Numerical Analysis

Ryan Tully-Doyle
University of New Haven

November 18, 2019

Abstract

The notes herein comprise a first course in numerical analysis, with a focus on implementation.

Numerical analysis is the area of mathematics concerned with computational solutions to *continuous* problems. That is, numerical analysis provides methods for approximation solutions to the problems encountered, for example, in calculus, differential equations, and linear algebra. We will be concerned with both methods and implemenation.

1 What you need - Octave/Matlab

This course is going to heavily emphasize programming - making mathematics useful by application. The only way to learn the techniques we are going to study is by actually using them. While standard techniques are prebuilt into various mathematical languages, we are going to be constructing our own implementations to solve problems.

The language that we are going to use is called Octave. Octave is an open-source (that is, free) language designed to be compatible with Matlab (which is an industry standard tool, but alas, not free. Matlab is installed on lab computers all over campus and on classroom computers. All code should work in both languages). Octave is available on Mac, PC, and Linux based computers. You should download and install it (or pay for a student license for Matlab, which is $50) as soon as possible.

You can find a link to Octave here: https://www.gnu.org/software/octave/
.

Alternatively, if you'd like to work in Matlab, you can find the student version here: https://www.mathworks.com/store/link/products/student/new?s_tid=ac_buy_sv_cta, with the relevant version being $49.99.

I will provide detailed help getting everything installed if you have trouble.

Addendum: You will need a current installation https://www.python.org/downloads/. Once it is installed, you can run the command `pip install sympy`, which will install all the symbolic support.

2 Motivation

You might reasonable ask why we need to learn numerical methods. After all, we've spent years learning explicit techniques for solving equations culminating in calculus. We know dozens of formulae for derivatives and integrals, powerfl techniques for evaluating them. We know how to use elimination and substitution to solve systems of linear equations. So why numerical methods? It's probably best to consider some examples. For instance, the function $f(x) = e^x$ is about as nice as functions come - it has a very smooth graph, since derivatives exist to all orders. Even better, its derivative is itself!

```
xlim = [0,2];
ylim = [0,e^2];

[x,y] = fplot('e^x', [xlim ylim]);

plot(x, y, 'linewidth',2);

set(gca,'xlim',xlim);
set(gca,'ylim',ylim);

grid on;
box on;
axis('nolabel','square');
title("Graph␣of␣f(x)␣=␣e^x")
```

That is one smooth looking plot. What if we replace x with x^2? (Functions of this form are very common in practice. The Gaussian or normal distribution from statistics is a very important example.)

```
xlim = [0,2];
ylim = [0,e^4];

[x,y] = fplot('e^(x^2)', [xlim ylim]);

plot(x, y, 'linewidth',2);

set(gca,'xlim',xlim);
set(gca,'ylim',ylim);

grid on;
box on;
axis('nolabel','square');
title("Graph␣of␣$g(x)␣=␣exp(x^2)")
```

It looks just as smooth, and in fact it is: all derivatives exist everywhere to all orders, and they are pleasing mixtures of polynomials and exponentials, just like Calculus I. Now, suppose I want to find the area under the graphs for $0 \leq x \leq 1$. From Calculus II, we know that we can evaluate the integral

$$\int_0^1 e^x \, dx = e - 1,$$

which is elementary. What about $g(x)$? Certainly we can still write

$$\int_0^1 e^{x^2} \, dx,$$

but now what?

There doesn't seem to be anywhere to go - the function e^{x^2} *does not have* a closed antiderivative. There is no way to use the techniques of basic calculus to deal with it. This should be alarming. After all, the function is about as nice as we could ask for. So what do we do? We'll need to look at **numerical calculus** for the answer.

What about other examples? From very early, we learn how to use algebra to solve equations. Are there equations that can't be solved with algebra? Consider the following question:

Question 2.1 Where do the graphs of $f(x) = 2x$ and $g(x) = \cos x$ intersect? $\square$

```
x = 0:.01:2;
y1 = 2*x;
y2 = cos(x);

plot(x, y1, 'r', x, y2, 'b')
```

A plot shows that they cross. But where? You learned a long time ago to find intersections between two graphs by setting the functions equal:

$$2x = \cos x$$

which really can be thought of as a root finding problem:

$$2x - \cos x = 0.$$

Can algebra solve this? The equation above is sometimes called a transcedental equation - that is, it involves functions that cannot be undone with the operations of algebra. Except in very special cases, there is no way to solve for x. And yet, we can see the intersection. To find it, we'll study a pile of techniques in an area called **root finding**.

Another area of interest is to construct a function from given data. That is, given some points, how can we find a function that passes through those points? This is a question known as **interpolation**. There are many approaches that we will discuss.

3 Taylor polynomials

3.1 Taylor's theorem

In calculus, you should have learned that an infinitely differentiable function possesses a representation in the form of an infinite power series. You are probably familiar with

$$\begin{aligned} e^x &= 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \dots \\ \cos x &= 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \dots \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \end{aligned}$$

Power series are an incredibly powerful tool for approximating functions, and they occur all over numerical mathematics. Typically, instead of using an infinite series, we truncate at a particular term, getting a polynomial that *approximates* the function rather than a series that is the function (where it converges). The general theorem that describes how to get a polynomial approximation for a function is called Taylor's theorem.

Theorem 3.1 *Let f be a function so that $n+1$ derivatives of f exist on an interval (a, b) and let $x_0 \in (a, b)$. Then for each $x \in (a, b)$, there exists a* *constant c strictly between x and x_0 such that*

$$f(x) = f(x_0) + \sum_{j=1}^{n} \left(\frac{f^{(j)}(x_0)}{j!}(x - x_0)^j \right) + \frac{f^{n+1}(c)}{(n+1)!}(x - x_0)^{n+1}.$$

The first part of the equation above is the **nth order Taylor polynomial for f at x_0**, and we use the notation

$$T_n(x) = f(x_0) + \sum_{j=1}^{n} \left(\frac{f^{(j)}(x_0)}{j!}(x - x_0)^j \right).$$

The second part is called the remainder term for T_n at x_0, and we use the notation

$$R_n(x) = \frac{f^{n+1}(c)}{(n+1)!}(x - x_0)^{n+1}.$$

You can think of the remainder term as containing the difference between f and T_n. We don't usually know the value of c for a given x_0, but we can the remainder term to put an upper bound on the worst possible error for a given Taylor approximation of f on (a, b).

So what's the upshot? Unwrapped from sigma notation, essentially any well-enough behaved function can be approximated at a point x_0 with a polynomial that is easy to calculate. That is, for values of x near x_0,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots,$$

and the worst that the approximation will be is the maximum value of $R_n(x)$ near x_0.

3.2 What is a Taylor approximation good for?

An obvious question should occur to you: why do we care about Taylor series and polynomials? The short answer is that Taylor polynomials behave much like their parent functions "near" the point of expansion. By including more terms, we typically expect to get a better approximation of the original function and a larger interval on which the approximation is well-behaved. A classical example of this phenomenon can be seen in the Taylor polynomials for $\sin x$ near $x_0 = 0$.

```
X = -7:.01:7;
Y1 = sin(X);
Y2 = X - X.^3/factorial(3) + X.^5/factorial(5) -
    X.^7/factorial(7) + X.^9/factorial(9);

plot(X, Y1, 'r', X, Y2, 'b')
axis( [-7 7 -2 2])
```

The idea of replacing a function with a polynomial built from its derivatives is a powerful computational technique. Indeed, if we're working with approximations, we often need not even compute formulae for the derivatives but instead work with difference quotients. This will require that we have a firm understanding of how error accumulates, which will be discussed in a future lecture.

3.3 Coding functions in Octave

Let's begin by talking a bit about how to use functions in Octave. Suppose that we are interested in the 3rd Taylor polynomial for the function $f(x) = e^x$ at $x_0 = 0$. A bit of computation will tell you that

$$T_3(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}.$$

First, we'll need to define the function T_3 so that we can use it like a function - we want to be able to plot and evaluate it at various points. To do so, we'll use the `inline` command.

```
T3 = inline('1␣+␣x␣+␣1/2*x^2␣+␣1/6*x^3')
```

Once we have the function defined, we can evaluate it anywhere.

```
T3 = inline('1␣+␣x␣+␣1/2*x^2␣+␣1/6*x^3')
T3(0)
T3(1)
T3(10)
T3(-1)
```

A standard technique that we need to be comfortable with is evaluating functions on arrays (that is, lists of numbers). Here are some example arrays.

```
#the first array generates the integers
#from 1 to 10 and stores the list
#in a variable A
A = 1:10

#the second example generates the numbers between 0 and 2
#incrementing by .02. it is long. we could suppress the
#output by adding a ; to the end of the line.
B = 0:.02:2
```

An incredible useful feature of Octave/Matlab is the ability to feed an array into a function. However, Octave assumes that we want matrix operations unless we tell it that we want to work entry by entry. The way we do that is to put a . before any operation that could be interpreted as a matrix operation to force Octave to work entrywise.

```
#the exponentiation needs a dot because it represents
#repeated multiplication of objects that might be matrices.
T3 = inline('1␣+␣x␣+␣1/2*x.^2␣+␣1/6*x.^3')
A = 1:10;
C = T3(A)
```

Plotting in Octave seems complicated, but really just exposes how mathematical software generates plots in general. Suppose that we want to compare the function $f(x) = e^x$ with the Taylor polynomial T_3. Our goal is plot both functions on the same set of axes. A useful command for this is `plot`.

```
#first, we make an array of numbers from -3 to 3,
    incrementing by .01
X = -3:.01:3;

#now, we define T3 (e^x already exists as a function called
    exp())
```

```
T3 = inline('1␣+␣x␣+␣1/2*x.^2␣+␣1/6*x.^3');

#we need some y-values to plot, so we feed T3 and e^x our
    X-value array
Y1 = T3(X);
Y2 = exp(X);

#finally, we plot the graphs.
plot(X, Y1, 'r', X, Y2, 'b')
legend('T_3(x)', 'e^x')
legend("boxoff")
legend("left")
```

4 Root finding

In elementary or middle school, we learn how to solve algebraic equations. Equations that are algebraic can be solved with familiar operations: addition, multiplication, exponentiation. However, there are very simple equations that cannot be solved with algebra. Consider

$$\cos x = x.$$

There is no way to extract the x from inside the cosine to isolate it on one side of the equation. Even algebraic equations may be impossible to solve. Consider

$$x^6 - x - 1 = 0.$$

There is no formula that can be used to factor this equation (like the quadractic formula). If we're not fortunate enough to get a polynomial that has obvious "nice" factors, the only way to proceed is to approximate the solutions. Questions like this fall under the general category of **root finding** problems. A root is a solution to the equation $f(x) = 0$.

4.1 Bisection method

We're going to start with a straightforward equation to illustrate our first method. Let's find solutions to the equation

$$x^2 - 2 = 0.$$

Now obviously, the answers are $\pm\sqrt{2}$, but how much good is that for computation or application? Where do the values for $\sqrt{2}$ even come from? (Hint: $\sqrt{2}$ is defined to be the solution to this equation).

```
X = -3:.1:3;
Y = X.^2 - 2;
plot(X, Y)
set(gca, "xaxislocation","origin")
set(gca, "yaxislocation", "origin")
```

It's clear from the plot generated by the code above that the solutions to $x^2 - 2 = 0$ are the x-intercepts of the function $f(x) = x^2 - 2$. If you were asked to guess what the roots were you might say "between -2 and -1 and also between 1 and 2". That observation is the first step in a set of methods called **bracketing**. At this point, we can take advantage of one of the core theorems of calculus: the Intermediate Value Theorem.

Theorem 4.1 *Let f be a continuous function on the interval $[a, b]$. Then for every y falling strictly between $f(a)$ and $f(b)$, there exists a number c strictly between a and b so that $f(x) = y$.*

A more familiar formulation might state "if a continuous f is positive at a point a and negative at a point b then it must have gone through 0 somewhere". This can be restated into a condition easy to check in an algorithm.

Theorem 4.2 *Suppose that f is continuous on $[a, b]$. Then $[a, b]$ contains a root of f if $f(a)f(b) < 0$.*

So we arrive at the main idea of the bisection method, which you can think of as a narrowing process. In short, we'll bracket the root with an interval, cut it in half, then use the condition to figure out which half contains the root. Repeating this process will produce an arbitrarily small interval containing the root, which we'll denote α. We can treat the midpoint of this tiny interval as an approximation for the root α.

Consider our example, which is to find a root of the function $f(x) = x^2 - 2$.

1. From the graph, we can bracket one of the roots with $a = 1, b = 2$.
2. We can check to see that we've chosen good values by looking at $f(a)f(b) = (-1)(2) = -2 < 0$, and so the IVT guarantees a root lies in $[1, 2]$.
3. Now bisect the interval. Compute the midpoint $m = \frac{a+b}{2} = 1.5$.
4. We'll check the interval $[1, 1.5]$ for the root. Since $f(1)f(1.5) = -.25$, this interval contains the root.
5. We can repeat the process with our new guess, $m = 1.25$, and so on until we're satisfied with the accuracy of our calculation.

4.2 Error

So how do we decide when to stop the process listed above? Since we don't know what the answer is, we can't use "distance from the correct answer" as a measure of error. Instead, we'll have to measure something about the process of iteration itself. We need to make the measurements of error relative to the process, not just absolute, to be useful. This will be discussed in more detail when we introduce Newton's method.

Definition 4.3 Given an iterative process that produces an approximation m_k at each step k, the **absolute relative approximate error** at step k is

$$|\epsilon_k| = \left| \frac{m_k - m_{k-1}}{m_k} \right|.$$

$\Diamond$

If an iterative approximation converges, then the error will decrease from step to step. Thus, we can use error to determine when to stop. Before we begin the process of approximation, we choose a **tolerance** - that is, a small number that is the maximum possible error we're willing to allow. Then, we can impose a stopping condition whenever $|\epsilon_k| < tol$.

4.3 Bisection algorithm

Algorithm 4.4 *Suppose that f is continuous on $[a, b]$ and that $f(a)f(b) < 0$. Let tol be a given tolerance. Let $maxiter$ be the maximum number of iterations. Set $m = \frac{a+b}{2}$ and $iter = 0$.*

1. *Set $iter = iter + 1$. If $iter > maxiter$, exit and return failed to converge.*

2. *If $f(a)f(m) < 0$, set $b = m$. Otherwise set $a = m$.*
3. *Set $m' = \frac{a+b}{2}$.*
4. *Set $|\epsilon| = \frac{m'-m}{m'}$.*
5. *If $|\epsilon| < tol$, return m'.*
6. *Otherwise, let $m = m'$ and go to step 1.*

The bisection method always converges, so we do not need to include a maximum iteration counter, though we do anyway as later methods may not converge and can potentially loop forever.

We'll now present an example of the bisection method used to compute the value of $\sqrt{2}$.

```
f = @(x) x.^2 - 2;
a = 1;
b = 2;
tol = .5*10^-4;
M = (a+b)/2;
max_iter = 1000;
iter = 0;
err = 1;


while err > tol
    iter = iter + 1;
    if iter > max_iter
        break
    endif

    if f(a)*f(M) < 0
        b = M;
    else
        a = M;
    endif

    m = (a+b)/2;
    err = abs((m - M)/m);
    M = m;
endwhile

printf('The␣approximation␣is␣%d', m)
```

4.4 False position (Regula Falsi)

The method of false position is one of the oldest methods for guessing the solution to an equation. We're going to use a version of it that is specifically built for root finding. This method can have significantly faster convergence than the bisection method, as it takes functional behavior into consideraton.

Consider the equation $x^2 - 2 = 0$, the solutions of which are the roots of the function $f(x) = x^2 - 2$.

```
f = @(x) x.^2 - 2
ezplot(f)
```

As in the bisection method, we notice that the interval $[1,2]$ must contain a root, as the function passes through the axis between them. (That is,

$f(1)f(2) < 0$.) To find an approximation of the zero, we construct the line between $(a, f(a))$ and $(b, f(b))$. We can approximate the root of $f(x)$ by the x-intercept of the line that we've drawn.

```
f = @(x) x.^2 - 2;
a = 1;
b = 2;
g = @(x) f(a) + (f(b) - f(a))/(b-a) * (x - a);
X = -3:.1:3;
XX = a:.1:b;
plot(X, f(X), 'r', XX, g(XX), 'b')
hold on;
plot(sqrt(2),0, 'r*')
plot((a*f(b) - b*f(a))/(f(b) - f(a)), 0, 'b*')
axis([.5 2.5 -1.5 2.5])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

Already from the picture above we can see that we've got a reasonable approximation of the zero.The slope the line is going to be given by $m = \frac{f(b)-f(a)}{b-a}$, and some quick algebra with the point slope form of the line will give you that the x-intercept of the line segment is

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

If we iterate this method, we should see the intercepts of the line segments "march" towards the actual zero. At this point, the method is identical to the bisection method: we identify which of the two subintervals created by the new approximation is the bracking subinterval, we replace the relevant endpoint, and we repeat the line trick.

Algorithm 4.5 *Suppose that f is continuous on $[a, b]$ and that $f(a)f(b) < 0$. Let tol be a given tolerance. Let maxiter be the maximum number of iterations. Set $m = \frac{af(b)-bf(a)}{f(b)-f(a)}$ and iter $= 0$.*

1. *Set iter $=$ iter $+ 1$. If iter $>$ maxiter, exit and return failed to converge.*
2. *If $f(a)f(m) < 0$, set $b = m$. Otherwise set $a = m$.*
3. *Set $m' = m = \frac{af(b)-bf(a)}{f(b)-f(a)}$.*
4. *Set $|\epsilon| = \frac{m'-m}{m'}$.*
5. *If $|\epsilon| <$ tol, return m'.*
6. *Otherwise, let $m = m'$ and go to step 1.*

4.5 Classes of differentiability

Our next set of techniques do not involve bracketing. In exchange for losing a bracket that is guaranteed to contain a root of a function, we get methods that converge significantly faster when certain hypotheses are met. Speed of convergence will be discussed in detail in a later section.

The main hypotheses of derivative based methods is that the functions being worked with are "nice enough". Nice as a mathematical terms is an amusing catchall for "whatever is needed to make this theorem run", but it usually refers to smoothness of some kind. That is, the slope of the function is well behaved as we move around the x-values.

In Calculus 1, we learn that

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

where we're using a form that emphasizes the fact that the derivative is the limit of secant lines. A function for which f' exists for every $x \in (a, b)$ is called differentiable on (a, b). Be careful. It's easy to be lulled into a false sense of security about differentiable functions.

Let's consider a family of functions called the **topologist's sine curves**. Here is the first member of the family.

$$\begin{aligned} f(x) &= \sin \frac{1}{x} && \text{if } x \neq 0 \\ &= 0 && \text{if } x = 0 \end{aligned}$$

Notably, this function does not have a limit as $x \to 0$ from the left or the right. We can try to fill the singularity in the function with the point $(0, 0)$, but that doesn't make the function continuous. The reason why should be clear from the graph below: as the function approaches $x = 0$ it oscillates increasingly quickly, essentially filling the area on the y-axis between -1 and 1. Since the function has a value of 0 at 0, but no limit, f is discontinuous there.

```
f = @(x) sin(1./x)
X = -2:.0001:2;
plot(X, f(X))
hold on;
plot(0,0,'r*')
axis([-1 1 -1 1])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

Let's look at the next member of the family:

$$\begin{aligned} f(x) &= x \sin \frac{1}{x} && \text{if } x \neq 0 \\ &= 0 && \text{if } x = 0 \end{aligned}$$

The x in front forces the function to go to 0, capturing it between the lines $y = x$ and $y = -x$.

```
f = @(x) x.*sin(1./x)
X = -2:.0001:2;
plot(X, f(X))
hold on;
plot(0,0,'r*')
axis([-1 1 -1 1])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

The graph above should make it obvious that now, the point $(0, 0)$ fills the hole in the function and makes f continuous at 0. What about the derivative?

```
f = @(x) x.*sin(1./x)
g = @(x) sin(1./x) - 1./x.*cos(1./x)
X = -2:.0001:2;
```

```
plot(X, f(X), 'b', X, g(X), 'g')
hold on;
plot(0,0,'r*')
axis([-1 1 -1 1])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

The green function above is the derivative. The formula for the derivative makes sense everywhere but at 0. So we'll use the defintion to see if the derivative is defined there.

$$\begin{aligned}
f'(0) &= \lim_{h\to 0} \frac{f(0+h)-f(0)}{h} \\
&= \lim_{h\to 0} \frac{f(h)}{h} \\
&= \lim \frac{h\sin(1/h)}{h} \\
&= \lim_{h\to 0} \sin\frac{1}{h}
\end{aligned}$$

which does not exist. So f is continuous, but not differentiable. Another member of the family:

$$\begin{aligned}
f(x) &= x^2\sin\frac{1}{x} && \text{if } x \neq 0 \\
&= 0 && \text{if } x = 0
\end{aligned}$$

```
f = @(x) x.^2.*sin(1./x)
g = @(x) 2*x.*sin(1./x) - cos(1./x)
X = -2:.0001:2;
plot(X, f(X), 'b', X, g(X), 'g')
hold on;
plot(0,0,'r*')
axis([-1 1 -1 1])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

Again, the derivative formula makes sense everywhere but 0. At 0, we can use the definition:

$$\begin{aligned}
f'(0) &= \lim_{h\to 0} \frac{f(0+h)-f(0)}{h} \\
&= \lim_{h\to 0} \frac{f(h)}{h} \\
&= \lim \frac{h^2\sin(1/h)}{h} \\
&= \lim_{h\to 0} h\sin\frac{1}{h} = 0.
\end{aligned}$$

Here's the complete graph of the derivative, given that $f'(0) = 1$. It should be obvious that even though the derivative exists everywhere, it is not continuous.

```
g = @(x) 2*x.*sin(1./x) - cos(1./x)
X = -2:.0001:2;
plot(X, g(X), 'g')
hold on;
plot(0,0,'r*')
```

```
axis([-1 1 -1.5 1.5])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

As we keep increasing the power of x, we get functions with better properties. What happens if we increase the power one more time?

$$\begin{aligned} f(x) &= x^3 \sin\frac{1}{x} && \text{if } x \neq 0 \\ &= 0 && \text{if } x = 0 \end{aligned}$$

```
f = @(x) x.^3.*sin(1./x)
g = @(x) 3*x.^2.*sin(1./x) - x.*cos(1./x)
X = -2:.0001:2;
plot(X, f(X), 'r', X, g(X), 'g')
hold on;
plot(0,0,'r*')
axis([-1 1 -1.5 1.5])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

As before, the green function is the derivative. Notice that now the derivative is also converging to 0 as $x \to 0$. That is, f is continuous, differentiable everywhere, AND the derivative is continuous.

This is what we mean by "nice" in the sense of approximation. In order for derivative methods to be well-behaved, the derivatives should exist AND be continuous

Definition 4.6 Let f be a function on an interval (a, b). The function f is said to be **of class** C^k if f can be differentiated k times and $f^{(k)}$ is continuous on (a, b). $\Diamond$

C^1 functions are also called **continuously differentiable**. Thus, by filling in the point at $(0, 0)$, we can say

- $\sin(1/x)$ is discontinuous on $(-a, a)$;
- $x \sin(1/x)$ is continuous on $(-a, a)$;
- $x^2 \sin(1/x)$ is differentiable on $(-a, a)$;
- $x^3 \sin(1/x)$ is C^1 on $(-a, a)$.

This can be continued. Generally, C^1 functions are the bare minimum we require to call a function "nice". You should also note that this is a hierarchy of regularity: every subsequent level has all the properties of the level that came prior.

4.6 Newton-Raphson method

We come to a very powerful method that illustrates perhaps the central idea of numerical analysis (and indeed most of continuous mathematics):

Every function is a line.

This might seem like an absurd statement, but with some slight adjustments, maybe we can be convinced.

Every (nice enough) function is a line (if you look closely enough).

In fact, this is the essential point of working with tangent lines. Indeed, the same thinking applies in more variables as well (where lines get replaced by planes).

Functions are difficult to find x-intercepts for, *unless those functions are lines.* So what we'll do is just pretend the function is a line, and find its intercept. In more familiar terms, we'll replace the function with its tangent line approximation.

As an easy example, consider $f(x) = x^2 - 2$, our old friend that lets us find the value of $\sqrt{2}$. Suppose that we guess that the root is close to $x = 2$.

```
f = @(x) x.^2 - 2;
T = @(x) 4*(x - 2) + 2;
X = 0:.01:3;
plot(X, f(X), 'r', X, T(X), 'b')
hold on;
plot(sqrt(2), 0, 'b*')
plot(1.5, 0, 'b*')
axis([1 3 -2 7])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

Indeed, the tangent line (in blue) lands very close to the root in question (on the red curve). We'd like to be able to iterate this procedure. Suppose that we're given a point $(x_0, f(x_0))$. The tangent line to f at that point is

$$y - f(x_0) = f'(x_0)(x - x_0).$$

Solving for x when $y = 0$ gives the equation

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

This simple equation is the heart of Newton's method. To iterate the approximation, let $x_1 = x$ and solve the equation again:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

The hope is that given a good enough guess, that this sequence of approximations will approach the root. Let's see how it bahaves in this case.

```
format long
f = @(x) x.^2 - 2;
fprime = @(x) 2*x;
g = @(x) x - f(x)./fprime(x);
guess = 2;
for i = 1:4
    new = g(guess);
    disp("Approximation␣is:"), disp(new)
    guess = new;
endfor

err = (sqrt(2) - guess)/sqrt(2);
disp("Relative␣error:")
disp(err)
```

What the output above shows is that starting from $x = 2$, Newton's method converges to within .0000000001% of the true value of $\sqrt{2}$ in just four steps.

At this point, excited, we decide to test another case.

```
f = @(x) x.^20 - 2;
X = -1.1:.01:1.1;
plot(X, f(X))
axis([-1.5 1.5 -3 5])
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

That is one flat function. But that means the tangent line is going to have a slope very close to 0 if we guess to the left of the root...

```
f = @(x) x.^20 - 2;
fprime = @(x) 20*x.^19;
guess = .7;
new = guess - f(guess)./fprime(guess);
disp(new)
disp(f(new))
```

That is, a guess of $x_0 = .7$, which is pretty close to the root displayed on the graph, gives a second approximation of 88.3929, which means computing the second tangent line at the point $(88.3928, 8.47884 \times 10^{38})$. Not good.

4.7 Newton's method requirements and problems

So we've discovered the first problem in the last example. Newton's method does not work well near places where functions are very flat. In particular, this means that we'll want to avoid local extrema -- hitting near one with an approximation could send the tangent line way beyond the approximation area.

What other problems might arise?

5 Interpolation

5.1 Naive polynomial interpolation

Given a set of points, say $P(x_1, y_1), Q(x_2, y_2), R(x_3, y_3)$, we say that a function f **interpolates** P, Q, R if the graph of f passes through each prescribed point - that is, $f(x_i) = y_i$ for each $i = 1, \ldots 3$ in this case. The function f can be used to predict the values of y for values of x that fall in between these points. (Predicting points outside of the data is called **extrapolation**.)

A first approach to this problem might take advantage of the following fact.

Theorem 5.1 $p_1, \ldots, p_n n - 1$

One possible proof of this fact is the construction of the Lagrange polynomials, to be discussed in the next section.

So suppose that we are given the points $(1, 1), (2, -3), (5, 10)$. A quick plot shows that they are clearly non-collinear. Thus, there should exist a degree two polynomial that interpolates the points, which will have the general form

$$y = ax^2 + bx + c.$$

The unknowns we need to find are the coefficients a, b, c. So we'll plug in the known data and get a system of equations.

$$\begin{aligned} 1 &= a + b + c \\ -3 &= 4a + 2b + c \\ 10 &= 25a + 5b + c \end{aligned}$$

We can solve the system using matrices.

```
A = [1, 1, 1, 1;  4, 2, 1, -3; 25, 5, 1, 10]
B = rref(A)
```

Doing this by hand would be potentially very time consuming, particularly if we had to work with many points, not just three. Our result says that the approximate polynomial that interpolates the data is

$$f(x) = 2.08333x^2 - 10.25x + 9.16667.$$

Note that this polynomial is unique (up to approximation error). That is, any other technique that finds a degree two polynomial through these three points will find the same result.

In the next few sections, we will discuss two more interpolation techniques, one that is very easy for humans to understand (and evaluate with a little modification in certain cases) and one that is easy to program into a computer technique.

```
f = @(x) 2.08333* x.^2 - 10.25 *x + 9.16667;
X = 0:.01:6;
Y = f(X);
plot([1, 2, 5],[1, -3, 10], 'b*')
hold on
plot(X,Y, 'r')
```

Figure 5.2 Data and interpolating function example

5.2 Lagrange interpolation

We'll begin with a method due to Lagrange that makes it very easy to write down by hand what the interpolating polynomal is (but doesn't give it in a computationally efficient form). Still, Lagrange interpolation is based on fundamental observations about the graphs of functions.

Suppose that we are given the points $(1, 0), (2, 0), (5, 0)$ and we are asked to provide an interpolating polynomial of degree two (these points are collinear, but they very soon won't be). One obvious choice is to form the polynomial

$$p(x) = (x - 1)(x - 2)(x - 5).$$

This polynomial is not a unique polynomial of degree two, but it certainly interpolates them. We are going to use the idea of interpolating roots, as we've done here, to build interpolating functions for data off of the x-axis.

This exercise can be made slightly harder by allowing one of the points to move off of the x-axis. Now consider the points $(1, 0), (2, 0), (5, 10)$. Ignoring the third point for the moment, we can still interpolate $(1, 0), (2, 0)$ in the same way as before: with $p(x) = (x-1)(x-2)$. Maybe we're lucky and p also passes through $(5, 10)$. We can check -- $p(5) = (5 - 1)(4 - 1) = 12$. That is, p misses $(5, 10)$.

```
f = @(x) (x - 1).*(x-2);
X = 0:.01:6;
Y = f(X);
plot([1, 2, 5],[0, 0, 10], 'b*')
hold on
plot(X,Y, 'r')
```

What we need is a way to adjust p that keeps the function passing through $(1,0)$ and $(2,0)$. We might notice that every function of the form $f(x) = c(x-1)(x-2) = c\cdot p(x)$ has this property, so let's choose a value of c that makes f pass through $(5,10)$.

$$\begin{aligned}10 &= f(5) = c\cdot p(5) = c(5-1)(5-4) = 12c\\ \frac{10}{12} &= c.\end{aligned}$$

So $f(x) = \frac{10}{12}p(x) = \frac{10}{12}(x-1)(x-2)$ interpolates the data.

```
f = @(x) 10/12*(x - 1).*(x-2);
X = 0:.01:6;
Y = f(X);
plot([1, 2, 5],[0, 0, 10], 'b*')
hold on
plot(X,Y, 'r')
```

Now, notice that 10 was the value we wanted to be output with input 5, and that $p(5) = 12$. In fact, if $(5,10)$ had been given the name (x_3, y_3), then the value of c would have been $c = \frac{y_3}{p(x_3)}$, and the interpolating function for $(1,0),(2,0)$ and (x_3,y_3) would have been

$$f(x) = \frac{y_3}{p(x_3)}p(x) = \frac{y_3}{p(x_3)}(x-1)(x-2).$$

We can modify this further: given initial data $(x_1,0),(x_2,0),(x_3,y_3)$, define

$$p(x) = (x-x_1)(x-x_2)$$

and

$$f(x) = \frac{y_3}{p(x_3)}p(x) = \frac{y_3}{p(x_3)}(x-x_1)(x-x_2).$$

f is the interpolating function for the data.

We're finally ready to tackle the full problem: using this approach to find the interpolating polynomial for $(1,1),(2,-3),(5,10)$. We'll approach the problem in three pieces: First, construct $f_3(x)$ for the points $(1,0),(2,0),(5,10)$ using the ideas from the last section.

$$f_3(x) = \frac{10}{12}(x-1)(x-2).$$

Second, construct $f_2(x)$ for the points $(1,0),(2,-3),(5,0)$. Since $p_2(x) = (x-1)(x-5)$, we get $c = \frac{-3}{p_2(2)} = \frac{-3}{-3} = 1$. So

$$f_2(x) = 1(x-1)(x-5)$$

interpolates this data.

For a third step, construct $f_1(x)$ for the points $(1,1),(2,0),(5,0)$. In this problem, $p_1(x) = (x-2)(x-5)$, and so $c = \frac{1}{p(1)} = \frac{1}{4}$. Thus,

$$f_1(x) = \frac{1}{4}(x-2)(x-5).$$

What follows is a plot of the three solutions to the subproblems.

```
f3 = @(x) 10/12*(x - 1).*(x - 2);
f2 = @(x) (x - 1).*(x-5);
```

```
f1 = @(x) 1/4*(x-2).*(x-5);
X = 0:.01:6;
plot(X, f1(X), X, f2(X), X, f3(X))
hold on
plot([1, 2, 5],[1, -3, 10], 'b*')
plot([1, 2, 5],[0,0,0], 'r*')
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

You should notice that each parabola passes through one of the real points and two of the substituted roots. We are at the magic step. Let

$$f = f_1 + f_2 + f_3.$$

```
f3 = @(x) 10/12*(x - 1).*(x - 2);
f2 = @(x) (x - 1).*(x-5);
f1 = @(x) 1/4*(x-2).*(x-5);
f = @(x) f1(x) + f2(x) + f3(x)
X = 0:.01:6;
plot(X, f(X))
hold on
plot([1, 2, 5],[1, -3, 10], 'b*')
plot([1, 2, 5],[0,0,0], 'r*')
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

What happened? Let's look at f more closely.

$$f(x) = \frac{1}{4}(x-2)(x-5) + (x-1)(x-5) + \frac{10}{12}(x-1)(x-2).$$

Notice that for a given input for one of the data points, only one term is non-zero, and that term has been built to evaluate to the correct y-value. That is,

$$\begin{aligned} f(1) &= \frac{1}{4}(-1)(-4) + 0 + 0 = 1 \\ f(2) &= 0 + (1)(-3) + 0 = -3 \\ f(5) &= 0 + 0 + \frac{10}{12}(3)(4) = 10 \end{aligned}$$

Furthermore, f must be unique, because f is a degree two polynomial. (In fact, f is the same function from the previous section if you multiply it out.)

So what is the general procedure? Start with a list of interpolants, $(x_1, y_1), \ldots, (x_n, y_n)$. For each i, let

$$p_i(x) = \prod_{j \neq i}(x - x_j) = (x - x_1)\ldots(x - x_{i-1})(x - x_{i+1})\ldots(x - x_n).$$

Then define f_i to be

$$f_i(x) = \frac{y_i}{p_i(x_i)} p_i(x),$$

and finally the **Lagrange interpolating polynomial** to be

$$f(x) = \sum_i f_i(x) = f_1(x) + \ldots + f_n(x).$$

5.3 Newton interpolation

Lagrange interpolation is easy to describe and relatively easy to write out by hand, even for many point, with some practice. However, the computation of the Lagrange interpolating polynomial doesn't easily lend itself to the structure of computer programs. We would like a method for writing down an interpolating polynomial that can be performed in a straightforward way in a recursive fashion. The approach that we present here is called **Newton interpolation**. Of course, the result will be the same, as interpolating polynomials of degree $n-1$ are unique for a given set of m points.

Suppose that we're given two points, $(x_0, y_0), (x_1, y_1)$. The lowest degree polynomial f through the first point is the horizontal line $f(x) = y_0 = f(x_0)$. The lowest degree polynomial f through the two points is

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0).$$

We call the quantities y_0 and $\frac{y_1-y_0}{x_1-x_0}$ **divided differences.** So now lets see what happens with three points.

Suppose that we're given three points to interpolate, $(x_0, y_0), (x_1, y_1), (x_2, y_2)$. We claim that we can construct a unique quadratic polynomial that interpolates the data of the form

$$f(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1),$$

for some as yet unknown constants b_0, b_1, b_2 that depend on the points. (You can easily show that for noncollinear points, you get a nonsingular coefficient matrix for the resulting system, which implies a unique solution.)

To find the values of the coefficients, we'll plug in our data points.

$$f(x_0) = b_0 + 0 + 0$$

and so $b_0 = f(x_0) = y_0$. Moving to x_1,

$$\begin{aligned} f(x_1) &= b_0 + b_1(x_1 - x_0) + 0 \\ y_1 &= y_0 + b_1(x_1 - x_0) \\ b_1 &= \frac{y_1 - y_0}{x_1 - x_0} \end{aligned}$$

Notice that so far, we've exactly reproduced the line connecting the first two points. Now let's look at b_2.

$$\begin{aligned} f(x_2) &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0) + b_2(x_2 - x_0)(x_2 - x_1) \\ y_2 - y_0 - \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0) &= b_2(x_2 - x_0)(x_2 - x_1) \\ b_2 &= \frac{\frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{y_1-y_0}}{x_2 - x_0}, \end{aligned}$$

where the last step requires a bit of non-obvious algebra, but provides a more useful form.

It turns out that this process can be repeated for additional points, though one can imagine the formula for the next step is quite complicated to express in x_i, y_i. So we introduce a notation that will make this process easy to write in recursive form. The mth divided difference is given by the recursive formula

$$f[x_i] = y_i;$$

$$f[x_m, \ldots, x_0] = \frac{f[x_m, \ldots, x_1] - f[x_{m-1}, \ldots, x_0]}{x_m - x_0}.$$

For example, in this notation,

$$f[x_1, x_0] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{y_2 - y_1}{x_2 - x_1},$$

and you should convince yourself that the expression for $f[x_2, x_1, x_0]$ agrees with our previous computation.

We want to use this approach because we'd prefer our computations to be purely in terms of array entries, not the application of general formulas. As an example, we will trace through the example of computing the coefficients of the Newton polynomial for the points $(1,1), (2,3), (5,10)$ in code.

```
nodeX = [1, 2, 5];
nodeY = [1, 3, 10];
#this section calculates the coefficients
data = zeros(3); #3x3 matrix of 0s
for i = 1:3
    data(1,i) = nodeY(i)
endfor
for i = 1:2
    data(2,i) = (data(1,i+1) -
        data(1,i))/(nodeX(i+1)-nodeX(i));
endfor
for i = 1:1
    data(3,i) = (data(2,i+1)- data(2,i))/(nodeX(i+2) -
        nodeX(i));
endfor
b = data(:,1)'
#this␣section␣creates␣the␣Newton␣polynomial.␣It␣is␣
    unenlightening␣at␣the␣moment,␣because␣I␣am␣trying␣to␣
    avoid␣using␣symbolic␣variables.␣To␣be␣rewritten.
structure␣=␣@(x,␣b)␣b(1)␣+␣b(2)*(x␣-␣nodeX(1))␣+␣b(3)*(x␣-␣
    nodeX(1)).*(x␣-␣nodeX(2));
poly␣=␣@(x)␣structure(x,␣b);
scatter(nodeX,␣nodeY,␣400)
hold␣on
plot(0:.01:6,␣poly(0:.01:6))
```

The code above should provide an example of how to implement the recursion without having to rely on increasingly complex formulas to compute the coefficients. As a goal, one could try to produce all of the data in the matrix "data" in one loop, instead of the many loops that I have (suggestively) used. One could also write a much more revealing computation of the final polynomial.

5.4 Problems with polynomials

Polynomial interpolation has a serious drawback in many cases: high degree polynomials can behave wildly between points in a way that doesn't reflect the expected behavior of the underlying function. To see this, consider the function

$$f(x) = \frac{1}{x^2 + 25}$$

which is a typical example of a rational function with no vertical asymptotes.

```
X = -1:.01:1;
f = @(x) 1./(1 + 25*x.^2);
plot(X, f(X))
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

We will sample this function and then attempt to reconstruct it using polynomial interpolation. Suppose that we choose six equally spaced points from f.

```
X = -1:.01:1;
f = @(x) 1./(1 + 25*x.^2);

P = [-1, -.6, -.2, .2, .6, 1];


plot(X, f(X))
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
hold on
scatter(P, f(P), 400)
```

Now suppose we find the unique polynomial that goes through these six points. (The code here uses the command polyfit.)

```
X = -1:.01:1;
f = @(x) 1./(1 + 25*x.^2);

P = [-1, -.6, -.2, .2, .6, 1];


scatter(P, f(P), 400)
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
hold on

plot(X, f(X))
p = polyfit(P, f(P), 5)
plot(X, polyval(p,X))
```

Notice that the interpolating polynomial doesn't appear to behave like the original function at all, oscillating through the end points. Perhaps we can fix the problem by choosing more points.

```
X = -1:.01:1;
f = @(x) 1./(1 + 25*x.^2);

Q = -1:.2:1;

scatter(Q, f(Q), 400)
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
hold on

plot(X, f(X))
q = polyfit(Q, f(Q), 11)
plot(X, polyval(q,X))
```

Unfortunately, this seems to have exacerbated the problem. The polynomial starts to oscillate wildly and to grow sharply between nodes near the outside of the domain. It turns out that this behavior is common in high degree polynomial interpolation of equally-spaced points. Replacing a complicated function with a polynomial interpolant isn't going to be as easy as sampling equidistant points, and we should expect that similar bad behavior is going to crop up in data sets with many points if we use a unique polynomial fit.

There are many approaches to dealing with this undesirable behavior. In the next section, we'll talk about building smooth curves that pass through all of the points by considering them three at a time. In this section, we should at least be aware of better sampling methods that underlie much of modern approximation theory. Note that this is only a flavor - approximation theory is a huge field with myriad applications and techniques.

One way that we can try to make a better approximating polynomial for a function is to sample more points near the edge - this should allow us to control the bad behavior that seems to occur there. But how should we pick the points? For various reasons, it turns out that a natural choice for sampling nodes (that is, the x-values that we'll use to sample the function we're trying to approximate) is the so-called **Chebyshev nodes**, which correspond to the x coordinates of equally spaced points on a unit circle.

```
n = 11;
nodes = cos(pi/n*((1:n)-.5));
X = -1:.01:1;
f = @(x) sqrt(1 - x.^2)
scatter(nodes, zeros(length(nodes),1), 400)
hold on
scatter(nodes, f(nodes), 400)
plot(X, f(X))
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
```

The following code will compare high degree approximations of f using equally spaced and Chebyshev nodes

```
X = -1:.01:1;
f = @(x) 1./(1 + 25*x.^2);

n = 11;
nodes = cos(pi/n*((1:n)-.5));
m = 21;
nodes2 = cos(pi/m*((1:m)-.5));



Q = -1:.2:1;
QQ = -1:.1:1;

l = polyfit(Q, f(Q), length(Q)-1);
p = polyfit(nodes, f(nodes), n-1);

ll = polyfit(QQ, f(QQ), length(QQ)-1);
pp = polyfit(nodes2, f(nodes2), m-1);

subplot(2, 2, 1)
scatter(Q, f(Q), 400)
```

```
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
axis([-1 1 -1 1])
title("Poly.␣intepolation␣on␣equidistant␣nodes,␣n␣=␣11")
hold on
plot(X, f(X))
plot(X, polyval(l,X))
hold off
subplot(2, 2, 2)
scatter(nodes, f(nodes), 400)
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
axis([-1 1 -1 1])
hold on
plot(X, f(X))
plot(X, polyval(p,X))
title("Poly.␣interpolation␣on␣Chebyshev␣nodes,␣n␣=␣11")
subplot(2, 2, 3)
scatter(QQ, f(QQ), 400)
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
title("Poly.␣intepolation␣on␣equidistant␣nodes,␣n␣=␣21")
axis([-1 1 -1 1])
hold on
plot(X, f(X))
plot(X, polyval(ll,X))
hold off
subplot(2, 2, 4)
scatter(nodes2, f(nodes2), 400)
set(gca,'xaxislocation','origin')
set(gca, 'yaxislocation','origin')
axis([-1 1 -1 1])
hold on
plot(X, f(X))
plot(X, polyval(pp,X))
title("Poly.␣interpolation␣on␣Chebyshev␣nodes,␣n␣=␣21")
hold off
```

5.5 Spline interpolation

Instead of trying to force a high degree polynomial to fit through many points, if we want an interpolating function that has smooth properties, we could consider a piecewise approach instead. That is, instead of a single formula that interpolates all of the points, we'll construct small segments that join together into a smooth(ish) function that changes definition as we move down the set of data points. Piecewise interpolation is usually done in early math classes with a ruler connecting points with lines - a more sophisticated name for this is linear interpolation. Because a given line only has two parameters (slope and intercept), the best we can expect is to meet two conditions - that is, the line passes through each of two points.

This is reflective of a general principle in mathematics that gets used but not often mentioned throughout college level courses - to meet one condition, an equation or set of equations needs one free parameter. In this case, since we're working with simultaneous equations, essentially we're using the invertible matrix theorem from linear algebra to guarantee a unique solutions.

What we're going to present is a method that breaks a set of data points into consecutive groups of two points. We'll need an interpolating function piece with at least two free parameters that can be chosen. Since we want the curve to be differentiable, we'll need each segment to meet the next with the same slope - that is, we need the first derivatives to match every time we change segments. With $n+1$ points, that means we have $n-1$ points that are connections (that is, not the endpoints of the data set). Since we're dictating a condition on derivatives, we know that we'll need at least three parameters. In order to make the changes between the segments curve naturally (which is important in many physical applications), we also want the second derivatives to match. (This is essentially dictating that the curvatures of the segments match, or more geometrically that there is an osculating circle where two pieces meet.) So each segment is actually dealing with four restrictions, and thus we need a function with four free parameters - a cubic polynomial fits the bill!

Where are we at? Suppose that we have $n+1$ points to interpolate. Then there are n segments, which requires n functions. Each segment has an associated cubic polynomial with 4 parameters, so there are $4n$ parameters available. How many restrictions are there?

- Two points to interpolate for each segment, so $2n$ conditions.
- $n-1$ points at which derivatives match.
- $n-1$ points at which second derivatives match.

So far, we have $4n$ parameters and $2n+n-1+n-1=4n-2$ conditions. Where are the last two conditions going to come from? The endpoints! Since we have two available conditions to impose, we'll get a nice square system if we impose them on the endpoints. There are a couple of common choices for how to do this. The first is called a **natural spline**, which is the case where the function has no curvature at the endpoints - that is, it's locally linear:

$$f''(x_1)=f''(x_{n+1})=0.$$

In some physical applications, the designer of the spline might want to have prescribed slopes at the endpoints instead. That is,

$$f'(x_1)=c_1, \qquad f'(x_{n+1})=c_2.$$

Let's take a look at a small example to see how spline fitting works in practice. Suppose we're given three points, $(1,2),(2,4),(3,1)$. Note that these are arranged in order of ascending x values. For each pair of points, we need a cubic spline - that is functions

$$\begin{aligned} f_1(x)&=a_1x^3+b_1x^2+c_1x+d_1,\\ f_2(x)&=a_2x^3+b_2x^2+c_2x+d_2. \end{aligned}$$

We'll organize the equations as above. First, we have the interpolating conditions (that is, the functions need to pass through the points.)

$$\begin{aligned} a_1+b_1+c_1+d_1&=2\\ 8a_1+4b_1+2c_1+d_1&=4\\ 8a_2+4b_2+2c_2+d_2&=4\\ 27a_2+9b_2+3c_2+d_2&=1 \end{aligned}$$

Second, we want first derivatives to match at the transition point at $(2,4)$.

$$12a_1+4b_1+c_1=12a_2+4b_2+c_2$$

Third, we want second derivatives to match at $(2, 4)$.

$$12a_1 + 2b_1 = 12a_2 + 2b_2$$

Finally, we impose the endpoint condition for a free spline:

$$6a_1 + b_1 = 0$$
$$18a_2 + 2b_2 = 0$$

There are various algebraic approaches to solving splines that we will not consider here. Instead, we will try to organize these equations into a matrix form.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 8 & 4 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 4 & 2 & 1 \\ 0 & 0 & 0 & 0 & 27 & 9 & 3 & 1 \\ 12 & 4 & 1 & 0 & -12 & -4 & -1 & 0 \\ 12 & 2 & 0 & 0 & -12 & -2 & 0 & 0 \\ 6 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 18 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 4 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

At this point, we can solve the system using octave and row reduction. As long as there are no three consecutive points that are collinear, this system will have a unique solution (that is, the coefficient matrix is invertible).

```
system =
    [1,1,1,1,0,0,0,0,2;8,4,2,1,0,0,0,0,4;0,0,0,0,8,4,2,1,4;0,0,0,0,27,9,3,1,1;12,4,1,0,-
coeefs = rref(system)(:,9) #this row reduces the system and
    takes the answer column, which is column 9
f1 = coeefs(1:4)';␣#these␣take␣the␣coefficients␣associated␣
    with␣the␣first␣and␣second␣spline
f2␣=␣coeefs(5:8)'; #and put the polynomials in vector form
X1 = 1:.01:2;
X2 = 2:.01:3;
Y1 = polyval(f1, X1);#evaluates each spline segment on the
    correct domain
Y2 = polyval(f2, X2);
plot(X1, Y1, 'r', X2, Y2, 'b')
hold on
scatter([1,2,3],[2,4,1],400) #plots the original points.
```

Next is a quick code demonstration of the addition of one more points, say at $(4, 10)$. You should look for the pattern in the equation sets, as this should give a clue about how to implement a general routine. By way of comparison, we will also plot the unique cubic that passes through the four points. You should note that the purple graph makes wider oscillations between the points than the multi-colored spline fit.

```
system = [1,1,1,1,0,0,0,0,0,0,0,0,2;
          8,4,2,1,0,0,0,0,0,0,0,0,4;
          0,0,0,0,8,4,2,1,0,0,0,0,4;
          0,0,0,0,27,9,3,1,0,0,0,0,1;
          0,0,0,0,0,0,0,0,27,9,3,1,1;
          0,0,0,0,0,0,0,0,64, 16, 4, 1, 10;
          12,4,1,0,-12,-4,-1,0,0,0,0,0,0;
          0,0,0,0,27,6,1,0,-27,-6,-1,0,0;
          12,2,0,0,-12,-2,0,0,0,0,0,0,0;
```

```
           0,0,0,0,18, 2, 0,0, -18, -2, 0,0,0;
           6,2,0,0,0,0,0,0,0,0,0,0,0;
           0,0,0,0,0,0,0,0,24,2,0,0,0];
coeefs = rref(system)(:,13) #this row reduces the system and
    takes the answer column, which is column 9
f1 = coeefs(1:4)';␣#these␣take␣the␣coefficients␣associated␣
    with␣the␣first␣and␣second␣spline
f2␣=␣coeefs(5:8)'; #and put the polynomials in vector form
f3 = coeefs(9:12)';
X1␣=␣1:.01:2;
X2␣=␣2:.01:3;
X3␣=␣3:.01:4;
Y1␣=␣polyval(f1,␣X1);#evaluates␣each␣spline␣segment␣on␣the␣
    correct␣domain
Y2␣=␣polyval(f2,␣X2);
Y3␣=␣polyval(f3,X3);

X␣=␣1:.01:4;
f␣=␣polyfit([1,2,3,4],[2,4,1,10],3);
Y␣=␣polyval(f,X);

plot(X1,␣Y1,␣X2,␣Y2,␣X3,␣Y3,␣X,␣Y)
hold␣on
scatter([1,2,3,4],[2,4,1,10],400)␣#plots␣the␣original␣points.
```

6 Numerical Integration

6.1 Integration review

Before we begin looking at numerical calculus, it is useful to recall some of the basic notions. In particular, we'll be reexamining the introduction to calculus most students see in the second semester of the course.

Here is a question - what is $\int_0^1 x^2\,dx$? Most people that have some experience with calculus will perform the following operation:

$$\int_0^1 x^2\,dx = \frac{1}{3}x^3|_0^1 = \frac{1}{3}.$$

However, this computation both misses the point of what the integral represents and uses a technique that will largely be unavailable in practice.

As to the first point, the **definite integral** represents a measurement of the **signed area** between a graph and the x-axis. We say signed area because area above the axis and area below the axis are considered to have opposite signs. Again, a definite integral is an area. Below, we plot the region corresponding to $\int_0^1 x^2\,dx$.

```
X = 0:.01:1;
X1 = -1:.01:2;
Y = X.^2;
Y1 = X1.^2;
area(X, Y, 'FaceColor', "red")
hold on
plot(X1, Y1)
```

Now, the area of this region is certainly $\frac{1}{3}$. To arrive at that conclusion, we used one of the most important theorems of continuous mathematics, the

fundamental theorem of calculus, which gives a connection between definite integrals, that is the signed area under a curve, with indefinite integrals, that is antiderivatives.

Theorem 6.1 Fundamental theorem of calculus, part II. *Suppose that f is a function on an interval $[a, b]$ with an antiderivative F such that $F'(x) = f(x)$ for all $x \in [a, b]$. If f is Riemann integrable, then*

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

One salient assumption present in the fundamental theorem of calculus is the existence of an antiderivative. Unfortunately, there are many functions that do not possess a (closed form) antiderivative, including some of the most useful functions in practice. For example, the **normal distribution** from statistics is essentially defined by the function $f(x) = e^{-x^2}$. A typical problem might wish to compute an integral like $\int_{-1}^{2} e^{-x^2}\,dx$, which is a simple area contained under a very nice curve, as shown below.

```
X = -4:.01:4;
XX = -1:.01:2;
plot(X, exp(-X.^2))
hold on
area(XX, exp(-XX.^2))
```

However, the fundamental theorem cannot be used to compute the area indicated by the definite integral because the function e^{-x^2} has no closed form antiderivative. So we need to approach the area finding problem with techniques related to the definition of definite integrals, which consist of breaking the area under functions up into approximating rectangles and then making the rectangles uniformly smaller in width.

```
X = 0:.01:1;
Y = X.^2;
rectangle("Position", [0,0, .5, .25], "facecolor","red",
    "facealpha",.1)
hold on
rectangle("Position", [.5,0, .5, 1], "facecolor","red",
    "facealpha",.1)
plot(X,Y)
area(X, Y, "facecolor", [0 0.6 .9])
plot(.5*ones(11,1), 0:.1:1, "k")
```

The **Riemann sum** that approximates the signed area under f on $[a, b]$ is given by the following formula. Let n be the number of approximating rectangles, and the width of each rectangle be $\Delta x = \frac{b-a}{n}$. Then we can define a partition of $[a, b]$ by $x_0 = a$, $x_i = x_0 + i\Delta x$, and $x_n = b$. On each subinterval $[x_i, x_{i+1}]$, we choose a point x_i^*. Then the area under f can be approximated by the expression

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} f(x_i^*)\Delta x.$$

Those functions for which $\lim_{n\to\infty} \sum_{i=0}^{n-1} f(x_i^*)\Delta x$ converges are called **Riemann integrable**.

Note, it need not be the case that the rectangles have equal width, which is an assumption made here to simplify the presentation.

6.2 The trapezoid rule

An immediate observation of a Riemann sum approximation for a definite integral might lead you to conclude that other shapes might provide more accurate approximations than rectangles. An easy shape to work with in this context is the trapezoid - it has a simple formula for area and allows us to avoid having to choose random points inside the subintervals. Compare the following pictures.

```
f = @(x) x.^2;
X = 0:.01:6;
Xi = linspace(0,6,4);
Y = X.^2;
for i = 1:(length(Xi)-1)
    rectangle("Position", [Xi(i),0, Xi(i+1) - Xi(i),
        f(Xi(i+1))], "facecolor",[0, .8, .8])
endfor
hold on
area(X,Y, "facecolor", [0, .6, .9])
for i = 1:length(Xi)
    plot([Xi(i), Xi(i)], [0, f(Xi(i))], 'k')
endfor
```

```
f = @(x) x.^2
X = 0:.01:6;
Xi = linspace(0,6,4);
Y = f(X);
Yi = interp1(X, Y, Xi);
plot(X,Y)
hold on
area(Xi, Yi, "facecolor", [0 .8 .8])
area(X,Y, "facecolor", [0, .6, .9])
for i = 1:length(Xi)
    plot([Xi(i), Xi(i)], [0, f(Xi(i))], 'k')
endfor
```

While the example might seem to be artificially chosen to make the trapezoids significantly more accurate than the rectangles, in fact, the vast majority of graphs of interest will look like the pictures above for small enough subintervals. This motivates the development of the **trapezoid rule** for approximating a definite integral.

Recall that the area of a trapezoid with height h and base widths b_1, b_2 is given by the formula

$$A = \frac{b_1 + b_2}{2} h.$$

Suppose that f is a function defined on the interval $I = [a, b]$ and let $x_0, \ldots, x_n$ be a uniform partition of I with subinterval width $\Delta x = \frac{b-a}{n}$. Consider the subinterval $[x_i, x_{i+1}]$. Then

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{f(x_i) + f(x_{i+1})}{2} \Delta x.$$

Thus,

$$\int_a^b f(x)\,dx = \sum_i \int_{x_i}^{x_{i+1}} f(x)\,dx$$

$$\approx \sum_i \frac{f(x_i)+f(x_{i+1})}{2}\Delta x$$
$$= \frac{\Delta x}{2}\left(f(x_0)+2\sum_{i=2}^{n-1} f(x_i)+f(x_n)\right)$$
$$= \frac{b-a}{2n}\left(f(x_0)+2\sum_{i=2}^{n-1} f(x_i)+f(x_n)\right)$$

and we define the n segement trapezoid approximation to the area under f by

$$T_n(f,[a,b]) = \frac{b-a}{2n}\left(f(x_0)+2\sum_{i=2}^{n-1} f(x_i)+f(x_n)\right).$$

Let's use the trapezoid rule to approximate the integral indicated above - $\int_0^6 x^2\,dx$. We'll use three trapezoids as in the example picture.

```
a = 0;
b = 6;
n = 3;
X = linspace(a,b,n+1);
f = @(x) x.^2;
Y = f(X);
approxArea = ((b-a)/(2*n))*(Y(1) + 2*Y(2) + 2*Y(3) + Y(4))
F = @(x) 1/3*x.^3;
trueArea = F(b) - F(a)
error = (approxArea - trueArea) / trueArea
```

We have included the true result, since we can use the fundamental theorem in this case. The result of the approximation with just 3 trapezoids is a relative error of just 5 percent.

6.3 A special case of Richardson's extrapolation (optional)

This section will have a different flavor than most of the rest of the notes. Here, we'll see the "analysis" part of numerical analysis - that is, we're going to use theoretical ideas and estimates to improve the approximation given in the trapezoid formula. The idea is that the application of mathematical reasoning can lead to significant improvements in our naive formulations (a theme common in approximation theory).

We'll first recall the **triangle inequality**, which says that $|a+b| \leq |a|+|b|$. In fact, we can apply this to a sum of any finite length, by induction:

$$\left|\sum_i a_i\right| \leq \sum_i |a_i|.$$

Now, Riemann sums are finite sums, and so the triangle inequality applies.

$$\left|\sum_{i=0}^{n-1} f(x_i^*)\Delta x\right| \leq \sum_{i=0}^{n-1} |f(x_i^*)|\,\Delta x,$$

and when the limit of the Riemann sum exists as the number of rectangles tends to infinity (that is, whenever f is Riemann integrable), we get the integral version of the triangle inequality:

$$\left|\int_a^b f(x)\,dx\right| \leq \int_a^b |f(x)|\,dx.$$

This is a theorem in real analysis and will be used here without a formal proof beyond the sketch above.

Let $I = \int_a^b f(x)\,dx$. Let T_n represent the n segment trapezoid approximation of I. Let E_T be the error in the approximation - that is

$$E_T = I - T_n.$$

Our first goal is to measure how large the error E_T is expected to be in terms of the number of trapezoids n.

Theorem 6.2 *Let f be Riemann integrable on $[a, b]$. Then*

$$|E_T| \sim \frac{1}{n^2}.$$

Proof. Let $\Delta x = x_{i+1} - x_i = \frac{b-a}{n}$. We first analyze the error in the trapezoid approximation on a single interval. A u-substitution gives

$$\int_{x_i}^{x_{i+1}} f(x)\,dx = \int_0^{\Delta x} f(t + x_i)\,dt.$$

Using integration by parts twice, we get

Table 6.3

u	v
$f(t + x_i)$	1
$f'(t + x_i)$	$t + A$
$f''(t + x_i)$	$\frac{(t+A)^2}{2} + B$

where we forgo the usual choice of 0 as the integration constant (hence the A, B in the table), which gives the formula

$$\begin{aligned}
&\int_0^{\Delta x} f(t + x_i)\,dx \\
&= [(t + A)f(t + x_i)]_0^{\Delta x} \\
&- \left[\left(\frac{(t + A)^2}{2} + B\right) f'(t + x_i)\right]_0^{\Delta x} \\
&+ \int_0^{\Delta x} \left(\frac{(t + A)^2}{2} + B\right) f''(t + x_i)\,dx.
\end{aligned}$$

From this point, the idea is to choose values of A, B that force each term to play a certain role, with the goal of concentrating the error of the approximation in the integral term. First, we choose A so that the first term above is equal to the trapezoid area - that is, we want

$$(\Delta x + A)f(\Delta x + x_i) - Af(x_i) = \frac{f(x_{i+1}) + f(x_i)}{2}\Delta x.$$

Algebra shows that $A = \frac{-\Delta x}{2}$ solves the equation.

Now, we want to choose B so that the second term is zero. That is, we want

$$\begin{aligned}
&\left[\left(\frac{(t + A)^2}{2} + B\right) f'(t + x_i)\right]_0^{\Delta x} \\
&= \left(\frac{(\Delta x)^2}{8} + B\right) [f'(x_{i+1}) - f'(x_i)] = 0.
\end{aligned}$$

This obviously holds when $B = \frac{-(\Delta x)^2}{8}$.

We conclude that the error on the ith segment, denoted $E_T(i)$ is given by

$$E_T(i) = \int_0^{\Delta x} \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) f''(t + x_i)\, dt$$

Now, we can get the total error in the trapezoid approximation by adding each of the individual errors.

$$\begin{aligned} E_T &= \sum E_T(i) \\ &= \sum_{i=0}^{n-1} \int_0^{\Delta x} \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) f''(t + x_i)\, dt \\ &= \int_0^{\Delta x} \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \left(\sum_{i=0}^{n-1} f''(t + x_i) \right) dt \end{aligned}$$

For a well-behaved function f (the precise assumption is that f is C^2 on $[a, b]$),the second derivative is bounded on $[a, b]$ - that is, we assume that there exists a constant K so that $|f''(x)| \leq K$ for all $x \in [a, b]$. Then, using the triangle inequality, we derive the approximation

$$\begin{aligned} |E_T| &= \left| \int_0^{\Delta x} \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \left(\sum_{i=0}^{n-1} f''(t + x_i) \right) dt \right| \\ &\leq \int_0^{\Delta x} \left| \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \right| \left| \left(\sum_{i=0}^{n-1} f''(t + x_i) \right) \right| dt \\ &\leq \int_0^{\Delta x} \left| \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \right| \left| \left(\sum_{i=0}^{n-1} K \right) \right| dt \\ &\leq nK \int_0^{\Delta x} \left| \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \right| dt \end{aligned}$$

The function $g(t) = \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right)$ is a parabola that opens upwards with zeros at $t = 0$ and $t = h$, and so

$$\begin{aligned} &\int_0^{\Delta x} \left| \left(\frac{(t - \frac{\Delta x}{2})^2}{2} - \frac{(\Delta x)^2}{8} \right) \right| dt \\ &\int_0^{\Delta x} \frac{(\Delta x)^2}{8} - \left(\frac{(t - \frac{\Delta x}{2})^2}{2} \right) dt \\ &= \left[\frac{(\Delta x)^2}{8} t - \frac{(t - \frac{\Delta x}{2})^3}{6} \right]_0^{\Delta x} \\ &= \frac{(\Delta x)^3}{12} \end{aligned}$$

Putting this together with the previous computation, we get

$$|E_T| \leq nK \frac{(\Delta x)^3}{12} = nK \frac{(b - a)^3}{12n^3} = \frac{K(b - a)^3}{12n^2},$$

where K was an absolute bound for f'' on $[a, b]$, and in the worst case, we get $|E_T| \leq \frac{K(b-a)^3}{12} \frac{1}{n^2} = \frac{C}{n^2}$ - that is, the error is proportional to $\frac{1}{n^2}$, which
Under the assumption of worst case error and a reasonable function f, we
establishes the claim. ■

conclude that the total trapezoidal error E_T is proportional to $\frac{1}{n^2}$, or in other words that

$$E_T = \frac{C}{n^2}.$$

So how can we use this to build a better process? Note that for n segments, we can write

$$I = T_n + \frac{C}{n^2}$$

and likewise for $2n$ segments, we have

$$I = T_{2n} + \frac{C}{(2n)^2},$$

which is a system of simultaneous equations. We'll prepare to eliminate C.

$$I = T_n + \frac{C}{n^2}$$
$$I = T_{2n} + \frac{C}{4n^2}$$

$$n^2 I = n^2 T_n + C$$
$$4n^2 I = 4n^2 T_{2n} + C$$

$$3n^2 I = 4n^2 T_{2n} - n^2 T_n$$

$$I = T_{2n} + \frac{T_{2n} - T_n}{3}$$

Thus, we have what is known as a first order Richardson's extrapolation -

$$\int_a^b f(x)\,dx \approx T_{2n} + \frac{T_{2n} - T_n}{3}.$$

Let's see how it performs with our existing example.

```
a = 0;
b = 6;
n = 3;
X = linspace(a,b,n+1);
f = @(x) x.^2;
Y = f(X);
T3 = ((b-a)/(2*n))*(Y(1) + 2*Y(2) + 2*Y(3) + Y(4))
m = 6;
XX = linspace(a, b, m+1);
YY = f(XX);
T6 = ((b-a)/(2*m))*(YY(1) + 2*(YY(2)+ YY(3) + YY(4) + YY(5)
    + YY(6)) + YY(7));
R6 = T6 + (T6 - T3)/(3)
F = @(x) 1/3*x.^3;
trueArea = F(b) - F(a)
errorT = (T6 - trueArea) / trueArea
errorR = (R6 - trueArea) / trueArea
```

6.4 Simpson's 1/3 rule

An alternative to using trapezoids is to use polynomials to interpolate sample points. It turns out that using quadratic polynomials on equally spaced interpolation points gives a very nice formula. We'll begin with a single segment and approximate $\int_a^b f(x)\,dx$.

Recall that there is a unique parabola through any three points - we'll use the points $(a, f(a)), ((a+b)/2, f((a+b)/2), (b, f(b))$. We have several techniques available for finding such an interpolation - we'll derive ours using Newton polynomials. The Newton polynomial through our points is

$$f(x) = b_0 + b_1(x-a) + b_2(x-a)(x-(\frac{a+b}{2})),$$

where

$$b_0 = a, b_1 = \frac{f((a+b)/2) - f(a)}{(a+b)/2 - a}, b_2 = \frac{\frac{f(b)-f((a+b)/2)}{b-(a+b)/2} - \frac{f((a+b)/2)-f(a)}{(a+b)/2-a}}{b-a}$$

7 Introduction to Fourier Analysis

7.1 Review of linear algebra

A **vector space** over a scalar field F is a collection of vectors together with operations that make it possible to do algebra on that collection. In particular, a set of vectors V is a vector space under the operations of addition and scalar multiplication if the following axioms are satisfied:

1. associativity of addition: $u, v, w \in V \Rightarrow u + (v + w) = (u + v) + w$
2. commutativity of addition: $u, v \in V \Rightarrow u + v = v + u$
3. additive identity: there is an element 0, the zero vector, so that $v \in V \Rightarrow v + 0 = 0 + v = 0$
4. additive inverses: every vector v has an inverse $-v \Rightarrow v + (-v) = 0$
5. compatibility: $\alpha, \beta \in F, u \in V \Rightarrow (\alpha\beta)u = \alpha(\beta u)$
6. multiplicative identity: $1 \in F, u \in V \Rightarrow 1u = u$
7. distribution over vector addition: $\alpha \in F, u, v \in V \Rightarrow \alpha(u + v) = \alpha u + \alpha v$
8. distribution over field addition: $\alpha, \beta \in F, u \in V \Rightarrow (\alpha + \beta)u = \alpha u + \beta u$

The prototypical example of a vector space is n-dimensional **Euclidean space**, $\mathbb{R}^n$, our usual notion of vectors. However, many other sets of objects constitute vector spaces with the appropriate operations - for example, $C([0, 1])$, the space of continuous functions on the interval $[0, 1]$ is a vector space over $\mathbb{R}$ under addition of functions. We are building to understanding spaces of functions.

Notice that the defintion of vector spaces doesn't include any way to multiply vectors. One notion of vector multiplication that you've likely seen before on $\mathbb{R}^n$ is the **dot product** of vectors. Let $v = (v_1, \ldots, v_n), u = (u_1, \ldots, u_n) \in \mathbb{R}^n$. Then

$$u \cdot v = \sum_{i=1}^{n} v_i u_i.$$

One of the most useful characteristics of the dot product on $\mathbb{R}^n$ is that it allows a the definition of the angle between two vectors:

$$u \cdot v = \|u\| \, \|v\| \cos\theta$$

where θ is the angle between u and v. Notice that when the vectors are perpendicular, this implies that the dot product is 0. We can generalize this geometry to the setting of general vector spaces.

The dot product is an example of a more general kind of product called an **inner product** on a vector space. Let V be a vector space.

8 Code examples

8.1 Lab 1 - Introduction

```
#Symbolic math in Octave/Matlab --
#September 4, 2019
#Numerical Analysis

#to use symbolic math, we first need to load the symbolic
    package
#using the command pkg load symbolic
#once the package is installed and loaded, we can use the
    symbolic capabilities
#installed in Octave

syms x; #defines x as a symbolic variable
fun = sin(x); #defines the variable fun as a symbolic
    object sin(x)

#one useful command that works on symbolic objects is diff
#which is Octave's built in command for differentiation.

#diff(fun) takes the symbolic derivative of the symbolic
    function
#sin(x).
diff(fun);

#this can be stored as another variable
dfun = diff(fun);

#note that this object cannot be evaluted. If we want to
    turn it
#into a function, we can use the command function_handle.

dfun = function_handle(dfun);

dfun(pi/3) #evaluates the derivative at pi/3

#dfun is now a function that can be evaluated on scalars or
    arrays.
#it can also be plotted. the simplest possible plotting
    command is
#ezplot, which takes care of a lot of the mechanics for
    you. ezplot
#doesn't care if an expression is symbolic or a function
ezplot(dfun)

#if you want to plot multiple graphs on the same figure,
    you can use the
# hold command to keep the figure in place before the next
    plot
ezplot(fun)
hold on;
ezplot(dfun)
hold off;

#another useful mathematical command is factorial
factorial(3)

#putting this together, we might plot sin x against a Taylor
#polynomial
fun = sin(x);
T = x - x.^3/factorial(3) + x.^5/factorial(5) -
    x.^7/factorial(7);
ezplot(fun)
hold on;
```

```
#Loops and arrays
#Numerical Analysis
#9/5/2019
#Ryan Tully-Doyle

#This set of examples will focus on loops and arrays, which
    are structures
#that we will be using constantly.
#A for loop runs over an index that goes through a
    prescribed set of numbers.
#The formatting is slightly different than other languages,
    but powerful
#in a mathematics context.

#the following loop will run for values of i starting at 1
    and ending at 10.
#Each iteration will perform the same command.

for i = 1:10
  disp(i) #display the current value of i
end

#unlike other langauges, non-integer indicies can be used
    in octave/matlab.

for i = 1:.1:10 #starts at 1, counts by .1 until reaching 10
  disp(i) #display the current value of i
end

#it is useful to be able to exit a loop on a condition. in
    octave, this
#command is called break

for i=1:10
  disp(i)
  if i == 5
    printf("You have to stop now!\n") #\n tells printf to
        break the line
    break
  end
end

#Next, we'll look at how octave deals with arrays, or lists
    of numbers.
#Functions in octave by default can act on arrays or
    scalars without using loops

X = 12:17;
disp(X)

#you might wish to know how long an array is

length(X)

#you might like to know the largest element in an array

max(X)

#If you want to refer to a specific element in an array,
    you can extract it
#by invoking its index.
```

```
X(3) #calls the third entry of X

#You can take slices of arrays by using index ranges

X(3:5) #calls the third through fifth entry of X
```

9 Assignments

9.1 Assignment Set 1

```
#Assignment set 1
#Numerical Analysis
#September 5, 2019

#############################################
#Exercise 1
#############################################
#Useful commands:
#inline(), plot(), abs(), max()
#############################################
#A) Plot the functions
#f(x) = sin(x)
#T7(x) = x - x^3/3! + x^5/5! - x^7/7!
#on the same set of axes.
#
#B) Then, given the domain you've chosen, calculate the
    maximum error.
#
#C)Finally, find the largest interval you can so that the
    maximum absolute true
#error is less than 0.1.





#############################################
#Exercise 2
#############################################
#Useful commands:
#sym, syms, diff, factorial, function_handle, plot
#############################################
#A) Write a loop that constructs the degree 7 Taylor
    expansion of
#f(x) = sin(x)
#about the point a = 0.
#
#B) Rewrite your loop so that it can find the Taylor series
    of degree n about
#any center a for f(x) = sin(x).
#
#C) Rewrite your loop so that it can take an arbitrary
    function, produce the
#degree n Taylor polynomial, and plot the function and the
    Taylor polynomial
#near a.




###############################################
#Exercise 3 (optional)
###############################################
```

```
#Modify your answer to Exercise 2 so that given an error
    tolerance,
#the program produces the largest domain near a for which
    the maximum error is
#less than that tolerance. The output should be both the
    Taylor polynomial
#and a domain where the approximation is good.
```

9.2 Assignment Set 2

```
#Assignment set 2
#Numerical Analysis
#September 13, 2019

##############################################
#Exercise 1
##############################################
##############################################
#useful commands:
#for/endfor, while/endwhile, break, @, if/elseif/else/endif
##############################################
#A) Plot the function f(x) = x^6 - x - 1. Use your plot
#to construct reasonable brackets for the roots of f.
#
#B) Use the bisection method to compute the roots of f. Set
#your tolerance so that the approximation is good to six
#significant figures.






##############################################
#Exercise 2
##############################################
#Useful commands:
#function, return, for, while, break
##############################################
#Write a function that takes as input a function,
#a bracket [a,b], a tolerance, and a maximum number
#of iterations and produces an approximation to the
#root in the interval [a,b]. You should consider
#adding a logic check to make sure that a root really
#is inside the bracket before executing.




################################################
#Exercise 3 (optional)
################################################
#Rewrite the function in Exercise 2 so that the user
    specifies
#a desired number of significant figures instead of a
    tolerance.
```

9.3 Assignment set 3

```
################################################
#Exercise 1
################################################
#Find an approximation for the square root of 2 using
#the method of false position.


################################################
#Exercise 2
################################################
#Find an approximation for the square root of 2 using
#the Newton-Raphson method. Compare the number of steps and
#the running time with Exercise 1.

################################################
#Exercise 3
################################################
#Find a function for which your bisection solver runs
    faster than
#your false position solver. Explain what you think is
    happening.

################################################
#Exercise 4
################################################
#Find a function that gives you extremely slow convergence
#in Newton's method. Explain what you think is happening.
```

9.4 Assignment set 4

```
################################################
#Exercise 1
################################################
#Use Newtons' method to find the roots of
#f(x) = x^6 - 9x^3 + 2x - 10


################################################
#Exercise 2
################################################
#Repeat exercise 1 but using the secant method.

################################################
#Exercise 3
################################################
#Turn your secant and Newton's method routines into
    funtions.
#You should build in failure checks (that is, the x's get
    huge
#or the answers don't converge).


################################################
#Exercise 4
################################################
```

```
#Write a hybrid method that first tries the secant method
    to find a root
#and then uses the bisection method if the secant method
    fails.
```

9.5 Assignment set 5 (including challenge set)

```
################################################
#Exercise 1 (challenge)
################################################
#Implement synthetic division. Input should be an array of
#numbers representing the coefficients and one number
    representing
#the point to be evaluated. Output should be the quotient
    and
#the function value.


################################################
#Exercise 2 (challenge)
################################################
#Use the synthetic division method and Newton's method to
    factor the polynomial
#f(x) = x^3 - 2x^2 - 5x + 6 completely.

################################################
#Exercise 3
################################################
#Find the Lagrange interpolating polynomial for the points
#(2,8), (4,2), (8,0.125).
#Plot the polynomial and the function f(x) = 2^(5-x),
#which generates the points, on the same graph.
#Is the Lagrange polynomial a good interpolant?
#Is it a good extrapolant?


################################################
#Exercise 4
################################################
#Find the Lagrange interpolating function for the points
#(-3, 4/10), (-2, 4/5), (-1, 2), (0,4), (1, 2), (2, 4/5),
    (3, 4/10)
#These points come from the function f(x) = 1/(x^2 + 1).
#What happens? Is the Lagrange polynomial a good
    approximation
#for f? Why or why not?
```

9.6 Assignment set 6 - Newton polynomials

```
################################################
#Exercise 1
################################################
#Write a script that produces the coefficients to the Newton
#polynomial corresponding to the points (1,1), (2,3),
    (5,10), (6, -2)
#and then produces the polynomial. Verify that the
    polynomial is
```

```
#correct by plotting the points and the polynomial on the
    same axes.


###############################################
#Exercise 2
###############################################
#Write a function that takes a matrix containing n
#interpolation points and produces the corresponding Newton
#polynomial.
```

9.7 Assignment set 7 - Cubic splines

```
###############################################3
#Exercise 1
###############################################
#Compute the cubic spline function that interpolates the
    points
#(-1, 3), (0,1), (1, 3). Plot your function and the points.


###################################################
#Exercise 2
##################################################
#Extend your previous result by appending the point (2,0).
    Pay attention to the organization
#of the matrix that you use to find the spline coefficients.


#######################################################
#Exercise 3
########################################################
#Use array appending to build a script that will compute
    and plot the spline interpolation for
#an arbitrary number of points.

##########################################################
#Exercise 4
#########################################################
#Sample the function f(x) = 1/(x^2 + 16) with an odd number
    of equally spaced points.
#Compare the graph of the original function, the graph of
    the unique polynomial interpolating function through
#the sample points, and the spline interpolating function
    through the sample points.
```

9.8 Assignment 8 - Numerical integration

```
###########################################################
#Exercise 1
##############################################################
#Write a function that implements the n segment trapezoid
    rule
#for inputs consisting of an interval (a,b), a number of
    segments n,
#and an anonymous function f. Then wrap that function into
    a driver
```

```
#that takes the above inputs plus a tolerance and increases
    the
#number of segments until a given tolerance is met (in
    absolute
#relative error). Compute the area under the function
    e^(-x^2) from
# x=0 to x=2.

################################################################
#Exercise 2
##################################################################
#Write a function that implements Richardson's first order
#extrapolation of the trapezoid rule. Compare the speed of
#convergence with the trapezoid rule for the function x^2
    sin(2x)
#on the interval [0,10].

#################################################################
#Exercise 3
###################################################################
#Write a function that implements Simpson's 1/3 rule. Then
    create a
#driver function that takes the same inputs plus a
    tolerance and
#increases the number of segments until the tolerance is
    met (in
#absolute relative error).
```