.show()
```

"ROOT TLorentzVector Class"

```
#
# This file explores the use of ROOT's TLorentzVector class, which is useful
# for working with energy-momentum 4-vectors.
#

from __future__ import division, print_function
from ROOT import TLorentzVector

# Create the energy-momentum vector for a photon with energy 5 GeV
photon = TLorentzVector(0, 0, 5.0, 5.0)

# Create the 4-vector for a proton at rest
proton = TLorentzVector(0, 0, 0, 0.938)

# Create vectors for two pi+ pions and one pi- pion
pip1, pip2, pim = TLorentzVector(), TLorentzVector(), TLorentzVector()

# Set the vector components for the pions
pip1.SetPxPyPzE(-0.226178, -0.198456, 1.946048, 1.974144)
pip2.SetPxPyPzE(0.554803, -0.301158, 1.301439, 1.453219)
pim.SetPxPyPzE(-0.07765, 0.072333, 1.372624, 1.38382)

# Print the magnitudes of each 4-vector
print("The magnitude of the photon 4-vector is", photon.Mag())
```

```
print("The magnitude of the proton 4-vector is", proton.Mag())
print("The magnitude of the pip1 4-vector is", pip1.Mag())
print("The magnitude of the pip2 4-vector is", pip2.Mag())
print("The magnitude of the pim 4-vector is", pim.Mag())

# Create a new 4-vector by adding/subtracting the others
diff = photon + proton - ( pip1 + pip2 + pim )

diffMass = diff.Mag()
print("The invariant mass of the missing particle is", diffMass)
```

"ROOT Histogram"

```
from __future__ import division, print_function
from ROOT import gRandom, TCanvas, TH1F

c1 = TCanvas('c1', 'Example', 200, 10, 700, 500)
hpx = TH1F('hpx', 'px', 100, -4, 4)

for i in xrange(100000):
    px = gRandom.Gaus()
    hpx.Fill(px)

hpx.Draw()
c1.Update()

# Save the plot as an image
c1.Print("myhistogram.eps")
```

"ROOT Histograms"

```
from __future__ import division, print_function
from ROOT import gRandom, TCanvas, TH1F

# Create a canvas
c1 = TCanvas('c1', 'My Basic Histograms', 200, 10, 700, 500)

# Divide the canvas into two plot areas
c1.Divide(1,2)

# Create a histograms
hist1 = TH1F('hist1', 'First', 100, -4, 4)
hist2 = TH1F('hist2', 'Second', 100, -4, 4)

for i in xrange(100000):
    px = gRandom.Gaus()
    hist1.Fill(px)

for i in xrange(100000):
    px = gRandom.Gaus()
    hist2.Fill(px)

# Go to plot area 1
c1.cd(1)

# Plot the first histogram
hist1.Draw()

# Go to plot area 2
```

```
c1.cd(2)

# Plot the second histogram
hist2.Draw()

c1.Update()

# Save the plot as an image
c1.Print("myhistogram.eps")
```

"Boundary Value Problem"

```
"""

1D_boundary_value_problem.py numerically solves the Sturm-Liouville equation

       -( p y' )' + qy = f

where p, q, and f are all functions of x, on the interval 0 < x < 1 with the
boundary conditions

       y(0) = y_0
       y'(1) = 0

"""
import matplotlib.pyplot as plt
import numpy as np


def p(x):
    return x**8


def q(x):
    return x**2 - 3*x + 1


def f(x):
    return np.exp(-x**2)


def Adotx(size, x, diagonal, upper, lower):
    """
    Computes the matrix-vector product Ax, where A is given as three
    diagonal band vectors.
    """
    z = np.zeros(size)
    z[0] = diagonal[0]*x[0] + upper[0]*x[1]
    for i in range(1, n-1):
        z[i] = diagonal[i]*x[i] + upper[i]*x[i+1] + lower[i-1]*x[i-1]
    z[n-1] = diagonal[n-1]*x[n-1] + lower[n-2]*x[n-2]
    return z


def residual(size, x, diagonal, upper, lower, b):
    """
    Computes the matrix-vector product Ax, where A is given as three
    diagonal band vectors.
    """
    res = np.abs(b - Adotx(size, x, diagonal, upper, lower))
    return res


def BICGSTAB(size, diagonal, upper, lower, b):
    """
    For a tridiagonal matrix, this function takes in the size (i.e. number
    of rows in the square matrix), the vector b, and the three vectors
    corresponding to the diagonal, upper diagonal row, and lower diagonal row
    of the matrix. It returns the approximation x, as well as the number of
```

```
    BICGSTAB iterations that were performed.
    """
    k = 0
    x = np.zeros(size)
    r = b - Adotx(size, x, diagonal, upper, lower)
    rhat = r.copy()
    rho0, alpha, omega = 1, 1, 1
    v, p = np.zeros(size), np.zeros(size)
    TOL = TOLfactor * np.mean(residual(n, x, D, U, L, b))
    while np.mean(residual(n, x, D, U, L, b)) > TOL:
        k += 1
        rho1 = np.dot(rhat, r)
        beta = (rho1/rho0)*(alpha/omega)
        p = r + beta*(p - omega*v)
        v = Adotx(size, p, diagonal, upper, lower)
        alpha = rho1/np.dot(rhat, v)
        h = x + alpha*p
        if np.mean(residual(n, h, D, U, L, b)) < TOL:
            return h, k
        s = r - alpha*v
        t = Adotx(size, s, diagonal, upper, lower)
        omega = np.dot(t, s)/np.dot(t, t)
        x = h + omega*s
        if np.mean(residual(n, x, D, U, L, b)) < TOL:
            return x, k
        r = s - omega*t
        rho0 = rho1
    return x, k


######################################################################
#                              MAIN
######################################################################

# Set the number of intervals
n = 100    # Number of intervals
h = 1/n    # Interval width
y0 = 1     # Left boundary condition

TOLfactor = 1e-7 # Error tolerance for BICGSTAB method

# Define the domain
x = np.linspace(0, 1, n+1)

# Define the b-vector
b = [0.5*h*f((x[0]+x[1])/2) + 0.5*h*f((x[1]+x[2])/2) + \
     (y0/h)*p((x[0]+x[1])/2) - (y0*h/6)*q((x[0]+x[1])/2)] # The b-vector
         containing the first element

for i in range(2, n):
    b += [0.5*h*f((x[i-1]+x[i])/2) + 0.5*h*f((x[i]+x[i+1])/2)]

b += [0.5*h*f((x[n-1]+x[n])/2)] # Add last element in b


# Define the diagonal elements of A
D = []
for i in range(1, n):
```

```
    a1 = (1/h)*p((x[i-1]+x[i])/2)
    a2 = (1/h)*p((x[i]+x[i+1])/2)
    a3 = (h/3)*q((x[i-1]+x[i])/2)
    a4 = (h/3)*q((x[i]+x[i+1])/2)
    D += [a1 + a2 + a3 + a4]

D += [(1/h)*p((x[n-1]+x[n])/2) + (h/3)*q((x[n-1]+x[n])/2)] # Last diagonal
    element


# Define the off-diagonal elements of A
U, L = [], []
for i in range(1, n):
    a1 = (-1/h)*p((x[i]+x[i+1])/2) + (h/6)*q((x[i]+x[i+1])/2)
    U += [a1]
    L += [a1]


# Compute the coefficients using the BICGSTAB algorithm for a sparse matrix
    system
y = BICGSTAB(n, D, U, L, b)[0]
y = np.hstack(([y0], y)) # Insert y0 as first element


#
# NOTE: Be mindful of the plotting scale. Python automatically scales the plots
# so that an effectively horizontal line looks nothing like a horizontal line.
#
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

"Schrodinger Equation"

```
"""
schrodinger.py gives the first several energy eigenvalues and plots the
    associated
wavefunctions for a given symmetric potential.

"""
from __future__ import division, print_function
import matplotlib.pyplot as plt
import numpy as np


def V(x):
    """
    The symmetric potential. You can change this to compute the
    energy eigenvalues and eigenstates for other symmetric potentials.

    Right now it is
        V(x) = 0.5*x^2
    for the quantum harmonic oscillator.
    """
    return 0.5*x**2


def f(r, x, E):
    """
    Write the set of first-order ODEs
        d(phi)/dx = 2[V(x) - E]psi
        d(psi)/dx = phi
    as a single vectorized function of the form
        f(r,x) = dr/dx.
    """
    psi = r[0]
    phi = r[1]
    fpsi = phi
    fphi = 2*(V(x) - E)*psi

    return np.array([fpsi, fphi], float)


def rk4(func, r, x, h, E):
    """
    4th order Runge-Kutta method for solving 1st order differential equations

    func: user defined function for the 1st order differential equations
    r: dependent variable
    x: independent variable
    h: independent variable step size
    """
    k1 = h*func(r, x, E)
    k2 = h*func(r + 0.5*k1, x + 0.5*h, E)
    k3 = h*func(r + 0.5*k2, x + 0.5*h, E)
    k4 = h*func(r+k3, x+h, E)
    return (k1 + 2*k2 + 2*k3 + k4)/6


def waveFunction(f, r, xValues, deltaX, E):
    """
```

```
    Solve for the wave function using the 4th order Runge Kuttta method
    to solve for the values for the wavefunction for a given value of the
    energy E.

    Note: The wave function is not necessarily an eigenfunction. It is only
    an eigenfunction if E happens to be an eigenvalue.
    an eigenvalue.
    """
    # make a copy of the initial values so that this function can be
    # repeatedly called with the same initial values
    s = np.copy(r)

    psi = []
    for x in xValues:
        psi += [s[0]]
        s += rk4(f, s, x, h, E)

    return np.array(psi, float)


def bracketingInterval(E, dE):
    """
    Given a starting energy and an increment value,
    find a bracketing interval for the energy eigenvalue.
    """
    bracket = False
    while bracket is False:
        E1, E2 = E, E + dE
        psi1 = waveFunction(f, r, xValues, h, E1)
        psi2 = waveFunction(f, r, xValues, h, E2)
        if (psi1[-1] < 0) != (psi2[-1] < 0):
            bracket = True

        if E2 > 1000:
            print("Bracketing interval not found!")
            break

        E += dE

    return [E1, E2]


def secantMethod(E1, E2):
    """
    Given a bracketing interval for the energy eigenvalue,
    find the energy eigenvalue and eigenstate using the secant method.
    """
    target = 1e-6

    while np.abs(E1 - E2) > target:
        # Get the wave function for energy for E1
        psiPoints = waveFunction(f, r, xValues, h, E1)

        # Get the wavefunction value at the boundary
        psiE1 = psiPoints[-1]

        # Get the wavefunction for energy E2
        psiPoints = waveFunction(f, r, xValues, h, E2)
```

```
        psiE2 = psiPoints[-1]

        # Use the secant method to get new estimates for E1 and E2
        E1, E2 = E2, E2 - psiE2 * (E2 - E1) / (psiE2 - psiE1)

    return [E2, psiPoints]


##########################################################################
#                              MAIN
##########################################################################

solutions = int(raw_input("How many solutions do you want. E.g. enter 3 if you "
                          "want the ground state and first two excited states: "))
limit = float(raw_input("Enter a plotting limit (E.g. 4): "))

vMin = 0.0 # Minimum potential energy
xMin, xMax, N = 0.0, limit, 1000
h = (xMax - xMin)/N

# The list of x-values
xValues = np.arange(xMin, xMax, h)

# Initialize the plot
plt.rc('text', usetex=True)
plt.title("Numerical Solutions of 1D Schrodinger Equation")


parity = 0 # Ground state is parity even for even V(x)
currentE = vMin # Ground state energy must be greater than vMin
for solutions in range(solutions):

    # Set the initial conditions, which depend on the parity
    if parity % 2 == 0:
        r = np.array([1.0, 0.0], float)
    else:
        r = np.array([0.0, 1.0], float)

    # Find a bracketing interval. If the energy levels are very
    # closely spaced, you may need to decrease the increment from
    # 0.1 to something smaller.
    interval = bracketingInterval(currentE, 0.1)
    E1, E2 = interval[0], interval[1]

    # Solve for the eigenvalue and wavefunction using the secant method
    solution = secantMethod(E1, E2)
    E, psi = solution[0], solution[1]

    # Extend the wavefunction to x < 0 using the symmetry of V(x)
    if parity % 2 == 0:
        psi = np.append(psi[::-1], psi[1:])
    else:
        psi = np.append(-psi[::-1], psi[1:])

    # Normalize the wavefunction
    norm = np.sqrt(np.dot(psi, psi)*h)
    psiN = psi/norm
```

```
    # Plot the normalized wavefunction
    x = np.append(-xValues[::-1], xValues[1:])
    label = r"$\psi_" + str(parity) + "(x)$"
    plt.plot(x, psiN, label=label)

    print("\nState", parity)
    print("E_", parity, " = ", E, sep="")
    print("Normalization constant: ", norm, sep="")

    # Expectation values
    print("<x^2>: ", np.dot(psiN, x*x*psiN)*h, sep="")

    # Increment the parity for the next solution
    parity += 1

    # Where to start the next bracketing interval
    currentE = E2

    # Check that the tail of psiN is approximately zero
    tail = psiN[len(psiN)-5:]
    if np.dot(tail, tail) > 1e-8:
        print("\nWARNING! Your plotting limit is probably too large or too small
            for this "
              "wavefunction, so your results may not be accurate! At the ends of "
              "your plotting region, the wavefunction should be zero. If the "
              "wavefunction has not yet decayed to zero, your plotting limit is "
              "too small. If the wavefunction shoots to +/- infinity, your
                  plotting"
              " limit is too large.")


# Plot the potential V(x) as well
xValues = np.append(-xValues[::-1], xValues[1:])
vValues = [V(i) for i in xValues]
plt.plot(xValues, vValues, label=r"$V(x)$", linestyle='dashed')

# Finish the plot
plt.legend(loc=1)
plt.xlabel(r"$x$")
plt.grid(True)
plt.ylim((-2, 2))
plt.show()
```

Index