

Numerical Methods

for Math 451

Dr. Randall Paul

Version 1.0
(Same as the text dated December 17, 2018)

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. The essence of the license is that

You are free:

- **to Share** to copy, distribute and transmit the work
- **to Remix** to adapt the work

Under the following conditions:

- **Attribution** You must attribute the work in the manner specified by the author (but not in any way that suggests that they endorse you or your use of the work). Please contact the author at randall.paul@oit.edu to determine how best to make any attribution.
- **Noncommercial** You may not use this work for commercial purposes.
- **Share Alike** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

With the understanding that:

- **Waiver** Any of the above conditions can be waived if you get permission from the copyright holder.
- **Public Domain** Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- **Other Rights** In no way are any of the following rights affected by the license:
 - ◊ Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
 - ◊ The author's moral rights;
 - ◊ Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- **Notice** For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page below.

To view a full copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

Contents

I	Problems in One Dimension	5
1	Precision and Error	7
1.1	Introduction	7
1.2	Python	9
1.2.1	Repeated commands and loops	10
1.2.2	Functions and Modules	13
1.3	Errors and Big ‘O’ Notation	15
1.4	Exercise Solutions and Problems	18
2	Zero Finding	21
2.1	Bisection	21
2.2	Programming the Bisection Method	23
2.3	Secant Lines	27
2.4	Programming the Secant Method	29
2.5	Newton’s Method	33
2.6	Programming Newton’s Method	37
2.7	Exercise Solutions and Problems	40
3	Taylor’s Theorem	45
3.1	Taylor Polynomials	46
3.2	Graphing in Python	48
3.3	Convergence of Newton’s Method	52
3.4	Derivative Estimates	54
3.5	Exercise Solutions and Problems	58
4	Numeric Integration	65
4.1	Rectangle Rule	66
4.2	Trapezoid Rule	69
4.3	Simpson’s Rule	71
4.4	Romberg Integration	75
4.5	Exercise Solutions and Problems	79
5	Initial Value Problems	87
5.1	Euler’s Method	88
5.2	Euler’s Method using Python	90

5.3	Taylor's Method	95
5.4	Runge-Kutta Methods	98
5.5	Exercise Solutions and Problems	101
II	Problems involving Linear Systems	109
6	Linear Systems: Elimination Methods	111
6.1	Naive Gaussian Elimination	112
6.1.1	Matrices in Python	114
6.1.2	Programming Elimination	116
6.2	Gaussian Elimination with Partial Pivoting	119
6.2.1	Programming Partial Pivoting	120
6.3	Ill-conditioned Matrices	123
6.4	Exercise Solutions and Problems	127
7	Linear Systems: Decomposition and Iteration	133
7.1	LU Factorization	133
7.1.1	Calculating the Factorization	134
7.1.2	Programming <i>LU</i> Factorization	135
7.2	Iterative Methods	137
7.2.1	Jacobi Iteration	138
7.2.2	Gauss-Seidel Iteration	140
7.2.3	Programming Jacobi Iteration	141
7.3	Exercise Solutions and Problems	141
8	Interpolation	147
8.1	Polynomial Curve Fitting	147
8.1.1	VanderMonte Method	148
8.1.2	Newton's Divided Difference Method	149
8.1.3	Programming Newton's Divided Difference	151
8.2	Splines	154
8.2.1	Linear Splines	154
8.2.2	Programming Linear Splines	155
8.2.3	Quadratic Splines	156
8.2.4	Programming Quadratic Splines	159
8.2.5	Natural Cubic Spline	163
8.3	Regression Curves	164
8.3.1	Linear Regression	165
8.3.2	Polynomial Regression	167
8.4	Exercise Solutions and Problems	167
A	Selected Proofs	175
A.1	Mean Value Theorem for Integrals	175
A.2	Taylor's Theorem	176

A.3	Improved Trapezoid Rule	177
A.4	Minimal Energy of the Natural Cubic Spline	179

Part I

Problems in One Dimension

Chapter 1

Precision and Error

1.1 Introduction

Numerical Analysis is the study of how to use the enormous memory and computational power of modern computers to solve mathematical problems. While modern computers are truly marvelous machines, it remains critical for the student to understand the mathematics behind the computation. You will do a number of fairly tedious computations in this course simply because this is the best way for the student to understand what the computer is doing.

Just as important is to understand the limitations of the computer. It is a powerful tool, but can certainly be misused. There is a tendency by the student to trust the output of a computer implicitly. One of the most important things you can learn in this course is that, under the right circumstances, a computer can produce highly inaccurate results. Or even occasionally, complete nonsense.

Consider the following example.

Example 1.1: Let $x = 0.2$. Subtract $3/16$ from x , then multiply the result by 16. What happens? What happens if we do this repeatedly?

On paper you should just get 0.2 back every time. This is because:

$$16 \left(0.2 - \frac{3}{16} \right) = 3.2 - 3 = 0.2$$

And using a simple calculator that's exactly what will happen. But if you do this on an expensive modern computer, something strange occurs.

```
In : 0.2 - 3/16
Out: 0.0125000000000000011
In : 16*(0.2 - 3/16)
Out: 0.200000000000000018
```

How can this be? What is this strange discrepancy? Disturbing as it is, you might comfort yourself with the thought that at least it's very small. But as we'll see in this course, small errors can become large errors.

If you repeat this process, for instance, then the error grows. That is, if you subtract $3/16$ from the result above, and then multiply by 16, you get:

Out: 0.200000000000000284

And if you repeat ten times, the error starts to get significant.

Out: 0.20012207031249988

The reason for this is that a modern computer does not store 0.2 exactly. Computers store numbers in *binary*, that is, base 2. Thus,

$$\begin{aligned} 0.2 = \frac{1}{5} &= \frac{1}{8} + \frac{1}{16} + \frac{1}{128} + \frac{1}{256} + \dots \\ &= (0.001100110011\dots)_2 \text{ (repeating)} \end{aligned}$$

In binary a seemingly very simple number like 0.2 has an infinite, repeating “decimal” expansion, similar to how in base 10, $1/9 = 0.1111\dots$. And while modern computers have enormous amounts of memory, that memory is still finite. The computer must cut off the sequence at some point in order to store it. As we’ll see in the next section, the computer happens to cut off this sequence between the two ‘0’s and the two ‘1’s, then *rounds up* the final digit from 0 to 1.

For the computer, then,

$$0.2 \approx (0.001100110011\dots001101)_2$$

Now, since $3/16 = (0.0011)_2$,

$$0.2 - \frac{3}{16} \approx (0.000000110011\dots001101)_2$$

Multiplying by 16 shifts the decimal point four places to the right. Hence,

$$16 * \left(0.2 - \frac{3}{16}\right) \approx (0.00110011\dots0011010000)_2$$

This effectively “clips off” the first four decimal digits from the computer’s expansion of 0.2 while adding four ‘0’s to the end. Repeating gives

$$16 * \left(16 * \left(0.2 - \frac{3}{16}\right) - \frac{3}{16}\right) \approx (0.0011\dots00110100000000)_2$$

Eventually,

$$16 * \left(\dots 16 * \left(16 * \left(0.2 - \frac{3}{16}\right) - \frac{3}{16}\right) \dots - \frac{3}{16}\right) \approx (0.010000\dots00000000)_2 = \frac{1}{4}$$

Repeating again results in a 1. Again results in a 13, and again a 205! We have moved very far from the obviously correct answer of 0.2.

There are two points I hope you take away from this example. You would do well to keep them in mind throughout this course.

1. A computer knows very few numbers **exactly**. This may not come as such a surprise for numbers like $\sqrt{2}$ or π , but it is true for even such a mundane number as 0.2.
2. Though we may begin with numbers that are accurate to many, many digits, this is no guarantee that our **results** will be accurate to many digits...or even that they will be accurate **at all**.

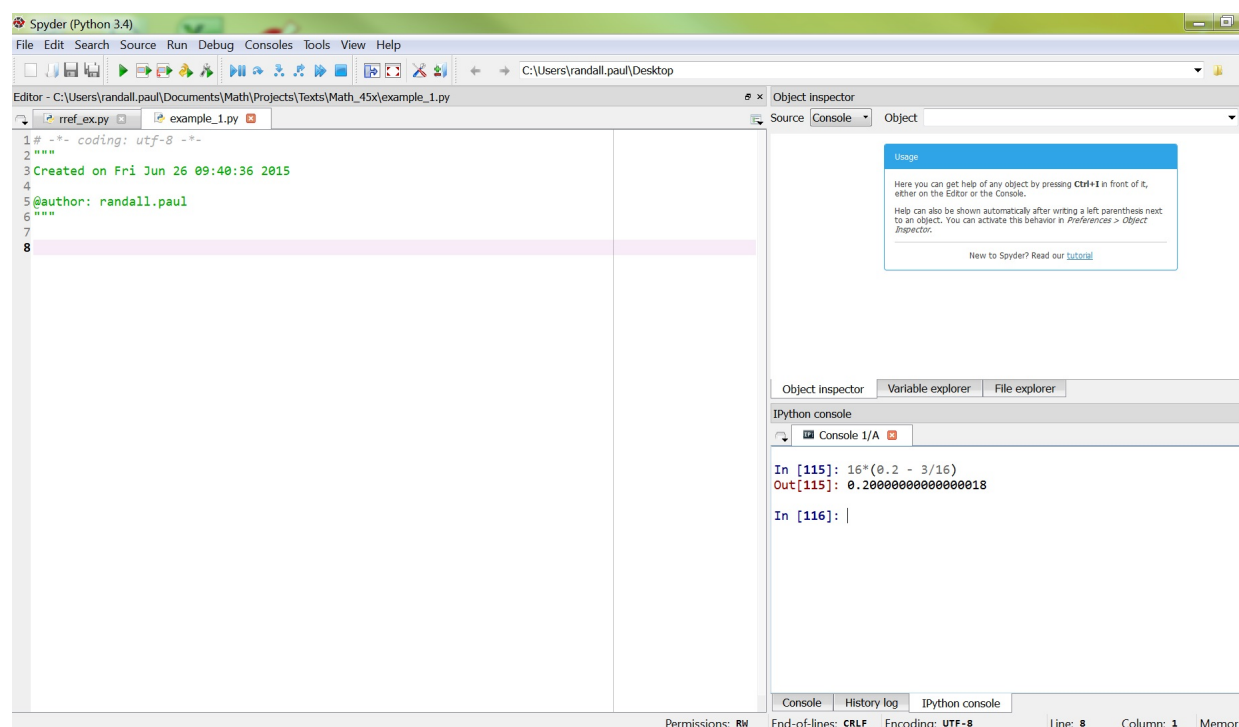
1.2 Python

For the programming portion of this class we will use the programming language *Python*. Python is an open source, scripting language which has been made much more powerful by various *modules* oriented towards mathematics. While the fact that Python is a scripting language means that it runs somewhat slower than formal compiled languages like *C* or *Fortran*, this is more than compensated for by the fact that it is easier to write in and, generally, easier to read. Code that is never written or debugged runs slowly regardless of the language.

Code presented in this text will be written as clearly as possible, even if that causes the code to be somewhat less efficient than it might otherwise be. Again the emphasis is for the student to be able to understand what the computer is doing. A dense, unreadable block of code—no matter how efficient—does not further that goal.

Examples and screenshots for this text will use the *Spyder* IDE with the *Anaconda* implementation of Python. It is not necessary that you use the same, but both are available free on-line (for Mac, Windows, or Linux). The standard installation of Anaconda also already contains all the mathematics modules that we'll need (e.g. *numpy*, *scipy*, *matplotlib*).

If you launch Spyder, something like the following window appears:



On the lower right are the *consoles*. These allow us to give commands directly to Python. Shown is the *IPython console*, which we will normally use. (In this text we will suppress the bracketed line numbers that label input and output from this console.)

Arithmetic expressions can be easily evaluated.

```
In : 3*(2 + 2)
Out: 12
```

Powers can be applied using the *double* `*`. An interesting feature of Python is it can handle integers as large as its computer memory can store. Thus if you really want the exact value of an enormous integer like 2^{1000} ,

```
In : 2**1000
Out:
1071508607186267320948425049060001810561404811705533607443750388370
3510511249361224931983788156958581275946729175531468251871452856923
1404359845775746985748039345677748242309854210746050623711418779541
8215304647498358194126739876755916554394607706291457119647768654216
7660429831652624386837205668069376
```

Python also supports some less well-known operations, such as *mod* (`%`) (divide and return the remainder) and *integer division* (`//`) (divide ignoring the remainder).

```
In : 16%5
Out: 1
In : 16//5
Out: 3
```

$16 = 3 * 5 + 1$, so 16 divided by 5 is 3 with a remainder of 1.

1.2.1 Repeated commands and loops

Let's return to the command we used in Example 1.1.

```
In : 16*(0.2 - 3/16)
Out: 0.200000000000000018
```

The example was to start with 0.2 and repeat this operation over and over. One way to do this is to first assign the value 0.2 to the variable `x`, then reassign to `x` the result when this operation is applied to `x`.

```
In : x = 0.2
In : x = 16*(x - 3/16)
In : print(x)
0.200000000000000018
```

We could type the last two commands again to repeat the calculation, but that is tiresome and unnecessary. One way to repeat a previous command is to use the ‘up arrow’ ↑ from the keyboard. If you hit it twice, then the command from two lines previously appears. You may edit it, or simply hit ‘return’ to repeat it.

```
In : x = 16*(x - 3/16)
In : print(x)
0.200000000000000284
```

This works well if you are repeating one or two earlier commands. However, if you are going to repeat a whole sequence of commands you should really use the *Editor*. (This is the big window on the left.) You write the sequence of commands that you want Python to execute, then save them to a file. (If you are working on a public machine, then you should save your files to a well-marked folder either on your **S:** drive or your personal thumbdrive.) You then *run* the file by selecting from the toolbar **Run - Run** or the green right-arrow **or** hitting **f5**. The effect is almost the same as if you had typed the commands into the console.

Example 1.2: Type the following into the editor:

```
x = 0.2
x = 16*(x - 3/16)
x = 16*(x - 3/16)
print(x)
```

Save as `example_1` and Run. A window giving you some options appears—just select ‘Run’ again. In the console you will see a `runfile` command with a long path ending with your file name. After that, the expected result:

```
0.200000000000000284
```

But what if we wanted to execute the command `x = 16*(x - 3/16)` ten times, rather than just twice? Or even one hundred times? In this case we need what is called a *for loop*.

Example 1.3: To see how loops work in Python, add a couple of lines to your file `example_1`:

```
for i in range(1,10):
    print(i)
```

Now Run the file. If you did everything correctly, then after `0.200000000000000284` the numbers from 1 to 9 will appear.

Note that the loop did not go to 10! This is a very smart and convenient way to do loops, but it takes some getting used to. In a Python **for** loop, the last number is **not** executed.

Two things are critically important here. First the colon at the end of the **for** line indicates that there is a block of commands that will be repeated. Second is the indentation before the **print(i)** command. Only indented commands will be repeated. While other languages use commands like **begin** and **end** or different types of parentheses to group commands, *indentation* is the principal means by which Python groups commands together. This is actually convenient and makes your code more readable, but you must get your indentation right if your code is to run correctly.

Example 1.4: So how would we execute the command from Example 1.1 ten times?

```
x = 0.2
for i in range(1,11):
    x = 16*(x - 3/16)
print(x)
```

This produces the output:

0.20012207031249988

Exercise 1.5: What would be the output if you wrote your code:

```
x = 0.2
for i in range(1,11):
    x = 16*(x - 3/16)
print(x)
```

Try to guess what will happen, then make the change and see if you're right.

Exercise 1.6: What would be the output if you wrote your code:

```
x = 0.2
for i in range(0,10):
    x = 16*(x - 3/16)
print(x)
```

Try to guess what will happen, then make the change and see if you're right.

1.2.2 Functions and Modules

It is inconvenient to have a separate file for each short snippet of code that we might want to use. For this reason (and others) we often define *functions*. A function is a set of commands with a name and possibly a set of *arguments*. When we wish to use this code, we call the function by its name and assign values to its arguments. Then the code executes, often producing output that can be assigned to other variables.

Example 1.7: Let's define a new function which performs some number of $x = 16*(x - 3/16)$ operations to 0.2. We'll call the function `strange02`. It will take as its argument `n`, which will be the number of times the operation is performed.

Edit the file `example_1` so you have:

```
def strange02(n):
    x = 0.2
    for i in range(0,n):
        x = 16*(x - 3/16)
    return(x)
```

Notice the indentation. There is a colon after `strange02(n)` and everything after that is indented. This means that all of these commands are part of the function. Our familiar $x = 16*(x - 3/16)$ operation is part of both the function and the loop, so it is indented **twice**.

After we run the file, nothing appears to happen. This is because we have not told Python to produce anything—we have just defined a function. To see the function in action, go to the console and write:

```
In : strange02(10)
```

Out will come the result of performing our operation ten times.

```
Out: 0.20012207031249988
```

If now you wrote:

```
In : y = strange02(5)
```

Nothing would happen. But if you then wrote:

```
In : print(y)
```

Python would produce:

```
0.200000000001164153
```

...which is the result of five operations.

Exercise 1.8: We saw in Exercise 1.1 that Python only keeps a finite sequence representation of 0.2, and that after a certain number of operations we get 0.25 rather than 0.2. Use `strange02` to find out how many operations that is. What does this tell us about how long the finite binary sequence that Python keeps for 0.2 is?

Besides the functions we may write for ourselves, there are vast libraries of specialized functions that already exist in collections called *modules*. Core Python does not automatically load these functions. If you want to use them, you have to tell Python to load them by *importing* them from the appropriate module. Core Python does not even understand such familiar mathematical functions as $\cos(x)$ or $\ln(x)$ —they’re defined in a module called (naturally enough) *math*.

Example 1.9: Use Python to evaluate $\cos(\pi/4)$.

If you go to the console and simply write...

```
In : cos(pi/4)
```

...you will get an error message ending in

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

If we then import the `cos` function from the `math` module... well things still don’t quite work.

```
In : from math import cos
```

```
In : cos(pi/4)
```

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

Now the constant `pi` isn’t defined. We could import `pi` as well, but it’s more convenient (if inefficient) to simply load the entire `math` module using the ‘wildcard’ `*`.

```
In : from math import *
```

```
In : cos(pi/4)
```

```
Out: 0.7071067811865476
```

Example 1.10: Use Python to calculate the *determinant* and *inverse* of the matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

We will use matrices extensively in this course. However, Python does not automatically load functions on matrices like `det` or `inv`. In fact, Python does not load the data structure “matrix” at all unless you tell it to. If you write:

```
In : y = matrix([[1,2],[3,4]])
```

Python will again return an error saying, basically, that it doesn’t know what you mean by `matrix`. However if you first load `matrix` from the module `scipy`, then all is well.

```
In : from scipy import matrix
In : y = matrix([[1,2],[3,4]])
In : print(y)
[[1 2]
 [3 4]]
```

Of course Python still doesn’t know the functions for determinants or matrix inverses unless you load them as well. Again it’s convenient to simply load the entire linear algebra portion of `scipy` (this is a *submodule* of `scipy` called `scipy.linalg`.)

```
In : from scipy.linalg import *
```

This imports **everything** in `scipy.linalg`. Then, say, to calculate the *determinant* of the matrix `y`, we write:

```
In : det(y)
Out: -2.0
```

```
In : inv(y)
Out:
array([[ -2. ,  1. ],
       [ 1.5, -0.5]])
```

1.3 Errors and Big ‘O’ Notation

The principal objective of this course is to estimate mathematical quantities. Our estimate is useless unless we have some idea of how well it approximates the quantity in question. Often we will compare different methods of estimation by applying them to problems where we already know the solution.

The *raw error* is the difference between our estimate and the true (exact) solution. If we know the true solution, then we can calculate the error directly. Even when we do not know the true solution, we can often estimate the error.

The raw error is not a very meaningful number by itself. An estimate with a raw error of 1.0 would be very bad if the quantity you were estimating was something on the order of 2.0. It would be excellent, on the other hand, if you were estimating a quantity on the order of 10^6 . For this reason we will often consider the *percentage error*:

$$\text{percent error} = 100 \cdot \left| \frac{x_e - x_s}{x_s} \right|$$

where x_e is our estimate and x_s is the exact solution.

Example 1.11: Let's return yet again to Example 1.1. If we apply the operation $x = 16 * (x - 3/16)$ thirteen times to $x = 0.2$ on a computer, what is the raw and percentage error of the result?

We saw that our computer produces $x = 0.25$. Using simple algebra, on the other hand, we saw that we **should** just get $x = 0.2$ back again no matter how many times we do the operation. Thus, $x_e = 0.25$ while $x_s = 0.2$, and

$$\text{raw error} = x_e - x_s = 0.05$$

$$\text{percent error} = 100 \cdot \left| \frac{0.25 - 0.2}{0.2} \right| = 25\%$$

Under some circumstance 0.05 might be quite a “small” error. However in this situation it comprises fully 25% of the correct answer, and so it quite a “large” error with respect to 0.2.

Often the error will depend on a some parameter from our numerical method. Sometimes this is an integer, such as the number of iterations or steps in the method. Sometimes it is a small number describing the length of a step or the distance between points in a grid. Usually the exact relationship between the parameter and the error is very complex. We can, though, often give a rough estimate of this relationship by using what is call *Big “O” notation*.

We say $f(x)$ is $O(b(x))$ (pronounced “ f is *on the order of* b ”) if there is some positive constant, K , so that

$$|f(x)| \leq K|b(x)|$$

This notation is useful in a variety of contexts, but for our purposes $f(x)$ will be an error depending in some complicated way on the parameter x . $b(x)$ will be some much simpler “bounding” function.

Just to get the idea, let's do a first example that does not have anything to do with errors or parameters.

Example 1.12: Let $d(r)$ be the number of points with integer coefficients within a circle of radius r centered at the origin. Let's also restrict $r \geq 1$. Use big O notation to estimate the size of $d(r)$.

We can see that $d(1) = 1$ as the only point with integer coefficients within the circle of radius 1 is the origin, $(0, 0)$. A slightly bigger circle, however, will also include the “compass points” $(0, 1), (1, 0), (0, -1), (-1, 0)$, so $d(1.1) = 5$. Once r becomes greater than $\sqrt{2}$, the circle also contains the points $(1, 1), (-1, 1), (1, -1), (-1, -1)$, so $d(1.5) = 9$.

While the exact relationship between r and $d(r)$ is fiendishly complex, we can see that $d(r)$ is going to be approximately proportional to the area of a circle of radius r , which is itself proportional to r^2 . The bigger the circle, the better this approximation gets.

r	$d(r)$	$d(r)/r^2$
1.0	1	1.0000
1.1	5	4.1322
1.5	9	4.0000
5.1	89	3.4218
10.1	325	3.1860
100.1	31473	3.1410

It's clear from the table, and in fact can be shown rigorously that

$$\lim_{r \rightarrow \infty} \frac{d(r)}{r^2} = \pi$$

For large values of r , then,

$$d(r) \approx \pi r^2 \quad \Rightarrow \quad d(r) \leq K r^2$$

for a constant K just a little larger than π . Allowing for smaller values of r requires K to be a little bigger, but it can be shown that, **for any** $r \geq 1$,

$$d(r) \leq 5r^2$$

Thus, $d(r)$ is $O(r^2)$.

Example 1.13: Use big O notation to describe the behavior of the function:

$$f(\theta) = \theta - \sin(\theta)$$

as θ becomes small. Make a table supporting your claim.

If we substitute some values of θ that approach zero, we notice that for each order of magnitude θ decreases, $f(\theta)$ drops by **three** orders of magnitude. This suggests that f is on the order of θ^3 . Making a table similar to the previous example,

θ	$\theta - \sin(\theta)$	$f(\theta)/\theta^3$
1.000	1.5853e-01	0.15852902
0.100	1.6658e-04	0.16658335
0.010	1.6667e-07	0.16666583
0.001	1.6667e-10	0.16666666

From the table it appears that:

$$\lim_{\theta \rightarrow 0} \frac{\theta - \sin(\theta)}{\theta^3} \approx 0.1666 \dots = \frac{1}{6} \quad \Rightarrow \quad |\theta - \sin(\theta)| \leq K |\theta|^3$$

for K around $\frac{1}{6}$. Thus we would guess that $\theta - \sin(\theta)$ is $O(\theta^3)$.

Another way to think of this is that,

$$\sin(\theta) = \theta + \text{Err}(\theta)$$

where $\text{Err}(\theta)$ is an error term that is on the order of θ^3 . We will often express this relationship using big O notation as:

$$\sin(\theta) = \theta + O(\theta^3)$$

For small values of θ , θ^3 will be very small indeed. Therefore the error will also be very small—as long as the constant K is not very large (which we can see it is not). This is the basis for the “small angle approximation” of $\sin(\theta)$ which states that:

$$\sin(\theta) \approx \theta \quad (\text{for small } \theta)$$

We will discuss these ideas much more rigorously in Chapter 3, when we consider *Taylor’s Theorem*.

1.4 Exercise Solutions and Problems

Solution to Exercise 1.5:

Since the command `print(x)` is **indented**, it is part of the loop. That means it is executed ten times, just like the operation. The output will be:

```
0.200000000000000018
0.2000000000000000284
0.20000000000000004547
0.20000000000000007276
0.200000000001164153
0.200000000018626451
0.20000000298023224
0.20000004768371582
0.20000076293945312
0.20001220703125
```

Solution to Exercise 1.6:

The variable `i` goes from 0 to 9, rather than 1 to 10 (as in Example 1.3), but that doesn’t matter as `i` does not appear explicitly anywhere inside the loop. The operation is executed ten times either way, so the output is the same:

```
0.2001220703125
```

Solution to Exercise 1.8:

By experimenting we can establish that

```
In : strange02(13)
Out: 0.25
```

So thirteen applications of the operation reduced the computer’s binary sequence representation of 0.2 to $(0.01)_2$. Each application of the operation removed **four** binary digits from the front of the sequence. Therefore thirteen applications removed $13 * 4 = 52$ digits, leaving two. Hence Python originally stored 0.2 as a binary sequence of **54** digits.

Problem 1.1: The binary expansion of $\frac{1}{7}$ is $(0.001001001\dots)_2$ (repeating).

- a) Assuming Python also stores $\frac{1}{7}$ with 54 digits, how many applications of the operation: $x = 8*(x - 1/8)$ would have to be applied before x was no longer a fraction? (Recall that $\frac{1}{8} = (0.001)_2$, while multiplying by 8 moves the binary “decimal point” three places to the right.)
- b) Verify your answer to part **a** by writing a short Python function called `strange17`, taking `n` as its argument, which applies this operation to $\frac{1}{7}$ `n` times and returns the result.

Problem 1.2: Use big O notation to describe the behavior of the function:

$$f(\theta) = 1 - \cos(\theta)$$

as θ becomes small. Make a table supporting your claim.

Chapter 2

Zero Finding

One of the most common tasks requiring a numerical method is that of solving an equation which has no analytic solution.

Example 2.1: Approximate the solution to the equation:

$$e^x = 2 - x$$

This equation has no analytic solution. We can apply the natural logarithm to both sides to obtain:

$$x = \ln(2 - x)$$

This does tell us that any solution would have to be less than 2, but it does not tell us what that solution is. Further, it's not immediately clear that there even **is** a solution.

We will express the problem of solving an equation as the equivalent problem of find a zero for a given function. Then Example 2.1 could be restated as:

$$f(x) = e^x + x - 2$$

Find a **zero** (that is, an x -intercept) for f .

First we note that $f(0) = -1$ while $f(2) = e^2$. Since f is a continuous function, $f(0) < 0$ and $f(2) > 0$, the **Intermediate Value Theorem** states that there must be an x_s between 0 and 2 where $f(x_s) = 0$. So we now know that there **is** a solution, but we still have no clear idea how to find it.

2.1 Bisection

Our first numerical method is really just a systematic variant of the “Guess-and-check method” that anyone with a calculator might apply. It is called the **Bisection Method**, and relies on the function being continuous, and that for two points a and b , $f(a)$ and $f(b)$ have different signs. This implies that a solution exists on some closed and bounded interval $[a, b]$. (We say such a zero has been *bracketed*.)

The method proceeds as follows:

Consider the midpoint of the interval, $x_m = (a+b)/2$. Now either $f(x_m) = 0$ (whereupon we are finished) or $f(x_m)$ has a different sign from either $f(a)$ or $f(b)$. Then a solution is bracketed by either the interval $[a, x_m]$ or $[x_m, b]$ (respectively). Since either of these intervals is half the length of the original, we have reduced the uncertainty of where the solution is (that is, the error) by half. We repeat the process on the new interval, reducing our error to one quarter the original length.

While this will almost never give an **exact** answer, eventually we will have confined the solution to such a small interval that we know it to as many decimal places as we desire. Further, that's usually all we can really hope to do on a computer since even numbers that we know "exactly" (like, for instance, 0.2) are really only stored approximately.

Example 2.2: Use the Bisection Method to approximate a solution to the equation

$$f(x) = e^x + x - 2$$

to three decimal places (that is, so that the raw error is less than 10^{-3}).

From our discussion after Example 2.1, we know we have bracketed a solution between 0 and 2. The midpoint of the interval $[0, 2]$ is 1. Evaluating $f(1) = e - 1 > 0$. Since $f(0) < 0$ we have now confined the solution to the interval $[0, 1]$. The midpoint of this new interval is 0.5 and $f(0.5) = e^{0.5} - 1.5 \approx 0.149 > 0$, so the solution is confined to $[0, 0.5]$.

If at each step our estimate of the solution is the midpoint, then the error is just half the width of the confining interval. We need to repeat this process until the error is less than 0.001.

n	[a, b]	width
0	[0.00000, 2.00000]	2.00000
1	[0.00000, 1.00000]	1.00000
2	[0.00000, 0.50000]	0.50000
3	[0.25000, 0.50000]	0.25000
4	[0.37500, 0.50000]	0.12500
5	[0.43750, 0.50000]	0.06250
6	[0.43750, 0.46875]	0.03125
7	[0.43750, 0.45312]	0.01562
8	[0.43750, 0.44531]	0.00781
9	[0.44141, 0.44531]	0.00391
10	[0.44141, 0.44336]	0.00195

We choose our estimated solution to be the midpoint of the final interval, so the error is half the width of the final interval.

Therefore $x_e \approx 0.4423$ and the raw error is less than $0.00195/2 = 0.000975 < 0.001$.

As a final check we can evaluate $f(0.4423) \approx -0.0012$. This is reasonably small, so it appears that x_e should in fact be fairly close to the actual solution x_s .

So, what can we say in general about the error for the bisection method? Well clearly it depends on how many times, **n**, we apply the method.

If we don't apply the method at all (so $n = 0$) and use the midpoint as our estimate, then

$$|\text{Error}| \leq \frac{|b-a|}{2}$$

Applying the method once and taking the midpoint of the resulting half-size interval results in an error half of that. Thus after n applications,

$$|\text{Error}(n)| \leq \frac{|b-a|}{2^{n+1}} \Rightarrow |\text{Error}(n)| \leq \left(\frac{|b-a|}{2}\right) 2^{-n}$$

Using big O notation, we say that the error is $O(2^{-n})$. (Here the constant K is just $|b-a|/2$.)

2.2 Programming the Bisection Method

Writing a Python program that performs the Bisection Method is actually a little bit involved, so we will work our way up to it. Let's start by defining our function $f(x)$ from Example 2.1. So open Spyder and type into the Editor window:

```
def f(x):  
    return exp(x) + x -2
```

Save it in new file called `zero_finding`. (Make sure you save it on your **S:** drive or a thumbdrive.) Run the file with the green arrow from the toolbar.

If you type into the IPython console, you should see:

```
In : f(0)  
Out: -1.0
```

```
In : f(2)  
Out: 7.3890560989306504
```

At first we'll use the up arrow (\uparrow) in the console to repeat commands. Later we'll turn this into a program.

Since we'll be changing the left and right endpoints to our interval, let's give them names and then use them to calculate the midpoint and f at the midpoint.

```
In : a = 0
```

```
In : b = 2
```

```
In : m = (a+b)/2
```

```
In : f(m)  
Out: 1.7182818284590451
```

This is $e - 1$ as we calculated earlier, but all we really care about is that it is **positive**. This tells me that I want to change the **right** endpoint to **m**, then use the up arrow to repeat those two earlier commands.

```
In : b = m
```

```
In : m = (a+b)/2
```

```
In : f(m)
```

```
Out: 0.14872127070012819
```

This is still positive, so repeat these three commands again.

```
In : b = m
```

```
In : m = (a+b)/2
```

```
In : f(m)
```

```
Out: -0.46597458331225861
```

Since $f(m)$ is now negative, we replace the left endpoint and repeat.

```
In : a = m
```

```
In : m = (a+b)/2
```

```
In : f(m)
```

```
Out: -0.17000858538179875
```

...and so we'd replace the left endpoint again. But this is getting tedious. How might we turn this into a program? Return to the Editor and **below** our definition of **f** write:

```
def bisection(a,b):  
    m = (a+b)/2  
    if f(m) > 0:  
        b = m  
    if f(m) < 0:  
        a = m  
    return a,b
```

Notice the `if` statements with a condition and a colon `:`. The statements indented after the `if` statement will only be executed if the condition is true. So `b` will be assigned the value of `m` **only if** `f(m)` is positive.

Run the file and return to the console.

```
In : bisectionce(0,2)
Out: (0,1.0)
```

```
In : bisectionce(0,1)
Out: (0,0.5)
```

Well that's better, but we're still having to copy our output from the first use of `bisectionce` into our second use by hand. Let's reset our endpoint variables, then use them when in `bisectionce`. In Python there's a slick way to do this using *tuples*. (These are just round parentheses.)

```
In : (a,b) = (0,2)

In : (a,b) = bisectionce(a,b)

In : print((a,b))
(0, 1.0)
```

The first line assigns 0 to `a` and 2 to `b`—all in one step! The second line assigns the first coordinate of the output of `bisectionce` to `a` and the second coordinate to `b`. It's all gotten a little bit confusing, so we can print out both endpoints at once with the third line. (Notice we used two sets of parentheses in the `print` statement—the outer ones enclosed the argument for the `print` function while the inner ones enclosed the tuple `(a,b)`.)

In fact having the `print` statement in there will be so useful we should add it to our program:

```
def bisectionce(a,b):
    m = (a+b)/2
    if f(m) > 0:
        b = m
    if f(m) < 0:
        a = m
    print((a,b))
    return a,b
```

Notice the `print` statement is not double indented—that would cause it to be part of the `if` statement, so it would only be executed when $f(m) < 0$. We want it to be executed every time.

Return to the console and execute `(a,b) = bisectionce(a,b)` a few times.

```
In : (a,b) = bisectionce(a,b)
(0, 1.0)
```

```
In : (a,b) = bisectionce(a,b)
(0, 0.5)
```

```
In : (a,b) = bisectionce(a,b)
(0.25, 0.5)
```

```
In : (a,b) = bisectionce(a,b)
(0.375, 0.5)
```

So finally let's write a new program that uses a `for` loop to execute `bisectionce` several times, prints the error, and then returns our estimate of the zero, x_e (the midpoint of the last interval). **Below `bisectionce`** write:

```
def bisectiontimes(a,b,n):
    for k in range(0,n):
        (a,b) = bisectionce(a,b)
    print('Error <= ', (b-a)/2)
    return (a+b)/2
```

Run and go to the console.

```
In : bisectiontimes(0,2,10)
(0, 1.0)
(0, 0.5)
(0.25, 0.5)
(0.375, 0.5)
(0.4375, 0.5)
(0.4375, 0.46875)
(0.4375, 0.453125)
(0.4375, 0.4453125)
(0.44140625, 0.4453125)
(0.44140625, 0.443359375)
Error <=  0.0009765625
Out: 0.4423828125
```

Though it now works well with **this** function on **this** interval, the program is far from finished. What if you gave the program values for **a** and **b** which did not bracket a zero? What if for some different function **f**, **f(a)** were positive while **f(b)** were negative? Then a zero would be bracketed, but the program would not function correctly. We will deal with these issues in the following exercises.

Exercise 2.3: Modify **bisectntimes** so that it tests whether the given values for **a** and **b** actually bracket a zero. If they do not, then the program should print an error message and stop. It should **not** enter the loop. Check that your program works by writing

```
In : bisectntimes(1,2,10)
Out: Zero is not bracketed by (1, 2)
```

Exercise 2.4: Modify the **if** statement in **bisectonce** so that **b** is replaced by **m** only if **f(m)** and **f(b)** have the **same sign**. Similarly modify the other **if** statement so that **a** is replaced by **m** only if **f(m)** and **f(a)** have the **same sign**. Check that your program works by changing **f** to **2-log(x)** and writing:

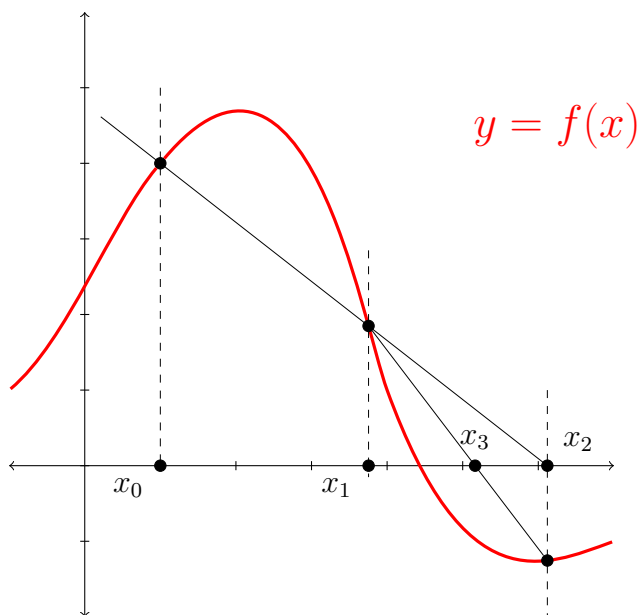
```
In : bisectntimes(7,8,5)
(7, 7.5)
(7.25, 7.5)
(7.375, 7.5)
(7.375, 7.4375)
(7.375, 7.40625)
Error <= 0.015625
Out: 7.390625
```

2.3 Secant Lines

The Bisection method has two big disadvantages and one big advantage. The first disadvantage is that you need to have bracketed a zero for the method to work. The second is the fact that it converges to the solution relatively slowly. The big advantage is that it does inevitably converge to a solution. We'll now discuss a method that is a sort of mirror image of the Bisection Method. This method does not require bracketing and converges very quickly to the solution...**when it converges at all**.

Like the Bisection Method, the Secant Method begins with two numbers, a and b . These two points do not, however, have to bracket a solution. A new number is found by drawing a line through the points $(a, f(a))$ and $(b, f(b))$ and finding the x -intercept, x_2 . A new line is then drawn through $(b, f(b))$ and $(x_2, f(x_2))$ whose x -intercept is x_3 . If we think of $x_0 = a$ and $x_1 = b$, we describe the process as follows:

Secant Method



The line through $(x_k, f(x_k))$ and $(x_{k-1}, f(x_{k-1}))$ has slope

$$m = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

and equation

$$y = mx + f(x_k) - mx_k$$

If we set $y = 0$ and solve, we will have the x -intercept which we call x_{k+1} ,

$$x_{k+1} = \frac{mx_k - f(x_k)}{m} = x_k - \frac{(x_k - x_{k-1})f(x_k)}{f(x_k) - f(x_{k-1})}$$

Example 2.5: Find to four decimal places a zero to the function $f(x) = e^x + x - 2$ (from Example 2.1). Use the Secant Method with $a = 0$ and $b = 2$.

The first iteration gives us x_2 ,

$$x_2 = x_1 - \frac{(x_1 - x_0)f(x_1)}{f(x_1) - f(x_0)} = 2 - \frac{(2 - 0)f(2)}{f(2) - f(0)} \approx 2 - \frac{14.77811}{8.38906} \approx 0.23841$$

The next gives us x_3 ,

$$x_3 = x_2 - \frac{(x_2 - x_1)f(x_2)}{f(x_2) - f(x_1)} \approx 0.23841 - \frac{0.86736}{-7.88143} \approx 0.34846$$

Putting the results into a table we can see that x_k jumps around a bit at first, then settles down. The change in x_k from one iteration to the next gets very small as the process converges to the actual solution.

For $x_e = 0.44285$ we have $f(x_e) \approx -1.13 \times 10^{-5}$. Subsequent iterations will make $f(x_k)$ smaller, but do not change the first five digits of x_e . We assume that x_e is accurate to at least 5 decimal places.

k	x_k	$ x_k - x_{k-1} $
0	0.0	N/A
1	2.0	2.0
2	0.23841	1.76159
3	0.34846	0.11005
4	0.44867	0.10021
5	0.44269	0.00598
6	0.44285	0.00017
7	0.44285	0.00000

The fact that $x_0 = 0$ and $x_1 = 2$ bracketed the solution was irrelevant. We see that the method works just as well if we take $x_0 = 1$ and $x_1 = 2$.

k	x_k	$ x_k - x_{k-1} $
0	1.0	N/A
1	2.0	1.0
2	0.69699	1.30301
3	0.55962	0.13737
4	0.45196	0.10767
5	0.44318	0.00878
6	0.44286	0.00032
7	0.44285	0.00000

If we look again at the recurrence relation for the Secant Method,

$$x_{k+1} = x_k - \frac{(x_k - x_{k-1})f(x_k)}{f(x_k) - f(x_{k-1})}$$

we can see how the method may fail. If $f(x_k) = f(x_{k-1})$ then the secant line is horizontal and thus never crosses the x -axis. Even if $f(x_k) \approx f(x_{k-1})$, then we are dividing by the very small number $f(x_k) - f(x_{k-1})$, and x_{k+1} may jump very far from x_k . In theory this jumping may cause the sequence $\{x_k\}$ to diverge.

Exercise 2.6: Find the first two iterations of the Secant Method applied to $f(x) = 2 - \ln(x)$ with $a = 1$ and $b = 2$.

2.4 Programming the Secant Method

Again before writing a formal program, let's start by performing the method from the console using the up arrow to repeat commands.

Let's work Example 2.5 with Python's help. We need to set a and b , then perform the first iteration.

```
In : a = 0
```

```
In : b = 2
```

```
In : x = b - ((b-a)*f(b))/(f(b)-f(a))
```

```
In : print(x)
0.238405844044
```

Now we need **b** to play the role of **a** and **x** to play the role of **b**. Then we use the up arrow to repeat the iteration.

```
In : a = b
```

```
In : b = x
```

```
In : x = b - ((b-a)*f(b))/(f(b)-f(a))
```

```
In : print(x)
0.348456492055
```

Now that we see what commands are to be repeated, it's relatively easy to write a function that does that. Open the file **zero_finding** in the Editor, and anywhere below the definition of **f** write:

```
def secant(a,b,n):
    for k in range(0,n):
        x = b - ((b-a)*f(b))/(f(b)-f(a))
        a = b
        b = x
        print(x)
    return x
```

Run the file, then in the console write: **secant(0,2,6)**

```
In : secant(0,2,6)
0.238405844044
0.348456492055
0.448667809859
0.44268777986
0.442854106034
0.442854401017
Out: 0.44285440101735246
```


So this is fine, but we'd like to improve it in a couple of ways. First from the tables above we note that it was illuminating to see the change from x_{k-1} to x_k . As this quantity became small, we could see that our method was converging.

Second, we don't really know (or particularly care) how many iterations the program should make. What we really want is for the program to continue working **until the change in x is very small**, however many iterations that may take. Rather than use a **for** loop which executes a set of commands a fixed number of times, we would like a **while** loop which executes a set of commands **for as long as some condition is satisfied**.

To see a **while** loop in action, write the following in the console:

```
In : x = 2

In : while x<2000:
...:     x = x**2
...:     print(x)
...:
4
16
256
65536
```

We start with a value of $x=2$ then square x repeatedly until the result is bigger than 2000. This turned out to take four executions of the loop, but we didn't need to know that when we wrote the code.

Let's edit our function `secant` so that instead of the number of iterations `n`, we take as our argument `tol`, which will be how small the change in x must be for us to decide the method has converged sufficiently.

```
def secant(a,b,tol):
    dx = abs(a-b)
    while dx > tol:
        x = b - ((b-a)*f(b))/(f(b)-f(a))
        a = b
        b = x
        dx = abs(a-b)
        print('x = ', x, 'change in x = ', dx)
    return x
```

Run the file and in the console write: `secant(0,2,1e-5)`

```

In : secant(0,2,1e-5)
x = 0.238405844044 change in x = 1.76159415596
x = 0.348456492055 change in x = 0.110050648011
x = 0.448667809859 change in x = 0.100211317804
x = 0.44268777986 change in x = 0.0059800299993
x = 0.442854106034 change in x = 0.000166326174526
x = 0.442854401017 change in x = 2.94983127758e-07
Out: 0.44285440101735246

```

The loop ran until the absolute value of the difference between x_k and x_{k-1} (which we're storing as the variable `dx`) was less than 10^{-5} . This turned out to be six iterations. If we only needed, say, three digits of accuracy we could write:

```

In : secant(0,2,0.001)
x = 0.238405844044 change in x = 1.76159415596
x = 0.348456492055 change in x = 0.110050648011
x = 0.448667809859 change in x = 0.100211317804
x = 0.44268777986 change in x = 0.0059800299993
x = 0.442854106034 change in x = 0.000166326174526
Out: 0.4428541060342247

```

This only took five executions of the loop.

Exercise 2.7: As we mentioned earlier, it is possible for the Secant Method to fail to converge. In that case `dx` would never become small, and the loop would continue forever. We therefore would want some sort of limit on the number of times the `while` loop executes.

How would we edit `secant` so that under no circumstances is the loop executed more than 20 times?

Exercise 2.8: It is also possible that `f(b)-f(a)` might be zero or so small that `x` becomes very large. How would we edit `secant` so that the program simply stops without returning a value if $|f(b) - f(a)|$ is less than 10^{-10} ?

2.5 Newton's Method

The best known method for finding a zero of a function is *Newton's Method*. It is similar to the Secant Method, in that it does not require that we have bracketed a zero. Also, while it is not guaranteed to converge, when it does converge it converges *rapidly*.

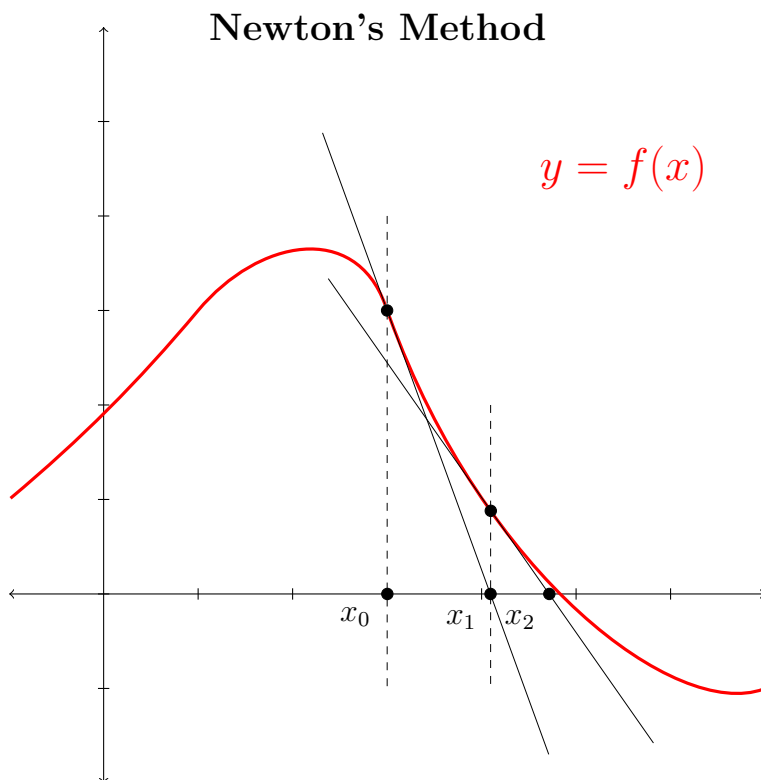
The Secant method uses a secant line drawn through two points, $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$, while Newton's Method uses a *tangent line* drawn through a single point on the graph of f , $(x_k, f(x_k))$. The slope of this line will be the *derivative* of f at x_k , so Newton's Method requires the function to be *differentiable*. If the derivative of f can be calculated, then Newton's Method is both simpler and faster than the Secant Method.

The slope of the tangent line is $m = f'(x_k)$ and the equation of the tangent line is

$$y = mx + f(x_k) - mx_k$$

As before, if we set $y = 0$ and solve, the x -intercept is x_{k+1} .

$$x_{k+1} = \frac{mx_k - f(x_k)}{m} = x_k - \frac{f(x_k)}{f'(x_k)}$$



Example 2.9: Find to four decimal places a zero to the function $f(x) = e^x + x - 2$ (from Example 2.1). Use Newton's Method with $x_0 = 0$.

To begin we need the derivative of f , $f'(x) = e^x + 1$. The first iteration gives us x_1 ,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 0 - \frac{f(0)}{f'(0)} = 0 - \frac{-1}{2} = 0.5$$

The next gives us x_2 ,

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \approx 0.5 - \frac{0.14872}{2.64872} \approx 0.44385$$

Putting the results into a table we can see that x_k converges rapidly to the solution. After only three iterations, we have our solution to six digits. After four we have it to thirteen digits.

k	x_k	$ x_k - x_{k-1} $
0	0.0	N/A
1	0.5	0.5
2	0.44358	0.05615
3	0.44285	0.00100
4	0.44285	3.02×10^{-7}
5	0.44285	2.78×10^{-14}

Exercise 2.10: Find the first two iterations of Newton's Method applied to $f(x) = 2 - \ln(x)$ with $x_0 = 1$.

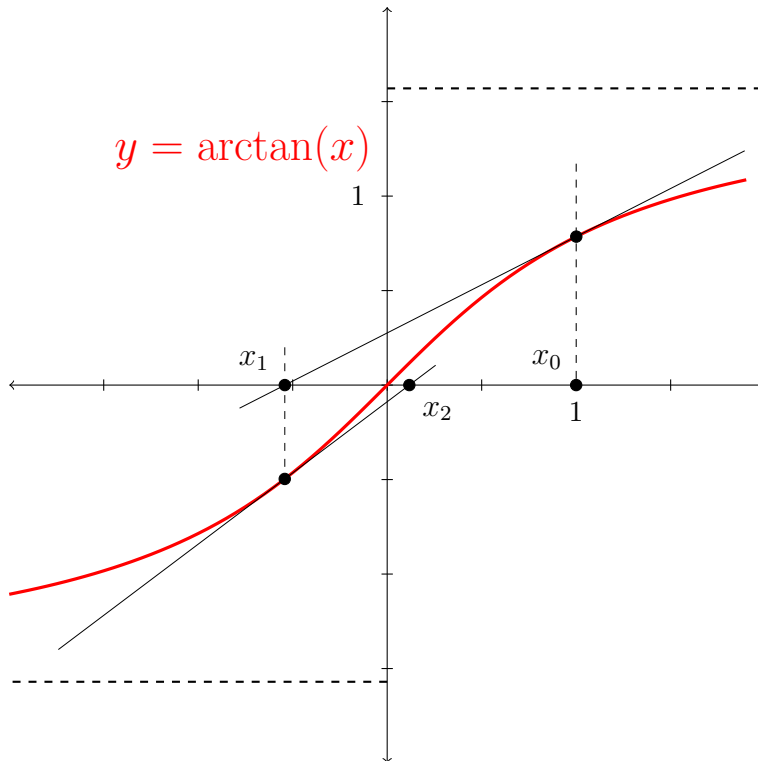
While we did not see it in Example 2.9, Newton's Method does not necessarily converge quickly and directly to the solution. If one of the iterates, x_k , happens to land in a place where f' is close to zero, then the tangent line will be nearly horizontal. In that case x_{k+1} may well be very far from x_k . The sequence $\{x_k\}$ may “bounce around” in such a way that it fails to converge. We mentioned this possibility when we were discussing the Secant Method, but let's now see an explicit example for Newton's Method.

Example 2.11: Consider $f(x) = \arctan(x)$. Clearly $f(x) = 0$ if and only if $x = 0$. Calculate two iterations of Newton's Method with $x_0 = 1.0$ and $x_0 = 1.5$. Explain what is happening **qualitatively** by looking at the graph of f .

$f'(x) = 1/(1 + x^2)$, so the recurrence relation for Newton's Method is

$$x_{k+1} = x_k - \frac{\arctan(x_k)}{1/(1 + x_k^2)} = x_k - (1 + x_k^2) \arctan(x_k)$$

Newton's Method Converges



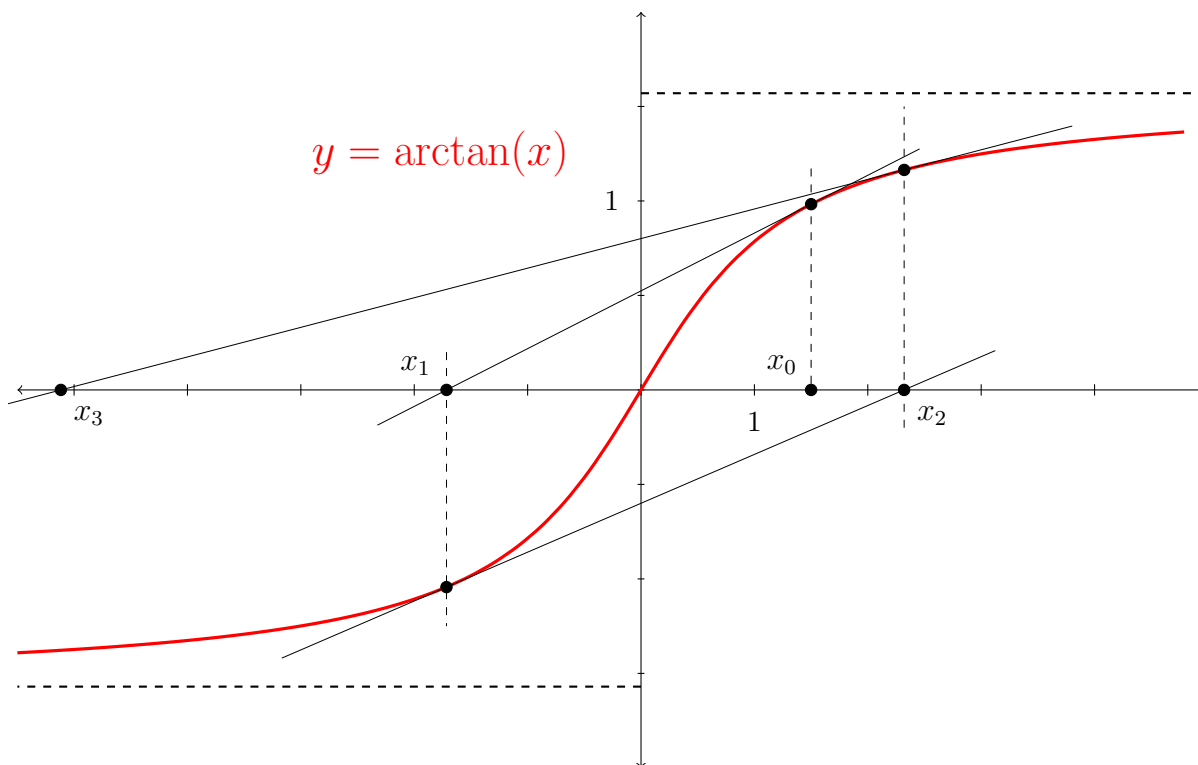
If $x_0 = 1.0$, then

$$x_1 = 1.0 - (1 + (1.0)^2) \arctan(1.0) \approx -0.5708$$

$$x_2 = -0.5708 - (1 + (-0.5708)^2) \arctan(-0.5708) \approx 0.1169$$

It appears Newton's Method is converging to the solution, $x = 0$, as expected.

Newton's Method Diverges



If $x_0 = 1.5$, then

$$x_1 = 1.5 - (1 + (1.5)^2) \arctan(1.5) \approx -1.6941$$

$$x_2 = -1.6941 - (1 + (-1.6941)^2) \arctan(-1.6941) \approx 2.3211$$

$$x_3 = 2.3211 - (1 + (2.3211)^2) \arctan(2.3211) \approx -5.1141$$

Now the iterations of Newton's Method are **diverging away** from the solution!

So what do we know about the convergence of Newton's Method? We will present an informal proof of the theorem below in Chapter 3, but this seems like a good time to state what's true.

Theorem 2.1: (Convergence of Newton's Method)

If: f is differentiable on an interval (a, b) and $f(\bar{x}) = 0$ for some $\bar{x} \in (a, b)$,

Then: for some $\epsilon > 0$ and any $x_0 \in (\bar{x} - \epsilon, \bar{x} + \epsilon)$,

$$\lim_{k \rightarrow \infty} x_k = \bar{x} \quad \text{where } x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

When you get past the rigorous mathematical language, all this theorem says is that if you choose a starting point x_0 for Newton's Method that is **close enough** to the zero that you're looking for, then Newton's Method will converge to that zero.

2.6 Programming Newton's Method

We'll use Example 2.11 to demonstrate how to use Python to apply Newton's Method. In the Editor, edit `f` to be

```
def f(x):  
    return atan(x)
```

We also need the derivative, `df`.

```
def df(x):  
    return 1/(x**2+1)
```

Now in the console,

```
In : x0 = 1.0
```

```
In : x1 = x0 - f(x0)/df(x0)
```

```
In : print(x1)  
-0.570796326794897
```

```
In : x0 = x1
```

```
In : x1 = x0 - f(x0)/df(x0)
```

```
In : print(x1)  
0.116859903998913
```

We're ready to write the program. Again we'll start with a simple `for` loop.

```
def newt(x0,n):  
    for k in range(0,n):  
        x1 = x0 - f(x0)/df(x0)  
        print('x1 = ', x1, 'Change in x = ',abs(x1-x0))  
        x0 = x1  
    return x1
```

Then at the console write `newt(1.0,4)`

```

In : newt(1.0,4)
x1 = -0.570796326794897 Change in x = 1.57079632679490
x1 = 0.116859903998913 Change in x = 0.687656230793810
x1 = -0.00106102211704472 Change in x = 0.117920926115958
x1 = 7.96309604410642e-10 Change in x = 0.00106102291335432
Out: 7.96309604410642e-10

```

This is actually kind of awkward to read. It would be nice if our data could be rounded off to a reasonable number of decimal places and organized into neat columns. For this we'll need *formatted output*, which can be accomplished with a `print` command.

Normally when you use the `print` command, everything inside the single quotes appears exactly as you have written it. You can, however, insert a number into this output using the *control character %*.

Of course we already know how to do this in an uncontrolled way.

```

In : x = sqrt(2)

In : print('The length of the diagonal is ',x,' if the sides are one.')
The length of the diagonal is 1.41421356237 if the sides are one.

```

But we can accomplish the same effect with better formatting using `%`.

```

In : print('The length of the diagonal is %5.3f if the sides are one.' % x)
The length of the diagonal is 1.414 if the sides are one.

```

The control sequence `%5.3f` needs some explaining. `%` starts the control sequence. The 5 says that there will be a minimum of five spaces for the output (there may be more). The 3 after the `.` says that we want the number rounded to three decimal places. Finally the `f` tells the computer to display the number as a *floating point number*. After the string is ended by the single quote, the `% x` tells the computer to insert `x` into the output where the `%5.3f` control sequence appears.

Exercise 2.12: What would be the output if you wrote:

```
In : print('The length of the diagonal is %3.5f if the sides are one.' % x)
```

Exercise 2.13: What would be the output if you wrote:

```
In : print('The length of the diagonal is %10.4f if the sides are one.' % x)
```

If we replace the `f` with an `e`, then the number is presented in *scientific notation*.


```
In : print('The length of the diagonal is %8.2e if the sides are one.' % x)
The length of the diagonal is 1.41e+00 if the sides are one.
```

Finally, we can insert more than one number into the output if we use a Python *tuple*.

```
In : r = 4.0
```

```
In : print('The area of a circle of radius %3.2f is %5.2f.' % (r,pi*r**2))
The area of a circle of radius 4.00 is 50.27
```

Returning to our `newt` program, we can clean up the output by using the formatting.

```
def newt(x0,n):
    for k in range(0,n):
        x1 = x0 - f(x0)/df(x0)
        print('x1 = %10.5f, change in x = %10.2e' % (x1, abs(x1-x0)))
        x0 = x1
    return x1
```

Then writing `newt(1.0,4)` at the console,

```
In : newt(1.0,4)
x1 = -0.57080, change in x = 1.57e+00
x1 = 0.11686, change in x = 6.88e-01
x1 = -0.00106, change in x = 1.18e-01
x1 = 0.00000, change in x = 1.06e-03
Out: 7.96309604410642e-10
```

Similarly,

```
In : newt(1.5,4)
x1 = -1.69408, change in x = 3.19e+00
x1 = 2.32113, change in x = 4.02e+00
x1 = -5.11409, change in x = 7.44e+00
x1 = 32.29568, change in x = 3.74e+01
Out: 32.2956839142100
```

Exercise 2.14: Use `newt` and trial and error to approximate the value of x_0 where $x_1 = -x_0$. That is, the point where Newton's method neither spirals out (diverging), nor spirals in (converging), but simply toggles back and forth.

2.7 Exercise Solutions and Problems

Solution to Exercise 2.3:

We want to rule out the possibility that either both $f(a)$ and $f(b)$ are positive, or that that they are both negative. We could, therefore, write our `if` statement as:

```
if (f(a)>0 and f(b)>0) or (f(a)<0 and f(b)<0):
```

But there's a much more elegant way to do it. The product of two numbers will be positive if and only if they have the same signs. Therefore we can just write:

```
if f(a)*f(b)>0:
```

The program will then be:

```
def bisectiontimes(a,b,n):
    if f(a)*f(b)>0:
        print('Zero is not bracketed by', (a,b))
        return
    for k in range(0,n):
        (a,b) = bisectiononce(a,b)
    print('Error <= ', (b-a)/2)
    return (a+b)/2
```

The `return` statement inside the `if` statement will cause the function `bisectiontimes` to end immediately without returning any value.

Solution to Exercise 2.4:

We can use the same trick from Exercise 2.3 for this change as well. $f(m)*f(b)>0$ will only be true when $f(m)$ and $f(b)$ have the same sign (likewise for $f(m)$ and $f(a)$). So the program should be:

```
def bisectiononce(a,b):
    m = (a+b)/2
    if f(m)*f(b) > 0:
        b = m
    if f(m)*f(a) > 0:
        a = m
    print((a,b))
    return a,b
```

Solution to Exercise 2.6:

$$x_2 = 2 - \frac{(2-1)f(2)}{f(2)-f(1)} \approx 2 - \frac{1.30685}{-0.69315} \approx 3.88539$$

$$x_3 \approx 3.88539 - \frac{1.21188}{-0.66408} \approx 5.71031$$

Solution to Exercise 2.7:

We'll need a *counter* to count how many times the loop has been executed, as well as a change to the *while* loop condition.

```
def secant(a,b,tol):
    dx = abs(a-b)
    count = 0
    while dx > tol and count < 21:
        x = b - ((b-a)*f(b))/(f(b)-f(a))
        a = b
        b = x
        dx = abs(a-b)
        count = count + 1
        print('x = ', x, 'change in x = ', dx)
    return x
```

Thus the *while* loop will stop if **either** $dx \leq tol$ **or** $count \geq 21$.

Solution to Exercise 2.8:

We just want an *if* statement right before we divide by $f(b)-f(a)$. If $f(b)-f(a)$ is too small the program should print a message and then end with a **return**.

```
def secant(a,b,tol):
    dx = abs(a-b)
    count = 0
    while dx > tol and count < 21:
        if abs(f(b)-f(a)) < 1e-10:
            print('|f(a)-f(b)| < 1e-10')
            return (b-a)/2
        x = b - ((b-a)*f(b))/(f(b)-f(a))
        a = b
        b = x
        dx = abs(a-b)
        count = count + 1
        print('x = ', x, 'change in x = ', dx)
    return x
```

Solution to Exercise 2.10:

$f'(x) = -1/x$, so

$$x_1 = 1 - \frac{2 - \ln(1)}{-1/1} = 3$$

$$x_2 = 3 - \frac{2 - \ln(3)}{-1/3} \approx 5.7042$$

Solution to Exercise 2.12:

The length of the diagonal is 1.41421 if the sides are one.

Python would have tried to put the number into a section of length 3 characters, but it was too long when there were five decimal places plus the leading 1 plus the decimal point.

Solution to Exercise 2.12:

The length of the diagonal is 1.4142 if the sides are one.

Python reserves a section of length 10 characters for the number. The number's four decimal places plus the leading 1 plus the decimal point occupy 6 spaces, so there are an additional 4 spaces in front of the number.

Solution to Exercise 2.14:

We saw that Newton's Method diverges for $x_0 = 1.5$, and converges for $x_0 = 1.0$, so the 'toggling' value must be between these two. Since,

```
In : newt(1.4,4)
x1 =  -1.41362, change in x =  2.81e+00
x1 =   1.45013, change in x =  2.86e+00
x1 =  -1.55063, change in x =  3.00e+00
x1 =   1.84705, change in x =  3.40e+00
Out: 1.84705408415019
```

Newton's Method appears to be very slowly diverging, while

```
In : newt(1.39,4)
x1 =  -1.38715, change in x =  2.78e+00
x1 =   1.37964, change in x =  2.77e+00
x1 =  -1.36002, change in x =  2.74e+00
x1 =   1.30949, change in x =  2.67e+00
Out: 1.30948824748913
```

Now Newton's Method appears to be very slowly converging, so we conclude the 'toggling' value is between 1.39 and 1.4. Further investigation shows it's slightly bigger than 1.39174.

```
In : newt(1.39174,4)
x1 =  -1.39173, change in x =  2.78e+00
x1 =   1.39171, change in x =  2.78e+00
x1 =  -1.39165, change in x =  2.78e+00
x1 =   1.39149, change in x =  2.78e+00
Out: 1.39149336326741
```

Problem 2.1: There is a variation on the Bisection Method called the **Method of False Position**. This method also requires that the function be continuous and that you have bracketed a zero. However, instead of considering the midpoint of the interval at each step, this method takes the x -intercept, x_i of the line from $(a, f(a))$ through $(b, f(b))$. As before, x_i becomes the new left or right endpoint of the new, smaller interval depending on whether $f(x_i)$ is positive or negative. However with this method, the interval will not, in general, approach zero width. Instead, one of the end points will approach the actual zero.

- a. Derive the formula for x_i :

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

- b. Write programs `falseposonce` and `falseposntimes` which implement the Method of False Position. As in the Bisection Method, the program should return the estimated zero and the error. As opposed to the Bisection Method, however, the estimated zero should be either the left or right endpoint. We will estimate the error as the distance between x_i and the appropriate endpoint.
- c. Test your program by finding the zero for $f(x) = e^x + x - 2$. How many iterations does it take to get an error less than 10^{-4} ? How does that compare to the Bisection Method?

Problem 2.2: Consider the function:

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

Clearly this function has zeros at $x = \pm 1$.

- Use our Newton's Method program to experimentally determine the interval $[a, b]$ around $x = 1$ so that if the starting value is in this interval, Newton's Method converges to $x = 1$.
- Experimentally determine the value c so that Newton's Method **diverges** if the starting value is greater than c .
- Investigate what happens between b and c . Explain what you see. (It might help to sketch the graph of f .)

Problem 2.3: Newton's Method's fast convergence relies on the derivative of f being non-zero when f is zero. That is, if $f(x_0) = 0$ then $f'(x_0) \neq 0$. We say such zeros have **multiplicity** = 1. In general, the multiplicity of a zero of f at x_0 is the lowest number of derivative of f which gives a non-zero result at x_0 .

So, for example if $f(x) = 1 - \cos(x)$, then $f(0) = 0$ and $f'(0) = \sin(0) = 0$, but $f''(0) = \cos(0) = 1 \neq 0$. Thus the multiplicity of 0 is 2.

To find higher multiplicity zeros efficiently we must modify Newton's Method so that:

$$x_{k+1} = x_k - m \frac{f(x_k)}{f'(x_k)} \quad (m = \text{multiplicity of zero})$$

- Make a copy of `newt`, call it `mnewt`, and modify it so that it also takes an argument `m` which is the multiplicity of the zero you're looking for.
- Let $f(x) = 1 - \cos(x)$. Use regular `newt` and our new program `mnewt` with $m = 2$ to find the zero at $x = 0$. Compare the number of iterations required.
- Let $f(x) = x^4 - x^3 - 3x^2 + 5x - 2$. Find the multiplicity of the zero of f at $x = 1$.
- Use regular `newt` and our new program `mnewt` to find the zero of f from the previous part at $x = 1$. Compare the number of iterations required.

Chapter 3

Taylor's Theorem

An enormous part of the theoretical underpinnings of numerical methods is based on a single theorem from calculus, *Taylor's Theorem*. The proof of Taylor's Theorem is not very illuminating, so we leave it to Appendix A.2.

Theorem 3.1: Taylor's Theorem

If: a function $f : \mathbb{R} \rightarrow \mathbb{R}$ has $n + 1$ continuous derivatives on some open interval (a, b) , **then:** for any $x, x_0 \in (a, b)$ there exists a number ξ between x and x_0 so that:

$$\begin{aligned} f(x) = & f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \dots \\ & \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \end{aligned}$$

The first n terms are called the *Taylor Polynomial* of f centered at x_0 . The final term involves a mysterious number ξ , which constitutes an *error* term. Generally the best we can do is to estimate it since we generally don't know exactly what number ξ is. The estimation goes something like this,

$$\left| \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \right| \leq \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} \right| |x - x_0|^{n+1} \leq K|x - x_0|^{n+1}$$

Since the $n + 1$ -th derivative of f is continuous in an interval around x_0 , it is bounded. This bound contributes to the constant, K . This inequality tells us that, in the “Big O ” notation from section 1.3, the error is **on the order of** $(x - x_0)^{n+1}$.

If we define the n -th Taylor polynomial centered at x_0 to be

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

then the conclusion to Taylor's Theorem implies

$$f(x) = P_n(x) + O(x - x_0)^{n+1}$$

While this may seem very abstract at this point, we'll see very soon that it has very real, very useful implications for the estimation of a variety of mathematical quantities.

3.1 Taylor Polynomials

The Taylor polynomial of a function can be described as the “best” n -th degree polynomial approximation to the function at the point x_0 . You might justifiably question what is meant by “best”. While there is a rigorous statement that can be made, it’s perhaps more illuminating to just look at the graphs of some functions and a couple of their associated Taylor polynomials.

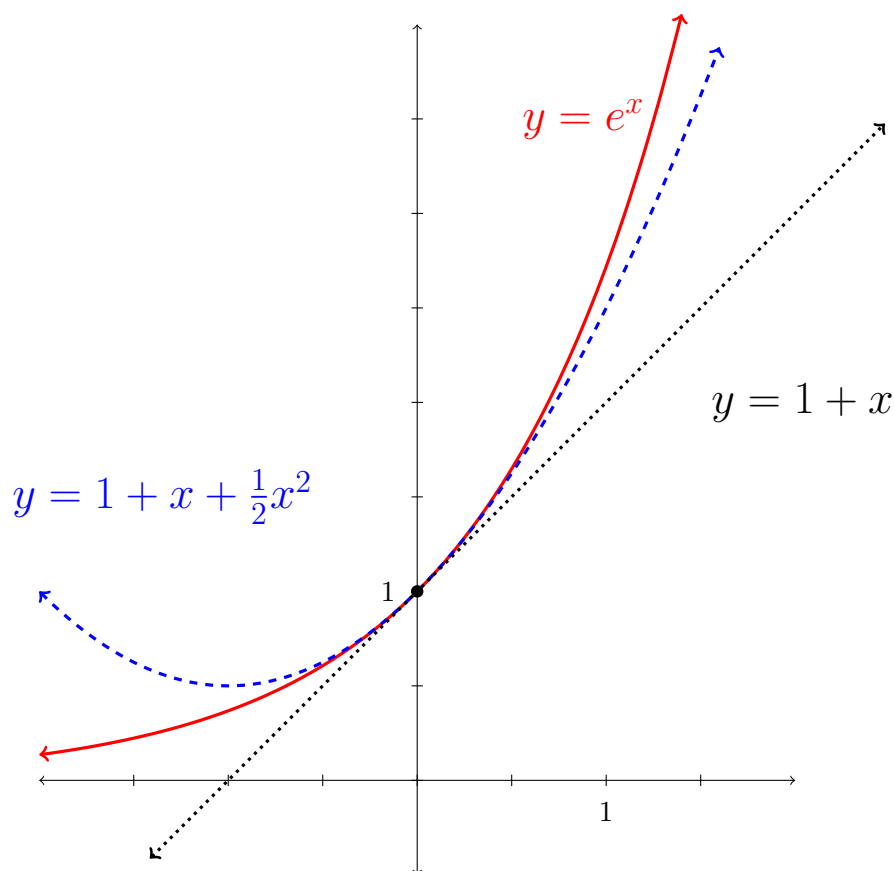
Example 3.1: Find and graph the first and second Taylor polynomials to the function $f(x) = e^x$ about the point $x_0 = 0$.

$f(0) = f'(0) = 1$, so the first Taylor polynomial is

$$P_1(x) = f(0) + f'(0)(x - 0) = 1 + x$$

$f''(0) = 1$ also, so the second Taylor polynomial is

$$P_2(x) = f(0) + f'(0)(x - 0) + \frac{f''(0)}{2}(x - 0)^2 = 1 + x + \frac{1}{2}x^2$$



Notice that, while both polynomials agree with the function at $x_0 = 0$, the quadratic Taylor polynomial stays closer to the function for longer.

In totally artificial situations like those in Example 3.1 we can actually calculate the number ξ from Theorem 3.1 (Taylor’s Theorem).

Example 3.2: Find ξ so that the conclusion of Taylor's Theorem is satisfied for $f(x) = e^x$, $x_0 = 0$, $x = 0.5$, and $n = 1$.

Taylor's Theorem says that there should be a number ξ so that

$$e^{0.5} = 1 + 0.5 + \frac{f''(\xi)}{2}(0.5)^2 = 1.5 + \frac{e^\xi}{8}$$

Solving for ξ ,

$$\xi = \ln(8 * (e^{0.5} - 1.5)) \approx 0.17376$$

This is a reassuring number as $0 \leq 0.17376 \leq 0.5$, so $x_0 \leq \xi \leq x$ just as Taylor's Theorem guarantees.

Exercise 3.3: Find ξ so that the conclusion of Taylor's Theorem is satisfied for $f(x) = e^x$, $x_0 = 0$, $x = 0.5$, and $n = 2$. (As above, your number should be between 0 and 0.5.)

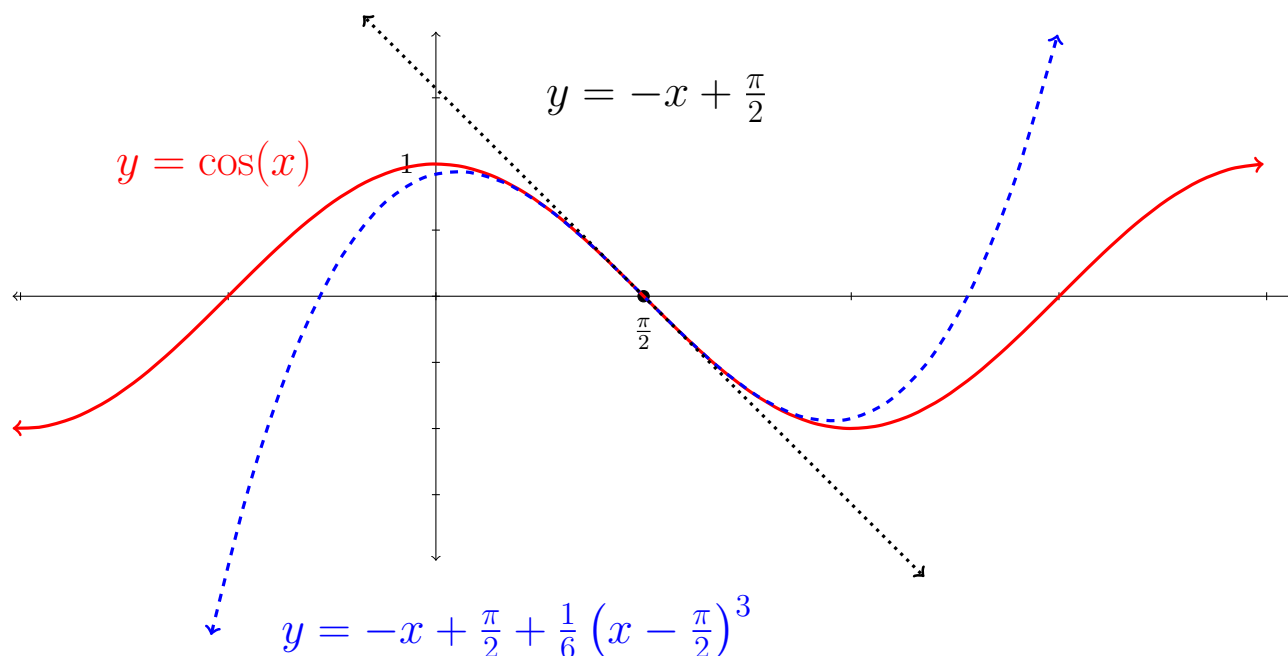
Example 3.4: Find and graph the first and third Taylor polynomials to the function $f(x) = \cos(x)$ about the point $x_0 = \pi/2$.

$f(\pi/2) = 0$ and $f'(\pi/2) = -1$, so the first Taylor polynomial is

$$P_1(x) = f\left(\frac{\pi}{2}\right) + f'\left(\frac{\pi}{2}\right)\left(x - \frac{\pi}{2}\right) = -x + \frac{\pi}{2}$$

$f''(\pi/2) = 0$ and $f'''(\pi/2) = 1$, so the third Taylor polynomial is

$$P_3(x) = f'\left(\frac{\pi}{2}\right)\left(x - \frac{\pi}{2}\right) + \frac{1}{6}f'''\left(\frac{\pi}{2}\right)\left(x - \frac{\pi}{2}\right)^3 = -x + \frac{\pi}{2} + \frac{1}{6}\left(x - \frac{\pi}{2}\right)^3$$



Exercise 3.5: Find ξ so that the conclusion of Taylor's Theorem is satisfied for $f(x) = \cos(x)$, $x_0 = \pi/2$, $x = \pi/4$, and $n = 5$. (Your number should be between $\pi/4$ and $\pi/2$.)

Exercise 3.6: Find the second and third Taylor polynomials to the function $f(x) = \ln(x)$ about the point $x_0 = 1$.

3.2 Graphing in Python

In the previous section we not only derived Taylor polynomials for several functions, but also produced their graphs. In this section we'll see how to produce these graphs in Python. The syntax for graphing in Python is based on the syntax in Matlab, so if you are familiar with plotting in Matlab, this should not seem very different.

Let's attempt to reproduce the graph in Example 3.1. First we need to graph the exponential function. To do that we will need to import some Python libraries, then produce a range of x values and evaluate the exponential function at each of those values. This will produce a sequence of points in the xy plane. Finally we have Python connect the dots together with a nice, smooth line. This will be the graph of $y = e^x$.

Open a new python file and save it as `graphing`. Then write the following code in the file and save.

```
from matplotlib.pyplot import *
from numpy import *

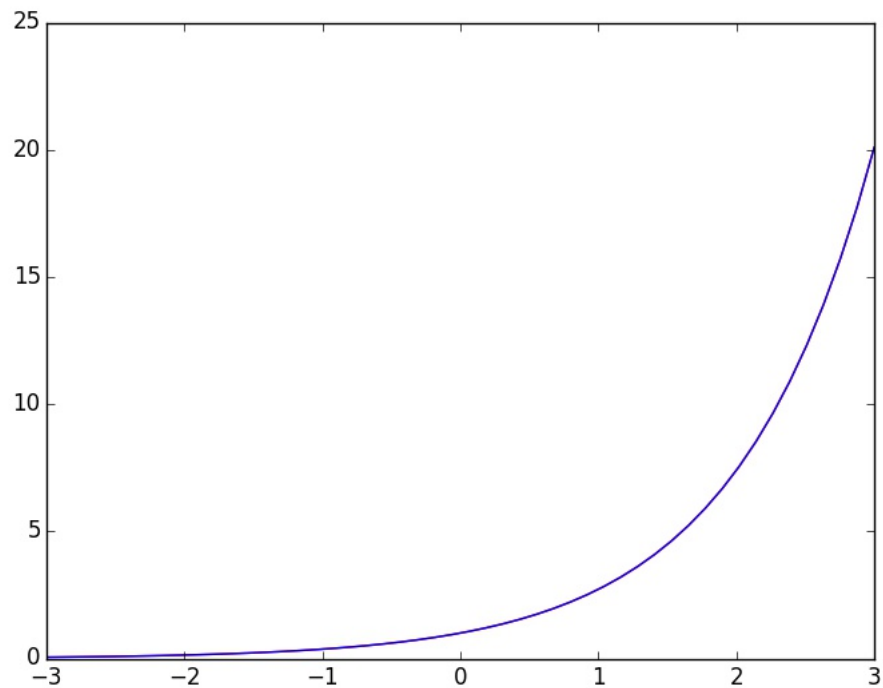
x = linspace(-3,3,50)
plot(x,exp(x))
```

`linspace` produces an *array* of fifty evenly spaced x values, starting at -3 and ending at 3 . `exp` produces another array of y values which is the exponential function applied to each x value. Finally the `plot` function connects the x - y pairs with lines and puts the output onto axes.

If you run this code, you will see a small graph appear in the IPython console. It works, but it's not very impressive. Before going any further, we'd like to have our graph in its own window, out where we can see it and save it. To do this, at the IPython prompt write:

```
In : %matplotlib
Using matplotlib backend: Qt4Agg
```

This tells Python to (among other things) direct its plotting output to the *backend* program QT4Agg. Now when you run the file, you should get a new window that looks like:

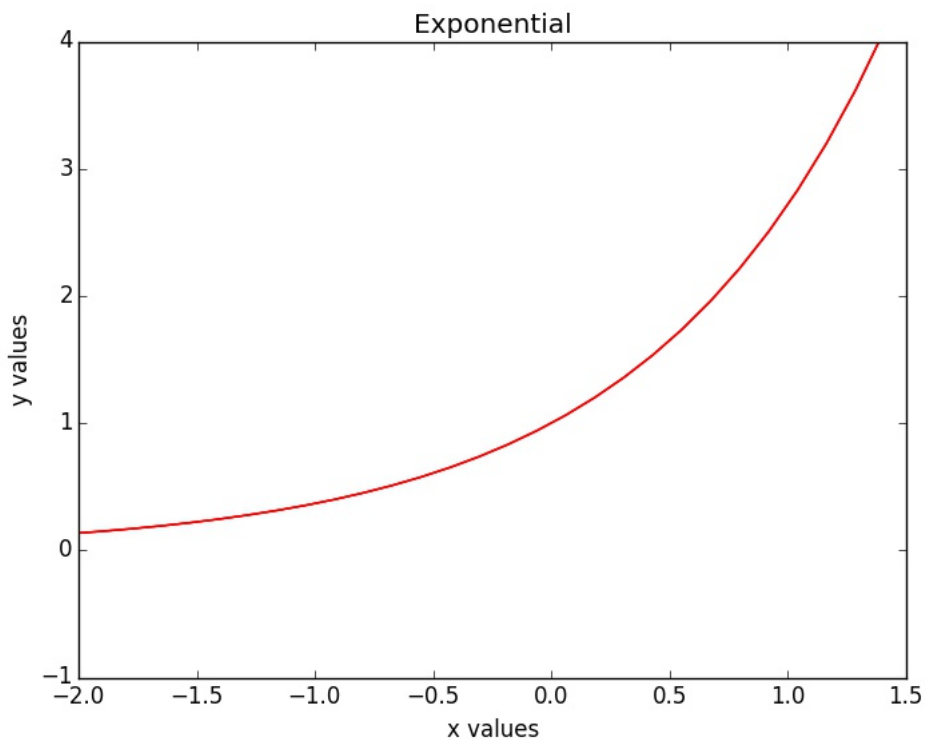


A good beginning, but let's add some things. First, we'd like the x -axis to go from -2 to 1.5 , while the y -axis should go from -1 to just 4 . This is accomplished with the `axis` command. We'd also like to label the axes and give the graph a title. Finally we'd like the graph to be red. Edit your file so that it reads:

```
from matplotlib.pyplot import *
from numpy import *

title('Exponential')
xlabel('x values')
ylabel('y values')
x = linspace(-3,3,50)
plot(x,exp(x),color='red')
axis([-2,1.5,-1,4])
```

Now when you run the file, the graphing window should change to:

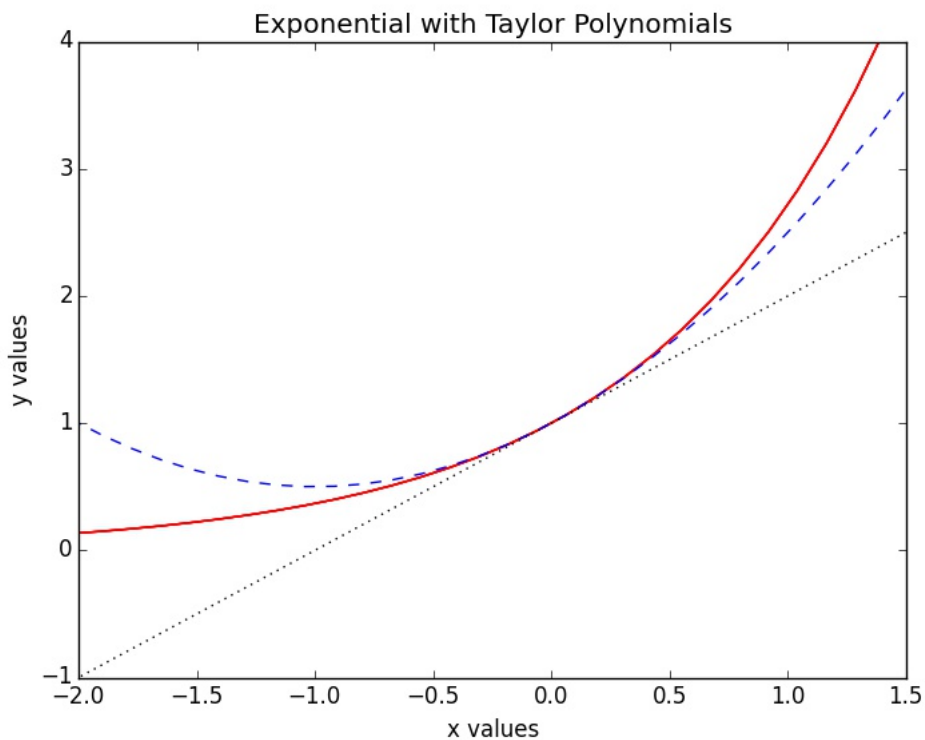


Much better. Now let's add the graphs of the two Taylor polynomials. Just as we did for the exponential function, we can adjust the color and the style of the graph lines by assigning appropriate string values to the variables `color` and `linestyle` inside the `plot` function. Notice, too, that we calculated the y values for the two polynomials separately and stored them in the arrays, `tp1` and `tp2`. We then used those arrays in the `plot` function.

```
from matplotlib.pyplot import *
from numpy import *

title('Exponential with Taylor Polynomials')
xlabel('x values')
ylabel('y values')
x = linspace(-3,3,50)
plot(x,exp(x),color='red')
axis([-2,1.5,-1,4])
tp1 = 1 + x
plot(x,tp1,color='black',linestyle=':')
tp2 = 1 + x + x**2/2
plot(x,tp2,color='blue',linestyle='--')
```

Now when you run the file, the graphing window should change to:

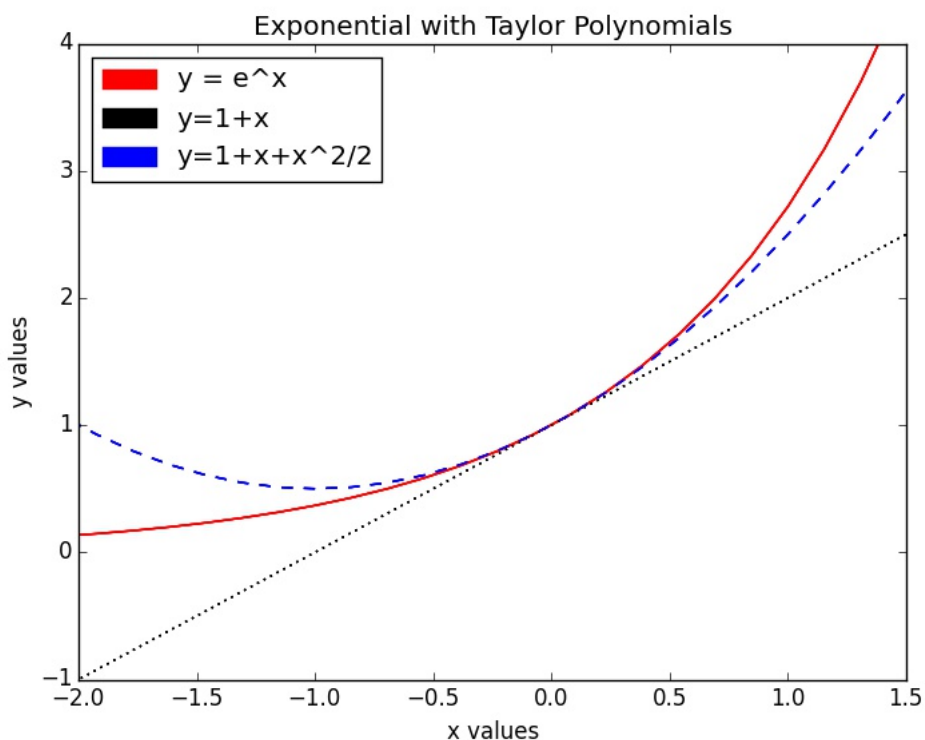


Finally we need a *legend* to remind us which graph is which. This actually requires a command from yet another library, but that's no trouble. The final code should be:

```
from matplotlib.pyplot import *
from numpy import *
from matplotlib.patches import Patch

title('Exponential with Taylor Polynomials')
xlabel('x values')
ylabel('y values')
x = linspace(-3,3,50)
plot(x,exp(x),color='red')
axis([-2,1.5,-1,4])
tp1 = 1 + x
plot(x,tp1,color='black',linestyle=':')
tp2 = 1 + x + x**2/2
plot(x,tp2,color='blue',linestyle='--')
L1 = Patch(color='red',label='y = e^x')
L2 = Patch(color='black',label='y=1+x')
L3 = Patch(color='blue',label='y=1+x+x^2/2')
legend(handles=[L1,L2,L3],loc='upper left')
```

Running it should produce a very attractive looking graph:



Exercise 3.7: Use Python to reproduce, as best you can, the graph from Example 3.4.

3.3 Convergence of Newton's Method

In section 2.5 we applied a rather cookbook algorithm called Newton's Method for finding the zeros of a smooth function. We got some intuition for why this method might work from looking at the graphs of functions and their tangent lines. We saw empirically that the method worked surprisingly well, but it wasn't really clear **why**.

We'll give a fairly informal "proof" of Theorem 2.1 in the hopes that this will give the reader a sense of when and why Newton's method works. Also, it should give us a more precise idea of just how quickly Newton's method converges once we are close to a zero.

Let's consider the smooth function, f , from Theorem 2.1 and a point \bar{x} where $f(\bar{x}) = 0$. Let x_0 be our "first guess" at the zero \bar{x} . Then by Taylor's Theorem (3.1),

$$f(\bar{x}) = 0 = f(x_0) + f'(x_0)(\bar{x} - x_0) + O(\bar{x} - x_0)^2$$

Solving for \bar{x} ,

$$\bar{x} = x_0 + \frac{-f(x_0) + O(\bar{x} - x_0)^2}{f'(x_0)}$$

We saw in problem 2.3 that Newton's Method converges much more slowly if $f'(\bar{x}) = 0$ (that is, if \bar{x} is a **higher multiplicity zero**). Theorem 2.1 is still true, but the convergence is much slower (and the proof is much harder). So let's assume that $f'(\bar{x}) \neq 0$. Then if x_0 is close to \bar{x} , $f'(x_0) \neq 0$ either. Dividing by $f'(x_0)$ then just changes the constant K associated to the big O term.

So,

$$\bar{x} = x_0 - \frac{f(x_0)}{f'(x_0)} + O(\bar{x} - x_0)^2$$

When we let

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

We see that

$$\bar{x} = x_1 + O(\bar{x} - x_0)^2 \Rightarrow |\bar{x} - x_1| = O(\bar{x} - x_0)^2$$

And in general that

$$|\bar{x} - x_{k+1}| = O(\bar{x} - x_k)^2$$

What this says is that the raw error for x_{k+1} is on the order of the raw error for x_k **squared**. Now if the raw error for x_k is large, then the raw error for x_{k+1} could be very large. This is the case when Newton's Method is "jumping around". On the other hand, if the raw error for x_k is **small** then the raw error for x_{k+1} will be **very small**. This is the case when Newton's Method is converging rapidly.

Example 3.8: Let $f(x) = e^x - 4$. Clearly the zero of this function is $\ln(4) \approx 1.386$. Apply Newton's Method to f with $x_0 = 2$. Comment on the error at each step.

Below are listed the iterations from Newton's Method and the associated errors.

k	x_k	$ x_k - \ln(4) $
0	2.00000	6.137e-01
1	1.54134	1.550e-01
2	1.39772	1.142e-02
3	1.38636	6.498e-05
4	1.38629	2.111e-09
5	1.38629	0.000e+00

After the first step, the method begins to converge rapidly. The error at $k = 1$ is approximately 10^{-1} while the error at $k = 2$ is approximately 10^{-2} , or the $k = 1$ error **squared**. The $k = 3$ error is $6.5 \times 10^{-5} \approx 10^{-4}$, or about the $k = 2$ error squared.

The $k = 4$ error is on the order of the expected error, 10^{-8} . We would expect the $k = 5$ error to be a little less than 10^{-16} , but this is less than the smallest machine precision ($\approx 10^{-16}$), so the computer gives an error of exactly 0.

3.4 Derivative Estimates

In this section we'll see how Taylor's Theorem allows us to estimate the derivatives of a function at a point based solely on a set of nearby points. While this isn't particularly useful in and of itself, we'll see that it is absolutely critical for estimating definite integrals (Chapter 4) and the solutions to initial value problems (Chapter 5).

For these applications it will be convenient for us to express Taylor's Theorem in a slightly different form. We'll let $h = x - x_0$, and drop the 0 from x_0 . Then the conclusion to Taylor's Theorem may be written,

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \dots + \frac{f^{(n)}(x)}{n!}h^n + O(h^{n+1})$$

Let's begin by finding the simplest estimate for the first derivative, $f'(x)$. From Taylor's Theorem with $n = 1$ we have

$$f(x+h) = f(x) + f'(x)h + O(h^2)$$

Solving for $f'(x)$ gives,

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

It can be shown that, intuitively enough, for $n \geq m$,

$$\frac{O(h^n)}{h^m} = O(h^{n-m})$$

So an h cancels from inside the big O term, leaving us with

Theorem 3.2: (Forward Difference Formula)

If f has two continuous derivatives on $[x, x+h]$, **then**

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

This theorem also tells us that this formula has an **error on the order of h** .

Example 3.9: Use the Forward Difference Formula to estimate $f'(1)$ for $f(x) = e^x$ for $h = 0.1, 0.01, 0.001$. Comment on the errors for each value of h .

$f'(x) = e^x$, so the true value of $f'(1) = e \approx 2.71828$.

Using the Forward Difference Formula

$$f'(1) \approx \frac{e^{(1+0.1)} - e^1}{0.1} \approx 2.85884$$

The raw error is $\approx 2.85884 - 2.71828 \approx 0.14056$.

Presenting the results in a table,

h	f' estimate	error
0.1	2.85884	0.14056
0.01	2.73192	0.01364
0.001	2.71964	0.00136

Note that when h is reduced by one tenth, the error is reduced by approximately one tenth. This is characteristic of systems where the error is of **the same order as h** .

Example 3.10: Create a Python file named `Derivative`, and define a function `f` as e^x . Then write a Python program called `FD` which takes `x` and `h` as arguments and returns an approximation to $f'(x)$ using the Forward Difference Formula.

You may need to insert a line that *imports* the exponential function, `exp`. The file `Derivative.py` should look similar to:

```
from math import exp

def f(x):
    return exp(x)

def FDD(x,h):
    dy = (f(x+h)-f(x))/h
    return dy
```

Now we run the file `Derivative` and write in the console:

```
In : FDD(1,0.1)
Out: 2.858841954873883
```

Exercise 3.11: Below `f`, but above `FD`, define a function `df` also as e^x . (In general this will be the actual derivative of `f`.) Now modify your program `FD` so that it calculates the *raw error*. Then use formatted output to print out h , $f'(x)$, and the error to five decimal places.

While h may be positive or negative, we usually interpret it as positive. In that case we can come up with a formula essentially identical to the Forward Difference Formula by putting a minus sign in front of the h . From Taylor's Theorem with $n = 1$ we have

$$f(x - h) = f(x) + f'(x)(-h) + O((-h)^2)$$

Simplifying $O(-h)^2$, and solving for $f'(x)$ gives,

Theorem 3.3: (Backwards Difference Formula)
If f has two continues derivatives on $[x - h, x]$ then

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

Again, this formula has an **error on the order of h** , so we really haven't gained anything from an accuracy standpoint. Of course we can always improve the accuracy by using a smaller value for h , but what we really want is a formula that will give us high accuracy from an only moderately small h .

To make progress toward a more accurate solution we need to use a higher degree Taylor polynomial and **more points**. Let's look at a longer expansion of both $f(x+h)$ and $f(x-h)$ and **subtract** the second from the first.

$$\begin{array}{rclcl} f(x+h) & = & f(x) & +f'(x)h & +\frac{1}{2}f''(x)h^2 & +O(h^3) \\ -f(x-h) & = & -f(x) & +f'(x)h & -\frac{1}{2}f''(x)h^2 & +O(h^3) \\ \hline f(x+h) - f(x-h) & = & 0 & +2f'(x)h & +0 & +O(h^3) \end{array}$$

Now solving for f' we have

Theorem 3.4: (Central Difference Formula)

If f has three continuous derivatives on $[x-h, x+h]$, then

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

Now the error is on the order of h^2 , a great improvement as we will see.

Example 3.12: Use the Central Difference Formula to estimate $f'(1)$ where $f(x) = e^x$ for $h = 0.1, 0.01, 0.001$. Comment on the errors for each value of h .

Using the Central Difference Formula

$$f'(1) \approx \frac{e^{(1+0.1)} - e^{(1-0.1)}}{2(0.1)} \approx 2.72281$$

The raw error is $\approx 2.72281 - 2.71828 \approx 0.00453$.

Presenting the results in a table,

h	f' estimate	error
0.1	2.72281	4.53×10^{-3}
0.01	2.71832	4.53×10^{-5}
0.001	2.71828	4.53×10^{-7}

Now when h is reduced by one tenth, the error is reduced by approximately one hundredth. This is characteristic of systems where the error is **on the order of h^2** .

Exercise 3.13: Add to the file **Derivative** another function **CD** which takes **x** and **h** as arguments and returns $f'(x)$ using the Central Difference Formula. It should use formatted output to print h and $f'(x)$ to five decimal places. It should also print the error in scientific notation to two decimal places.

Exercise 3.14: Use Taylor's Theorem to expand $f(x+2h)$ up to $O(h^3)$. Then subtract the expansion of $f(x-h)$ and derive another $O(h^2)$ formula for $f'(x)$.

Exercise 3.15: Verify that the formula derived in Exercise 3.14 is $O(h^2)$ by estimating $f'(1)$ where $f(x) = e^x$ for $h = 0.1, 0.01, 0.001$.

Taylor's Theorem may also be used to estimate higher derivatives. If we consider Taylor expansions up to $O(h^4)$ of $f(x+h)$ and $f(x-h)$ and add them, then we may produce a formula for f'' .

$$\begin{array}{rcl} f(x+h) & = & f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + O(h^4) \\ +f(x-h) & = & f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + O(h^4) \\ \hline f(x+h) + f(x-h) & = & 2f(x) + 0 + f''(x)h^2 + 0 + O(h^4) \end{array}$$

Since both the f' and f''' terms drop out, we may solve for $f''(x)$, giving

Theorem 3.5: (Second Derivative Central Difference Formula)

If f has four continuous derivatives on $[x-h, x+h]$, **then**

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

Here again we used the fact that

$$\frac{O(h^4)}{h^2} = O(h^2)$$

to show our formula is accurate to order h^2 .

Example 3.16: Use the Central Difference Formula to estimate $f''(0)$ where $f(x) = \cos(x)$ for $h = 0.1, 0.01, 0.001$. Comment on the errors for each value of h .

Using the Central Difference Formula for $h = 0.1$,

$$f''(0) \approx \frac{\cos(0+0.1) - 2\cos(0) + \cos(0-0.1)}{(0.1)^2} \approx -0.999167$$

We know $f''(0) = -\cos(0) = -1$, so the raw error is 0.000833.

Assembling the data in a table with the other values of h and the corresponding errors,

h	f'' estimate	error
0.1	-0.999167	8.33×10^{-4}
0.01	-0.999992	8.33×10^{-6}
0.001	-1.000000	8.33×10^{-8}

Again the error decreases by two orders of magnitude for each order of magnitude that h is decreased.

Exercise 3.17: Add to the file `Derivative` another function `CDD` which takes `x` and `h` as arguments and returns $f''(x)$ using the Central Difference Formula for the Second Derivative. It should use formatted output to print h and $f''(x)$ to five decimal places. It should also print the error in scientific notation to two decimal places.

3.5 Exercise Solutions and Problems

Solution to Exercise 3.3:

Taylor's Theorem says that there should be a number ξ so that

$$e^{0.5} = 1 + 0.5 + \frac{1}{2}(0.5)^2 + \frac{f'''(\xi)}{6}(0.5)^3 = 1.625 + \frac{e^\xi}{48}$$

Solving for ξ ,

$$\xi = \ln(48(e^{0.5} - 1.625)) \approx 0.12982$$

$0 \leq 0.12982 \leq 0.5$, so again $x_0 \leq \xi \leq x$ just as Taylor's Theorem guarantees.

Solution to Exercise 3.5:

Since $f^{(4)}(x) = \cos(x)$, $f^{(4)}(\pi/2) = 0$. So,

$$P_4(x) = P_3(x) = -x + \frac{\pi}{2} + \frac{1}{6}\left(x - \frac{\pi}{2}\right)^3$$

Taylor's Theorem says that there should be a number ξ so that

$$\cos\left(\frac{\pi}{4}\right) = -\frac{\pi}{4} + \frac{\pi}{2} + \frac{1}{6}\left(\frac{\pi}{4} - \frac{\pi}{2}\right)^3 + \frac{f^{(5)}(\xi)}{5!}\left(\frac{\pi}{4} - \frac{\pi}{2}\right)^5$$

$\cos^{(5)}(\xi) = -\sin(\xi)$, so this simplifies to

$$\frac{\sqrt{2}}{2} = \frac{\pi}{4} - \frac{\pi^3}{384} + \frac{\pi^5}{122880} \sin(\xi)$$

Or, approximating,

$$0.7071068 \approx 0.7046526 + 0.0024904 \sin(\xi)$$

Solving for ξ ,

$$\xi = \sin^{-1}(0.99995) \approx 1.56065$$

or just less than $\pi/2 \approx 1.57080$, as Taylor's Theorem requires.

Solution to Exercise 3.6:

$f(1) = 0$ and $f'(1) = 1$, so the first Taylor polynomial is

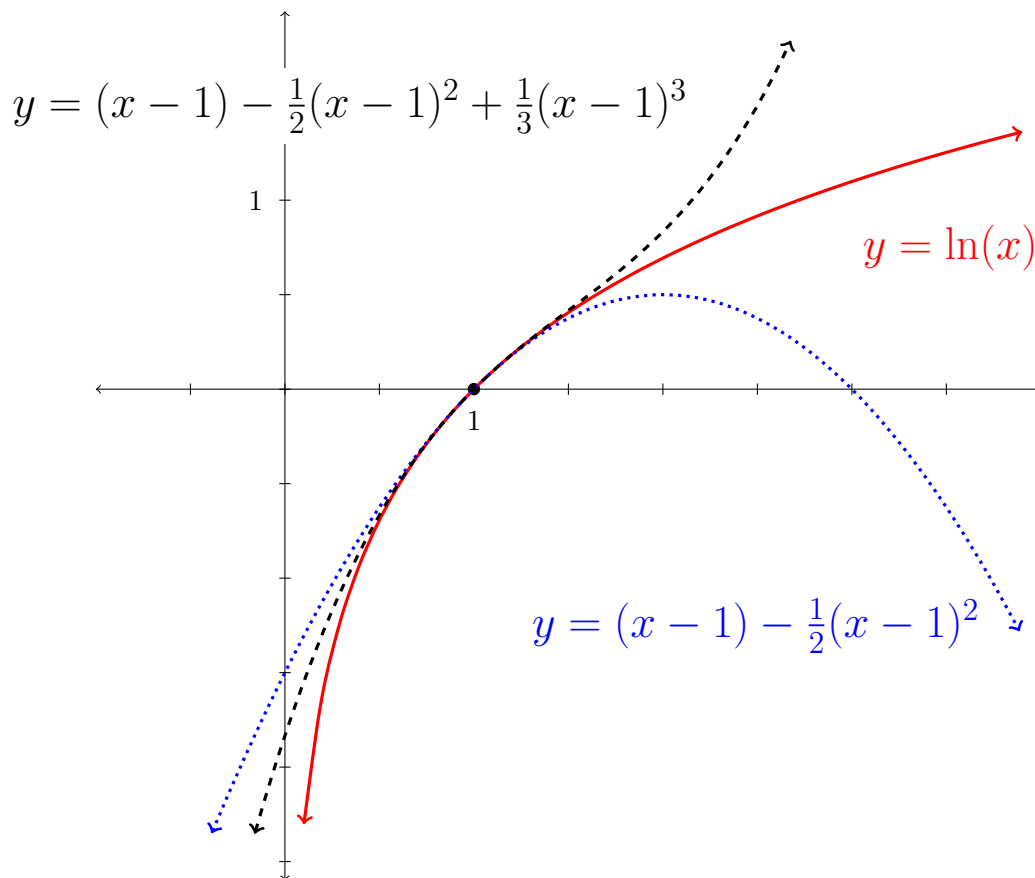
$$P_1(x) = f(1) + f'(1)(x - 1) = x - 1$$

$f''(0) = -1$, so the second Taylor polynomial is

$$P_2(x) = (x - 1) + \frac{f''(1)}{2}(x - 1)^2 = (x - 1) - \frac{1}{2}(x - 1)^2$$

And $f'''(1) = 2$, so the third Taylor polynomial is

$$P_3(x) = (x - 1) - \frac{1}{2}(x - 1)^2 + \frac{f'''(1)}{6}(x - 1)^3 = (x - 1) - \frac{1}{2}(x - 1)^2 + \frac{1}{3}(x - 1)^3$$

**Solution to Exercise 3.7**

Your code should be similar to:

```
from matplotlib.pyplot import *
from numpy import *
from matplotlib.patches import Patch

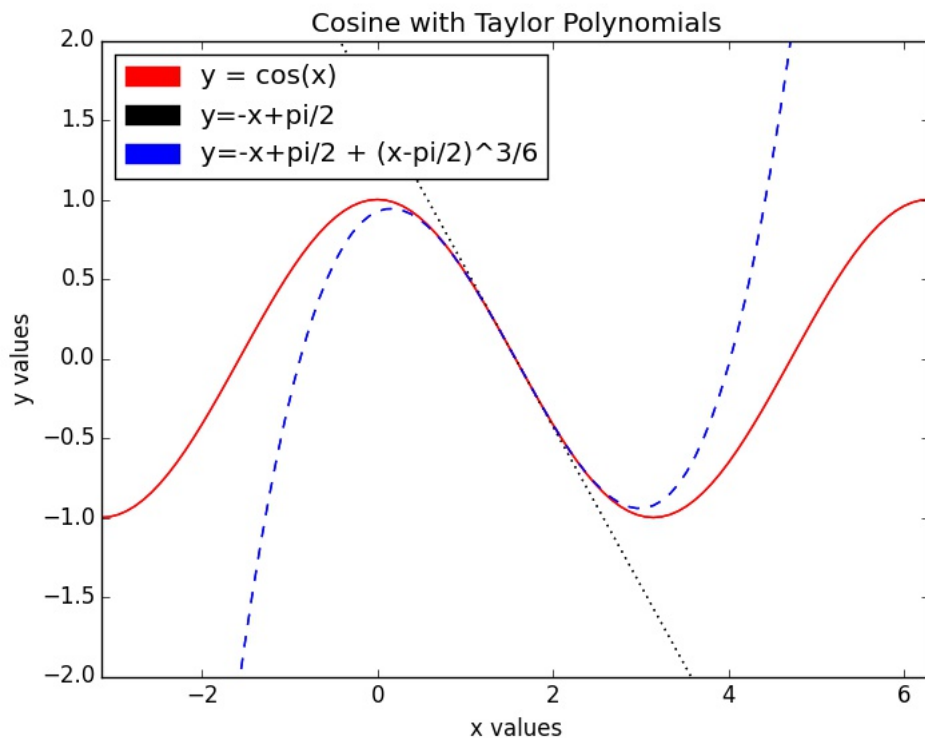
title('Cosine with Taylor Polynomials')
xlabel('x values')
```

```

ylabel('y values')
x = linspace(-pi,2*pi,100)
plot(x,cos(x),color='red')
axis([-pi,2*pi,-2,2])
tp1 = -x+pi/2
plot(x,tp1,color='black',linestyle=':')
tp2 = -x+pi/2 + (x-pi/2)**3/6
plot(x,tp2,color='blue',linestyle='--')
L1 = Patch(color='red',label='y = cos(x)')
L2 = Patch(color='black',label='y=-x+pi/2')
L3 = Patch(color='blue',label='y=-x+pi/2 + (x-pi/2)^3/6')
legend(handles=[L1,L2,L3],loc='upper left')

```

Which should produce:



Solution to Exercise 3.11:

The file `Derivative.py` should look similar to:

```
from math import exp
```

```
def f(x):
    return exp(x)
```

```
def df(x):
```

```

    return exp(x)

def FD(x,h):
    dy = (f(x+h)-f(x))/h
    err = abs(dy-df(x))
    print('h = %4.3f, df = %7.5f, error = %7.5f' % (h,dy,err))
    return dy

```

Now we run the file `Derivative` and write in the console:

```

In : FD(1,0.1)
h = 0.100, df = 2.85884, error = 0.14056
Out: 2.858841954873883

```

Solution to Exercise 3.17:

```

def CD(x,h):
    dy = (f(x+h)-f(x-h))/(2*h)
    err = abs(dy-df(x))
    print('h = %4.3f, df = %7.5f, error = %7.2e' % (h,dy,err))
    return dy

```

Now we run the file `Derivative` and write in the console:

```

In : CD(1,0.1)
h = 0.100, df = 2.72281, error = 4.53e-03
Out: 2.7228145639474177

```

Solution to Exercise 3.14:

Expanding $f(x + 2h)$ gives

$$f(x + 2h) = f(x) + f'(x)(2h) + \frac{1}{2}f''(x)(2h)^2 + O(h^3) = f(x) + 2f'(x)h + 2f''(x)h^2 + O(h^3)$$

Subtracting $4f(x - h)$,

$$\begin{array}{rcl}
 f(x + 2h) & = & f(x) + 2f'(x)h + 2f''(x)h^2 + O(h^3) \\
 -4f(x - h) & = & -4f(x) + 4f'(x)h - 2f''(x)h^2 + O(h^3) \\
 \hline
 f(x + 2h) - 4f(x - h) & = & -3f(x) + 6f'(x)h + 0 + O(h^3)
 \end{array}$$

As in the Central Difference estimate, the f'' term drops out, allowing us to solve for f' in terms of f evaluated at different points. Moving the $f(x)$ term to the other side and solving,

$$f'(x) = \frac{f(x + 2h) + 3f(x) - 4f(x - h)}{6h} + O(h^2)$$

This is called *Richardson's Formula*.

Solution to Exercise 3.15:

Using Richardson's Formula

$$f'(1) \approx \frac{e^{(1+0.1)} + 3e^1 - 4e^{(1-0.1)}}{0.6} \approx 2.72759$$

And the raw error would be $\approx 2.72759 - 2.71828 \approx 0.00930$.

h	f' estimate	error
0.1	2.72759	9.30×10^{-3}
0.01	2.71837	9.08×10^{-5}
0.001	2.71828	9.06×10^{-7}

When h is reduced by one order of magnitude, the error is reduced by approximately two orders of magnitude. This indicates the error is on the order of h^2 .

Solution to Exercise 3.17:

At the top of the file we'll need to define the actual second derivative, as well as change our definitions for `f` and `df`. (We don't really need to change our definition of `df` for just this exercise, but let's do it to be consistent.)

```
def f(x,y):
    return cos(x)

def df(x,y):
    return -sin(x)

def ddf(x,y):
    return -cos(x)
```

Then we use `ddf` to calculate our error.

```
def CDD(x,h):
    ddy = (f(x+h)-2*f(x)+f(x-h))/(h**2)
    err = abs(ddy-ddf(x))
    print('h = %4.3f, ddf = %7.5f, error = %7.2e' % (h,ddy,err))
    return ddy
```

Now we run the file `Derivative` and write in the console:

```
In : CDD(1,0.1)
h = 0.100, df = 2.72055, error = 2.27e-03
Out: 2.720547818529306
```

This result agrees with the table in Example 3.16.

Problem 3.1: Consider $f(x) = x^{\frac{2}{3}}$

- a) Find the first and second Taylor polynomials, p_1 and p_2 , for f with $x_0 = 1$.
- b) Find ξ so that the conclusion of Taylor's Theorem is satisfied for $x_0 = 1$, $x = 1.5$, and $n = 2$.
- c) Use Python to plot f , p_1 , and p_2 . Choose scales on your axes so that the shapes of the different graphs are evident. The graphs should use different linestyles and colors, and there should also be a legend. You need not submit your code, just the final graph.

Problem 3.2:

- a) Use the Central Difference formula to approximate $f'(2)$ for $f(x) = \ln(x)$ and $h = 0.2$ and $h = 0.1$. What sort of reduction in the error do you expect? Why?
- b) Use Taylor's Theorem to show

$$f'(x) = \frac{f(x+2h) - 8f(x+h) + 8f(x-h) - f(x-2h)}{-12h} + O(h^4)$$

- c) Write a short Python program called C8D which takes x and h as arguments and returns the approximation of $f'(x)$ obtained from the rule introduced in part(b). It should also print the error between this estimate and the true value of $f'(x)$.
- d) Use the program from part(c) to approximate $f'(2)$ for $f(x) = \ln(x)$ with $h = 0.2$ and $h = 0.1$.
- e) Compare the errors in the previous two parts.

Problem 3.3: By using Taylor's Theorem (you don't need to check this) one can estimate the second derivative via:

$$f''(x) \approx \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2}$$

- a) Write a short Python program called C16DD which takes x and h as arguments and returns the $f''(x)$. It should also print the error between $f''(x)$ and the approximation of $f''(x)$ obtained from the rule above.
- b) Use your program to approximate $f''(2)$ for $f(x) = \ln(x)$ with $h = 0.1$, $h = 0.01$, and $h = 0.001$.
- c) What is the order of accuracy for this estimate? How do you know?

Chapter 4

Numeric Integration

In this chapter we will use Taylor's Theorem to approximate the value of a definite integral. As in the case of derivative estimates, keeping more terms from the Taylor expansion will give us more complex, but also more accurate formulas.

Let's state the problem more carefully. For a function f with some number of continuous derivatives on an interval $[a, b]$, we wish to approximate $\int_a^b f(t) dt$. All of the methods we'll discuss here begin by dividing the interval $[a, b]$ evenly into n subintervals $[x_i, x_{i+1}]$ (for $0 \leq i \leq n$). Here $x_0 = a, x_n = b$. If we define h as the width of each interval, then $x_{i+1} = x_i + h$ and

$$\int_a^b f(t) dt = \sum_{i=0}^{n-1} \int_{x_i}^{x_i+h} f(t) dt$$

Next we define the *anti-derivative* of f ,

$$F(x) = \int_{x_0}^x f(t) dt$$

From the Fundamental Theorem of Calculus we know that $F'(x) = f(x)$ and

$$\int_{x_i}^{x_i+h} f(t) dt = F(x_i + h) - F(x_i)$$

If f has N derivatives, then F will have $N + 1$ derivatives, and we can write the Taylor expansion of F as

$$F(x_i + h) = F(x_i) + F'(x_i)h + F''(x_i)\frac{h^2}{2} + \dots + F^{(N)}(x_i)\frac{h^N}{N!} + O(h^{N+1})$$

Then since $F' = f$,

$$F(x_i + h) - F(x_i) = f(x_i)h + f'(x_i)\frac{h^2}{2} + \dots + f^{(N-1)}(x_i)\frac{h^N}{N!} + O(h^{N+1})$$

Let's go ahead and write this down as a theorem.

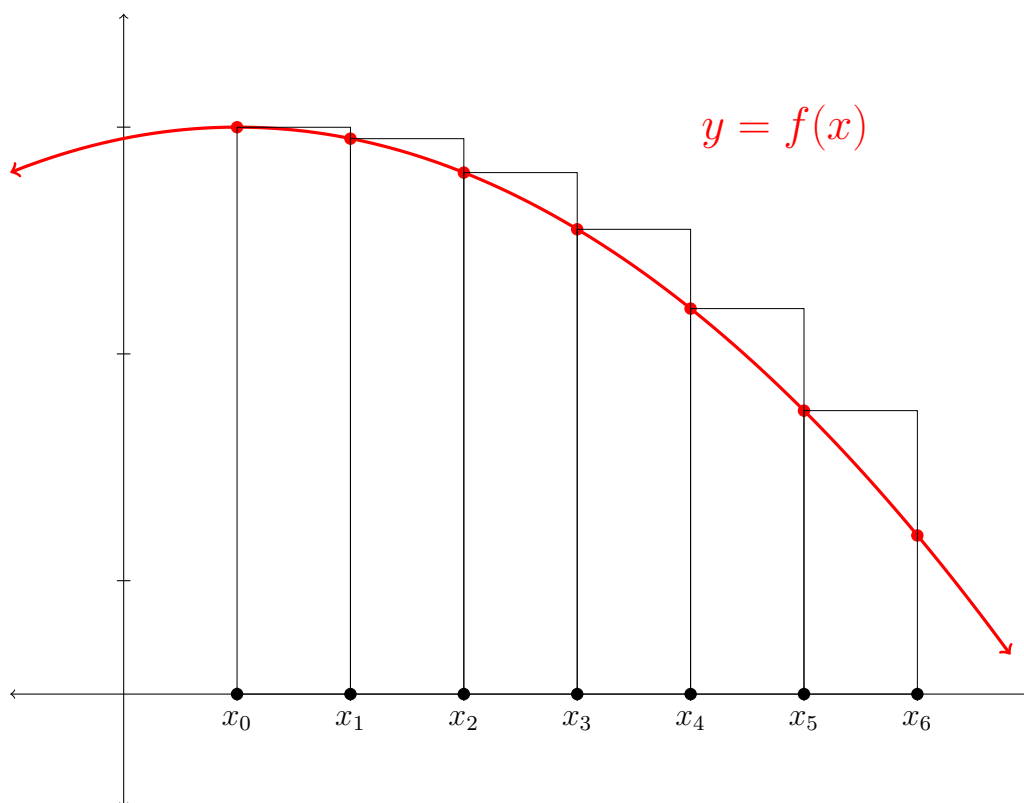
Theorem 4.1: If: f has N continuous derivatives on an interval $[x_i, x_i + h]$,
Then:

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + f'(x_i)\frac{h^2}{2} + \dots + f^{(N-1)}(x_i)\frac{h^N}{N!} + O(h^{N+1})$$

All of these methods will become more accurate as the number of intervals n gets larger (that is, as h gets smaller), but **how much more accurate** will depend on the order of the estimate, N . Different values of N give rise to different methods.

4.1 Rectangle Rule

We begin with the simplest method for approximating the value of a definite integral—rectangles. When we learn integral calculus, the definite integral is introduced as a limit of a sum of the areas of rectangles. This makes sense for a theoretical definition, but it leaves a lot to be desired as a numerical method. Nevertheless it is familiar and a good introduction to how we derive better, more complex methods.



The *Rectangle Rule* simply corresponds to Theorem 4.1 with $N = 1$. This gives

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + O(h^2)$$

Then our estimate of the integral is just,

$$\int_a^b f(t) dt = \sum_{i=0}^{n-1} \left(f(x_i)h + O(h^2) \right)$$

The error terms require some care. We are adding together n errors, each of which is $O(h^2)$. That gives a total error which is $O(nh^2)$. However recall that $h = (b-a)/n \Rightarrow n = (b-a)/h$. Therefore since $n = O(h^{-1})$, $O(nh^2) = O(h)$. That gives us

Theorem 4.2: (Rectangle Rule)

If f has one continuous derivative on $[a, b]$, **then**

$$\int_a^b f(t) dt = h \left(\sum_{i=0}^{n-1} f(x_i) \right) + O(h)$$

Note that each $hf(x_i)$ term just corresponds to the area of a rectangle with height $f(x_i)$ and base h .

Example 4.1: Use the Rectangle Rule to estimate $\int_1^2 \ln(t) dt$. Use $n = 4$ and $n = 8$ intervals. Compare the corresponding errors.

From integral calculus we know the exact solution is

$$\int_1^2 \ln(t) dt = t \ln(t) - t \Big|_{t=1}^2 = 2 \ln(2) - 1 \approx 0.386294$$

Using $n = 4$ intervals gives $h = (2 - 1)/4 = 0.25$. Applying the Rectangle Rule with $n = 4$ gives us

$$\int_1^2 \ln(t) dt \approx (.25) \left(\ln(1) + \ln(1.25) + \ln(1.5) + \ln(1.75) \right) \approx 0.297056$$

Using $n = 8$ intervals gives $h = (2 - 1)/8 = 0.125$, and the Rectangle Rule now gives us

$$\int_1^2 \ln(t) dt \approx (.125) \left(\ln(1) + \ln(1.125) + \ln(1.25) + \dots + \ln(1.875) \right) \approx 0.342322$$

Summarizing the results in a table,

n	h	Value	Error
4	0.250	0.29706	-0.08924
8	0.125	0.34232	-0.04397

Thus dividing h by 2 has the effect of dividing the error by 2 (approximately). This is characteristic of an **order h method**.

Exercise 4.2: Use the Rectangle Rule to estimate $\int_0^{\pi/4} t \cos(t) dt$. Use $n = 6$ and $n = 12$ intervals. Compare the corresponding errors.

Example 4.3: Open a new Python file called `integral.py`. Import `math`, then define `f`, `df`, and `F` to be `log(x)`, `1/x`, and `x*log(x)-x` respectively (the integrand, its derivative, and an anti-derivative).

Then write a short python program called `rect` which takes `a`, `b`, and `n` as arguments and returns the Rectangle Rule approximation to $\int_a^b f(t) dt$ using n intervals. It should also print the error in exponential notation with two significant figures. Check your program by comparing your results to those of Example 4.1.

```
from math import *

def f(x):
    return log(x)

def df(x):
    return 1/x

def F(x):
    return x*log(x) - x

#Rectangle Rule
def rect(a,b,n):
    h = abs(a-b)/n
    area = 0
    #Evaluate f at left endpoints and sum
    x = a
    for k in range(0,n):
        area = area + f(x)
        x = x + h
    area = area*h
    ex = F(b) - F(a)
    print('Error = %3.2e' % (area-ex))
    return area
```

The lines beginning with a `#` are *comments*. They play no part in the execution of the code, but are included to make the code more readable. It is good programming practice to include lots of comments in your code.

We check our program by running the file `integral`, then in the console

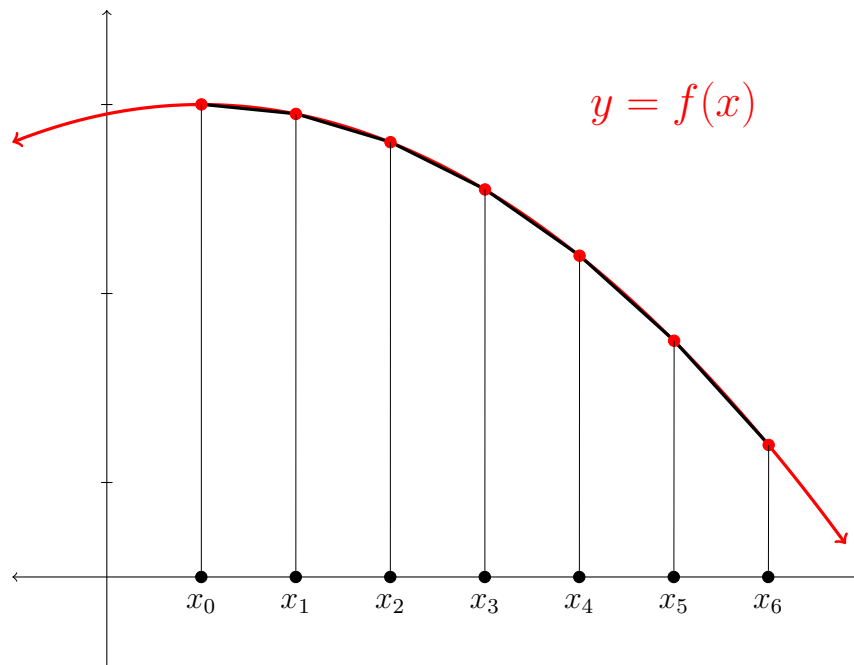
```
In : rect(1,2,4)
Error = -8.92e-02
Out: 0.2970561118394492
```

```
In : rect(1,2,8)
Error = -4.40e-02
Out: 0.3423222111670987
```

The results agree with Example 4.1.

4.2 Trapezoid Rule

The *Trapezoid Rule* is similar, but much more accurate than the Rectangle Rule. Here we approximate the area under the curve as a sum of trapezoids.



We begin constructing the Trapezoid Rule by considering Theorem 4.1 with $N = 2$. This gives

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + f'(x_i)\frac{h^2}{2} + O(h^3)$$

Next we use the Forward Difference Formula (Theorem 3.2) for $f'(x_i)$.

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + \left(\frac{f(x_i+h) - f(x_i)}{h} + O(h) \right) \frac{h^2}{2} + O(h^3)$$

Distributing the h^2 term and combining the $O(h^3)$ terms give us,

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + \frac{h}{2} \left(f(x_i+h) - f(x_i) \right) + O(h^3)$$

We can see at this point why this formula gives the Trapezoid Rule. We are estimating the integral by adding the area of a rectangle $hf(x_i)$ plus the area of a triangle of height $f(x_i+h) - f(x_i)$ and base h . This is the area of a trapezoid.

Simplifying the expression,

$$\int_{x_i}^{x_i+h} f(t) dt = \frac{h}{2} \left(f(x_i+h) + f(x_i) \right) + O(h^3)$$

Then our estimate of the integral is

$$\int_a^b f(t) dt = \sum_{i=0}^{n-1} \left(\frac{h}{2} (f(x_i+h) + f(x_i)) + O(h^3) \right)$$

Again we are adding together n errors, each of which is now $O(h^3)$. That gives a total error which is $O(nh^3) = O(h^2)$ since $n = O(h^{-1})$. That gives us that the Trapezoid Rule is

$$\int_a^b f(t) dt = \frac{h}{2} \left(\sum_{i=0}^{n-1} f(x_i+h) + f(x_i) \right) + O(h^2)$$

We can simplify this a bit more by noticing that all the interior points $(x_1, x_2 \dots x_{n-1})$ appear in the sum **twice**. For instance $f(x_2)$ appears once when $i = 1$ (as the right endpoint) and again when $i = 2$ (as the left endpoint). Only the full interval endpoints, $x_0 = a$ and $x_n = b$ appear once. The final result is then,

Theorem 4.3: (Trapezoid Rule)

If f has two continuous derivatives on $[a, b]$, **then**

$$\int_a^b f(t) dt = \frac{h}{2} \left(f(a) + f(b) \right) + h \sum_{i=1}^{n-1} f(x_i) + O(h^2)$$

Example 4.4: Use the Trapezoid Rule to estimate $\int_1^2 \ln(t) dt$. Use $n = 4$ and $n = 8$ intervals. Compare the corresponding errors.

Applying the Trapezoid Rule with $n = 4$ gives us

$$\int_1^2 \ln(t) dt \approx \frac{0.25}{2} (\ln(1) + \ln(2)) + (0.25) (\ln(1.25) + \ln(1.5) + \ln(1.75)) \approx 0.383700$$

Applying the Trapezoid Rule with $n = 8$ gives us

$$\int_1^2 \ln(t) dt \approx \frac{0.125}{2} (\ln(1) + \ln(2)) + (0.125) (\ln(1.125) + \ln(1.25) + \dots + \ln(1.875)) \approx 0.385644$$

Summarizing the results in a table,

n	h	Value	Error
4	0.250	0.38370	-0.00259
8	0.125	0.38564	-0.00065

Thus dividing h by 2 has the effect of dividing the error by 4 (approximately). This is characteristic of an **order h^2 method**.

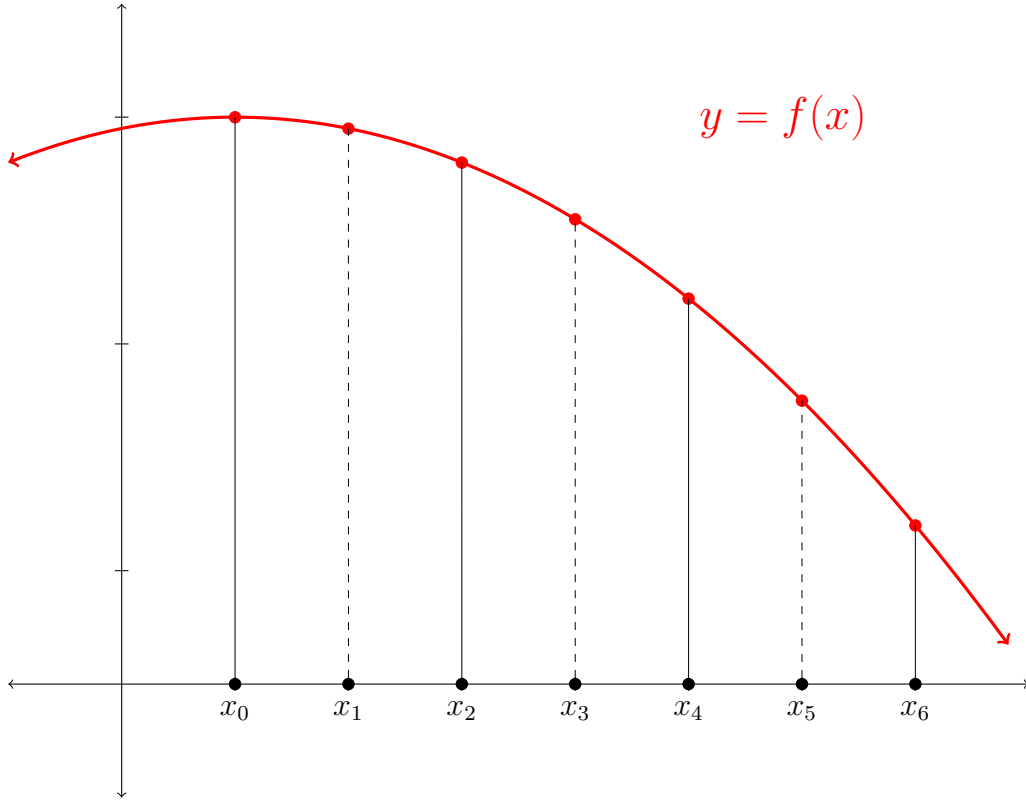
Exercise 4.5: Use the Trapezoid Rule to estimate $\int_0^{\pi/4} t \cos(t) dt$. Use $n = 6$ and $n = 12$ intervals. Compare the corresponding errors.

Exercise 4.6: Add to your Python file `integral.py` by writing a short python program called `trap` which takes `a`, `b`, and `n` as arguments and returns the Trapezoid Rule approximation to $\int_a^b f(t) dt$ using n intervals. It should also print the error in exponential notation with two significant figures. Check your program by comparing your results to those of Example 4.4.

4.3 Simpson's Rule

The next available “rule” would seem to come from Theorem 4.1 with $N = 3$. In fact for *Simpson's Rule* we let $N = 4$.

To implement Simpson's Rule, we divide the interval into pairs of sub-intervals and then approximate the area by topping the sub-intervals with parabolic sections.



Returning to Theorem 4.1 with $N = 4$, we have

$$\int_{x_i}^{x_i+h} f(t) dt = f(x_i)h + f'(x_i)\frac{h^2}{2} + f''(x_i)\frac{h^3}{6} + f'''(x_i)\frac{h^4}{24} + O(h^5)$$

There is a complication. We will need to use the Second Derivative Central Difference formula (Theorem ??) to substitute for the $f''(x_i)$ term. Inconveniently, this formula requires **three** points while the formula above only involves x_i and x_{i+1} .

The solution to this problem is to consider the integral over **two** intervals.

$$\int_{x_i-h}^{x_i+h} f(t) dt = \int_{x_i}^{x_i+h} f(t) dt + \int_{x_i-h}^{x_i} f(t) dt$$

We can apply Theorem 4.1 to the second integral, giving

$$\begin{aligned} \int_{x_i-h}^{x_i} f(t) dt &= - \int_{x_i}^{x_i-h} f(t) dt \\ &= - \left(f(x_i)(-h) + f'(x_i)\frac{(-h)^2}{2} + f''(x_i)\frac{(-h)^3}{6} + f'''(x_i)\frac{(-h)^4}{24} + O(-h)^5 \right) \\ &= f(x_i)h - f'(x_i)\frac{h^2}{2} + f''(x_i)\frac{h^3}{6} - f'''(x_i)\frac{h^4}{24} + O(h^5) \end{aligned}$$

Simplifying and combining the expressions for the two integrals together we get a formula for the integral over both intervals.

$$\begin{array}{rcl} \int_{x_i}^{x_i+h} f(t) dt & = & f(x_i)h + f'(x_i)\frac{h^2}{2} + f''(x_i)\frac{h^3}{6} + f'''(x_i)\frac{h^4}{24} + O(h^5) \\ + \int_{x_i-h}^{x_i} f(t) dt & = & f(x_i)h - f'(x_i)\frac{h^2}{2} + f''(x_i)\frac{h^3}{6} - f'''(x_i)\frac{h^4}{24} + O(h^5) \\ \hline \int_{x_i-h}^{x_i+h} f(t) dt & = & 2f(x_i)h + 0 + f''(x_i)\frac{h^3}{3} + 0 + O(h^5) \end{array}$$

Happily the first and third derivative terms dropped out, so we need only use Theorem ?? for the $f''(x_i)$ term.

$$\int_{x_i-h}^{x_i+h} f(t) dt = 2f(x_i)h + \left(\frac{f(x_i+h) - 2f(x_i) + f(x_i-h)}{h^2} + O(h^2) \right) \frac{h^3}{3} + O(h^5)$$

Distributing the h^3 factor and combining the $O(h^5)$ terms gives us, (with some simplification)

$$\int_{x_i-h}^{x_i+h} f(t) dt = \frac{h}{3} \left(f(x_i+h) + 4f(x_i) + f(x_i-h) \right) + O(h^5)$$

Now, since Simpson's rule is based on **pairs** of intervals we must have an **even** number of intervals in order to apply it. Further, when we add all the intervals together to get the full integral, we have to add them in pairs. That is, the first interval we add will be from x_0 to x_2 . The second from x_2 to x_4 , etc. The even numbered points represent the endpoints of these pairs of intervals while the odd numbered points represent the midpoints of these intervals. Therefore we may write the total integral by summing over the **odd** numbered points.

$$\begin{aligned} \int_a^b f(t) dt &= \sum_{k=1}^{n/2} \int_{x_{2k-1}-h}^{x_{2k-1}+h} f(t) dt \\ &= \sum_{k=1}^{n/2} \frac{h}{3} \left(f(x_{2k-1}+h) + 4f(x_{2k-1}) + f(x_{2k-1}-h) \right) + O(h^5) \end{aligned}$$

In this sum we can see that each $f(x_i)$ makes a different contribution depending on whether the point x_i is a **midpoint** or an **endpoint**. The midpoints are multiplied by four, while the endpoints are not. Further the **interior midpoints** ($x_2, x_4 \dots x_{n-2}$) are counted **twice** (as in the sum for the Trapezoid Rule) because they appear in two different terms in the sum (once as the left endpoint and once as the right).

Also like our previous theorems, the summing of the error terms reduces the order of the estimate by one factor of h . That is, the total error is $O(nh^5) = O(h^4)$.

Taken all together, this gives us

Theorem 4.4: (Simpson's Rule)

If f has four continuous derivatives on $[a, b]$ and n is even, **then**

$$\int_a^b f(t) dt = \frac{h}{3} \left(f(a) + f(b) + 4 \sum_{k=1}^{n/2} f(x_{2k-1}) + 2 \sum_{k=1}^{n/2-1} f(x_{2k}) \right) + O(h^4)$$

Example 4.7: Use Simpson’s Rule to estimate $\int_1^2 \ln(t) dt$. Use $n = 4$ and $n = 8$ intervals. Compare the corresponding errors.

To apply Simpson’s Rule we need only substitute into the formula, but for the $n = 4$ case let us include some “unnecessary” details to reinforce where the formula comes from.

Four intervals means we are summing two pairs of integrals, then applying the formula above to each a pair.

$$\begin{aligned} \int_1^2 \ln(t) dt &= \int_1^{1.5} \ln(t) dt + \int_{1.5}^2 \ln(t) dt \\ &\approx \frac{0.25}{3} \left(\ln(1) + 4 \ln(1.25) + \ln(1.5) \right) + \frac{0.25}{3} \left(\ln(1.5) + 4 \ln(1.75) + \ln(2) \right) \end{aligned}$$

We can rearrange the terms to get the formula from Theorem 4.7.

$$\int_1^2 \ln(t) dt \approx \frac{0.25}{3} \left(\ln(1) + \ln(2) + 4(\ln(1.25) + \ln(1.75)) + 2\ln(1.5) \right) \approx 0.38625956$$

Applying Simpson’s Rule with $n = 8$ (and omitting the unnecessary details) gives us

$$\begin{aligned} \int_1^2 \ln(t) dt &\approx \frac{0.125}{3} \left(\ln(1) + \ln(2) + 4(\ln(1.125) + \ln(1.375) + \ln(1.625) + \ln(1.875)) \right. \\ &\quad \left. + 2(\ln(1.25) + \ln(1.5) + \ln(1.75)) \right) \\ &\approx 0.38629204 \end{aligned}$$

Summarizing the results in a table,

n	h	Value	Error
4	0.250	0.386260	-3.48e-05
8	0.125	0.386292	-2.32e-06

Thus dividing h by 2 has the effect of dividing the error by 16 (approximately). This is characteristic of an **order h^4 method**.

Exercise 4.8: Use Simpson’s Rule to estimate $\int_0^{\pi/4} t \cos(t) dt$. Use $n = 6$ and $n = 12$ intervals. Compare the corresponding errors.

Exercise 4.9: Add to your Python file `integral.py` by writing a short python program called `simp` which takes `a, b`, and `n` as arguments and returns the Simpson’s Rule approximation to $\int_a^b f(t) dt$ using n intervals. It should also print the error in exponential notation with two significant figures. Check your program by comparing your results to those of Example 4.7.

4.4 Romberg Integration

We could continue to generate new methods by applying Theorem 4.1 with higher and higher values for N , but as our derivation of Simpson's Rule suggested, the derivations get more and more complicated. We would like to come up with a method that can be generalized to high accuracy without involved calculations for each value of N .

To do this, we'll now take a somewhat different approach and look closely at the Trapezoid Rule. We saw earlier that the Trapezoid Rule has an error that is $O(h^2)$, but a more careful analysis can tell us much more about the exact form of this error. We leave the details to Appendix A.3, but if we re-derive the Trapezoid Rule keeping track of more error terms we produce

Theorem 4.5: (Improved Trapezoid Rule)

If f has four continuous derivatives on $[a, b]$, then

$$\int_a^b f(t) dt = \frac{h}{2} (f(a) + f(b)) + h \sum_{i=1}^{n-1} f(x_i) + \left(\frac{f'(a) - f'(b)}{12} \right) h^2 + O(h^4)$$

The punchline of this theorem is that the $O(h^2)$ error can be written explicitly as a constant involving f' , but **independent of the number of intervals n** . What remains after correcting for this error is much smaller—on the order of h^4 .

Example 4.10: Use the Improved Trapezoid Rule to estimate $\int_1^2 \ln(t) dt$. Use $n = 4$ and $n = 8$ intervals. Compare the corresponding errors.

The “improved” version of the Trapezoid Rule is just the old version plus a correction of the form:

$$\frac{\ln'(1) - \ln'(2)}{12} (0.1)^2 = \left(\frac{1}{1} - \frac{1}{2} \right) \frac{0.01}{12} \approx 0.00041667$$

The improved result is then

$$\begin{aligned} \int_1^2 \ln(t) dt &\approx \frac{0.1}{2} (\ln(1) + \ln(2)) + 0.1 (\ln(1.25) + \ln(1.5) + \ln(1.75)) + \frac{\ln'(1) - \ln'(2)}{12} (0.1)^2 \\ &\approx 0.3863037 \end{aligned}$$

Once again summarizing our results in a table,

n	h	Value	Error
4	0.250	0.3863037	9.31e-06
8	0.125	0.3862950	5.90e-07

Thus dividing h by 2 has the effect of dividing the error by 16 (approximately). Again this is characteristic of an **order h^4 method**.

In fact, at least in this case, the raw errors are smaller than those obtained for Simpson's Rule. The point here, though, is not to come up with yet another rule that is accurate to

some order of h . Rather we'd like to make use of the fact that the h^2 portion of the error in the Trapezoid Rule is some constant times h^2 **without having to actually calculate the constant**. This can be accomplished in a clever way known as *Romberg Integration*.

In it's simplest form, Romberg Integration uses the results of the Trapezoid Rule for h and $h/2$ to obtain a $O(h^4)$ estimate. Let T_n be the Trapezoid Rule approximation of some integral with n intervals. We've just learned that

$$\int_a^b f(t) dt = T_n + c_2 h^2 + O(h^4)$$

where $c_2 = (f'(a) - f'(b))/12$. Now if we double the number of intervals, n , we halve h , so

$$\int_a^b f(t) dt = T_{2n} + c_2 \left(\frac{h}{2}\right)^2 + O(h^4) = T_{2n} + \frac{1}{4}c_2 h^2 + O(h^4)$$

Taking four times the second formula and subtracting the first, we get

$$\begin{array}{rcl} 4 \int_a^b f(t) dt & = & 4T_{2n} + c_2 h^2 + O(h^4) \\ - \int_a^b f(t) dt & = & -T_n - c_2 h^2 - O(h^4) \\ \hline 3 \int_a^b f(t) dt & = & 4T_{2n} - T_n + 0 + O(h^4) \end{array}$$

That gives us

$$\int_a^b f(t) dt = \frac{4T_{2n} - T_n}{3} + O(h^4)$$

Example 4.11: Use the formula above to estimate $\int_1^2 \ln(t) dt$ for $n = 2$ and $n = 4$. Compare the errors.

For $n = 2$ we first need the two interval Trapezoid Rule.

$$T_2 = \frac{0.5}{2}(\ln(1) + \ln(2)) + 0.5 \ln(1.5) \approx 0.376019$$

We have from Example 4.4 that $T_4 \approx 0.383700$ while $T_8 \approx 0.385644$. Using the Romberg formula for $n = 2$,

$$\int_1^2 \ln(t) dt \approx \frac{4(0.383700) - (0.376019)}{3} \approx 0.386260$$

which has a raw error of -3.48×10^{-5} .

Using the Romberg formula for $n = 4$,

$$\int_1^2 \ln(t) dt \approx \frac{4(0.385644) - (0.383700)}{3} \approx 0.386292$$

which has a raw error of -2.32×10^{-6} . Doubling n reduced h by half and had the effect of reducing the error by approximately a factor of 16. This is again consistent with an error that is $O(h^4)$.

But there is still more to Romberg Integration. It can be shown that

$$\int_a^b f(t) dt = T_n + c_2 h^2 + c_4 h^4 + \dots + c_{2N} h^{2N} + O(h^{2N+2})$$

where all of the constants c_{2k} are independent of the number of intervals n . We've just seen how taking linear combinations of trapezoid rule results can be used to eliminate the h^2 terms. In fact you can take further linear combinations of those results to eliminate the h^4 terms and h^6 terms and so on.

To introduce the full Romberg Algorithm we need some notation. Let the “zero-th” Romberg estimate just be the Trapezoid Rule estimate for some number of intervals which is a power of two. We'll write this as $R_{0,n} = T_{2^n}$. So $R_{0,0} = T_1$ while $R_{0,1} = T_2$ and $R_{0,2} = T_4$ etc. The higher order Romberg estimates are defined to be linear combinations of lower order Romberg estimates which eliminate the lowest power of h in the error. For instance, we already saw that the linear combination that eliminated the h^2 portion of the error was

$$R_{1,n} = \frac{4R_{0,n} - R_{0,n-1}}{3}$$

This new estimate has an h^4 error. That is,

$$\int_a^b f(t) dt = R_{1,n} + \bar{c}_4 h^4 + O(h^6)$$

As before we can compare the results for some number of intervals and twice that number of intervals (that is, n and $n+1$ since the number of intervals is 2^n). Then,

$$\int_a^b f(t) dt = R_{1,n+1} + \bar{c}_4 \left(\frac{h}{2}\right)^4 + O(h^6) = R_{1,n+1} + \frac{1}{16} \bar{c}_4 h^4 + O(h^6)$$

Using a technique similar to our calculation above to remove the h^4 term,

$$\begin{array}{rcl} 16 \int_a^b f(t) dt & = & 16R_{1,n+1} - \bar{c}_4 h^4 + O(h^6) \\ - \int_a^b f(t) dt & = & -R_{1,n} - \bar{c}_4 h^4 - O(h^6) \\ \hline 15 \int_a^b f(t) dt & = & 16R_{1,n+1} - R_{1,n} + 0 + O(h^6) \end{array}$$

That gives us the next level of Romberg estimates,

$$\int_a^b f(t) dt = \frac{16R_{1,n+1} - R_{1,n}}{15} + O(h^6)$$

Now we define

$$R_{2,n} = \frac{16R_{1,n} - R_{1,n-1}}{15}$$

and have the estimate

$$\int_a^b f(t) dt = R_{2,n} + \tilde{c}_6 h^6 + O(h^8)$$

In general we have

Theorem 4.6: (Romberg Integration)

If: f has $2m + 4$ continuous derivatives, and the numbers $R_{m,n}$ are defined recursively so that

$$R_{0,n} = T_{2^n}, \quad \text{and} \quad R_{m,n} = \frac{4^m R_{m-1,n} - R_{m-1,n-1}}{4^m - 1}$$

then for $h = (b-a)/2^n$ and some constant c_{2m+2} (depending on the derivatives of f , but independent of n)

$$\int_a^b f(t) dt = R_{m,n} + c_{2m+2} h^{2m+2} + O(h^{2m+4})$$

All these subscripts and estimates can become confusing. One way to organize them is to make a table.

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	$R_{0,0}$	-	-	-
1	2	$R_{0,1}$	$R_{1,1}$	-	-
2	4	$R_{0,2}$	$R_{1,2}$	$R_{2,2}$	-
3	8	$R_{0,3}$	$R_{1,3}$	$R_{2,3}$	$R_{3,3}$

Since higher order R 's are based on lower order R 's, we fill out this table by calculating each column starting from the left.

Example 4.12: Find the Romberg estimate $R_{3,3}$ for $\int_1^2 \ln(t) dt$.

We have already calculated many of the entries in the table. We know $R_{0,2}$ and $R_{0,3}$ from Example 4.4. We have $R_{0,1}$, $R_{1,2}$, and $R_{1,3}$ from Example 4.11. The table starts as

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	$R_{0,0}$	-	-	-
1	2	0.376019	$R_{1,1}$	-	-
2	4	0.383700	0.386260	$R_{2,2}$	-
3	8	0.385644	0.386292	$R_{2,3}$	$R_{3,3}$

$$R_{0,0} = \frac{1}{2}(\ln(1) + \ln(2)) \approx 0.346574$$

$$R_{1,1} = \frac{4(0.376019) - (0.346574)}{3} \approx 0.385834$$

Updating the table,

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	0.346574	-	-	-
1	2	0.376019	0.385834	-	-
2	4	0.383700	0.386260	$R_{2,2}$	-
3	8	0.385644	0.386292	$R_{2,3}$	$R_{3,3}$

Now we proceed to the third column,

$$R_{2,2} = \frac{16(0.386260) - (0.385834)}{15} \approx 0.386288$$

$$R_{2,3} = \frac{16(0.386292) - (0.386260)}{15} \approx 0.386294$$

And finally the fourth,

$$R_{3,3} = \frac{64(0.386294) - (0.386288)}{63} \approx 0.386294$$

So the completed table is

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	0.346574	-	-	-
1	2	0.376019	0.385834	-	-
2	4	0.383700	0.386260	0.386288	-
3	8	0.385644	0.386292	0.386294	0.386294

We did not keep enough decimal places to see the difference between them, but the raw error for $R_{2,3}$ was -1.52×10^{-7} , while the raw error for $R_{3,3}$ was -5.20×10^{-8} . It is a bit awkward to compare $R_{2,3}$ with $R_{3,3}$ because both the number of intervals and the orders of estimate are different. It suffices to say that the lower-right corner will always contain the most accurate estimate.

Exercise 4.13: Use a table to find the $R_{3,3}$ Romberg approximation to the integral $\int_0^{\pi/4} t \cos(t) dt$. Compare with Simpson's rule applied using eight intervals (the same as $R_{3,3}$).

4.5 Exercise Solutions and Problems

Solution to Exercise 4.2:

For $n = 6$, $h = (\pi/4 - 0)/6 = \pi/24$. Then

$$\begin{aligned} \int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi}{24} \left(0 \cos(0) + \frac{\pi}{24} \cos\left(\frac{\pi}{24}\right) + \frac{2\pi}{24} \cos\left(\frac{2\pi}{24}\right) + \dots + \frac{5\pi}{24} \cos\left(\frac{5\pi}{24}\right) \right) \\ &\approx 0.2249071 \end{aligned}$$

For $n = 12$, $h = (\pi/4 - 0)/12 = \pi/48$. Then

$$\begin{aligned}\int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi}{48} \left(0 \cos(0) + \frac{\pi}{48} \cos\left(\frac{\pi}{48}\right) + \frac{2\pi}{48} \cos\left(\frac{2\pi}{48}\right) + \dots + \frac{11\pi}{48} \cos\left(\frac{11\pi}{48}\right) \right) \\ &\approx 0.2439902\end{aligned}$$

Integrating By-Parts gives the exact answer,

$$\int_0^{\pi/4} t \cos(t) dt = t \sin(t) + \cos(t) \Big|_0^{\pi/4} \approx 0.2624671$$

Thus the raw error for $n = 6$ is approximately -3.76×10^{-2} , while the raw error for $n = 12$ is approximately -1.85×10^{-2} . The error for $n = 12$ is about about half the error for $n = 6$ which is what you would expect from a $O(h)$ method.

Solution to Exercise 4.5:

For $n = 6$, $h = \pi/24$. Then

$$\begin{aligned}\int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi/24}{2} \left(0 \cos(0) + \frac{\pi}{4} \cos\left(\frac{\pi}{4}\right) \right) \\ &\quad + \frac{\pi}{24} \left(\frac{\pi}{24} \cos\left(\frac{\pi}{24}\right) + \frac{2\pi}{24} \cos\left(\frac{2\pi}{24}\right) + \dots + \frac{5\pi}{24} \cos\left(\frac{5\pi}{24}\right) \right) \\ &\approx 0.2612553\end{aligned}$$

For $n = 12$, $h = \pi/48$. Then

$$\begin{aligned}\int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi/48}{2} \left(0 \cos(0) + \frac{\pi}{4} \cos\left(\frac{\pi}{4}\right) \right) \\ &\quad + \frac{\pi}{48} \left(\frac{\pi}{48} \cos\left(\frac{\pi}{48}\right) + \frac{2\pi}{48} \cos\left(\frac{2\pi}{48}\right) + \dots + \frac{11\pi}{48} \cos\left(\frac{11\pi}{48}\right) \right) \\ &\approx 0.2621643\end{aligned}$$

Now the raw error for $n = 6$ is approximately -1.21×10^{-3} , while the raw error for $n = 12$ is approximately -3.03×10^{-4} . The error for $n = 12$ is about about a quarter the error for $n = 6$ which is what you would expect from a $O(h^2)$ method.

Solution to Exercise 4.6:

```
#Trapezoid Rule
def trap(a,b,n):
    h = abs(a-b)/n
    #Exterior endpoints only count once
    area = (f(a) + f(b))/2
    #Interior endpoints count twice
    x = a
```

```

for k in range(1,n):
    x = x + h
    area = area + f(x)
area = h*area
ex = F(b) - F(a)
print('Error = %3.2e' % (area-ex))
return area

```

Checking,

```

In : trap(1,2,4)
Error = -2.59e-03
Out: 0.38369950940944236

```

```

In : trap(1,2,8)
Error = -6.50e-04
Out: 0.3856439099520953

```

Solution to Exercise 4.8:

For $n = 6$, $h = \pi/24$. Then

$$\begin{aligned}
 \int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi/24}{3} \left(0 \cos(0) + \frac{\pi}{4} \cos\left(\frac{\pi}{4}\right) \right) \\
 &\quad + \frac{\pi/24}{3} \left(4 \cdot \frac{\pi}{24} \cos\left(\frac{\pi}{24}\right) + 4 \cdot \frac{3\pi}{24} \cos\left(\frac{3\pi}{24}\right) + 4 \cdot \frac{5\pi}{24} \cos\left(\frac{5\pi}{24}\right) \right) \\
 &\quad + \frac{\pi/24}{3} \left(2 \cdot \frac{2\pi}{24} \cos\left(\frac{2\pi}{24}\right) + 2 \cdot \frac{4\pi}{24} \cos\left(\frac{4\pi}{24}\right) \right) \\
 &\approx 0.2624695
 \end{aligned}$$

For $n = 12$, $h = \pi/48$. Then

$$\begin{aligned}
 \int_0^{\pi/4} t \cos(t) dt &\approx \frac{\pi/48}{3} \left(0 \cos(0) + \frac{\pi}{4} \cos\left(\frac{\pi}{4}\right) \right) \\
 &\quad + \frac{\pi/48}{3} \left(4 \cdot \frac{\pi}{48} \cos\left(\frac{\pi}{48}\right) + 4 \cdot \frac{3\pi}{48} \cos\left(\frac{3\pi}{48}\right) + \dots + 4 \cdot \frac{11\pi}{48} \cos\left(\frac{11\pi}{48}\right) \right) \\
 &\quad + \frac{\pi/48}{3} \left(2 \cdot \frac{2\pi}{48} \cos\left(\frac{2\pi}{48}\right) + 2 \cdot \frac{4\pi}{48} \cos\left(\frac{4\pi}{48}\right) + \dots + 2 \cdot \frac{10\pi}{48} \cos\left(\frac{10\pi}{48}\right) \right) \\
 &\approx 0.2624673
 \end{aligned}$$

Now the raw error for $n = 6$ is approximately 2.35×10^{-6} , while the raw error for $n = 12$ is approximately 1.46×10^{-7} . The error for $n = 12$ is about one sixteenth the error for $n = 6$ which is what you would expect from a $O(h^4)$ method.

Solution to Exercise 4.9:

```
#Simpson's Rule
def simp(a,b,n):
    h = abs(a-b)/n
    #Exterior endpoints only count once
    area = f(a) + f(b)
    #Midpoints are weighted by a factor of 4
    x = a+h
    for k in range(0,n//2):
        area = area + 4*f(x)
        x = x + 2*h
    #Interior endpoints count twice
    x = a+2*h
    for k in range(1,n//2):
        area = area + 2*f(x)
        x = x + 2*h
    area = area*h/3
    ex = F(b) - F(a)
    print('Error = %3.2e' % (area-ex))
    return area
```

Checking,

```
In : simp(1,2,4)
Error = -3.48e-05
Out: 0.38625956281456697
```

```
In : simp(1,2,8)
Error = -2.32e-06
Out: 0.3862920434663129
```

Solution to Exercise 4.13:

Applying the Trapezoid Rule for 1, 2, 4, and 8 intervals completes the first column.

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	0.2180895	-	-	-
1	2	0.2515186	$R_{1,1}$	-	-
2	4	0.2597389	$R_{1,2}$	$R_{2,2}$	-
3	8	0.2617857	$R_{1,3}$	$R_{2,3}$	$R_{3,3}$

$$R_{1,1} = \frac{4R_{0,1} - R_{0,0}}{3}, \quad R_{1,2} = \frac{4R_{0,2} - R_{0,1}}{3}, \quad R_{1,3} = \frac{4R_{0,3} - R_{0,2}}{3}$$

gives us the second column

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	0.2180895	-	-	-
1	2	0.2515186	0.2626616	-	-
2	4	0.2597389	0.2624791	$R_{2,2}$	-
3	8	0.2617857	0.2624679	$R_{2,3}$	$R_{3,3}$

$$R_{2,2} = \frac{16R_{1,2} - R_{1,1}}{15}, \quad R_{2,3} = \frac{16R_{1,3} - R_{1,2}}{15}$$

gives us the third column, and

$$R_{3,3} = \frac{64R_{2,3} - R_{2,2}}{63}$$

gives the fourth

n	2^n	$R_{0,n}$	$R_{1,n}$	$R_{2,n}$	$R_{3,n}$
0	1	0.2180895	-	-	-
1	2	0.2515186	0.2626616	-	-
2	4	0.2597389	0.2624791	0.2624669	-
3	8	0.2617857	0.2624679	0.2624671	0.2624671

The raw error for $R_{2,3}$ was -3.85×10^{-9} while the raw error for $R_{3,3}$ was 7.81×10^{-11} . For comparison, Simpson's Rule using eight intervals produced a raw error of 7.41×10^{-7} —clearly inferior to $R_{3,3}$ though it uses the same number of intervals.

Problem 4.1: By hand, showing your work, use six intervals to approximate the integral

$$\int_0^{\pi} \sin(x) dx$$

- a) using the Trapezoidal rule
- b) using Simpson's rule
- c) using the Improved Trapezoidal rule
- d) Compare the errors of these methods.

Problem 4.2: Add to your Python file `integral.py` by writing a short python program called `imtrap` which takes `a`, `b`, and `n` as arguments and returns the Improved Trapezoid Rule approximation to $\int_a^b f(t) dt$ using n intervals. It should also print the error in exponential notation with two significant figures. Check your program by comparing your results to those of Example 4.10.

Problem 4.3: Consider the integral

$$\int_0^2 5x^6 - 12x^5 + 30x^3 - 90x^2 + 4x + 1 dx$$

- a) Use `trap` to apply the Trapezoid Rule to estimate this integral for $n = 10$, $n = 100$, and $n = 1000$.
- b) What is the order of the estimate? How do you know?
- c) Is this behavior surprising? Do you have an explanation? (*Hint:* Consider the Improved Trapezoid Rule)

Problem 4.4: Use a table to find the $R_{3,3}$ Romberg approximation to the integral $\int_1^4 \sqrt{t} dt$. Compare with Simpson's rule applied using eight intervals (the same as $R_{3,3}$).

Problem 4.5: There is a method for estimating a definite integral similar to Simpson's rule which we will call *Paul's Peculiar Rule*. It requires that we divide the interval into a number of subintervals divisible by **four**. Then,

$$\begin{aligned} \int_a^b f(x) dx \approx & \frac{2h}{9} \left(-f(a) - f(b) + 16 \sum_{k=0}^{n/4-1} f(x_{4k+1}) \right. \\ & \left. - 12 \sum_{k=0}^{n/4-1} f(x_{4k+2}) + 16 \sum_{k=0}^{n/4-1} f(x_{4k+3}) - 2 \sum_{k=1}^{n/4-1} f(x_{4k}) \right) \end{aligned}$$

- a) Add to your Python file `integral.py` by writing a short python program called `PPR` which takes `a`, `b`, and `n` as arguments and estimates $\int_a^b f(x)dx$ according to Paul's Peculiar Rule using `n` subintervals.
- b) Estimate $\int_1^4 \ln(x) dx$ using the Trapezoidal Rule, Simpson's Rule, and Paul's Peculiar Rule for $n = 12$, $n = 120$, and $n = 1200$. Compare the methods.
- c) What is the order of accuracy of Paul's Peculiar Rule? How do you know?

Problem 4.6: There is another type of method for estimating definite integrals called **Gauss Quadrature**. There are different formulas depending on the number of points used, but the strength of the method is that you get a pretty good estimate with only a **very small number of points**.

For **two points** the formula is:

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

For **three points** the formula is:

$$\int_{-1}^1 f(x) dx \approx \frac{5}{9}f\left(-\sqrt{\frac{3}{5}}\right) + \frac{8}{9}f(0) + \frac{5}{9}f\left(\sqrt{\frac{3}{5}}\right)$$

- a) By hand estimate $\int_{-1}^1 e^x dx$ using the two point and three point formulas. Compare the errors.
- b) Write a **Python** program called `gaussQuad` which takes as its argument `n`, and estimates $\int_{-1}^1 f(x) dx$ using the two point formula if $n = 2$, and the three point formula if $n = 3$.
- c) Test your program by using $f(x) = e^x$. Then use it to estimate

$$\int_{-1}^1 x^5 + 5x^4 - 2x + 1 dx$$

Again find the error resulting from both formulas.

- d) For $\int_{-1}^1 e^x dx$ find the size of the h needed so that the Trapezoid rule matches the error produced by the two point formula. Repeat for the three point formula.

Chapter 5

Initial Value Problems

In this chapter we want to apply Taylor's Theorem to the problem of estimating the solution to a first order ordinary differential equation (ODE) given a starting value. These types of problems are known as *Initial Value Problems* (IVP). The more general problem of estimating the solution to a higher order differential equation with more initial conditions will be addressed in later chapters.

These problems are spiritually similar to the definite integrals estimated in chapter 4 in that we are trying, in some way, to find an *anti-derivative* of some given function. However, in the previous chapter our answer was simply a number—the value of the definite integral. For problems in this chapter our answer will be a **function**...or at least a set of points approximating the graph of a function.

Another similarity to calculating integrals is that, while we spent whole classes learning techniques to solve them analytically, the sad fact is that many (if not most) interesting “real world” differential equations have **no analytic solution**. That is, the only way to learn about their solution is through some sort of numerical scheme.

Nevertheless, let's begin by reviewing briefly a method for solving an IVP called *separation*.

Example 5.1: Solve the IVP,

$$\frac{dy}{dt} = -2y, \quad y(0) = 3$$

The equation is *separable*, meaning that all the y variables may be put on one side of the equation, while all the t variables may be put on the other. We then integrate both sides.

$$\begin{aligned} \frac{dy}{y} &= -2dt \\ \int \frac{dy}{y} &= \int -2dt + C \\ \Rightarrow \ln|y| &= -2t + C \end{aligned}$$

where C is an arbitrary constant of integration. We solve for y giving

$$y(t) = Ae^{-2t}$$

where $A = \pm e^C$, but since C is arbitrary, so is A . We then use the initial value to solve for A (and forget about C).

$$\begin{aligned} y(0) &= Ae^{-2(0)} \\ \Rightarrow 3 &= A \end{aligned}$$

Thus the solution to this IVP is

$$y(t) = 3e^{-2t}$$

We'll use this result to estimate the accuracy of our numerical methods.

Exercise 5.2: Solve the IVP,

$$\frac{dy}{dt} = ty^2, \quad y(1) = 1$$

As in chapter 4, our general approach will be the same for each of our specific methods. We will look for an approximation to the function y , a solution to

$$\frac{dy}{dt} = f(t, y(t)), \quad y(t_0) = y_0$$

Further, we will only try to approximate the solution on some fixed interval $[t_0, t_n]$. We begin by dividing this interval into n sub-intervals, $[t_i, t_{i+1}]$, each of width $h = (t_n - t_0)/n$. Then we attempt to approximate the value of the solution, $y(t)$, on the endpoints of these sub-intervals. That is, we want to find values y_i satisfying $y_i \approx y(t_i)$ for $i = 1, 2, \dots, n$.

Again Taylor's Theorem is the key, and the more terms from Taylor's Theorem that we keep, the more accurate will be our method. Applying Taylor's Theorem and substituting f for y' ,

Theorem 5.1:

$$\begin{aligned} y(t_i + h) &= y(t_i) + y'(t_i)h + y''(t_i)\frac{h^2}{2} + \dots + y^{(N)}(t_i)\frac{h^N}{N!} + O(h^{N+1}) \\ &= y(t_i) + f(t_i, y(t_i))h + \frac{d}{dt}[f(t_i, y(t_i))]\frac{h^2}{2} + \dots + \frac{d^{N-1}}{dt^{N-1}}[f(t_i, y(t_i))]\frac{h^N}{N!} \\ &\quad + O(h^{N+1}) \end{aligned}$$

5.1 Euler's Method

The first, simplest, and least accurate method is called Euler's Method. We begin by writing Theorem 5.1 in the case where $N = 1$.

$$y(t_i + h) = y(t_i) + f(t_i, y(t_i))h + O(h^2)$$

If we drop the error term, we can write down a *recurrence relation* for the our approximating values $\{y_i\}_{i=0}^n$,

$$y_{i+1} = y_i + hf(t_i, y_i)$$

Thus we may first use $y_0 = y(t_0)$ to find that $y_1 = y_0 + hf(t_0, y_0)$. Then use our newly calculated y_1 to find y_2 , and so on.

Note that at each step we pick up an error on the order of h^2 . By the time we have reached the right endpoint of our interval, t_n , we have picked up n such errors, so the accumulated error should be on the order of nh^2 . But, as before in chapter 4, $n = O(h^{-1})$, so the final error at the right endpoint should be $O(nh^2) = O(h)$.

Example 5.3: Use Euler's Method with $n = 4$ to approximate $y(1)$ for y the solution to the IVP

$$\frac{dy}{dt} = -2y, \quad y(0) = 3$$

The width of the intervals will be $h = (1 - 0)/4 = 0.25$. We have from the initial condition that $y_0 = 3$. Thus

$$\begin{aligned} y_1 &= y_0 + h(-2y_0) \\ &= 3 + 0.25(-2(3)) \\ &= 1.5 \end{aligned}$$

Similarly,

$$\begin{aligned} y_2 &= y_1 + h(-2y_1) \\ &= 1.5 + 0.25(-2(1.5)) \\ &= 0.75 \end{aligned}$$

Presenting all the results in a table, including the exact solution values obtained from our analytic solution calculated in example 5.1.

i	t_i	y_i	$y(t_i)$	Error
0	0.00	3.000	3.000	0
1	0.25	1.500	1.820	-3.196×10^{-1}
2	0.50	0.750	1.104	-3.536×10^{-1}
3	0.75	0.375	0.669	-2.944×10^{-1}
4	1.00	0.188	0.406	-2.185×10^{-1}

Our estimate for $y(1)$ is $y_4 \approx 0.188$. This is a poor estimate as we know from example 5.1 that the exact answer is $y(1) = 3e^{-2(1)} \approx 0.406$.

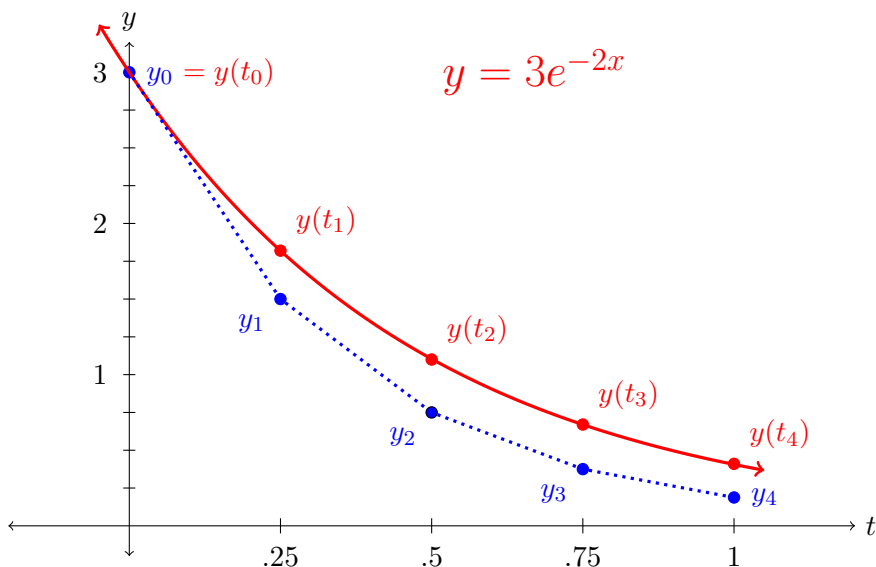
Exercise 5.4: Use Euler's Method with $n = 4$ to approximate $y(1.5)$ for y the solution to the IVP

$$\frac{dy}{dt} = ty^2, \quad y(1) = 1$$

What Euler's Method is really doing is approximating the solution, y , as a series of line segments. We know that the solution passes through the point (t_0, y_0) since that is what our initial value means. We also know the slope of the tangent line to y at that point since that is just $y'(t_0) = f(t_0, y_0)$. Euler's Method just follows that line for a short time period h to

give us our first estimate y_1 . In general y will not be a line, so $y(t_1) \neq y_1$, but if h is small enough they should be close. We then follow a new line through the new point (t_1, y_1) with new slope $f(t_1, y_1)$ to the next approximate value y_2 .

We can see this by comparing the graph of the solution to example 5.1 with our Euler estimate calculated in example 5.3.



Notice that since the solution y is **concave up**, Euler's Method always underestimates the true values. Another way to think about this is to notice that according to Taylor's Theorem the constant associated to the $O(h^2)$ error will depend on the **second derivative** of y . Since y'' is always positive in this example, y_i will always be less than $y(t_i)$.

5.2 Euler's Method using Python

Since any of these methods will become soul-killingly tedious for n of any size, we really want to use a computer to do the calculations. Since our estimate $\{y_i\}$ is not a single value, but a list of values we will need to introduce the idea of an *array*. This is an indexed list of numbers similar to a *vector* from physics or linear algebra.

There are several ways to define an array in Python. The ones we will use require the **numpy** library, so our first line will import this library. The command `y = zeros(n+1)` produces an array of $n + 1$ zeros and assigns it to the variable `y`. The command `t = linspace(t0,tn,n+1)` produces an array of $n + 1$ equally spaced numbers, starting with `t0` and ending with `tn`, and assigns it to the variable `t`.

To see these in action, open a new file called `ODEsolvers` and write:

```
from numpy import *
```

```
y = zeros(5)
```

```
t = linspace(0,1,5)
```

Run the file, then in the console write

```
In : y
Out: array([ 0.,  0.,  0.,  0.,  0.] )
```

```
In : t
Out: array([ 0.   ,  0.25,  0.5  ,  0.75,  1.   ])
```

We can refer to, or change the values in an array by using the index. So again in the console we may write

```
In : y[1] = 14
```

```
In : y
Out: array([ 0.,  14.,  0.,  0.,  0.] )
```

Notice that the index 1 referred to the **second** number in the list. This because the first number is indexed as 0.

With these tools in mind, let's first define the function **f** which constitutes the right hand side of the differential equation $y' = f(t, y)$. Then let's define a function **euler** which takes as arguments the starting time (**t0**), the ending time (**tn**), the number of intervals (**n**), and the initial value (**y0**). It should return the solution array **y**.

In the file **ODEsolvers** erase the commands for **y** and **t**, and write

```
#f in the IVP y' = f(t,y), y(t0)=y0
def f(t,y):
    return -2*y

#Euler's Method on interval [t0,tn] using n intervals and initial value y0
def euler(t0,tn,n,y0):
    h = abs(tn-t0)/n
    t = linspace(t0,tn,n+1)
    y = zeros(n+1)
    y[0] = y0
    for i in range(0,n):
        y[i+1] = y[i] + h*f(t[i],y[i])
    return y
```

Notice that we've chosen $f(t, y) = -2y$ so the differential equation is the same as the one appearing in examples 5.1 and 5.3. To reproduce the results from example 5.3, we'll just let $n = 4$ and $y_0 = 3$. Therefore run the file and in the console write

```
In : euler(0,1,4,3)
Out: array([ 3.   ,  1.5   ,  0.75  ,  0.375 ,  0.1875])
```

These are, indeed, the $\{y_i\}$ values from example 5.3. It would be nice, however, to reproduce the whole table of results—including the errors. We can write a function called **result_table** and use the *formatted print* command to do exactly that.

First define a new function that gives you the exact solution calculated in example 5.1.

```
#analytic solution to the IVP  $y' = f(t,y)$ ,  $y(t_0)=y_0$ 
def sol(t):
    return 3*exp(-2*t)
```

Then,

```
#produces a table comparing an approximation array y with the true solution
#sol(t) defined above
def result_table(t0,tn,y):
    n = len(y)-1
    t = linspace(t0,tn,n+1)
    for i in range(0,n+1):
        print('y_%2d = %5.3f, y(t_%2d) = %5.3f, Error = %10.3e' %
              (i,y[i],i,sol(t[i]),y[i]-sol(t[i])))
    return
```

Recall that the `print` command prints a string with the current value of `i` inserted for `%2d`, the current value of `y[i]` inserted for `%5.3f`, etc.

Run the file and in the console write

```
In : y = euler(0,1,4,3)
In : result_table(0,1,y)
y_ 0 = 3.000, y(t_ 0) = 3.000, Error = 0.000e+00
y_ 1 = 1.500, y(t_ 1) = 1.820, Error = -3.196e-01
y_ 2 = 0.750, y(t_ 2) = 1.104, Error = -3.536e-01
y_ 3 = 0.375, y(t_ 3) = 0.669, Error = -2.944e-01
y_ 4 = 0.188, y(t_ 4) = 0.406, Error = -2.185e-01
```

The first command generates the Euler approximation to the true solution and stores the result in the variable `y`. The second command produces a table comparing `y` to the actual solution `sol(t)` evaluated at the same points.

Exercise 5.5: Use your commands `euler` and `result_table` to reproduce the table in exercise 5.4. (You will have to change `f` and `sol`.)

Now that we have an easy way to execute Euler's Method, we may look at the errors produced for different numbers of intervals.

Example 5.6: Consider the Euler's Method approximations to the solution to the IVP in example 5.1 with $n = 10$ intervals, $n = 100$ intervals, and $n = 1000$ intervals. Compare the errors at $t = 1$.

After making sure `f` and `sol` are consistent with example 5.1, we write in the console

```
In : y10 = euler(0,1,10,3)
In : y100= euler(0,1,100,3)
In : y1000= euler(0,1,1000,3)
In : print('Error    10 = ',y10[10]-sol(1))
```

```
Error 10 = -0.0838833025098
In : print('Error 100 = ',y100[100]-sol(1))
Error 100 = -0.00814718202558
In : print('Error 1000 = ',y1000[1000]-sol(1))
Error 1000 = -0.000812282369786
```

Increasing the number of intervals by a factor of 10 decreases the error by approximately a factor of 10, just as it should for a $O(h)$ method.

Exercise 5.7: Consider the Euler's Method approximations to the solution to the IVP in example 5.2 with $n = 10$ intervals, $n = 100$ intervals, and $n = 1000$ intervals. Compare the errors at $t = 1.5$.

Finally, while it's all well and good to look at the values of a function in a table, it's often better to look at its graph. Here we can recall the techniques we learned in chapter 3 to write a function called `result_graph` which plots the solution and our approximation to the solution in the same window.

Example 5.8: Write a Python function which graphs the exact solution calculated in example 5.1 and the Euler approximation calculated in example 5.3.

At the top of our file `ODEsolvers` include the plotting libraries.

```
from matplotlib.pyplot import *
from matplotlib.patches import Patch
```

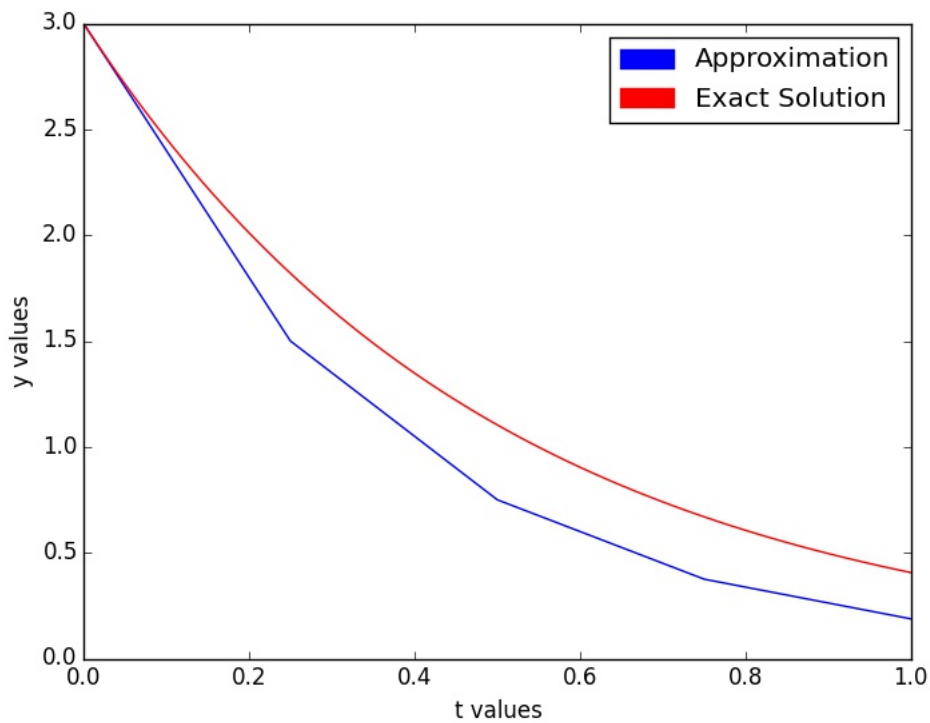
Then below, define our graphing function `result_graph`

```
#Produces a plot of y and true solution on interval [t0,tn]
def result_graph(t0,tn,y):
    xlabel('t values')
    ylabel('y values')
    #plot approximation
    n = len(y)-1
    t = linspace(t0,tn,n+1)
    plot(t,y,color='blue')
    L1 = Patch(color='blue',label='Approximation')
    #plot true solution
    t = linspace(t0,tn,101)
    ysol = sol(t)
    plot(t,ysol,color='red')
    L2 = Patch(color='red',label='Exact Solution')
    #make legend
    legend(handles=[L1,L2],loc='best')
    return
```

Now in the console write

```
In : %matplotlib
Using matplotlib backend: Qt4Agg
In : y = euler(0,1,4,3)
In: result_graph(0,1,y)
```

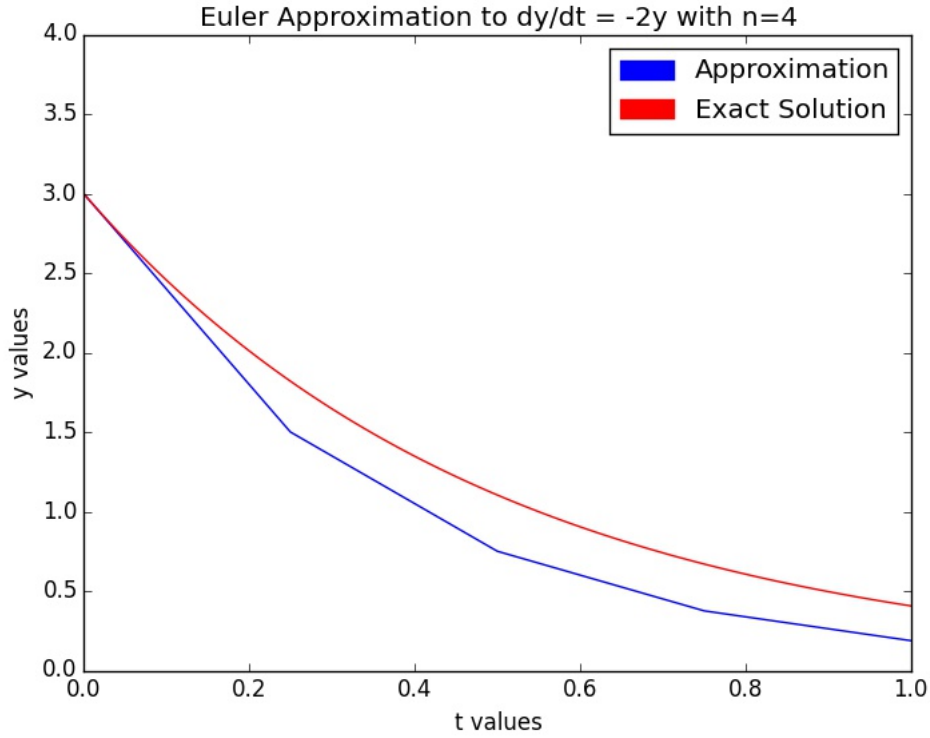
This should produce a new window showing



We can still add a title and play with the axes, if we are so inclined. In the console write

```
In : title('Euler Approximation to dy/dt = -2y with n=4')
In : axis([0,1,0,4])
```

Then the window becomes



Exercise 5.9: Use the Python function `result_graph` to graph the exact solution calculated in exercise 5.2 and the Euler approximation calculated in exercise 5.4.

5.3 Taylor's Method

To find a more accurate method than Euler's Method, we write Theorem 5.1 in the case where $N = 2$.

$$y(t_i + h) = y(t_i) + f(t_i, y(t_i))h + \frac{d}{dt}[f(t, y(t))]\bigg|_{t=t_i} \frac{h^2}{2} + O(h^3)$$

The chain rule tells us that

$$\frac{d}{dt}[f(t, y(t))] = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} \frac{dy}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f$$

Substituting and ignoring the $O(h^3)$ error term gives us a newer, more accurate recurrence relation.

$$y_{i+1} = y_i + fh + \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f \right) \frac{h^2}{2}$$

where all the functions are evaluated at (t_i, y_i) . As before in Euler's Method, there is an error at each step, so the value of our approximate solution at the right end point has accumulated n errors of size $O(h^3)$. This results in accumulated error of $O(nh^3) = O(h^2)$.

I'm unaware of a name for this method. Since it requires us to evaluate partial derivatives of f it is not really a practical method. We will call it *Taylor's Method* since it follows from Taylor's Theorem, but this is not a standard designation.

Example 5.10: Write a new Python function called `taylor` which takes the initial time `t0`, the terminal time `tn`, the number of intervals `n`, and the initial value `y0` and calculates the Taylor's Method approximation to the corresponding initial value problem.

The function `taylor` will look just like the function `euler` except that the command:

```
y[i+1] = y[i] +...
```

will have more terms. In particular, we will need new functions for the partial derivatives of f , call them `dft` for the partial of f with respect to t and `dfy` for the partial of f with respect to y . Then the new program will be

```
def taylor(t0,tn,n,y0):
    h = abs(tn-t0)/n
    t = linspace(t0,tn,n+1)
    y = zeros(n+1)
    y[0] = y0
    for i in range(0,n):
        y[i+1] = y[i] + h*f(t[i],y[i]) + (dft(t[i],y[i])
        +f(t[i],y[i])*dfy(t[i],y[i]))*h**2/2
    return y
```

Example 5.11: Approximate the solution to the IVP from example 5.1 using your function `taylor`. Use $n = 10$, $n = 100$, and $n = 1000$ intervals and consider the errors at $y(1)$.

First we need to make sure $f(t, y) = -2y$, and define the partial derivatives properly. $f_t = 0$ and $f_y = -2$. Thus in the file `ODEsolvers` write

```
def dft(t,y):
    return 0

def dfy(t,y):
    return -2
```

Then in the console we write:

```

In : y10 = taylor(0,1,10,3)
In : y100= taylor(0,1,100,3)
In : y1000= taylor(0,1,1000,3)
In : print('Error 10 = ',y10[10]-sol(1))
Error 10 = 0.00633824429804
In : print('Error 100 = ',y100[100]-sol(1))
Error 100 = 5.49563392115e-05
In : print('Error 1000 = ',y1000[1000]-sol(1))
Error 1000 = 5.42154156513e-07

```

Note that the error drops by two orders of magnitude for each order of magnitude you increase n . This indicates an order h^2 method.

Example 5.12: Use our function `result_graph` to graph the Taylor's Method approximate solution to example 5.1 with $n = 4$.

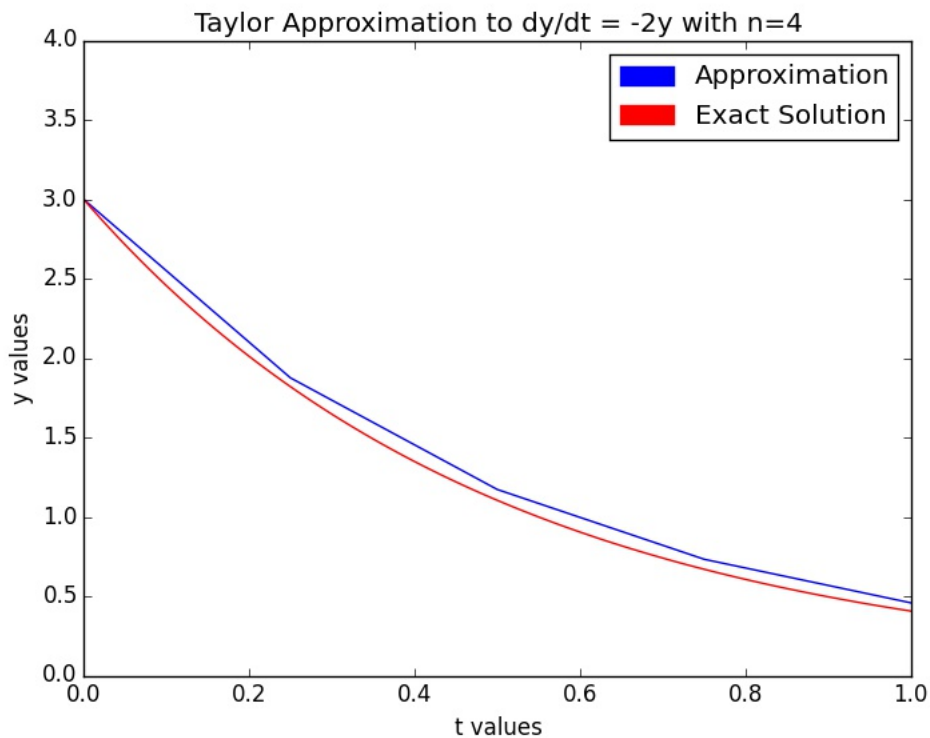
In the console we write

```

In : y = taylor(0,1,4,3)
In : result_graph(0,1,y)
In : title('Taylor Approximation to dy/dt = -2y with n=4')
In : axis([0,1,0,4])

```

This should produce a new window showing



Notice that now, since the $O(h^3)$ error is related to the **third derivative** of y , which is always negative, the approximation is now always **above** the true solution...and, obviously, much closer than the Euler's Method approximation.

Exercise 5.13: Approximate the solution to the IVP from example 5.2 using your function `taylor`. Use $n = 10$, $n = 100$, and $n = 1000$ intervals and consider the errors at $y(1.5)$.

Exercise 5.14: Use our function `result_graph` to graph the Taylor's Method approximate solution to example 5.2 with $n = 4$.

5.4 Runge-Kutta Methods

Taylor's Method gives us a much more accurate method than Euler's Method, but at the cost of finding and evaluating derivatives of f (as well as f itself). This is actually a serious cost, so we would like to have a more accurate method which only involves evaluating f and not its derivatives. We may do this by evaluating f at intermediate points between the step points. Such schemes are called *Runge-Kutta Methods* after the German mathematicians Carl Runge and Martin Kutta.

There are a whole host of different Runge-Kutta methods, which use more or fewer intermediate points, thus achieving different levels of accuracy. In this section we will only consider the simplest such method, The *Modified Euler's Method*. This method uses only one intermediate point (the midpoint) and produces an approximation which, at the right end point, is $O(h^2)$ (where h is the step size). This makes it a Runge-Kutta method of *order 2*.

Consider a solution to an ODE, $y' = f(t, y)$, on an interval of length h . We know from Theorem 3.4 that

$$y' \left(t + \frac{h}{2} \right) = \frac{y(t+h) - y(t)}{h} + O(h^2)$$

Solving for $y(t+h)$ and substituting f for y' gives

$$y(t+h) = y(t) + hf \left(t + \frac{h}{2}, y(t + \frac{h}{2}) \right) + O(h^3)$$

Taylor's Theorem gives us

$$y(t + \frac{h}{2}) = y(t) + y'(t)\frac{h}{2} + O(h^2)$$

We may substitute this into the term for f and apply Taylor's Theorem to only the portion of f which depends on y .

$$\begin{aligned} f \left(t + \frac{h}{2}, y(t + \frac{h}{2}) \right) &= f \left(t + \frac{h}{2}, y(t) + y'(t)\frac{h}{2} + O(h^2) \right) \\ &= f \left(t + \frac{h}{2}, y(t) + f(t, y(t))\frac{h}{2} \right) + \frac{\partial f}{\partial y} \cdot O(h^2) + O(h^4) \end{aligned}$$

If we substitute this expression back into our expression for $y(t+h)$, we see that the $\partial f/\partial y$ term is absorbed into the other $O(h^3)$ terms giving us

$$y(t+h) = y(t) + hf\left(t + \frac{h}{2}, y(t) + f(t, y(t))\frac{h}{2}\right) + O(h^3)$$

Ignoring the $O(h^3)$ term we can use this formula to make the recurrence relation for the Modified Euler Method:

$$y_{i+1} = y_i + hf\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i)\right)$$

So, spiritually what's going on with this formula?

Well first we're using Euler's method to estimate the solution at the midpoint of the interval. We then evaluate f at that point to get a better estimate for the change in y over the interval. Finally, we use this better estimate of y' to estimate y at the right end point.

Example 5.15: Write a new Python function called `modeuler` which takes the initial time `t0`, the terminal time `tn`, the number of intervals `n`, and the initial value `y0` and calculates the Modified Euler's Method approximation to the corresponding initial value problem.

As in example 5.10, we only need to change the `y[i+1] = y[i] + ...` line.

```
#Modified Euler's Method on interval [t0,tn] using n intervals and initial value y0
def modeuler(t0,tn,n,y0):
    h = abs(tn-t0)/n
    t = linspace(t0,tn,n+1)
    y = zeros(n+1)
    y[0] = y0
    for i in range(0,n):
        ym = y[i] + h*f(t[i],y[i])/2
        y[i+1] = y[i] + h*f(t[i]+h/2,ym)
    return y
```

Here we introduced the variable `ym` for the Euler's method estimate of y at the midpoint. This was just to make the code a bit more readable.

Example 5.16: Approximate the solution to the IVP from example 5.1 using your function `modeuler`. Use $n = 10$, $n = 100$, and $n = 1000$ intervals and consider the errors at $y(1)$.

In the console we write:

```
In : y10 = modeuler(0,1,10,3)
In : y100= modeuler(0,1,100,3)
In : y1000= modeuler(0,1,1000,3)
In : print('Error    10 = ',y10[10]-sol(1))
```

```
Error 10 = 0.00633824429804
In : print('Error 100 = ',y100[100]-sol(1))
Error 100 = 5.49563392115e-05
In : print('Error 1000 = ',y1000[1000]-sol(1))
Error 1000 = 5.42154156513e-07
```

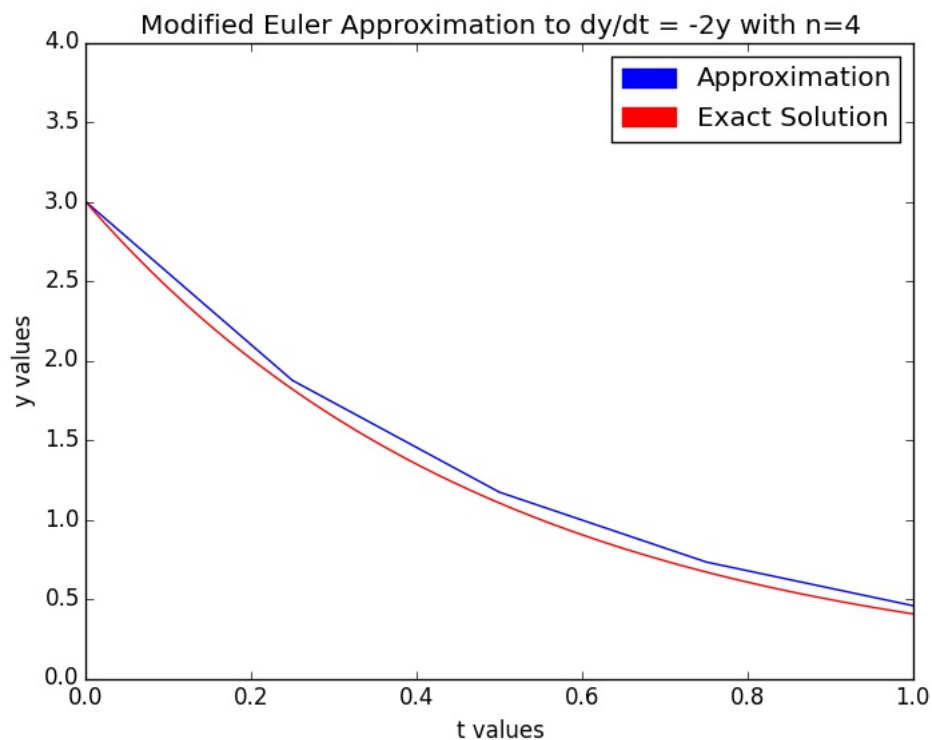
Note that the errors are virtually identical to those using Taylor's Method in example 5.11. Again we see errors dropping by two orders of magnitude for each order of magnitude you increase n . This is consistent with the Modified Euler being a second order Runge-Kutta method.

Example 5.17: Use our function `result_graph` to graph the Modified Euler's Method approximate solution to example 5.1 with $n = 4$.

In the console we write

```
In : y = modeuler(0,1,4,3)
In : result_graph(0,1,y)
In : title('Modified Euler Approximation to dy/dt = -2y with n=4')
In : axis([0,1,0,4])
```

This should produce a new window showing



Exercise 5.18: Approximate the solution to the IVP from example 5.2 using your function `modeuler`. Use $n = 10$, $n = 100$, and $n = 1000$ intervals and consider the errors at $y(1.5)$.

Exercise 5.19: Use our function `result_graph` to graph the Modified Euler's Method approximate solution to example 5.2 with $n = 4$.

5.5 Exercise Solutions and Problems

Solution to Exercise 5.2

Separating and integrating,

$$\begin{aligned}\frac{dy}{y^2} &= t \, dt \\ \int y^{-2} \, dy &= \int t \, dt + C \\ \Rightarrow -y^{-1} &= \frac{t^2}{2} + C\end{aligned}$$

Solving for y gives

$$y(t) = \frac{1}{-t^2/2 - C} = \frac{2}{-2C - t^2} = \frac{2}{A - t^2}$$

Here the constant A is just as arbitrary as the original integration constant C . Applying the initial value to find A ,

$$\begin{aligned}y(1) &= \frac{2}{A-1^2} \\ 1 &= \frac{2}{A-1} \\ \Rightarrow A &= 3\end{aligned}$$

So our solution is

$$y(t) = \frac{2}{3 - t^2}$$

Here $-\sqrt{3} < t < \sqrt{3}$ since the solution exists at $t = 1$ and “blows up” at $t = \pm\sqrt{3}$.

Solution to Exercise 5.4

The width of the intervals will be $h = (1.5 - 1)/4 = 0.125$. We have from the initial condition that $y_0 = 1$. Thus

$$\begin{aligned}y_1 &= y_0 + h \cdot (t_0 y_0^2) \\ &= 1 + 0.125(1(1)^2) \\ &= 1.125\end{aligned}$$

Similarly,

$$\begin{aligned} y_2 &= y_1 + h \cdot (t_1 y_1^2) \\ &= 1.125 + 0.125(1.125(1.125)^2) \\ &\approx 1.303 \end{aligned}$$

Presenting all the results in a table, including the exact solution values obtained from our analytic solution calculated in exercise 5.2.

i	t_i	y_i	$y(t_i)$	Error
0	1.000	1.000	1.000	0
1	1.125	1.125	1.153	-2.815×10^{-2}
2	1.250	1.303	1.391	-8.833×10^{-2}
3	1.375	1.568	1.803	-2.346×10^{-1}
4	1.500	1.991	2.667	-6.757×10^{-1}

Solution to Exercise 5.5

From exercise 5.2 we know that we have to change two functions in our file `ODEsolvers`.

```
def f(t,y):
    return t*y**2

def sol(t):
    return 2/(3-t**2)
```

Run the file, then write in the console

```
In : y = euler(1,1.5,4,1)
In : result_table(1,1.5,y)
y_ 0 = 1.000, y(t_ 0) = 1.000, Error = 0.000e+00
y_ 1 = 1.125, y(t_ 1) = 1.153, Error = -2.815e-02
y_ 2 = 1.303, y(t_ 2) = 1.391, Error = -8.833e-02
y_ 3 = 1.568, y(t_ 3) = 1.803, Error = -2.346e-01
y_ 4 = 1.991, y(t_ 4) = 2.667, Error = -6.757e-01
```

Solution to Exercise 5.7

Having edited `f` and `sol` appropriately, we may just write in the console

```
In : y10 = euler(1,1.5,10,1)
In : y100 = euler(1,1.5,100,1)
In : y1000 = euler(1,1.5,1000,1)
In : print('Error 10 = ',y10[10]-sol(1.5))
Error 10 = -0.381585887714
In : print('Error 100 = ',y100[100]-sol(1.5))
Error 100 = -0.0520986111207
In : print('Error 1000 = ',y1000[1000]-sol(1.5))
Error 1000 = -0.00541920171307
```

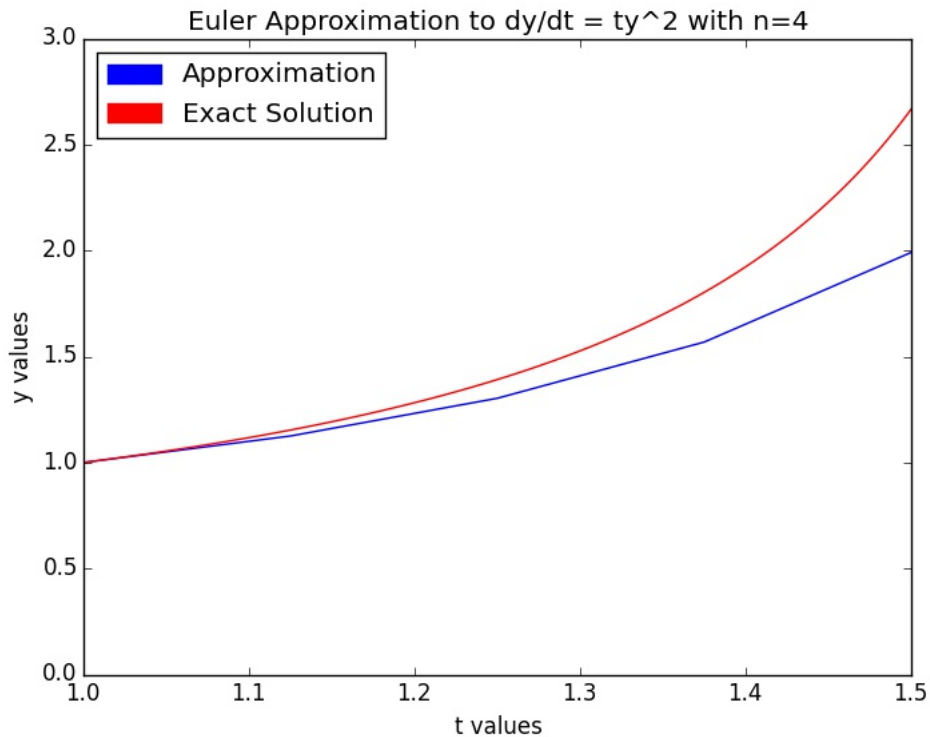

Increasing the number of intervals by a factor of 10 decreases the error by approximately a factor of 10, just as it should for a $O(h)$ method.

Solution to Exercise 5.9

Having edited `f` and `sol` appropriately, in the console write

```
In : y = euler(1,1.5,4,1)
In: result_graph(1,1.5,y)
In: title('Euler Approximation to dy/dt = ty^2 with n=4')
In : axis([1,1.5,0,3])
```

(Note that you don't need to write `%matplotlib` again.) This should produce a new window showing



Solution to Exercise 5.13

First we need to edit the functions so that we are dealing with exercise 5.2.

```
#f in the IVP  $y' = f(t,y)$ ,  $y(t_0)=y_0$ 
def f(t,y):
    return t*y**2

def dft(t,y):
    return y**2

def dfy(t,y):
    return 2*t*y

#analytic solution to the IVP  $y' = f(t,y)$ ,  $y(t_0)=y_0$ 
def sol(t):
    return 2/(3 - t**2)
```

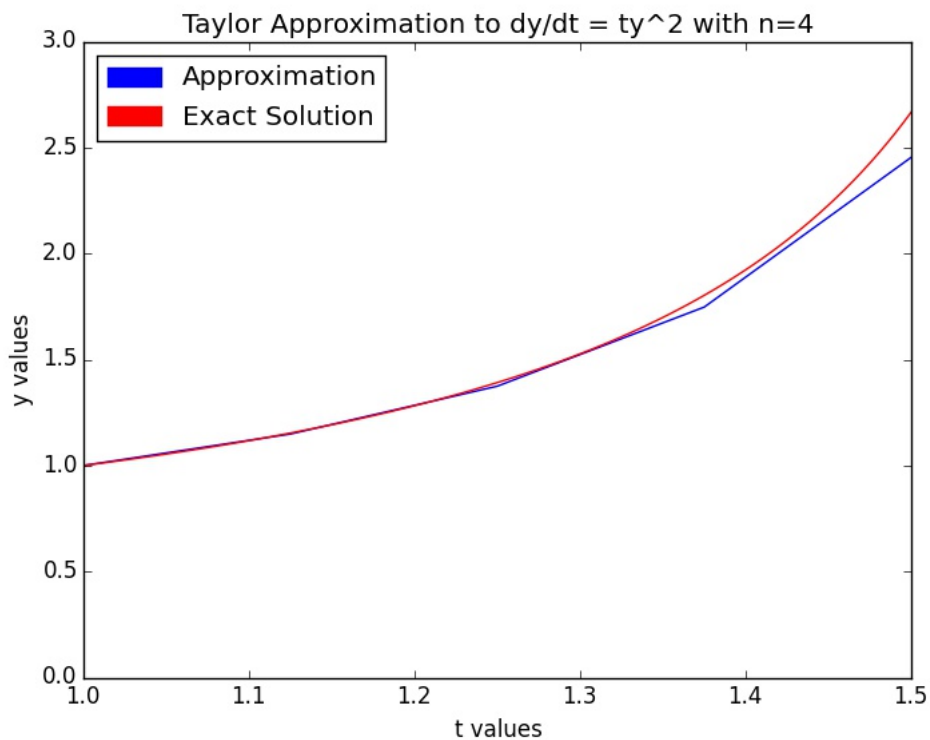
Then at the console

```
In : y10 = taylor(1,1.5,10,1)
In : y100 = taylor(1,1.5,100,1)
In : y1000 = taylor(1,1.5,1000,1)
In : print('Error 10 = ',y10[10]-sol(1.5))
Error 10 = -0.0530153639991
In : print('Error 100 = ',y100[100]-sol(1.5))
Error 100 = -0.000694881823891
In : print('Error 1000 = ',y1000[1000]-sol(1.5))
Error 1000 = -7.12557045013e-06
```

Again, for each increase of an order of magnitude in n , we decrease the error by two orders of magnitude.

Solution to Exercise 5.14

```
In : y = taylor(1,1.5,4,1)
In : result_graph(1,1.5,y)
In : title('Taylor Approximation to  $dy/dt = ty^2$  with  $n=4$ ')
In : axis([1,1.5,0,3])
```



Solution to Exercise 5.18

First we need to edit the functions so that we are dealing with exercise 5.2. (We don't need to worry about the derivative functions.)

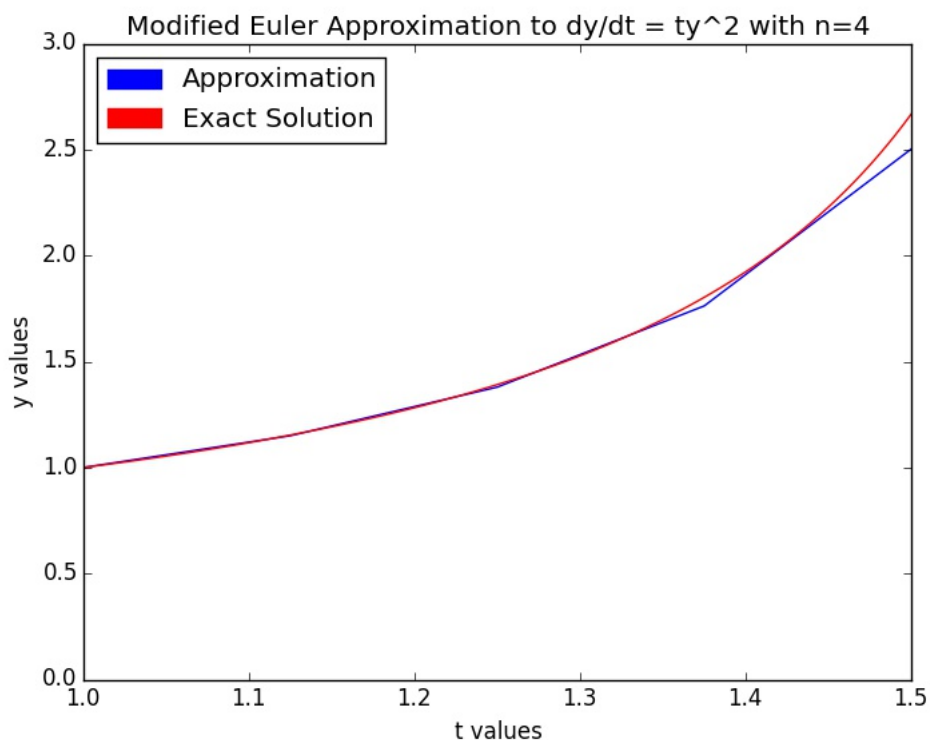
Then at the console

```
In : y10 = modeuler(1,1.5,10,1)
In : y100 = modeuler(1,1.5,100,1)
In : y1000 = modeuler(1,1.5,1000,1)
In : print('Error 10 = ',y10[10]-sol(1.5))
Error 10 = -0.0384756724517
In : print('Error 100 = ',y100[100]-sol(1.5))
Error 100 = -0.000481218651142
In : print('Error 1000 = ',y1000[1000]-sol(1.5))
Error 1000 = -4.91192917984e-06
```

Again, for each increase of an order of magnitude in n , we decrease the error by two orders of magnitude.

Solution to Exercise 5.19

```
In : y = modeuler(1,1.5,4,1)
In : result_graph(1,1.5,y)
In : title('Modified Euler Approximation to dy/dt = ty^2 with n=4')
In : axis([1,1.5,0,3])
```



Problem 5.1: For the initial value problem:

$$y' = y^2 \sin(t), \quad y(0) = -3$$

which has the exact solution:

$$y(t) = \frac{3}{3 \cos(t) - 4}$$

- Approximate by hand $y(\pi/2)$, using Euler's method with $n = 2$ steps.
- Approximate by hand $y(\pi/2)$, using the Taylor's Method with $n = 2$ steps.
- Approximate by hand $y(\pi/2)$, using the Modified Euler method with $n = 2$ steps.
- Compare the errors from the different methods at $t = \pi/2$. Use the **percentage error**:

$$\text{Percentage Error} = \left| \frac{\text{estimate} - \text{exact}}{\text{exact}} \cdot 100\% \right|$$

Problem 5.2: Consider the IVP $y' = t^2 - y - 2$, $y(-2) = 0$ which has the exact solution $y(t) = t^2 - 2t - 8e^{-t-2}$. Produce a single Python plot for $-2 \leq t \leq 3$ which shows:

- The Euler approximation for $n = 10$ in blue
- The Modified Euler approximation for $n = 10$ in red
- The actual solution (with enough points that it appears smooth) in green
- Including a legend with the three different plots labeled correctly

Include the code and the graph in your write-up.

Problem 5.3: One of the most popular “real life” methods for solving a differential equation is the Runge-Kutta method described below.

At each step we first calculate the four constants:

- $A = f(t_i, y_i)$
- $B = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}A\right)$
- $C = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}B\right)$
- $D = f(t_i + h, y_i + hC)$

Then we use a weighted mean of these four constants as our optimal change in y for calculating y_{i+1} .

$$y_{i+1} = y_i + \frac{h}{6}(A + 2B + 2C + D)$$

- a) Write a python program called **RK** which takes as arguments an initial time **t0**, a terminal time **tn**, a number of intervals **n**, and an initial value **y0**. It should return an approximation to the IVP

$$y' = f(t, y), \quad y(t_0) = y_0$$

using the method outlined above.

- b) Test your program on the IVPs described in example 5.1 (with right endpoint $t_n = 1.0$) and exercise 5.2 (with right endpoint $t_n = 1.5$) using $n = 10$ and $n = 100$. What are the errors at the right endpoint? (Make a table.)
- c) What is the order of accuracy for RK? How do you know?

Part II

Problems involving Linear Systems

Chapter 6

Linear Systems: Elimination Methods

Quite probably the most broadly encountered problem in mathematics is a system of linear equations. The solution of such systems is fundamental to a bewildering array of technical fields as well as in abstract mathematics. The theory of linear equations can (and does) account for the content of many mathematics courses. We could easily spend a year discussing just the numerical methods for solving such systems, but we'll restrict ourselves to this chapter.

The problem is basically this. For m equations with mn known coefficients a_{ij} and n unknown variables $\{x_i\}_{i=1}^n$ we wish to solve for the vector \vec{x} :

$$\begin{array}{cccccc} a_{11}x_1 & +a_{12}x_2 & +\dots & +a_{1n}x_n & = & b_1 \\ a_{21}x_1 & +a_{22}x_2 & +\dots & +a_{2n}x_n & = & b_2 \\ \vdots & & & \vdots & & \vdots \\ a_{m1}x_1 & +a_{m2}x_2 & +\dots & +a_{mn}x_n & = & b_m \end{array}$$

This system is usually rewritten as a *matrix equation*:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Or simply as an *augmented matrix*:

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & & & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{array} \right]$$

There are two main approaches to solving linear systems: *direct methods* and *iterative methods*. Direct methods use *elementary row operations* to turn the system into an *upper triangular* system which can be solved by simple back-substitution. Iterative methods for linear systems, like their cousins that we used in chapter 2, take an initial guess and repeatedly perform some process that causes the result to converge to the solution. In this chapter we'll address direct methods.

6.1 Naive Gaussian Elimination

Gaussian elimination is one of the first textbook methods for solving a linear system. We produce zeros in the i -th column by using an elementary row operation. The row operation replaces the j -th row by the j -th row plus a constant times the i -th row. The application of this operation changes many of the coefficients in the matrix. We'll keep track of these changes by using a superscript. For instance $a_{ij}^{(0)}$ will simply be the original a_{ij} . $a_{ij}^{(1)}$ will be the coefficient after the application of one row operation, etc.

Consider a standard “square” system of n equations with n unknown variables. Applying to each row j , the row operation

$$R_j = R_j - \frac{a_{21}}{a_{11}} R_1, \quad j = 2 \dots m$$

reduces the augmented matrix such that

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{array} \right] \sim \left[\begin{array}{cccc|c} a_{11}^{(0)} & a_{12}^{(0)} & \dots & a_{1n}^{(0)} & b_1^{(0)} \\ 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & a_{m2}^{(1)} & \dots & a_{mn}^{(1)} & b_m^{(1)} \end{array} \right]$$

We then apply the row operations:

$$R_j = R_j - \frac{a_{32}^{(1)}}{a_{22}^{(1)}} R_2, \quad j = 3 \dots n$$

to the right-hand-matrix. Eventually (hopefully) we reduce the augmented matrix to an upper-triangular matrix.

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array} \right] \sim \left[\begin{array}{cccc|c} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & \dots & a_{1n}^{(0)} & b_1^{(0)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \dots & a_{3n}^{(2)} & b_3^{(2)} \\ \vdots & & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{array} \right]$$

From here we may find the variables by back-substitution. That is,

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

and

$$x_{n-1} = \frac{b_{n-1}^{(n-2)} - a_{(n-1)n}^{(n-2)} x_n}{a_{(n-1)(n-1)}^{(n-2)}}$$

and so forth.

The method will fail if any of the divisors $a_{ii}^{(i-1)}$ are zero. In fact, we'll see that even if $a_{ii}^{(i-1)}$ is not zero but merely small, the division will amplify errors. The method is "naive" because we perform the algorithm and just hope that nothing bad happens.

Example 6.1: Use Naive Gaussian Elimination to solve the system:

$$\begin{array}{rrcr} 2x_1 & +3x_2 & -x_3 & = & 9 \\ x_1 & -x_2 & +3x_3 & = & -4 \\ 4x_1 & +x_2 & +2x_3 & = & 4 \end{array}$$

The augmented matrix for this system is:

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 9 \\ 1 & -1 & 3 & -4 \\ 4 & 1 & 2 & 4 \end{array} \right]$$

We begin by subtracting one half of the first row from the second row and twice the first row from the third row. These elementary row operations produce the new matrix:

$$\begin{array}{l} R_2 = R_2 - \frac{1}{2}R_1 \\ R_3 = R_3 - \frac{4}{2}R_1 \end{array} \left[\begin{array}{ccc|c} 2.0 & 3.0 & -1.0 & 9.0 \\ 0.0 & -2.5 & 3.5 & -8.5 \\ 0.0 & -5.0 & 4.0 & -14.0 \end{array} \right]$$

We continue by subtracting twice the second row from the third row:

$$R_3 = R_3 - \frac{-5.0}{-2.5}R_2 \left[\begin{array}{ccc|c} 2.0 & 3.0 & -1.0 & 9.0 \\ 0.0 & -2.5 & 3.5 & -8.5 \\ 0.0 & 0.0 & -3.0 & 3.0 \end{array} \right]$$

We complete the solution by back-substitution.

$$-3.0x_3 = 3.0 \quad \Rightarrow \quad x_3 = \frac{3.0}{-3.0} = -1.0$$

$$-2.5x_2 + 3.5x_3 = -8.5 \quad \Rightarrow \quad x_2 = \frac{-8.5 - 3.5(-1.0)}{-2.5} = 2.0$$

$$2.0x_1 + 3.0x_2 - 1.0x_3 = 9.0 \quad \Rightarrow \quad x_1 = \frac{9.0 - 3.0(2.0) + (-1.0)}{2.0} = 1.0$$

It is interesting to note that a mathematics student would probably apply the row operation of switching the first and second row before starting the elimination. (This would be to avoid the fractions that arise from dividing by two.) As we'll see later, however, **numerically** it is better to divide by as large a number as possible. Numerically, the smartest thing to do is switch the first and **third** equations so that you begin the elimination by dividing by **four**!

6.1.1 Matrices in Python

Recall in section 5.2 we introduced the idea of a Python *array*. This was an ordered list of numbers similar to a vector in physics. We already see some arrays in the outline of Naive Gaussian Elimination above. The constants on the right hand side of each equation constitute an array, as does the solution (x_1, x_2, x_3) . But to do Gaussian Elimination we will also need the idea of a two-dimensional box of numbers. That is, a *matrix*.

Let's begin by setting up the augmented matrix $[A : b]$ from example 6.1. As before we first need to import from the `numpy` package. In a new file called `linsys` write:

```
from numpy import *
```

```
Ab = matrix([[2,3,-1,9],[1,-1,3,-4],[4,1,2,4]],double)
```

Then in the console write:

```
In : Ab
```

```
Out:
```

```
matrix([[ 2.,  3., -1.,  9.],
        [ 1., -1.,  3., -4.],
        [ 4.,  1.,  2.,  4.]])
```

The modifier `double` which appears in the `matrix` command casts the entries of the matrix as double-precision floating point numbers (rather than, say, integers). The decimal point that appears after each emphasizes that these are floating point numbers which happen to be integers.

We can pull a row or a column out of the matrix `Ab` by using the `:` operator. This is a sort of 'wild card' which takes on all possible index values. For instance `Ab[1,:]` will be all the elements of `A` with first coordinate 1. That is, the second row. (Recall that the first row is indexed with a 0.)

```
In : Ab[1,:]
```

```
Out: matrix([[ 1., -1.,  3., -4.]])
```

```
In : Ab[:,2]
```

```
Out:
```

```
matrix([[ -1.],
        [  3.],
        [  2.]])
```

The entries of `Ab` with second coordinate 2 are, of course, the third *column*.

There are also some functions in `numpy` which produce matrices of a particular type. For instance `zeros((n,m))` produces an $n \times m$ matrix of zeros.

```
In : zeros((3,4))
```

```
Out:
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Similarly `ones((n,m))` produces a matrix of ones. Note the double parentheses. This is because the functions take a tuple as their argument. There is a third function, `eye(n,m)`, which produces an *identity* matrix of the appropriate dimensions. (For reasons passing understanding, this function does **not** take a tuple as an argument, so there are no doubled parentheses.)

In : `eye(3,4)`

Out:

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.]])
```

Often `zeros` is used to create a matrix of the appropriate dimensions, then new values are assigned to the entries of the matrix.

Example 6.2: Write a Python function called `makesys` which takes `n` as an argument, and returns an $n \times (n + 1)$ matrix of the form:

$$\left[\begin{array}{cccc|c} 1 & 3 & \dots & 2n-1 & 0 \\ 2 & 4 & \dots & 2n & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ n & n+2 & \dots & 3n-2 & 0 \end{array} \right]$$

```
def makesys(n):
    M = zeros((n,n+1))
    for k in range(0,n):
        for l in range(0,n):
            M[k,l] = k + 2*l + 1
    return M
```

Then in the console,

In : `makesys(4)`

Out:

```
array([[ 1.,  3.,  5.,  7.,  0.],
       [ 2.,  4.,  6.,  8.,  0.],
       [ 3.,  5.,  7.,  9.,  0.],
       [ 4.,  6.,  8., 10.,  0.]])
```

Example 6.3: Modify `makesys` so that the far right-hand column is the *sum* of the numbers in that row.

```
def makesys(n):
    M = zeros((n,n+1))
    for k in range(0,n):
        for l in range(0,n):
            M[k,l] = k + 2*l + 1
    for k in range(0,n):
        M[k,n] = sum(M[k,0:n])
    return M
```

Note that the `:` operator works in a similar way to the `range` function. `0:n` produces a range of indices from 0 up to `n-1`, but **not** `n`.

Again in the console,

```
In : makesys(4)
Out:
array([[ 1.,  3.,  5.,  7., 16.],
       [ 2.,  4.,  6.,  8., 20.],
       [ 3.,  5.,  7.,  9., 24.],
       [ 4.,  6.,  8., 10., 28.]])
```

Exercise 6.4: Modify `makesys` so that it takes two arguments, `n` and `C`. It should return the same matrix as above, but with `C` added to each diagonal element (`M[1,1]+C`, `M[2,2]+C`, etc.) and the right-hand column should still be the sum of the elements in each row.

6.1.2 Programming Elimination

Let's start seeing how to program Gaussian Elimination, but first doing a single row operation. Recall that we begin the elimination by subtracting a multiple of the first row from the second row. That is,

$$R_2 = R_2 - cR_1$$

where $c = a_{21}/a_{11}$. How would we do this in Python? Well, first we define `c`, then we use the `:` operator to manipulate the rows of `Ab`.

```
In : c = Ab[1,0]/Ab[0,0]
In : Ab[1,:] = Ab[1,:] - c*Ab[0,:]
In : Ab
Out:
matrix([[ 2. ,  3. , -1. ,  9. ],
        [ 0. , -2.5,  3.5, -8.5],
        [ 4. ,  1. ,  2. ,  4. ]])
```

Exercise 6.5: What Python commands would you write to further reduce **A** to an upper triangular matrix?

Performing Gaussian Elimination will require two loops, one over rows and one over columns. The indices can get fairly confusing, so let's work our way up. Begin by writing a single loop that produces the zeros in the first column. Re-run the file `linsys` (to restore **Ab**), then write in the console:

```
In : for j in range(1,3):
      c = Ab[j,0]/Ab[0,0]
      Ab[j,:] = Ab[j,:] - c*Ab[0,:]
```

In : **Ab**

Out:

```
matrix([[ 2. ,  3. , -1. ,  9. ],
        [ 0. , -2.5,  3.5, -8.5],
        [ 0. , -5. ,  4. , -14. ]])
```

In the code above *j* takes on the value 1, performing the row operation to produce the 0 in the second row, first column. Later *j*=2 produces the 0 in the third row, first column.

To complete the reduction we need another loop, starting with the first **column** and proceeding to the right. We're ready to write our program in `linsys`.

```
#Naive Gaussian Elimination for Ax = b => Ab = [A:b]
def ngauss(Ab):
    (n,m) = Ab.shape
    #Row Reduce [A:b]
    for k in range(0,n-1): #produce k-th column of zeros
        for j in range(k+1,n): #j-th row operation
            c = Ab[j,k]/Ab[k,k]
            Ab[j,:] = Ab[j,:] - c*Ab[k,:]
    return Ab
```

Note that we get the number of rows *n* from the *shape* command, which produces a tuple containing the number of rows and columns. Now run the file and write in the console:

```
In : print(ngauss(Ab))
[[ 2.   3.  -1.   9. ]
 [ 0. -2.5  3.5 -8.5]
 [ 0.   0. -3.   3. ]]
```

Finally we need to actually solve the system via back-substitution. Recall that this requires us to first find x_3 , then x_2 , and finally x_1 . In other words, we need a loop that runs *backwards*.

Adding to our program, we first change the return, then apply back-substitution.

```
#Naive Gaussian Elimination for Ax = b => Ab = [A:b]
def ngauss(Ab):
```

```

(n,m) = Ab.shape
#Row Reduce [A:b]
for k in range(0,n-1): #produce k-th column of zeros
    for j in range(k+1,n): #j-th row operation
        c = Ab[j,k]/Ab[k,k]
        Ab[j,:] = Ab[j,:] - c*Ab[k,:]
#Solve system via back substitution
x = zeros(n)
for k in range(n-1,-1,-1):
    sum = Ab[k,n]
    for j in range(k+1,n):
        sum = sum - Ab[k,j]*x[j]
    x[k] = sum/Ab[k,k]
return x

```

The line for `k in range(n-1,-1,-1)` defines `k` first as 2 (since `n=3`), then 1, then 0. The extra ‘-1’ comes from the fact that the loop is “stepping backward”.

Run the file, and in the console write:

```

In : print('x = ', ngauss(Ab))
x = [ 1.  2. -1.]

```

Note that $x_1 = 1, x_2 = 2, x_3 = -1$ was, in fact, the correct solution to the system in example 6.1.

Note also, by the way, that our program has row reduced the matrix `Ab`. This is a consequence of Python using *pass by reference* for its function arguments. This is as opposed to other languages, such as Matlab, which are *pass by value*. That is, when a matrix is passed as an argument to a function, the function makes of copy of the matrix and manipulates that copy, leaving the original matrix unaffected.

```

In : Ab
matrix([[ 2.   3.  -1.   9. ]
        [ 0.  -2.5  3.5 -8.5]
        [ 0.   0.  -3.   3. ]])

```

Exercise 6.6: Use your program `ngauss` to solve the system:

$$\begin{array}{rrcr}
 3x_1 & +4x_2 & -5x_3 & +6x_4 & = & 41.7 \\
 2x_1 & -x_2 & +6x_3 & -x_4 & = & -16.0 \\
 5x_1 & +2x_2 & -2x_3 & +3x_4 & = & 23.9 \\
 -x_1 & +3x_2 & -x_3 & +4x_4 & = & 19.0
 \end{array}$$

6.2 Gaussian Elimination with Partial Pivoting

We saw earlier that the naivete in Naive Gaussian Elimination came from the fact that we blithely divide by the diagonal elements a_{ii} at each step. If $a_{ii} = 0$ then the elimination will immediately fail. Even if $a_{ii} \neq 0$, but is very small, then dividing by a small number will tend to “blow up” the errors that occur naturally in any numeric calculation (recall how even 0.2 is only known approximately in a machine). In fact, we’ll see that this problem can occur even if a_{ii} is not particularly small, but is small *relative to* the other coefficients in the i -th row.

Partial pivoting is an addition to the algorithm of elimination that addresses this problem. At each step we determine which row would be “best” for doing the next elimination. We then exchange that row with the current highest unused row, and proceed with the elimination as before.

We chose the “best” row by first calculating the maximum absolute value for each element in a row (not counting the last column which is composed of the constants in the system of equations). We then divide each element in the i -th column by its corresponding maximum absolute value. The row in which the absolute value of this ratio is largest is defined to be the “best” row to use for elimination.

Example 6.7: Use Gaussian Elimination with Partial Pivoting to solve the system:

$$\begin{array}{rrcr} 2x_1 & +3x_2 & -x_3 & = & 9 \\ x_1 & -x_2 & +3x_3 & = & -4 \\ 4x_1 & +x_2 & +2x_3 & = & 4 \end{array}$$

The augmented matrix for this system is:

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 9 \\ 1 & -1 & 3 & -4 \\ 4 & 1 & 2 & 4 \end{array} \right]$$

To begin, consider the vector of maximum absolute values for each row (not counting the last column).

$$\vec{c} = [3, 3, 4]$$

Then we consider the left-most column and divide each value by its corresponding c_i and take the absolute value.

$$\vec{r} = \left[\frac{2}{3}, \frac{1}{3}, 1 \right]$$

Clearly the third row has the largest r value, so we exchange the first and third rows. The new augmented matrix is

$$\left[\begin{array}{ccc|c} 4 & 1 & 2 & 4 \\ 1 & -1 & 3 & -4 \\ 2 & 3 & -1 & 9 \end{array} \right]$$

Now we perform the elimination operations, $R_2 = R_2 - \frac{1}{4}R_1$ and $R_3 = R_3 - \frac{1}{2}R_1$. This produces the matrix

$$\left[\begin{array}{ccc|c} 4 & 1 & 2 & 4 \\ 0 & -1.25 & 2.5 & -5 \\ 0 & 2.5 & -2 & 7 \end{array} \right]$$

The first row is no longer available for elimination, so we do not consider it. From the second and third rows we have,

$$\vec{c} = [2.5, 2.5] \Rightarrow \vec{r} = [0.5, 1]$$

Again the third row is best, so we exchange the second and third rows,

$$\left[\begin{array}{ccc|c} 4 & 1 & 2 & 4 \\ 0 & 2.5 & -2 & 7 \\ 0 & -1.25 & 2.5 & -5 \end{array} \right]$$

...and perform the elimination $R_3 = R_3 + \frac{1}{2}R_2$. This produces

$$\left[\begin{array}{ccc|c} 4 & 1 & 2 & 4 \\ 0 & 2.5 & -2 & 7 \\ 0 & 0 & 1.5 & -1.5 \end{array} \right]$$

Finally back-substitution allows us to reproduce the correct answer from example 6.1, $x_3 = -1, x_2 = 2, x_1 = 1$.

Exercise 6.8: Use partial pivoting to solve the system:

$$\begin{array}{rrrrr} 3x_1 & +4x_2 & -5x_3 & +6x_4 & = & 41.7 \\ 2x_1 & -x_2 & +6x_3 & -x_4 & = & -16.0 \\ 5x_1 & +2x_2 & -2x_3 & +3x_4 & = & 23.9 \\ -x_1 & +3x_2 & -x_3 & +4x_4 & = & 19.0 \end{array}$$

6.2.1 Programming Partial Pivoting

To modify `ngauss` for partial pivoting, we'll need a function which performs row exchanges.

```
def swaprow(i,j,M):
    (n,m) = M.shape
    temp = zeros(m)
    temp[:] = M[i,:]
    M[i,:] = M[j,:]
    M[j,:] = temp[:]
    return
```

This function switches the i th row and the j th row in the matrix M . Again, notice that we do not need to return the modified matrix M . Since Python functions are *pass by reference*, whatever matrix is passed in the third argument will be directly modified by the function.

It is for this reason, by the way, that we cannot simply write `temp = M[i,:]` in the function `swaprow`. If we had done so, then `temp` would be simply a reference (or *pointer*) to the i th row of M . Then any changes to `M[i,:]` will also change `temp`. This defeats the purpose of creating a temporary variable `temp` to store the contents of the i th row, before over-writing it with the j th row. To make an actual copy of the i th row, we need to create a new array with the `zeros` command, and then copy the contents of the i th row with the `:` operator.

Exercise 6.9: Write a Python program called `revsys` which takes a matrix and reverses the rows. That is, the top row becomes the bottom row while the bottom row becomes the top. The second row becomes the second-to-last row, and vice versa, etc.

Before proceeding with programming partial pivoting, let's pause a moment to see that we actually need it. The matrix generating programs we wrote in exercises 6.4 and 6.9 will be very useful for this. In the console write

```
In : A = makesys(7,10)
In : A
Out:
array([[ 11.,   3.,   5.,   7.,   9.,  11.,  13.,  59.],
       [  2.,  14.,   6.,   8.,  10.,  12.,  14.,  66.],
       [  3.,   5.,  17.,   9.,  11.,  13.,  15.,  73.],
       [  4.,   6.,   8.,  20.,  12.,  14.,  16.,  80.],
       [  5.,   7.,   9.,  11.,  23.,  15.,  17.,  87.],
       [  6.,   8.,  10.,  12.,  14.,  26.,  18.,  94.],
       [  7.,   9.,  11.,  13.,  15.,  17.,  29., 101.]])
In : B = makesys(7,10)
In : revsys(B)
In : B
Out:
array([[ 7.,   9.,  11.,  13.,  15.,  17.,  29., 101.],
       [ 6.,   8.,  10.,  12.,  14.,  26.,  18.,  94.],
       [ 5.,   7.,   9.,  11.,  23.,  15.,  17.,  87.],
       [ 4.,   6.,   8.,  20.,  12.,  14.,  16.,  80.],
       [ 3.,   5.,  17.,   9.,  11.,  13.,  15.,  73.],
       [ 2.,  14.,   6.,   8.,  10.,  12.,  14.,  66.],
       [11.,   3.,   5.,   7.,   9.,  11.,  13.,  59.]])
```

The augmented matrix A represents a system of seven linear equations and seven unknowns. For instance,

$$11x_1 + 3x_2 + 5x_3 + 7x_4 + 9x_5 + 11x_6 + 13x_7 = 59$$

is the first equation in this system. The matrix **B** represents the same system, the equations are simply written in a different order.

Since for each equation the constant term is just the sum of the coefficients, the solution is just

$$x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = x_7 = 1$$

And when we apply Naive Gauss elimination to **A**, that's exactly what we get. However, when we apply it to **B**—where the larger coefficients are generally not on the main diagonal—the errors from not using the best row for elimination accumulate, and our solutions are not very good.

```
In : ngauss(A)
```

```
Out: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

```
In : ngauss(B)
```

```
Out:
```

```
array([-0.23951049,  2.15909091,  0.5          ,  0.97027972,  1.01398601,
        1.05769231,  1.1013986  ])
```

To introduce partial pivoting we need only modify our existing program **ngauss**. Cutting and pasting into our file **linsys**,

```
#Gaussian Elimination with Partial Pivoting for Ax = b => Ab = [A:b]
```

```
def ppgauss(Ab):
```

```
    (n,m) = Ab.shape
```

```
    #Row Reduce [A:b]
```

```
    for k in range(0,n-1): #produce k-th column of zeros
```

```
        #Find best pivot
```

```
        c = amax(abs(Ab[:,k:n]),1)
```

```
        MAX = abs(Ab[i,k])/c[i]
```

```
        I = k
```

```
        for i in range(k+1,n):
```

```
            mx = abs(Ab[i,k])/c[i]
```

```
            if mx > MAX:
```

```
                MAX = mx
```

```
                I = i
```

```
        swaprow(I,k,Ab)
```

```
        #Continue with row reduction
```

```
        for j in range(k+1,n): #j-th row operation
```

```
            c = Ab[j,k]/Ab[k,k]
```

```
            Ab[j,:] = Ab[j,:] - c*Ab[k,:]
```

```
    #Solve system via back substitution
```

```
    x = zeros(n)
```

```
    for k in range(n-1,-1,-1):
```

```
        sum = Ab[k,n]
```

```

    for j in range(k,n):
        sum = sum - Ab[k,j]*x[j]
    x[k] = sum/Ab[k,k]
return x

```

Here we've used the `amax` function which returns an array consisting of the maximum element in each row. (That is, when the second argument is 1. `amax(abs(Ab[:,k:n]),0)` would have returned an array consisting of the largest element in each **column**.)

Now when we apply our new program to the problematic matrix B, we get a more gratifying result.

```

In : B = makesys(7,10)
In : revsys(B)
In : ppgauss(B)
Out: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.])

```

Example 6.10: Compare the effects of `ngauss` and `ppgauss` on the matrix generated by `makesys(7,100)` with its rows reversed.

```

In : A = makesys(7,100)
In : revsys(A)
In : ngauss(A)
Out:
array([-20.26996528,  37.68489583, -24.          ,  1.38346354,
        1.4594184  ,  1.53537326,  1.61132812])

```

```

In : A = makesys(7,100)
In : revsys(A)
In : ppgauss(A)
Out: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.])

```

Note that with `revsys(makesys(7,100))`, the off-diagonal elements are much larger, and the solutions found from Naive Gauss Elimination are correspondingly much worse.

6.3 Ill-conditioned Matrices

One might get the impression from the previous section that, once you include partial pivoting, any reasonably small linear system with a unique solution can be solved fairly accurately by elimination. This is not so.

Example 6.11: Consider the following system of n linear equations.

$$\begin{array}{rcl}
 x_1 - 16x_2 & = & -3 \\
 x_2 - 16x_3 & = & -3 \\
 & \vdots & \\
 x_{n-1} - 16x_n & = & -3 \\
 5x_n & = & 1
 \end{array}$$

This corresponds to the augmented matrix

$$\left[\begin{array}{ccccc|c} 1 & -16 & 0 & 0 & \dots 0 & -3 \\ 0 & 1 & -16 & 0 & \dots 0 & -3 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & 1 & -16 & -3 \\ 0 & \dots & \dots & 0 & 5 & 1 \end{array} \right]$$

This matrix is already row-reduced, so we may start solving this system by back-substitution.

$$x_n = \frac{1}{5} = 0.2$$

$$x_{n-1} = 16x_n - 3 = 0.2$$

And, in general,

$$x_{k-1} = 16x_k - 3 = 0.2$$

until

$$x_1 = 16x_2 - 3 = 0.2$$

So the solution is just $x_1 = x_2 = \dots = x_n = 0.2\dots$ but perhaps this 0.2 business seems familiar. In fact we analyzed a very similar situation all the way back in chapter 1. Then we were iterating the operation $16(x - 3/16)$ from a starting value of $x = 0.2$. This is essentially the same since,

$$x_{k-1} = 16x_k - 3 = 16 \left(x_k - \frac{3}{16} \right)$$

where $x_k \approx 0.2$. And in fact when we solve this system we see the same creeping error that we first noticed in example 1.1.

Let's begin by writing a Python program to generate the augmented matrix above.

```
def bad02sys(n):
    M = eye(n,n+1)
    for k in range(0,n-1):
        M[k,k+1] = -16
        M[k,n] = -3
    M[n-1,n-1] = 5
    M[n-1,n] = 1
    return M
```

In the console we may generate a small example matrix and solve the corresponding system.

```
In : bad02sys(4)
Out:
array([[ 1., -16.,  0.,  0., -3.],
```

```

[ 0.,  1., -16.,  0., -3.],
[ 0.,  0.,  1., -16., -3.],
[ 0.,  0.,  0.,  5.,  1.]]

```

```
In : A = bad02sys(4)
```

```
In : ppgauss(A)
```

```
Out: array([ 0.2,  0.2,  0.2,  0.2])
```

All well and good so far. But recalling our early example, we found the problems became really obvious after 14 iterations.

```
In : A = bad02sys(14)
```

```
In : ppgauss(A)
```

```
Out:
array([ 0.25      ,  0.203125   ,  0.20019531,  0.20001221,  0.20000076,
        0.20000005,  0.2        ,  0.2        ,  0.2        ,  0.2        ,
        0.2        ,  0.2        ,  0.2        ,  0.2        ])
```

We notice that the partial pivoting doesn't help since, for one thing, there was no elimination. It turns out that the matrix A is just numerically unfriendly. The errors from solving this linear system are small if the system is not too large. However, if the system is large enough, then the problems inherent in storing numbers in a computer will overwhelm our solution method.

We say that this is an *ill-conditioned* system. The matrix of coefficients is called an *ill-conditioned* matrix. The way we measure the degree to which a matrix is ill-conditioned is with a quantity called the *condition number*.

A careful discussion of how the condition number is calculated is beyond the scope of this course, however we can say a few things. The condition number of a matrix which is not invertible is defined to be infinity. This corresponds to the fact that a system with this matrix as its coefficient matrix may not be consistent. That is, it may not have a solution at all, so it is not unreasonable to characterize such a matrix as being *perfectly ill-conditioned*.

However, as we saw in example 6.11, a system may have a perfectly reasonable, invertible coefficient matrix and yet still be numerically pathological. Such matrices will have large, but finite condition numbers. The actual size of the condition number corresponds roughly with how much accuracy may be lost in solving the corresponding system numerically. Once the condition number is larger than the double precision accuracy at which the computer stores numbers (about 10^{16}), then the answers produced will be essentially meaningless.

Example 6.12: Let's modify our program `bad02sys` so that it just produces the coefficient matrix. We'll call our new program `bad02mat`. We may then find the condition number for different sized matrices.

```
def bad02mat(n):
    M = eye(n,n)
    for k in range(0,n-1):
        M[k,k+1] = -16
    M[n-1,n-1] = 5
    return M

```

We also need to import the condition number function `cond` from the submodule `numpy.linalg`, so at the top of our file `linsys` write `from numpy.linalg import cond`.

Now in the console

```
In : A = bad02mat(6)
In : A
Out:
array([[ 1., -16.,  0.,  0.,  0.,  0.],
       [ 0.,  1., -16.,  0.,  0.,  0.],
       [ 0.,  0.,  1., -16.,  0.,  0.],
       [ 0.,  0.,  0.,  1., -16.,  0.],
       [ 0.,  0.,  0.,  0.,  1., -16.],
       [ 0.,  0.,  0.,  0.,  0.,  5.]])
```

```
In : cond(A)
Out: 3759523.6039952473
```

```
In : A = bad02mat(4)
In : cond(A)
Out: 14667.802723842738
```

```
In : A = bad02mat(14)
In : cond(A)
Out: 16150718532929392.0
```

Note that $n = 14$ was the point when the iterative process in example 1.1 first began to really depart from reality. This was because, at that point, the machine accuracy of 0.2 had been essentially “used up”. Note that the 14×14 case above was the point when the condition number reached 1.61×10^{16} , right around the double precision accuracy of the machine.

Finally, let’s look at the condition numbers for some of the more friendly systems we saw earlier in this chapter.

Example 6.13: Let’s modify our program `makesys` so that it just produces the coefficient matrix. We’ll call our new program `makemat`. We may then find the condition number for different matrices.

```
def makemat(n,C):
    M = zeros((n,n))
    for k in range(0,n):
        for l in range(0,n):
            M[k,l] = k + 2*l + 1
    for k in range(0,n):
        M[k,k] = M[k,k] + C
    return M
```


Now in the console

```
In : A = makemat(7,10)
```

```
In : A
```

```
Out:
```

```
array([[ 11.,   3.,   5.,   7.,   9.,  11.,  13.],
       [  2.,  14.,   6.,   8.,  10.,  12.,  14.],
       [  3.,   5.,  17.,   9.,  11.,  13.,  15.],
       [  4.,   6.,   8.,  20.,  12.,  14.,  16.],
       [  5.,   7.,   9.,  11.,  23.,  15.,  17.],
       [  6.,   8.,  10.,  12.,  14.,  26.,  18.],
       [  7.,   9.,  11.,  13.,  15.,  17.,  29.]])
```

```
In : cond(A)
```

```
Out: 18.278624621891101
```

```
In : revsys(A)
```

```
In : cond(A)
```

```
Out: 18.278624621891101
```

```
In : A = makemat(7,100)
```

```
In : cond(A)
```

```
Out: 1.8650833377736278
```

Note that the large matrix produced by `makemat` still had quite a small condition number. This large matrix just happens to be numerically friendly. Notice as well that reversing the order of the rows—an operation that caused such havoc for our naive gaussian elimination method—does not change the condition number.

And finally, when we used `makemat` to produce a matrix with even larger diagonal elements (which we know makes the numerical solution to this system more accurate) this caused the condition number to *decrease*.

6.4 Exercise Solutions and Problems

Solution to Exercise 6.4

There are really two ways to do this. One is with another loop.

```
def makesys(n,C):
    M = zeros((n,n+1))
    for k in range(0,n):
        for l in range(0,n):
            M[k,l] = k + 2*l + 1
    for k in range(0,n):
        M[k,k] = M[k,k] + C
    for k in range(0,n):
```

```

        M[k,n] = sum(M[k,0:n])
    return M

```

The other is with the `eye` command.

```

def makesys(n,C):
    M = zeros((n,n+1))
    for k in range(0,n):
        for l in range(0,n):
            M[k,l] = k + 2*l + 1
    M = M + C*eye(n,n+1)
    for k in range(0,n):
        M[k,n] = sum(M[k,0:n])
    return M

```

Either way in the console,

```

In : makesys(4,10)
Out:
array([[ 11.,   3.,   5.,   7.,  26.],
       [  2.,  14.,   6.,   8.,  30.],
       [  3.,   5.,  17.,   9.,  34.],
       [  4.,   6.,   8.,  20.,  38.]])

```

Solution to Exercise 6.5

```

In : c = Ab[2,0]/Ab[0,0]
In : Ab[2,:] = Ab[2,:] - c*Ab[0,:]
In : Ab
Out:
matrix([[ 2. ,   3. ,  -1. ,   9. ],
        [ 0. ,  -2.5,   3.5,  -8.5],
        [ 0. ,  -5. ,   4. , -14. ]])

```

```

In : c = Ab[2,1]/Ab[1,1]
In : Ab[2,:] = Ab[2,:] - c*Ab[1,:]
In : Ab
Out:
matrix([[ 2. ,   3. ,  -1. ,   9. ],
        [ 0. ,  -2.5,   3.5,  -8.5],
        [ 0. ,   0. ,  -3. ,   3. ]])

```

Solution to Exercise 6.6

We need only change the matrix `Ab` and run `ngauss`. In the file `linsys` write:

```
Ab = matrix([[3,4,-5,6,41.7],[2,-1,6,-1,-16],[5,2,-2,3,23.9],
[-1,3,-1,4,19]],double)
```

Run the file, then in the console:

```
In : print('x = ',ngauss(Ab))
```

```
x = [ 1.2  0.5 -2.3  4.1]
```

So the solution is $x_1 = 1.2, x_2 = 0.5, x_3 = -2.3, x_4 = 4.1$.

Solution to Exercise 6.8

The augmented matrix is

$$\left[\begin{array}{cccc|c} 3 & 4 & -5 & 6 & 41.7 \\ 2 & -1 & 6 & -1 & -16.0 \\ 5 & 2 & -2 & 3 & 23.9 \\ -1 & 3 & -1 & 4 & 19.0 \end{array} \right]$$

$$c = [6, 6, 5, 3] \Rightarrow r = \left[\frac{1}{2}, \frac{1}{3}, 1, \frac{1}{3} \right]$$

So we switch the first and third rows, then eliminate.

$$\left[\begin{array}{cccc|c} 5 & 2.0 & -2.0 & 3.0 & 23.90 \\ 0 & -1.8 & 6.8 & -2.2 & -25.56 \\ 0 & 2.8 & -3.8 & 4.2 & 27.36 \\ 0 & 3.4 & -1.4 & 4.6 & 23.78 \end{array} \right]$$

$$c = [6.8, 4.2, 4.6] \Rightarrow r \approx [0.265, 0.667, 0.739]$$

So we switch the second and fourth rows, then eliminate.

$$\left[\begin{array}{cccc|c} 5 & 2.0 & -2.000 & 3.000 & 23.900 \\ 0 & 3.4 & -1.400 & 4.600 & 23.780 \\ 0 & 0.0 & -2.647 & 0.412 & 7.776 \\ 0 & 0.0 & 6.059 & 0.235 & -12.971 \end{array} \right]$$

$$c = [2.647, 6.059] \Rightarrow r = [1, 1]$$

Since the r 's are the same, we need not switch. After the last elimination,

$$\left[\begin{array}{cccc|c} 5 & 2.0 & -2.000 & 3.000 & 23.900 \\ 0 & 3.4 & -1.400 & 4.600 & 23.780 \\ 0 & 0.0 & -2.647 & 0.412 & 7.776 \\ 0 & 0.0 & 0.000 & 1.178 & 4.829 \end{array} \right]$$

After back-substitution,

$$x_1 = 1.2, x_2 = 0.5, x_3 = -2.3, x_4 = 4.1$$

Solution to Exercise 6.9

```
def revsys(M):
    (n,m)=M.shape
    for i in range(0,n//2):
        swaprow(i,n-i-1,M)
    return
```

Problem 6.1: Consider the linear system

$$\begin{array}{rrrrr} 4.3x_1 & +6.6x_2 & -5.3x_3 & +6.8x_4 & = & 48.81 \\ 2.5x_1 & -1.2x_2 & +6.6x_3 & -2.0x_4 & = & -30.50 \\ 5.4x_1 & +2.2x_2 & -2.6x_3 & +3.5x_4 & = & 45.69 \\ -7.2x_1 & +5.3x_2 & -1.3x_3 & +4.9x_4 & = & -18.15 \end{array}$$

- a.) Solve the system by hand showing your steps and using the method of Naive Gaussian Elimination. After each arithmetic operation in the method, round to two decimal places.
(So, for instance, when performing the first elimination we calculate the multiplier, $c = 2.5/4.3 \approx 0.5813953 \rightarrow 0.58$.
Then, $a_{22}^{(1)} = -1.2 - (0.58)6.6 = -5.028 \rightarrow -5.03$.)
- b.) Solve the system again by hand showing your steps and using the method of Gaussian Elimination with Partial Pivoting. Again round to two decimal places after each operation.
- c.) Use `ppgauss` to solve this system. Compare your answers to parts a.) and b.) to this solution.

Problem 6.2: The *Hilbert Matrix* is a matrix of the form:

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{bmatrix}$$

for some integer n .

- a.) Write a Python program called `hilbmat` which takes an integer `n` as its argument and returns the $n \times n$ Hilbert matrix.

- b.) Use `cond` to calculate the condition number of different sizes of Hilbert matrix. Determine `n` so that the $n \times n$ Hilbert matrix has condition number around 10^{16} .
- c.) Write a Python program called `hilbsys` which takes an integer `n` as its argument and returns an $n \times (n + 1)$ augmented matrix whose coefficient matrix is the Hilbert matrix and whose right-hand most column is just the sum of the elements in each row of the Hilbert matrix.
- d.) Use `ppgauss` to solve the linear system with augmented matrix given in part c for different values of n . Notice that the solutions should always be

$$x_1 = x_2 = \dots = x_n = 1$$

For what values of n does `ppgauss` produce solutions with errors greater than 0.01, 0.1, and 1.0? Compare to your answer for part b.

Chapter 7

Linear Systems: Decomposition and Iteration

7.1 LU Factorization

To solve the system $A\vec{x} = \vec{b}$ we want to first *factorize* the coefficient matrix A into a *unit lower triangular* matrix L and an *upper triangular* matrix U . That is, we want to find L and U such that $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n1} & l_{n2} & \dots & l_{n,n-1} & 1 \end{bmatrix} \quad U = \begin{bmatrix} u_{11} & u_{12} & \dots & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & u_{nn} \end{bmatrix}$$

Once we have this factorization, solving the system is a straight-forward substitution, not requiring any elimination. First we solve $L\vec{y} = \vec{b}$, then $U\vec{x} = \vec{y}$.

Example 7.1: Use the fact that

$$\begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix}$$

to solve the system

$$\begin{aligned} 5x_1 + 3x_2 &= 4 \\ 3x_1 + 2x_2 &= 6 \end{aligned}$$

We first write this system as a matrix equation, then substitute the factorization.

$$\begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

Now we solve the system,

$$\begin{bmatrix} 1 & 0 \\ \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \Rightarrow y_1 = 4, y_2 = 6 - \frac{3}{5}(4) = \frac{18}{5}$$

And finally,

$$\begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ \frac{18}{5} \end{bmatrix} \Rightarrow \frac{1}{5}x_2 = \frac{18}{5} \Rightarrow x_2 = 18$$

$$5x_1 + 3x_2 = 4 \Rightarrow x_1 = \frac{4 - 3(18)}{5} = -10$$

Exercise 7.2: Use the fact that

$$\begin{bmatrix} 2 & 1 & -3 \\ 4 & 1 & 5 \\ -6 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -6 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -3 \\ 0 & -1 & 11 \\ 0 & 0 & 58 \end{bmatrix}$$

to solve the system

$$\begin{array}{rrcr} 2x_1 & +x_2 & -3x_1 & = & 9 \\ 4x_1 & +x_2 & +5x_3 & = & 5 \\ -6x_1 & +3x_2 & +x_3 & = & -7 \end{array}$$

7.1.1 Calculating the Factorization

Calculating the LU factorization is no more than performing elimination on the coefficient matrix. To see this consider our matrix from example 7.1.

To row-reduce the matrix, we may perform the row operation

$$\begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \quad R_2 = R_2 - \frac{3}{5}R_1 \quad \rightarrow \quad \begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix}$$

This may also be accomplished by multiplying by an elementary matrix,

$$\begin{bmatrix} 1 & 0 \\ -\frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix}$$

The inverse of the elementary matrix is just the same matrix with the sign of the off-diagonal changed, so if we multiply by this inverse,

$$\begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{3}{5} & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 0 & \frac{1}{5} \end{bmatrix}$$

which is, of course, the LU factorization we just used. In fact, in general, the U matrix is just the row-echelon form of the matrix while the sub-diagonal entries of the L matrix are just the negatives of the constants used to perform the eliminations.

Example 7.3: Find an LU factorization for the matrix

$$\begin{bmatrix} 2 & 4 & -1 & 5 \\ -4 & -5 & 3 & -8 \\ -6 & 0 & 8 & -3 \\ 2 & -5 & -4 & 1 \end{bmatrix}$$

To row reduce this matrix we start with the operations, $R_2 = R_2 + 2R_1$, $R_3 = R_3 + 3R_1$, and $R_4 = R_4 - R_1$. Then,

$$\rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 \\ 0 & 3 & 1 & 2 \\ 0 & 12 & 5 & 12 \\ 0 & -9 & -3 & -4 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -3 & ? & 1 & 0 \\ 1 & ? & ? & 1 \end{bmatrix}$$

Next $R_3 = R_3 - 4R_2$ and $R_4 = R_4 + 3R_2$, so

$$\rightarrow \begin{bmatrix} 2 & 4 & -1 & 5 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 2 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -3 & 4 & 1 & 0 \\ 1 & -3 & ? & 1 \end{bmatrix}$$

Since the matrix is in row-echelon form we do not need to perform the last row operation that would be represented in the final undetermined entry in L . So,

$$\begin{bmatrix} 2 & 4 & -1 & 5 \\ -4 & -5 & 3 & -8 \\ -6 & 0 & 8 & -3 \\ 2 & -5 & -4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -3 & 4 & 1 & 0 \\ 1 & -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -1 & 5 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Exercise 7.4: Find an LU factorization for the matrix

$$\begin{bmatrix} 2 & 1 & -3 \\ 4 & 1 & 5 \\ -6 & 3 & 1 \end{bmatrix}$$

7.1.2 Programming LU Factorization

To program the solution of a linear system by LU factorization, we'll divide the two portions of the algorithm into two different functions. First we want to write a function which decomposes the coefficient matrix A into L and U . Then we want a separate function which takes L , U , and the constant vector \vec{b} , and solves the system for \vec{x} .

Let's start a new file called `morelinsys`, import `numpy`, and input the matrix from exercise 7.4, and the constant vector from example 7.2.

```
from numpy import *
```

```
A = array([[2, 1, -3],[4, 1, 5],[-6, 3, 1]],double)
b = array([9,5,-7],double)
```

The function `LUdecomp` will take a coefficient matrix `M` and return the factors `L` and `U`. For the sake of clarity we'll create brand new matrices for the factorization. (If we'd wished to use computer memory efficiently, we would row reduce the matrix `M` (as we did in `ngauss`

and `ppgauss`), and call this reduced matrix U . We could then store the interesting elements of L (those below the diagonal) in the subdiagonal positions of row reduced M . Thus, the LU factorization uses no more memory than the original coefficient matrix M .)

The code for `LUdecomp` is a straight-forward modification of our `ngauss` elimination code. The differences are that we create the matrix L initially as an identity matrix, and make a copy of M that will become U after the row reduction. The constants c used in `ngauss` code will be stored as the interesting elements in L ,

```
def LUdecomp(M):
    (n,m) = M.shape
    L = eye(n,m)
    U = zeros((n,m))
    U[:,:] = M[:,:]
    for k in range(0,n-1): #produce k-th column of zeros
        for j in range(k+1,n): #produce j-th row operation
            L[j,k] = U[j,k]/U[k,k]
            U[j,:] = U[j,:] - L[j,k]*U[k,:]
    return (L,U)
```

Example 7.5: Use `LUdecomp` to find an LU decomposition of the coefficient matrix from exercise 7.4.

Run the file `morelinsys`, then in the console write:

```
In : (V,W) = LUdecomp(A)
In : V
Out:
array([[ 1.,  0.,  0.],
       [ 2.,  1.,  0.],
       [-3., -6.,  1.]])

In: W
array([[ 2.,  1., -3.],
       [ 0., -1., 11.],
       [ 0.,  0., 58.]])
```

Besides the fact that these answers agree with those found in exercise 7.4, we can check by multiplying these matrices together. We should, of course, get A back. However, as opposed to *Matlab*, the default multiplication of *arrays* in Python is *element-wise*, not as matrices. To matrix-multiply in Python, we use the function `dot`.

```
In : V*W
Out:
array([[ 2.,  0., -0.],
       [ 0., -1.,  0.]])
```

```
[ -0., -0., 58.]])
```

```
In : dot(V,W)
```

```
Out:
```

```
array([[ 2.,  1., -3.],  
       [ 4.,  1.,  5.],  
       [-6.,  3.,  1.]])
```

... the second of which is, of course, A .

Exercise 7.6: Use `LUdecomp` to find the LU decomposition of the matrix from example 7.3.

The function `LUsolve` takes L , U , and the constant vector b , and returns the solution vector x . For the code we need only modify our back-substitution code from `ngauss`.

```
def LUsolve(L,U,b):  
    n = len(B)  
    x = y = zeros(n)  
    for k in range(0,n):  
        y[k] = b[k]  
        for j in range(0,k):  
            y[k] = y[k] - L[k,j]*y[j]  
    for k in range(n-1,-1,-1):  
        x[k] = y[k]  
        for j in range(k+1,n):  
            x[k] = x[k] - U[k,j]*x[j]  
        x[k] = x[k]/U[k,k]  
    return x
```

Example 7.7: Use the function `LUsolve` to solve the system from exercise 7.2.

In the console write

```
In : (V,W) = LUdecomp(A)
```

```
In : LUsolve(V,W,b)
```

```
Out: array([ 2.,  2., -1.]])
```

7.2 Iterative Methods

Up to this point we've discussed what are known as *direct methods* in which was solve, more or less directly, for the solution of the system (if the solution exists). Unfortunately these methods become more and more cumbersome as the size of the system increases. To solve really large systems efficiently and effectively, we generally need to make some

useful assumptions about the system. One such assumption is that the system is *diagonally dominant*.

We say that a matrix A is *diagonally dominant* if, for every row i in A ,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Intuitively, diagonal dominance means that in each row the diagonal element is bigger than all the other elements in the row combined. In the case of a diagonally dominant matrix, an *iterative* method, similar to Newton's Method, can be used to approximate the solution.

Like Newton's method, we begin with a guess at the true solution, then perform a calculation that—if the system is diagonally dominant—gives us a better approximation. We then repeat the calculation until we are satisfied with the accuracy of the approximation.

7.2.1 Jacobi Iteration

The first method that we will discuss is called *Jacobi iteration*. Consider the linear system

$$\begin{array}{cccccc} a_{11}x_1 & +a_{12}x_2 & +\dots & +a_{1n}x_n & = & b_1 \\ a_{21}x_1 & +a_{22}x_2 & +\dots & +a_{2n}x_n & = & b_2 \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{n1}x_1 & +a_{n2}x_2 & +\dots & +a_{nn}x_n & = & b_n \end{array}$$

Now let us solve for the *diagonal* x_i s.

$$\begin{array}{cccccc} x_1 & = & \frac{b_1}{a_{11}} & -\frac{a_{12}}{a_{11}}x_2 & -\frac{a_{13}}{a_{11}}x_3 & -\dots & -\frac{a_{1n}}{a_{11}}x_n \\ x_2 & = & \frac{b_2}{a_{22}} & -\frac{a_{21}}{a_{22}}x_1 & -\frac{a_{23}}{a_{22}}x_3 & -\dots & -\frac{a_{2n}}{a_{22}}x_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ \vdots & & \vdots & & \vdots & & -\frac{a_{n-1,n}}{a_{n-1,n-1}}x_n \\ x_n & = & \frac{b_n}{a_{nn}} & -\frac{a_{n1}}{a_{nn}}x_1 & -\dots & -\frac{a_{n,n-1}}{a_{nn}}x_{n-1} \end{array}$$

Written in matrix form, we've taken our original matrix equation $A\vec{x} = \vec{b}$ and written it in the form $\vec{x} = C\vec{x} + \vec{c}$ where $C_{ij} = -a_{ij}/a_{ii}$ and $c_i = b_i/a_{ii}$. That is,

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 & -\frac{a_{12}}{a_{11}} & \dots & \dots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & -\frac{a_{23}}{a_{22}} & \dots & -\frac{a_{2n}}{a_{22}} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -\frac{a_{n-1,1}}{a_{n-1,n-1}} & \dots & -\frac{a_{n-1,n-2}}{a_{n-1,n-1}} & 0 & -\frac{a_{n-1,n}}{a_{n-1,n-1}} \\ -\frac{a_{n1}}{a_{nn}} & \dots & \dots & -\frac{a_{n,n-1}}{a_{nn}} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \vdots \\ \frac{b_{n-1}}{a_{n-1,n-1}} \\ \frac{b_n}{a_{nn}} \end{bmatrix}$$

To execute Jacobi iteration we make a first guess at the solution, $\vec{x}^{(0)}$ and plug it into the right hand side of the matrix equation above. The left side is then the new estimate, $\vec{x}^{(1)}$. In general

$$\vec{x}^{(k+1)} = C\vec{x}^{(k)} + \vec{c}$$

As long as A is diagonally dominant, this process converges to the true solution regardless of the initial guess $\vec{x}^{(0)}$.

Example 7.8: Use Jacobi iteration to solve the system

$$\begin{array}{rrcr} 3x_1 & +x_2 & +x_3 & = 4 \\ -2x_1 & +4x_2 & & = 1 \\ -x_1 & +2x_2 & -6x_3 & = 2 \end{array}$$

First note that the matrix of coefficients (not the vector of constants on the right hand side) is diagonally dominant, since: $3 > 1 + 1$, $4 > 2 + 0$, and $6 > 1 + 2$. Thus, Jacobi iteration should, in fact, work.

Begin by solving for the diagonal x_i s.

$$\begin{array}{rcll} x_1 & = & -\frac{1}{3}x_2 & -\frac{1}{3}x_3 & +\frac{4}{3} \\ x_2 & = & \frac{1}{2}x_1 & & +\frac{1}{4} \\ x_3 & = & -\frac{1}{6}x_1 & +\frac{1}{3}x_2 & -\frac{1}{3} \end{array}$$

Thus,

$$C = \begin{bmatrix} 0 & -\frac{1}{3} & -\frac{1}{3} \\ \frac{1}{2} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix}, \quad c = \begin{bmatrix} \frac{4}{3} \\ \frac{1}{4} \\ -\frac{1}{3} \end{bmatrix}$$

Since it doesn't really matter where you begin, it is traditional to begin with the zero vector. Then,

$$\vec{x}^{(1)} = C\vec{0} + \vec{c} = \vec{c}$$

The next iterations give:

$$\begin{aligned} \vec{x}^{(2)} &= \begin{bmatrix} 0 & -\frac{1}{3} & -\frac{1}{3} \\ \frac{1}{2} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix} \begin{bmatrix} \frac{4}{3} \\ \frac{1}{4} \\ -\frac{1}{3} \end{bmatrix} + \begin{bmatrix} \frac{4}{3} \\ \frac{1}{4} \\ -\frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{49}{36} \\ \frac{11}{12} \\ -\frac{17}{36} \end{bmatrix} \approx \begin{bmatrix} 1.36111 \\ 0.91667 \\ -0.47222 \end{bmatrix} \\ \vec{x}^{(3)} &= \begin{bmatrix} 0 & -\frac{1}{3} & -\frac{1}{3} \\ \frac{1}{2} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix} \begin{bmatrix} \frac{49}{36} \\ \frac{11}{12} \\ -\frac{17}{36} \end{bmatrix} + \begin{bmatrix} \frac{4}{3} \\ \frac{1}{4} \\ -\frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{32}{27} \\ \frac{67}{72} \\ -\frac{55}{216} \end{bmatrix} \approx \begin{bmatrix} 1.18519 \\ 0.93056 \\ -0.25463 \end{bmatrix} \end{aligned}$$

It certainly appears that the iteration is converging on the actual solution:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{8}{7} \\ \frac{23}{28} \\ -\frac{1}{4} \end{bmatrix} \approx \begin{bmatrix} 1.14286 \\ 0.82143 \\ -0.25000 \end{bmatrix}$$

7.2.2 Gauss-Seidel Iteration

There is a slightly faster iterative method which is very similar to Jacobi iteration called *Gauss-Seidel Iteration*. In this version we update each individual variable, then use the updated version to update subsequent variables.

For instance, as before, $x_1^{(k+1)}$ is updated based on the values of $\vec{x}^{(k)}$.

$$x_1^{(k+1)} = \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}}x_2^{(k)} - \frac{a_{13}}{a_{11}}x_3^{(k)} - \dots - \frac{a_{1n}}{a_{11}}x_n^{(k)}$$

However $x_2^{(k+1)}$ uses the newly calculated $x_1^{(k+1)}$.

$$x_2^{(k+1)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1^{(k+1)} - \frac{a_{23}}{a_{22}}x_3^{(k)} - \dots - \frac{a_{2n}}{a_{22}}x_n^{(k)}$$

And so on for the other variables.

$$x_i^{(k+1)} = \frac{b_i}{a_{ii}} - \frac{a_{i1}}{a_{ii}}x_1^{(k+1)} - \dots - \frac{a_{i,i-1}}{a_{ii}}x_{i-1}^{(k+1)} - \frac{a_{i,i+1}}{a_{ii}}x_{i+1}^{(k)} - \dots - \frac{a_{in}}{a_{ii}}x_n^{(k)}$$

Example 7.9: Use Gauss-Seidel Iteration to solve the system

$$\begin{array}{rrcr} 3x_1 & +x_2 & +x_3 & = 4 \\ -2x_1 & +4x_2 & & = 1 \\ -x_1 & +2x_2 & -6x_3 & = 2 \end{array}$$

This is the same system as example 7.8. It is diagonally dominant, as just as diagonal dominance forces Jacobi Iteration to converge, so does it force Gauss-Seidel Iteration to converge.

We again begin by solving for the diagonal x_i s.

$$\begin{array}{rcll} x_1 & = & -\frac{1}{3}x_2 & -\frac{1}{3}x_3 + \frac{4}{3} \\ x_2 & = & \frac{1}{2}x_1 & + \frac{1}{4} \\ x_3 & = & -\frac{1}{6}x_1 & + \frac{1}{3}x_2 - \frac{1}{3} \end{array}$$

As before we'll begin with $\vec{x}^{(0)} = \vec{0}$. Thus as before,

$$x_1^{(1)} = -\frac{1}{3}(0) - \frac{1}{3}(0) + \frac{4}{3} = \frac{4}{3}$$

Now, however,

$$x_2^{(1)} = \frac{1}{2}\left(\frac{4}{3}\right) + \frac{1}{4} = \frac{11}{12}$$

and

$$x_3^{(1)} = -\frac{1}{6}\left(\frac{4}{3}\right) + \frac{1}{3}\left(\frac{11}{12}\right) - \frac{1}{3} = -\frac{1}{4}$$

The second iteration gives:

$$\begin{aligned} x_1^{(2)} &= -\frac{1}{3} \left(\frac{11}{12} \right) - \frac{1}{3} \left(-\frac{1}{4} \right) + \frac{4}{3} = \frac{10}{9} \approx 1.11111 \\ x_2^{(2)} &= \frac{1}{2} \left(\frac{10}{9} \right) + \frac{1}{4} = \frac{29}{36} \approx 0.80556 \\ x_3^{(2)} &= -\frac{1}{6} \left(\frac{10}{9} \right) + \frac{1}{3} \left(\frac{29}{36} \right) - \frac{1}{3} = -\frac{1}{4} \approx -0.25000 \end{aligned}$$

7.2.3 Programming Jacobi Iteration

7.3 Exercise Solutions and Problems

Solution to Exercise 7.2

We first write this system as a matrix equation, then substitute the factorization.

$$\begin{bmatrix} 2 & 1 & -3 \\ 4 & 1 & 5 \\ -6 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ -7 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -6 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -3 \\ 0 & -1 & 11 \\ 0 & 0 & 58 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 1 \\ -7 \end{bmatrix}$$

Then we solve for \vec{y} ,

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -6 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 5 \\ -7 \end{bmatrix} \Rightarrow \begin{cases} y_1 = 9 \\ y_2 = 5 - 2(9) = -13 \\ y_3 = -7 + 3(9) + 6(-13) = -58 \end{cases}$$

And finally,

$$\begin{bmatrix} 2 & 1 & -3 \\ 0 & -1 & 11 \\ 0 & 0 & 58 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ -13 \\ -58 \end{bmatrix} \Rightarrow \begin{cases} x_3 = \frac{-58}{58} = -1 \\ x_2 = \frac{-13 - 11(-1)}{-1} = 2 \\ x_1 = \frac{9 - (2) + 3(-1)}{2} = 2 \end{cases}$$

Solution to Exercise 7.4

$R_2 = R_2 - 2R_1$ and $R_3 = R_3 + 3R_1$, so

$$\rightarrow \begin{bmatrix} 2 & 1 & -3 \\ 0 & -1 & 11 \\ 0 & 6 & -8 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & ? & 1 \end{bmatrix}$$

and $R_3 = R_3 + 6R_2$,

$$U = \begin{bmatrix} 2 & 1 & -3 \\ 0 & -1 & 11 \\ 0 & 0 & 58 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -6 & 1 \end{bmatrix}$$

Solution to Exercise 7.6

In the console write:

```

In : A = array([[2, 4, -1, 5], [-4, -5, 3, -8], [-6, 0, 8, -3],
...: [2, -5, -4, 1]],double)
In : (V,W) = LUdecomp(A)
In : V
Out:
array([[ 1.,  0.,  0.,  0.],
       [-2.,  1.,  0.,  0.],
       [-3.,  4.,  1.,  0.],
       [ 1., -3.,  0.,  1.]])

In : W
Out:
array([[ 2.,  4., -1.,  5.],
       [ 0.,  3.,  1.,  2.],
       [ 0.,  0.,  1.,  4.],
       [ 0.,  0.,  0.,  2.]])

In : dot(V,W)
Out:
array([[ 2.,  4., -1.,  5.],
       [-4., -5.,  3., -8.],
       [-6.,  0.,  8., -3.],
       [ 2., -5., -4.,  1.]])

```

Problem 7.1:

a.) For the matrix

$$A = \begin{bmatrix} 4 & 8 & 20 \\ 8 & 13 & 16 \\ 20 & 16 & -91 \end{bmatrix}$$

Decompose A into unit lower triangular matrix L and upper triangular matrix U . Show each step. Check your work with the LU decomposition program `LUdecomp`.

b.) Use your answer to part(a) to solve the system:

$$\begin{bmatrix} 4 & 8 & 20 \\ 8 & 13 & 16 \\ 20 & 16 & -91 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 24 \\ 18 \\ -119 \end{bmatrix}$$

Again, show each step and check your work with the LU program `LUsolve`.

Problem 7.2: Consider the system:

$$\begin{bmatrix} -2 & 5 & 9 \\ 7 & 1 & 1 \\ -3 & 7 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ -26 \end{bmatrix}$$

- a.) Perform three iterations of the Jacobi method with your initial guess being the zero vector. (You may use a calculator or computer to do the matrix multiplications, but show each step.) Why does it not appear to be converging?
- b.) Reorder the equations so that the Jacobi method will converge, and perform three iterations with your initial guess being the zero vector. (Again show each step.) Compare to the solution found by `ppgauss`.
- c.) Now perform three iterations on the reordered set of equations, using *Gauss-Seidel* iteration. Recall from class that for Gauss-Seidel iteration, the calculation of $x_1^{(k+1)}$ is the same as in Jacobi iteration, but $x_2^{(k+1)}$ uses the new $x_1^{(k+1)}$ along with the other old $x_i^{(k)}$ values. Similarly $x_3^{(k+1)}$ uses the new $x_1^{(k+1)}$ and $x_2^{(k+1)}$ along with the remaining older values. Etc. Compare with the `ppgauss` solution and the Jacobi solution calculated above.

Problem 7.3: Gauss-Seidel iteration can be expressed in a matrix form as follows. Let $M = L + U$ where L is lower triangular (including the diagonal) and U is upper triangular (**not** including the diagonal). Then, for the system $M\vec{x} = \vec{b}$,

$$M\vec{x} = (L + U)\vec{x} = \vec{b} \Rightarrow L\vec{x} = -U\vec{x} + \vec{b}$$

The iteration is

$$L\vec{x}^{(k+1)} = -U\vec{x}^{(k)} + \vec{b}$$

which we solve by letting $\vec{y} = -U\vec{x}^{(k)} + \vec{b}$, then solving $L\vec{x}^{(k+1)} = \vec{y}$ by substitution.

- a.) Write a Python program called `Gausseid` which takes as arguments a matrix `M`, a vector `b`, and an integer `n`. It should perform `n` iterations of Gauss-Seidel iteration on the system $M\vec{x} = \vec{b}$.

The program should:

- Copy the contents of `M` which are above the diagonal into a new matrix called `U` (the rest of the entries of `U` should be zero), and the diagonal and lower contents of `M` into a new matrix called `L` (again the upper parts of `L` should be zero).
- For each iteration, use `dot` to calculate

$$\vec{y} = -U\vec{x}^{(k)} + \vec{b}$$

- For each iteration, solve

$$L\vec{x}^{(k+1)} = \vec{y}$$

by substitution similar—but not identical—to how we solved for \vec{y} in `LUsolve`.

b.) For the system

$$\begin{bmatrix} 12 & -2 & 3 & 1 \\ -2 & 15 & 6 & -3 \\ 1 & 6 & 20 & -4 \\ 0 & -3 & 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 20 \\ 0 \end{bmatrix}$$

find the solution using the programs **LUdecomp** and **LUsolve**. Then determine the number of iterations necessary for **Jacobi** to find solutions to within 10^{-5} of the **LUsolve** solutions.

c.) Now use our newly written program **Gausseid** to determine how many iterations are necessary for it to find solutions to within 10^{-5} of the **LUsolve** solutions. Compare with **Jacobi**.

Chapter 8

Interpolation

It is a common problem in many disciplines to take a discrete set of data points and infer from that data some functional relationship between the variables in the data. For instance it is plausible that there should be a relationship between the weight of a vehicle and the fuel economy of that vehicle. To investigate this relationship we might directly measure the weight and fuel economy of several modern vehicles. We would like to use these measurements to guess the fuel economy of a different vehicle based only on its weight. This process is called *interpolation*.

There are two approaches to this problem. The first is to find a function whose values exactly agree with the given data, then simply evaluate this function at the variable of interest. This is known as *curve fitting*. There are many difficulties with this approach, beginning with the fact that, for a relatively large number of data points, the function could be complicated and cumbersome to evaluate.

A second approach, known well to statisticians, is to find a relatively simple function which, while not agreeing with the data exactly, passes near those points (in a sense that we will make explicit). This is known as *regression*.

8.1 Polynomial Curve Fitting

Given a data set of n data points,

$$\mathcal{S} = \{(x_0, y_0), (x_1, y_1), \dots (x_{n-1}, y_{n-1})\}$$

we want to find a polynomial of degree at most $n - 1$

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x + a_0$$

so that for every k , $p(x_k) = y_k$. (That is, the graph of the polynomial p passes through each of the data points.) To find the polynomial, we need only find the constants $\{a_k\}$.

There are two approaches to solving this problem. The first is to simply substitute the x_k values into p and produce a system of linear equations in the coefficients a_k . We then solve the system using techniques from earlier in the class (or use Python's build-in linear equation solver, `solve`). this is known as the *VanderMonte Method*. Unfortunately the matrices that

arise from this approach tend to be badly conditioned, even for a relatively small number of data points. A second more subtle, but numerically better behaved method is known as *Newton's Divided Difference Method*.

8.1.1 VanderMonte Method

If we evaluate the polynomial at the data points, we produce a system of equations:

$$\begin{array}{ccccccccc} p(x_0) = & a_{n-1}x_0^{n-1} & +a_{n-2}x_0^{n-2} & +\dots & +a_1x_0 & +a_0 & = & y_0 \\ p(x_1) = & a_{n-1}x_1^{n-1} & +a_{n-2}x_1^{n-2} & +\dots & +a_1x_1 & +a_0 & = & y_1 \\ & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ p(x_{n-1}) = & a_{n-1}x_{n-1}^{n-1} & +a_{n-2}x_{n-1}^{n-2} & +\dots & +a_1x_{n-1} & +a_0 & = & y_{n-1} \end{array}$$

This can be rewritten as a matrix equation.

$$\begin{bmatrix} x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_{n-1}^{n-1} & x_{n-1}^{n-2} & \dots & x_{n-1} & 1 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

We then just solve this system to find the coefficients a_k . The matrix of coefficients in this system is called the *VanderMonte matrix*. (We will see in a homework problem that the VanderMonte matrix is quite poorly conditioned even for relatively small values of n .)

Example 8.1: Find the third degree polynomial through the data points:

$$\mathcal{S} = \{(0, 1), (2, 5), (3, 0), (5, 8)\}$$

As there are four data points, we are looking for a third degree polynomial of the form:

$$p(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

The system of equations is then:

$$\begin{array}{ccccccccc} p(0) = & a_3(0)^3 & +a_2(0)^2 & +a_1(0) & +a_0 & = & 1 \\ p(2) = & a_3(2)^3 & +a_2(2)^2 & +a_1(2) & +a_0 & = & 5 \\ p(3) = & a_3(3)^3 & +a_2(3)^2 & +a_1(3) & +a_0 & = & 0 \\ p(5) = & a_3(5)^3 & +a_2(5)^2 & +a_1(5) & +a_0 & = & 8 \end{array}$$

Which leads to the matrix equation:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \\ 125 & 25 & 5 & 1 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \\ 8 \end{bmatrix}$$

We may solve this system to find $a_3 = \frac{16}{15}$, $a_2 = -\frac{23}{3}$, $a_1 = \frac{196}{15}$, $a_0 = 1$. Thus the polynomial is:

$$p(x) = \frac{16}{15}x^3 - \frac{23}{3}x^2 + \frac{196}{15}x + 1$$

8.1.2 Newton's Divided Difference Method

For this method we look for the interpolating polynomial in a particular form:

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \dots + c_n(x - x_0)(x - x_1) \dots (x - x_{n-2})$$

where the constant coefficients c_k are the unknowns.

First note that, since the factor $(x - x_0)$ appears in every term except the first, we have that

$$p(x_0) = c_0 = y_0$$

Similarly,

$$p(x_1) = c_0 + c_1(x_1 - x_0) = y_1$$

Which may be solved, showing

$$c_1 = \frac{y_1 - y_0}{x_1 - x_0}$$

The first constant is just the value of p at x_0 while the second is just the slope of the line through the points (x_0, y_0) and (x_1, y_1) . Things become a more complicated for the higher constants.

$$\begin{aligned} p(x_2) &= c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) = y_2 \\ \Rightarrow y_0 + (x_2 - x_0) \left(\frac{y_1 - y_0}{x_1 - x_0} + (x_2 - x_1)c_2 \right) &= y_2 \end{aligned}$$

With clever manipulation we may solve this as:

$$c_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}$$

We can broadly understand this formula as a “rate of change” of slopes. There is a pattern for the constants c_i , but to represent it we need to introduce some further notation.

We define the interpolating polynomial for a consecutive sequence of points $\{(x_{i-k}, y_{i-k}), (x_{i-k+1}, y_{i-k+1}), \dots, (x_i, y_i)\}$ to be:

$$p(x) = f[x_{i-k}] + f[x_{i-k}, x_{i-k+1}](x - x_{i-k}) + f[x_{i-k}, x_{i-k+1}, x_{i-k+2}](x - x_{i-k})(x - x_{i-k+1}) + \dots \\ \dots + f[x_{i-k}, \dots, x_i](x - x_{i-k}) \dots (x - x_{i-1})$$

It's a bit intimidating, but notice that we're just replacing the constants c_i with the symbols $f[x_0, \dots, x_i]$. It is a more general notation, though, because we could start the sequence someplace other than x_0 . The general formula for these constants is:

$$f[x_{i-k}, \dots, x_i] = \frac{f[x_{i-k+1}, \dots, x_i] - f[x_{i-k}, \dots, x_{i-1}]}{x_i - x_{i-k}} \quad (8.1)$$

This is a recursive formula because the $f[\dots]$ terms with more points are defined in terms of $f[\dots]$ terms with one fewer point. The $f[\dots]$ terms with only one point are defined directly to be:

$$f[x_i] = y_i$$

Example 8.2: Use Newton's Divided Difference Method to find the third degree polynomial through the data points:

$$\mathcal{S} = \{(0, 1), (2, 5), (3, 0), (5, 8)\}$$

We can read the one-point f s right off the data:

$$f[x_0] = 1, \quad f[x_1] = 5, \quad f[x_2] = 0, \quad f[x_3] = 8$$

The two-point f s can be easily calculated using equation 8.1,

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{5 - 1}{2 - 0} = 2$$

$$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{0 - 5}{3 - 2} = -5$$

$$f[x_2, x_3] = \frac{f[x_3] - f[x_2]}{x_3 - x_2} = \frac{8 - 0}{5 - 3} = 4$$

Similarly for the three-point $f[\dots]$:

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{-5 - 2}{3 - 0} = -\frac{7}{3}$$

$$f[x_1, x_2, x_3] = \frac{f[x_2, x_3] - f[x_1, x_2]}{x_3 - x_1} = \frac{4 - (-5)}{5 - 2} = 3$$

And finally,

$$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{x_3 - x_0} = \frac{3 - (-7/3)}{5 - 0} = \frac{16}{15}$$

All this information can be organized more clearly into a table:

i	x_i	$f[x_i]$	$f[x_{i-1}, x_i]$	$f[x_{i-2}, x_{i-1}, x_i]$	$f[x_{i-3}, x_{i-2}, x_{i-1}, x_i]$
0	0	1	—	—	—
1	2	5	2	—	—
2	3	0	-5	$-\frac{7}{3}$	—
3	5	8	4	3	$\frac{16}{15}$

We can simply read the coefficients of the Newton's Divided Difference Polynomial off of the diagonal:

$$p(x) = 1 + 2(x - 0) - \frac{7}{3}(x - 0)(x - 2) + \frac{16}{15}(x - 0)(x - 2)(x - 3)$$

Note that if we were to expand and simplify this polynomial, we would see the same polynomial as we found by the VanderMonte Method in example 8.1.

Note also that if we now wanted to find the interpolating polynomial for just the points $\{(2, 5), (3, 0), (5, 8)\}$, we would need to do no more work. Reading off of the sub-diagonal,

$$p(x) = 5 - 5(x - 2) + 3(x - 2)(x - 3)$$

Exercise 8.3: Construct the Newton's Divided Difference Polynomial for the data:

$$S = \{(-1, 4), (4, -2), (7, 9), (10, -1)\}$$

8.1.3 Programming Newton's Divided Difference

A simple and intuitive way to program Newton's Divided Difference Method is to represent the f portions of the table above as a two-dimensional array. We then write loops to fill in the values of the table, one column at a time.

Let's begin by creating the function, the array, and filling in the first column.

```
from numpy import *

def NewtonsDD(x,y):
    n = len(x)
    M = zeros((n,n))
    M[:,0] = y[:]
    return M
```

For testing purposes we may enter the points given in example 8.2.

```
x = array([0, 2, 3, 5])
y = array([1, 5, 0, 8])
print(NewtonsDD(x,y))
```

Running the file should now produce:

```
[[ 1.  0.  0.  0.]
 [ 5.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 8.  0.  0.  0.]]
```

Now we want a loop which fills in the columns, starting from the second, using the recurrence relation given above. The indexing of the denominator (a difference of x s) is a little tricky here.

```
def NewtonsDD(x,y):
    n = len(x)
    M = zeros((n,n))
    M[:,0] = y[:]
    #Calculate table for Newtons Divided Difference
    for j in range(1,n):
        for i in range(j,n):
            M[i,j] = (M[i,j-1] - M[i-1,j-1])/(x[i]-x[i-j])
    return M
```

Running the file should now yield:

```
[[ 1.  0.  0.  0.  ]
 [ 5.  2.  0.  0.  ]
 [ 0. -5. -2.3333333 0.  ]
 [ 8.  4.  3.  1.06666667]]
```

Which agrees with the result in example 8.2.

Finally it is gratifying (and a good check) to plot the points and the interpolating polynomial. In order to do that we must first see how to evaluate the interpolating polynomial in its somewhat peculiar Newton's Divided Difference form. To do this evaluation we want to think of the polynomial in a "more factored" form. That is:

$$p(x) = f[x_{i-k}] + (x - x_{i-k}) \left[f[x_{i-k}, x_{i-k+1}] + (x - x_{i-k+1}) \left[f[x_{i-k}, x_{i-k+1}, x_{i-k+2}] + \dots \right. \right. \\ \left. \left. \dots + f[x_{i-k}, \dots, x_{i-1}] + (x - x_{i-k}) \left[f[x_{i-k}, \dots, x_{i-1}] \dots \right] \right] \right]$$

For instance the polynomial from example 8.2 would appear as:

$$p(x) = 1 + x \left[2 + (x - 2) \left[-\frac{7}{3} + (x - 3) \left[\frac{16}{15} \right] \right] \right]$$

We may efficiently evaluate this polynomial by starting with the innermost constant, then multiplying by $x - x_{n-2}$ and adding the next constant. We thus work our way from the inside out.

In our code we will define a new variable, `t`, for graphing the polynomial (as `x` is taken). Remember that to plot things we need to import the `matplotlib.pyplot` library.

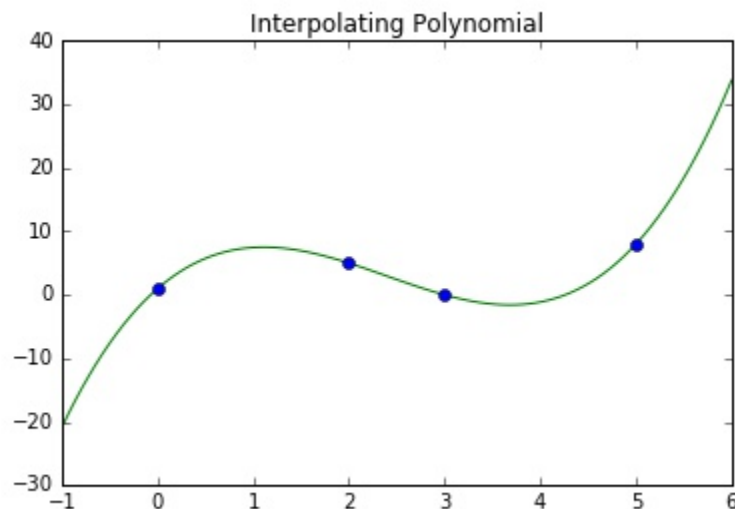
```

from numpy import *
from matplotlib.pyplot import *

def NewtonsDD(x,y):
    n = len(x)
    M = zeros((n,n))
    M[:,0] = y[:]
    #Calculate table for Newtons Divided Difference
    for j in range(1,n):
        for i in range(j,n):
            M[i,j] = (M[i,j-1] - M[i-1,j-1])/(x[i]-x[i-j])
    #Plot resulting polynomial
    a = min(x)-1
    b = max(x)+1
    t = linspace(a,b,100)
    p = zeros(100)+M[n-1,n-1]
    for i in range(n-2,-1,-1):
        p = p*(t-x[i])+M[i,i]
    plot(t,p,'green')
    title('Interpolating Polynomial')
    #Plot data points for reference
    plot(x,y,'o')
    return M

```

Besides the table, this should produce the graph:



Note that the green curve (the graph of the polynomial) really does pass through the data points (blue circles).

8.2 Splines

For large data sets the interpolating polynomial will have very high degree. This makes the polynomial awkward to work with as there may be very sharp peaks and valleys in the corresponding graph. A more natural class of interpolating functions are the *splines*.

An *n-spline* is a piece-wise, n -th degree polynomial function which passes through the given data points, and is thus continuous. Further it will have $n - 1$ continuous derivatives. For example, a 1-spline will be a continuous, piece-wise linear function with (in general) no continuous derivatives. A 1-spline is obtained by simply connecting the data points with lines (the lines will generally meet at the data points in non-differentiable “corners”). These are often also called *Linear Splines*.

A 2-spline will be a piece-wise quadratic function which will be smooth, in that the slopes of the tangent lines to the parabolas will agree at the data points (the second derivative in general will not be continuous). Similarly a 3-spline is made up of cubic functions and has two continuous derivatives. These are also often referred to as *Quadratic* and *Cubic* splines, respectively.

8.2.1 Linear Splines

A 1-spline through n points, $S = \{(x_0, y_0), (x_1, y_1), \dots (x_{n-1}, y_{n-1})\}$ will have the form:

$$s(x) = \begin{cases} a_0 + b_0(x - x_0) & \text{if } x_0 \leq x \leq x_1 \\ a_1 + b_1(x - x_1) & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ a_{n-2} + b_{n-2}(x - x_{n-2}) & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

where the constants $\{a_i\}$, $\{b_i\}$ are to be determined. A 1-spline is only required to pass through the data and be continuous. If we consider the requirement that the spline pass through the data at the left endpoint of each sub-interval, we have:

$$s(x_i) = a_i + b_i(x_i - x_i) = y_i \Rightarrow a_i = y_i$$

The continuity requirement is satisfied if the right endpoints agree as well, giving $n - 1$ equations:

$$s(x_{i+1}) = a_i + b_i(x_{i+1} - x_i) = y_{i+1} \Rightarrow b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

Naturally enough, the constants $\{b_i\}$ are just the slopes of the lines through consecutive endpoints.

Example 8.4: Construct the 1-spline through the data points:

$$S = \{(1, 2), (3, 4), (4, 1), (6, 3)\}$$

We may just read off the $\{a_i\}$ from the data: $a_0 = 2$, $a_1 = 4$, $a_2 = 1$. The $\{b_i\}$ are easily calculated:

$$b_0 = \frac{4-2}{3-1} = 1, \quad b_1 = \frac{1-4}{4-3} = -3, \quad b_2 = \frac{4-2}{3-1} = 1$$

The 1-spline is therefore the function:

$$s(x) = \begin{cases} 2 + (x-1) & \text{if } 1 \leq x \leq 3 \\ 4 - 3(x-3) & \text{if } 3 \leq x \leq 4 \\ 1 + (x-4) & \text{if } 4 \leq x \leq 6 \end{cases}$$

8.2.2 Programming Linear Splines

In fact, we could just plot the 1-spline through a given data set by simply defining the data set and using Python's `plot(x,y)` command. Python automatically connects the points with lines, and that's all that a 1-spline is. Instead, though, let's work harder than we need to and create a function that calculates and plots the 1-spline. This will give us a template to work from when we plot the more complex, higher order splines.

Any spline-plotting program will really have two parts. First it must calculate the constants, and second it must plot the function sub-interval by sub-interval.

The calculation of the $\{b_i\}$ is straight forward for the 1-spline. (We won't even bother to name the $\{a_i\}$ as they are just the $\{y_i\}$.)

```
from numpy import *

def spline1(x,y):
    n = len(x)
    b = zeros(n-1)
    #Continuous at right endpoints: n-1 equ's
    for i in range(0,n-1):
        b[i] = (y[i+1]-y[i])/(x[i+1]-x[i])
    return b
```

To test our program we can use the data from example 8.4.

```
x = array([1,3,4,6])
y = array([2,4,1,3])
print(spline1(x,y))
```

This should produce the correct values for the $\{b_i\}$,

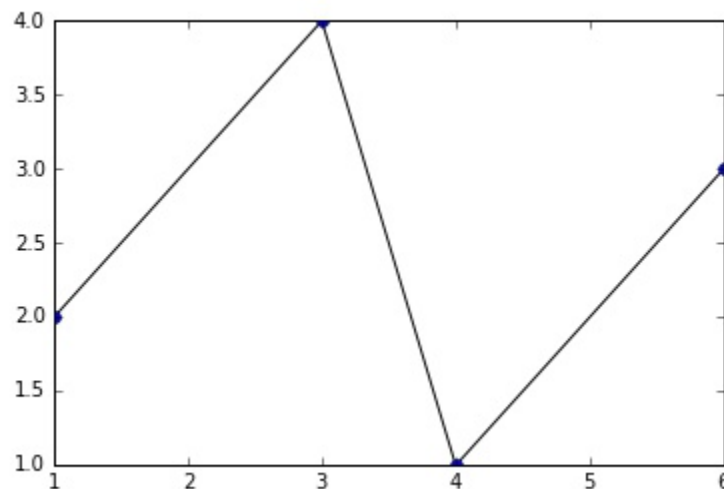
```
[ 1. -3.  1.]
```

To plot the spline, we need only loop through each sub-interval, plotting each line. As we did for the interpolating polynomial, we will define a new variable t to use for the plotting.

```
from numpy import *
from matplotlib.pyplot import *

def spline1(x,y):
    n = len(x)
    b = zeros(n-1)
    #Continuous at right endpoints: n-1 equ's
    for i in range(0,n-1):
        b[i] = (y[i+1]-y[i])/(x[i+1]-x[i])
    #Plot spline
    for i in range(0,n-1):
        t = linspace(x[i],x[i+1],20)
        s = y[i] + b[i]*(t-x[i])
        plot(t,s,'black')
    #Plot Data for reference
    plot(x,y,'bo')
    return b
```

Besides the vector b , this should produce the graph:



8.2.3 Quadratic Splines

A 2-spline through n points, $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ will have the form:

$$s(x) = \begin{cases} a_0 + b_0(x - x_0) + c_0(x - x_0)^2 & \text{if } x_0 \leq x \leq x_1 \\ a_1 + b_1(x - x_1) + c_1(x - x_1)^2 & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ a_{n-2} + b_{n-2}(x - x_{n-2}) + c_{n-2}(x - x_{n-2})^2 & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

where the constants $\{a_i\}$, $\{b_i\}$, $\{c_i\}$ are to be determined. As before, the requirement that the left endpoints pass through the data implies that $a_i = y_i$. The other constants are now more challenging to calculate, however.

The requirement that the spline be continuous at the right endpoints gives us $n - 1$ equations of the form:

$$s(x_{i+1}) = y_{i+1} \Rightarrow y_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = y_{i+1}$$

But a 2-spline is also required to be smooth at the $n - 2$ interior endpoints. We have that:

$$s'(x) = \begin{cases} b_0 + 2c_0(x - x_0) & \text{if } x_0 \leq x \leq x_1 \\ b_1 + 2c_1(x - x_1) & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ b_{n-2} + 2c_{n-2}(x - x_{n-2}) & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

So there are $n - 2$ equations of the form:

$$s'(x_{i+1}^-) = s'(x_{i+1}^+) \Rightarrow b_i + 2c_i(x_{i+1} - x_i) = b_{i+1}$$

There are $n - 1$ b_i s and $n - 1$ c_i s, giving us a total of $2n - 2$ variables which must satisfy $n - 1 + n - 2 = 2n - 3$ linear equations. There is, therefore, necessarily a free variable, and so to hope to get a uniquely defined solution we must add another equation. There are many possible additional constraints which we could add, but we will simply impose the condition that the derivative of the spline at the left endpoint must be a given value m_0 . The final equation is then

$$s'(x_0) = m_0 \Rightarrow b_0 = m_0$$

We must then solve this linear system to find the constants for the 2-spline.

Example 8.5: Construct the 2-spline through the data points:

$$S = \{(1, 2), (3, 4), (4, 1), (6, 3)\}$$

with derivative at $x = 1$ equal to 2.

As before $a_0 = 2$, $a_1 = 4$, $a_2 = 1$. The spline will have the form:

$$s(x) = \begin{cases} 2 + b_0(x - 1) + c_0(x - 1)^2 & \text{if } 1 \leq x \leq 3 \\ 4 + b_1(x - 3) + c_1(x - 3)^2 & \text{if } 3 \leq x \leq 4 \\ 1 + b_2(x - 4) + c_2(x - 4)^2 & \text{if } 4 \leq x \leq 6 \end{cases}$$

We must construct and solve a linear system to find the $\{b_i\}$, $\{c_i\}$.

The continuity equations are:

$$2 + b_0(3 - 1) + c_0(3 - 1)^2 = 4 \Rightarrow 2b_0 + 4c_0 = 2$$

$$4 + b_1(4 - 3) + c_1(4 - 3)^2 = 1 \Rightarrow b_1 + c_1 = -3$$

$$1 + b_2(6 - 4) + c_2(6 - 4)^2 = 3 \Rightarrow 2b_2 + 4c_2 = 2$$

Now the derivative of the spline will have the form:

$$s'(x) = \begin{cases} b_0 + 2c_0(x - 1) & \text{if } 1 \leq x \leq 3 \\ b_1 + 2c_1(x - 3) & \text{if } 3 \leq x \leq 4 \\ b_2 + 2c_2(x - 4) & \text{if } 4 \leq x \leq 6 \end{cases}$$

Therefore the smoothness equations are:

$$b_0 + 2c_0(3 - 1) = b_1 \Rightarrow b_0 - b_1 + 4c_0 = 0$$

$$b_1 + 2c_1(4 - 3) = b_2 \Rightarrow b_1 - b_2 + 2c_1 = 0$$

And our additional constraint that $s'(1) = 2$ gives:

$$b_0 = 2$$

We can write the resulting system as a matrix equation,

$$\begin{bmatrix} 2 & 0 & 0 & 4 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 4 \\ 1 & -1 & 0 & 4 & 0 & 0 \\ 0 & 1 & -1 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 2 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

This system may be solved to obtain the constants:

$$b_0 = 2, \quad b_1 = 0, \quad b_2 = -6, \quad c_0 = -\frac{1}{2}, \quad c_1 = -3, \quad c_2 = \frac{7}{2}$$

The spline is thus:

$$s(x) = \begin{cases} 2 + 2(x - 1) - \frac{1}{2}(x - 1)^2 & \text{if } 1 \leq x \leq 3 \\ 4 - 3(x - 3)^2 & \text{if } 3 \leq x \leq 4 \\ 1 - 6(x - 4) + \frac{7}{2}(x - 4)^2 & \text{if } 4 \leq x \leq 6 \end{cases}$$

Exercise 8.6: Write down the system of linear equation satisfied by constants in the 2-spline through the data points:

$$S = \{(-1, 4), (4, -2), (7, 9), (10, -1)\}$$

with the additional constraint that the derivative at the left end point equals the derivative at the right. That is: $s'(-1) = s'(10)$.

8.2.4 Programming Quadratic Splines

We will follow the template of our program for plotting a 1-spline, but the 2-spline will be much more complicated. Before we can solve for the constants $\{b_i\}$ and $\{c_i\}$, we will first have to construct the matrix of coefficients and the constant vector for the linear system we need to solve. Then we will solve the system using Python's `solve` command (or one of the programs we wrote earlier in the course). Finally we will plot the resulting spline sub-interval by sub-interval just as we did for the 1-spline.

We begin by copying over our 1-spline program, change its name, and import the library which contains the `solve` command. We will use the additional constraint that the slope at the left endpoint is a given value `m0`, so we will need that as an argument. We'll need to define the matrix of coefficients `A` and the vector on the right side of the equation, `B`. Along with the existing constant vector `b`, we will also need a new vector for the constants `c`.

To test our code, we'll leave the definition of `x` and `y` from example 8.4, changing only the name of the function to `spline2`, and adding the argument that $m0 = 2$. We'll comment out the code for the solving and plotting portions which we haven't gotten to yet. As our first task is to find `A` and `B` we'll return those.

```
from numpy import *
from numpy.linalg import solve
from matplotlib.pyplot import *

def spline2(x,y,m0):
    n = len(x)
    b = zeros(n-1)
    c = zeros(n-1)
    A = zeros((2*(n-1),2*(n-1)))
    B = zeros(2*(n-1))
    # #Continuous at right endpoints: n-1 equ's
    # for i in range(0,n-1):
    #     b[i] = (y[i+1]-y[i])/(x[i+1]-x[i])
    # #Plot spline
    # for i in range(0,n-1):
    #     t = linspace(x[i],x[i+1],20)
    #     s = y[i] + b[i]*(t-x[i])
    #     plot(t,s,'black')
    # #Plot Data for reference
```

```
#    plot(x,y,'bo')
    return A,B

x = array([1,3,4,6])
y = array([2,4,1,3])
print(spline2(x,y,2))
```

When we run this code, we simply get a 6×6 zero matrix and a 6×1 zero vector.

As we saw in the previous section, there are $n - 1$ continuity equations,

$$(x_{i+1} - x_i)b_i + (x_{i+1} - x_i)^2 c_i = y_{i+1} - y_i, \quad 0 \leq i \leq n - 2$$

The first $n - 1$ columns in **A** correspond to the **b[i]** variables, while the next $n - 1$ correspond to the **c[i]** variables. Thus,

```
A[i,      i] = x[i+1]-x[i]
A[i, (n-1)+i] =(x[i+1]-x[i])**2
```

assigns the correct values for the coefficients of **b[i]** (first line) and **c[i]** (second line) for the i th continuity equation. Similarly, the right hand side of the equation is assigned to **B[i]**.

```
B[i] = y[i+1] - y[i]
```

The next $n - 2$ rows of **A** come from the smoothness equations:

$$b_i - b_{i+1} + 2(x_{i+1} - x_i)c_i = 0, \quad 0 \leq i \leq n - 3$$

```
A[(n-1)+i,      i]      = 1
A[(n-1)+i,      i+1]    = -1
A[(n-1)+i, (n-1)+i]     = 2*(x[i+1]-x[i])
```

Note that you need to add $n - 1$ to the row number to get past the rows corresponding to continuity equations. Finally we have the additional constraint: $b_0 = m_0$. This is the final row, so we need to skip past both the $n - 1$ continuity equations and the $n - 2$ smoothness equations, thus:

```
A[(n-1)+(n-2), 0] = 1
B[(n-1)+(n-2)] = m0
```

Putting these assignments into our code (using appropriate loops in **i**), we have:

```
def spline2(x,y,m0):
    n = len(x)
    b = zeros(n-1)
    c = zeros(n-1)
    A = zeros((2*(n-1),2*(n-1)))
    B = zeros(2*(n-1))
    #Continuous at right endpoints: n-1 equ's
```

```

for i in range(0,n-1):
    A[i,          i] = (x[i+1]-x[i])
    A[i,  (n-1)+i] = (x[i+1]-x[i])**2
    B[i] = y[i+1]-y[i]
#Smooth at internal endpoints: n-2 equ's
for i in range(0,n-2):
    A[(n-1)+i,          i] = 1
    A[(n-1)+i,          i+1] = -1
    A[(n-1)+i,  (n-1)+i] = 2*(x[i+1]-x[i])
#Set initial slope: 1 equ
A[(n-1)+(n-2),0] = 1
B[(n-1)+(n-2)] = m0
# #Plot spline
# for i in range(0,n-1):
#     t = linspace(x[i],x[i+1],20)
#     s = y[i] + b[i]*(t-x[i])
#     plot(t,s,'black')
# #Plot Data for reference
# plot(x,y,'bo')
return A,B

```

Running this code should produce the matrix and vector found in example 8.5.

```

(array([[ 2.,  0.,  0.,  4.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  2.,  0.,  0.,  4.],
       [ 1., -1.,  0.,  4.,  0.,  0.],
       [ 0.,  1., -1.,  0.,  2.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.]]),
 array([ 2., -3.,  2.,  0.,  0.,  2.]))

```

Next we need to solve the system, then assign the first $n - 1$ elements of the solution to \mathbf{b} , and the next $n - 1$ elements to \mathbf{c} . We can now return \mathbf{b}, \mathbf{c} rather than \mathbf{A}, \mathbf{B} .

```

z = solve(A,B)
b[:] = z[ 0:  n-1]
c[:] = z[n-1:2*(n-1)]

```

Finally, we need only add the $c_i(x - x_i)^2$ terms to the function to be plotted. (We'll also change the color to green, to distinguish it from the 1-spline plot.) The final program should be:

```

def spline2(x,y,m0):
    n = len(x)
    b = zeros(n-1)
    c = zeros(n-1)
    A = zeros((2*(n-1),2*(n-1)))

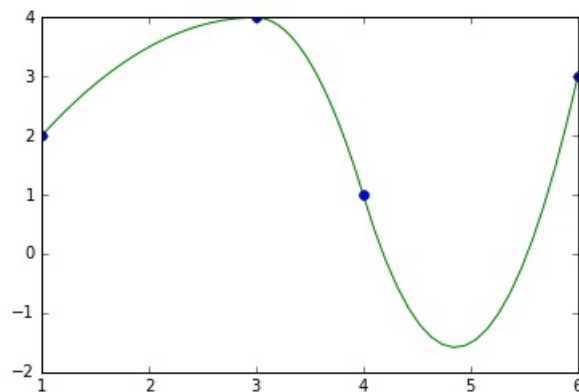
```

```

B = zeros(2*(n-1))
#Continuous at right endpoints: n-1 equ's
for i in range(0,n-1):
    A[i,          i] = (x[i+1]-x[i])
    A[i,  (n-1)+i] = (x[i+1]-x[i])**2
    B[i] = y[i+1]-y[i]
#Smooth at internal endpoints: n-2 equ's
for i in range(0,n-2):
    A[(n-1)+i,          i] = 1
    A[(n-1)+i,          i+1] = -1
    A[(n-1)+i,  (n-1)+i] = 2*(x[i+1]-x[i])
#Set initial slope: 1 equ
A[(n-1)+(n-2),0] = 1
B[(n-1)+(n-2)] = m0
#Solve system and set constants
z = solve(A,B)
b[0:n-1] = z[    0:   n-1]
c[0:n-1] = z[  n-1:2*(n-1)]
print('b = ',b)
print('c = ',c)
#Plot spline
for i in range(0,n-1):
    t = linspace(x[i],x[i+1],20)
    s = y[i] + b[i]*(t-x[i]) + c[i]*(t-x[i])**2
    plot(t,s,'green')
#Plot Data for reference
plot(x,y,'bo')
return b,c

```

Running this code should produce the correct $\{b_i\}$ and $\{c_i\}$ from example 8.5,
 $(\text{array}([2., 0., -6.]), \text{array}([-0.5, -3., 3.5]))$
as well as a nice plot:



Exercise 8.7: Make a copy of `spline2` and call it `spline2bal`. Edit `spline2bal` so that it implements the additional constraint from exercise 8.6, that $s'(x_0) = s'(x_{n-1})$. Test your program for the data in exercise 8.6.

8.2.5 Natural Cubic Spline

The “natural” cubic spline is a 3-spline with the additional constraints that there is no “bending” at the left and right endpoints. This is equivalent to requiring that $s''(x_0) = 0$ and $s''(x_{n-1}) = 0$. The resulting spline is especially “quiet” (shallow peaks and valleys), as well as having an interesting physical interpretation. If one were to place nails in a board, then thread a flexible strip of metal through the nails (so the strip is forced to be in contact with each nail, but may slide), then the strip will assume the shape of the natural cubic spline through the data set represented by the nails. This is because the natural cubic spline minimizes the energy contained in the tension on the strip of metal. (A proof of this interesting fact appears in the appendix.)

A 3-spline through n points, $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ will have the form:

$$s(x) = \begin{cases} y_0 + b_0(x - x_0) + c_0(x - x_0)^2 + d_0(x - x_0)^3 & \text{if } x_0 \leq x \leq x_1 \\ y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ y_{n-2} + b_{n-2}(x - x_{n-2}) + c_{n-2}(x - x_{n-2})^2 + d_{n-2}(x - x_{n-2})^3 & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

where the constants $\{b_i\}$, $\{c_i\}$, $\{d_i\}$ are to be determined. Again, the requirement that the left endpoints pass through the data implies that $a_i = y_i$. The other constants are now much more challenging to calculate, however.

As before, the requirement that the spline be continuous at the right endpoints gives us $n - 1$ equations of the form:

$$s(x_{i+1}) = y_{i+1} \Rightarrow y_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3 = y_{i+1}$$

We'll rewrite these as,

$$(x_{i+1} - x_i)b_i + (x_{i+1} - x_i)^2c_i + (x_{i+1} - x_i)^3d_i = y_{i+1} - y_i$$

Like the 2-spline, a 3-spline is required to be smooth at the $n - 2$ interior endpoints. We have that:

$$s'(x) = \begin{cases} b_0 + 2c_0(x - x_0) + 3d_0(x - x_0)^2 & \text{if } x_0 \leq x \leq x_1 \\ b_1 + 2c_1(x - x_1) + 3d_1(x - x_1)^2 & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ b_{n-2} + 2c_{n-2}(x - x_{n-2}) + 3d_{n-2}(x - x_{n-2})^2 & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

There are again $n - 2$ smoothness equations of the form:

$$s'(x_{i+1}^-) = s'(x_{i+1}^+) \Rightarrow b_i + 2c_i(x_{i+1} - x_i) + 3d_i(x_{i+1} - x_i)^2 = b_{i+1}$$

We'll rewrite these as:

$$b_i - b_{i+1} + 2(x_{i+1} - x_i)c_i + 3(x_{i+1} - x_i)^2 d_i = 0$$

For the 3-spline, there are an additional $n - 2$ concavity equations stemming from the requirement that the second derivative also be continuous at the interior endpoints.

$$s''(x) = \begin{cases} 2c_0 + 6d_0(x - x_0) & \text{if } x_0 \leq x \leq x_1 \\ 2c_1 + 6d_1(x - x_1) & \text{if } x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ 2c_{n-2} + 6d_{n-2}(x - x_{n-2}) & \text{if } x_{n-2} \leq x \leq x_{n-1} \end{cases}$$

$$s''(x_{i+1}^-) = s''(x_{i+1}^+) \Rightarrow 2c_i + 6d_i(x_{i+1} - x_i) = 2c_{i+1}$$

We'll rewrite these as:

$$2c_i - 2c_{i+1} + 6d_i(x_{i+1} - x_i) = 0$$

There are $n - 1$ b_i s, $n - 1$ c_i s, and now $n - 1$ d_i s, giving us a total of $3n - 3$ variables which must satisfy $n - 1 + 2(n - 2) = 3n - 5$ linear equations. The “natural” constraints give us the necessary two additional equations for a unique solution. The final equations are then

$$s''(x_0) = 0 \Rightarrow c_0 = 0$$

and

$$s''(x_{n-1}) = 0 \Rightarrow 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2}) = 0$$

Now we would solve this linear system to find the constants for the natural cubic spline.

Exercise 8.8: Construct the matrix equation for the natural cubic spline through the data points:

$$S = \{(1, 2), (3, 4), (4, 1), (6, 3)\}$$

8.3 Regression Curves

Up to this point we have constructed fairly complex functions which exactly agree with the given data. We will now construct fairly simple functions which are only “close” to the given data.

To use regression, we define a function of a particularly simple form—say a line or an exponential—which depends on some small set of constant parameters, \vec{a} . Then we choose the parameters so as to minimize the error between the actual data and the values predicted by the function.

If the function is $f(\vec{a}, x)$, then one convenient way to measure the error is to calculate:

$$E(\vec{a}) = \sqrt{\sum_{i=0}^{n-1} (f(\vec{a}, x_i) - y_i)^2}$$

(Other types of error measurement could be used to describe the “distance” between the function and the data, but this form is most often used because it is both simple and smooth.)

We are looking for values of the parameters \vec{a} which minimize E . A necessary condition for a stationary point (which may or may not be a minimum) is that, for each j ,

$$\frac{\partial E}{\partial a_j} = 0$$

This gives a system of equations:

$$\frac{1}{E} \sum_{i=0}^{n-1} (f(\vec{a}, x_i) - y_i) \frac{\partial f}{\partial a_j} = 0$$

Which we may simplify to:

$$\sum_{i=0}^{n-1} f(\vec{a}, x_i) \frac{\partial f}{\partial a_j} = \sum_{i=0}^{n-1} y_i \frac{\partial f}{\partial a_j} \quad (8.2)$$

If there are m parameters a_j , then for simple choices of f this can be made to be a linear system of m equation in m unknowns—which we can easily solve. Curves calculated by minimizing this form of the error are called *Least Squares Regression Curves*.

8.3.1 Linear Regression

The simplest useful function to use to interpolate data is a line,

$$f(a_0, a_1, x) = a_0 + a_1 x$$

The parameters are then just the y -intercept and the slope of the line. Applying equation 8.2 for $j = 0$, we have

$$\sum_i (a_0 + a_1 x_i) \frac{\partial}{\partial a_0} (a_0 + a_1 x_i) = \sum_i y_i \frac{\partial}{\partial a_0} (a_0 + a_1 x_i) \Rightarrow \sum_i (a_0 + a_1 x_i)(1) = \sum_i y_i(1)$$

or, after distributing the sum,

$$\left(\sum_i 1 \right) a_0 + \left(\sum_i x_i \right) a_1 = \left(\sum_i y_i \right)$$

Similarly for $j = 1$

$$\sum_i (a_0 + a_1 x_i) \frac{\partial}{\partial a_1} (a_0 + a_1 x_i) = \sum_i y_i \frac{\partial}{\partial a_1} (a_0 + a_1 x_i) \Rightarrow \sum_i (a_0 + a_1 x_i)(x_i) = \sum_i y_i(x_i)$$

or

$$\left(\sum_i x_i \right) a_0 + \left(\sum_i x_i^2 \right) a_1 = \left(\sum_i x_i y_i \right)$$

Written as a matrix equation (and replacing the sum of 1s with the number of points n), we have:

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$

Example 8.9: Find the least squares regression line for the data:

$$\{(1, 3), (2, 6), (4, 10), (5, 9)\}$$

Often it is easiest to find the matrix equation by first making a table with columns for the data and related functions of the data. We then simply find the sum of each column.

x_i	y_i	x_i^2	$x_i y_i$
1	3	1	3
2	6	4	12
4	10	16	40
5	9	25	45
12	28	46	100

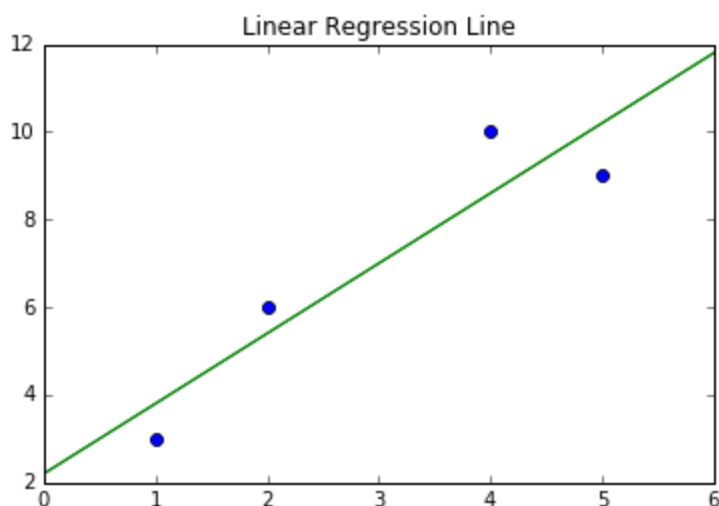
The matrix equation is thus,

$$\begin{bmatrix} 4 & 12 \\ 12 & 46 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 28 \\ 100 \end{bmatrix}$$

Solving this equation gives $a_0 = 2.2$, $a_1 = 1.6$. Thus the least squares regression line is:

$$y = 2.2 + 1.6x$$

Plotting the data points in blue and the regression line in green,



8.3.2 Polynomial Regression

The line is just the simplest form of a polynomial regression function

$$f(\vec{a}, x) = a_0 + a_1x + \dots + a_{m-1}x^{m-1}$$

Using this function we may find least square parabolas, cubics, etc. The equations for the parameters \vec{a} are still very much linear. The j -th equation will be:

$$\sum_i (a_0 + a_1x + \dots + a_{m-1}x_i^{m-1})(x_i)^j = \sum_i y_i(x_i)^j$$

or

$$\left(\sum x_i^j\right) a_0 + \left(\sum x_i^{j+1}\right) a_1 + \dots + \left(\sum x_i^{j+m-1}\right) a_{m-1} = \left(\sum x_i^j y_i\right)$$

the matrix equation is therefore

$$\begin{bmatrix} n & \sum x_i & \dots & \sum x_i^{m-1} \\ \sum x_i & \sum x_i^2 & \dots & \sum x_i^m \\ \vdots & & & \vdots \\ \sum x_i^{m-1} & \sum x_i^m & \dots & \sum x_i^{2m-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \dots \\ \sum x_i^{m-1} y_i \end{bmatrix}$$

Exercise 8.10: Find the least squares regression parabola for the data:

$$\{(1, 3), (2, 6), (4, 10), (5, 9)\}$$

8.4 Exercise Solutions and Problems

Solution to Exercise 8.3

$$f[x_0] = 4, f[x_1] = -2, f[x_2] = 9, f[x_3] = -1$$

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{-2 - 4}{4 - 1} = -\frac{6}{3}$$

$$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{9 - (-2)}{7 - 4} = \frac{11}{3}$$

$$f[x_2, x_3] = \frac{f[x_3] - f[x_2]}{x_3 - x_2} = \frac{-1 - 9}{10 - 7} = -\frac{10}{3}$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{\frac{11}{3} - -\frac{6}{5}}{7 - -1} = \frac{73}{120}$$

$$f[x_1, x_2, x_3] = \frac{f[x_2, x_3] - f[x_1, x_2]}{x_3 - x_1} = \frac{-\frac{10}{3} - \frac{11}{3}}{10 - 4} = -\frac{7}{6}$$

$$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{x_3 - x_0} = \frac{-\frac{7}{6} - \frac{73}{120}}{10 - -1} = -\frac{213}{1320}$$

All this information can be organized into a table:

i	x_i	$f[x_i]$	$f[x_{i-1}, x_i]$	$f[x_{i-2}, x_{i-1}, x_i]$	$f[x_{i-3}, x_{i-2}, x_{i-1}, x_i]$
0	-1	4	—	—	—
1	4	-2	$-\frac{6}{5}$	—	—
2	7	9	$\frac{11}{3}$	$\frac{73}{120}$	—
3	10	-1	$-\frac{10}{3}$	$-\frac{7}{6}$	$-\frac{213}{1320}$

So the Newton's Divided Difference Polynomial is:

$$p(x) = 4 + -\frac{6}{5}(x+1) + \frac{73}{120}(x+1)(x-4) - \frac{213}{1320}(x+1)(x-4)(x-7)$$

Solution to Exercise 8.6

The spline will have the form:

$$s(x) = \begin{cases} 4 + b_0(x+1) + c_0(x+1)^2 & \text{if } -1 \leq x \leq 4 \\ -2 + b_1(x-4) + c_1(x-4)^2 & \text{if } 4 \leq x \leq 7 \\ 9 + b_2(x-7) + c_2(x-7)^2 & \text{if } 7 \leq x \leq 10 \end{cases}$$

We must construct and solve a linear system to find the $\{b_i\}$, $\{c_i\}$.

The continuity equations are:

$$4 + b_0(4+1) + c_0(4+1)^2 = -2 \Rightarrow 5b_0 + 25c_0 = -6$$

$$-2 + b_1(7-4) + c_1(7-4)^2 = 9 \Rightarrow 3b_1 + 9c_1 = 11$$

$$9 + b_2(10-7) + c_2(10-7)^2 = -1 \Rightarrow 3b_2 + 9c_2 = -10$$

Now the derivative of the spline will have the form:

$$s'(x) = \begin{cases} b_0 + 2c_0(x+1) & \text{if } -1 \leq x \leq 4 \\ b_1 + 2c_1(x-4) & \text{if } 4 \leq x \leq 7 \\ b_2 + 2c_2(x-7) & \text{if } 7 \leq x \leq 10 \end{cases}$$

Therefore the smoothness equations are:

$$b_0 + 2c_0(4 + 1) = b_1 \Rightarrow b_0 - b_1 + 10c_0 = 0$$

$$b_1 + 2c_1(7 - 4) = b_2 \Rightarrow b_1 - b_2 + 6c_1 = 0$$

And our additional constraint that $s'(-1) = s'(10)$ gives:

$$b_0 = b_2 + 2c_2(10 - 7) \Rightarrow b_0 - b_2 - 6c_2 = 0$$

We can write the resulting system as a matrix equation,

$$\begin{bmatrix} 5 & 0 & 0 & 25 & 0 & 0 \\ 0 & 3 & 0 & 0 & 9 & 0 \\ 0 & 0 & 3 & 0 & 0 & 9 \\ 1 & -1 & 0 & 10 & 0 & 0 \\ 0 & 1 & -1 & 0 & 6 & 0 \\ 1 & 0 & -1 & 0 & 0 & -6 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} -6 \\ 11 \\ -10 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This system may be solved to obtain the constants:

$$b_0 = -\frac{41}{5}, b_1 = \frac{29}{5}, b_2 = \frac{23}{15}, c_0 = \frac{7}{5}, c_1 = -\frac{32}{45}, c_2 = -\frac{73}{45}$$

The spline is thus:

$$s(x) = \begin{cases} 4 - \frac{41}{5}(x+1) + \frac{7}{5}(x+1)^2 & \text{if } -1 \leq x \leq 4 \\ -2 + \frac{29}{5}(x-4) - \frac{32}{45}(x-4)^2 & \text{if } 4 \leq x \leq 7 \\ 9 + \frac{23}{15}(x-7) - \frac{73}{45}(x-7)^2 & \text{if } 7 \leq x \leq 10 \end{cases}$$

Solution to Exercise 8.7

We need only remove the argument `m0` and change the last row to implement the new constraint.

The equation $s'(x_0) = s'(x_{n-1})$, means

$$b_0 = b_{n-2} + 2c_{n-2}(x_{n-1} - x_{n-2}) \Rightarrow b_0 - b_{n-2} - 2(x_{n-1} - x_{n-2})c_{n-2} = 0$$

Thus we need to assign,

```
A[(n-1)+(n-2), 0] = 1
A[(n-1)+(n-2), n-2] = -1
A[(n-1)+(n-2), (n-1)+n-2] = -2*(x[n-1]-x[n-2])
```

We need not modify `B` as it defaults to the correct entry, zero. The new program with the data from exercise 8.6 is:

```
def spline2bal(x,y):
    n = len(x)
    b = zeros(n-1)
```

```

c = zeros(n-1)
A = zeros((2*(n-1),2*(n-1)))
B = zeros(2*(n-1))
#Continuous at right endpoints: n-1 equ's
for i in range(0,n-1):
    A[i,      i] = (x[i+1]-x[i])
    A[i,  (n-1)+i] = (x[i+1]-x[i])**2
    B[i] = y[i+1]-y[i]
#Smooth at internal endpoints: n-2 equ's
for i in range(0,n-2):
    A[(n-1)+i,      i] = 1
    A[(n-1)+i,      i+1] = -1
    A[(n-1)+i,  (n-1)+i] = 2*(x[i+1]-x[i])
#Set slope of left endpoint equal to slope of right endpoint
A[(n-1)+(n-2),      0] = 1
A[(n-1)+(n-2),      n-2] = -1
A[(n-1)+(n-2),  (n-1)+n-2] = -2*(x[n-1]-x[n-2])
#Solve system and set constants
z = solve(A,B)
b[0:n-1] = z[ 0:  n-1]
c[0:n-1] = z[ n-1:2*(n-1)]
#Plot spline
for i in range(0,n-1):
    t = linspace(x[i],x[i+1],20)
    s = y[i] + b[i]*(t-x[i]) + c[i]*(t-x[i])**2
    plot(t,s,'green')
#Plot Data for reference
plot(x,y,'bo')
return b,c

x = array([-1, 4, 7, 10],double)
y = array([ 4,-2, 9, -1],double)
print(spline2bal(x,y))

```

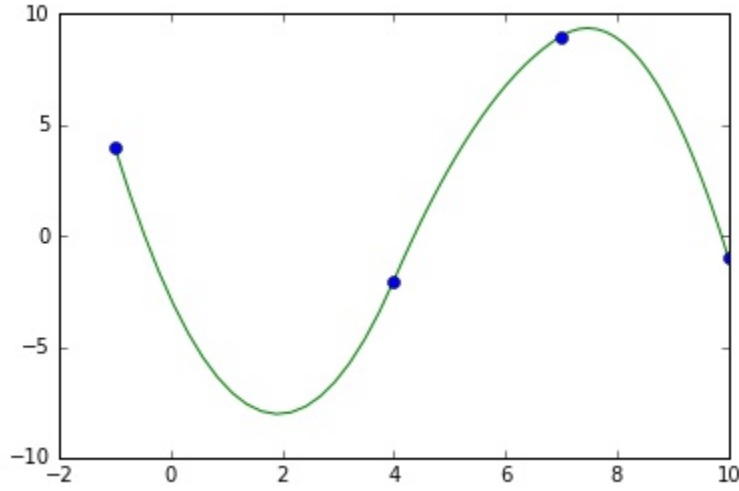
When run, this produces the correct b,c values,

```

(array([-8.2      ,  5.8      ,  1.53333333]),
array([ 1.4      , -0.71111111, -1.62222222]))

```

as well as the graph,



Note that the slopes of the tangent lines at the left and right endpoints seem to be the same.

Solution to Exercise 8.8

The spline will have the form:

$$s(x) = \begin{cases} 2 + b_0(x-1) + c_0(x-1)^2 + d_0(x-1)^3 & \text{if } 1 \leq x \leq 3 \\ 4 + b_1(x-3) + c_1(x-3)^2 + d_1(x-3)^3 & \text{if } 3 \leq x \leq 4 \\ 1 + b_2(x-4) + c_2(x-4)^2 + d_2(x-4)^3 & \text{if } 4 \leq x \leq 6 \end{cases}$$

We must construct and solve a linear system to find the constants $\{b_i\}$, $\{c_i\}$, $\{d_i\}$.

The continuity equations are:

$$2 + b_0(3-1) + c_0(3-1)^2 + d_0(3-1)^3 = 4 \Rightarrow 2b_0 + 4c_0 + 8d_0 = 2$$

$$4 + b_1(4-3) + c_1(4-3)^2 + d_1(4-3)^3 = 1 \Rightarrow b_1 + c_1 + d_1 = -3$$

$$1 + b_2(6-4) + c_2(6-4)^2 + d_2(6-4)^3 = 3 \Rightarrow 2b_2 + 4c_2 + 8d_2 = 2$$

Now the derivative of the spline will have the form:

$$s'(x) = \begin{cases} b_0 + 2c_0(x-1) + 3d_0(x-1)^2 & \text{if } 1 \leq x \leq 3 \\ b_1 + 2c_1(x-3) + 3d_1(x-3)^2 & \text{if } 3 \leq x \leq 4 \\ b_2 + 2c_2(x-4) + 3d_2(x-4)^2 & \text{if } 4 \leq x \leq 6 \end{cases}$$

Therefore the smoothness equations are:

$$b_0 + 2c_0(3-1) + 3d_0(3-1)^2 = b_1 \Rightarrow b_0 - b_1 + 4c_0 + 12d_0 = 0$$

$$b_1 + 2c_1(4-3) + 3d_1(4-3)^2 = b_2 \Rightarrow b_1 - b_2 + 2c_1 + 3d_1 = 0$$

The second derivative of the spline has the form:

$$s''(x) = \begin{cases} 2c_0 + 6d_0(x-1) & \text{if } 1 \leq x \leq 3 \\ 2c_1 + 6d_1(x-3) & \text{if } 3 \leq x \leq 4 \\ 2c_2 + 6d_2(x-4) & \text{if } 4 \leq x \leq 6 \end{cases}$$

So the concavity equations are:

$$2c_0 + 6d_0(3-1) = 2c_1 \Rightarrow 2c_0 - 2c_1 + 12d_0 = 0$$

$$2c_1 + 6d_1(4-3) = 2c_2 \Rightarrow 2c_1 - 2c_2 + 6d_1 = 0$$

The natural constraints give us:

$$s''(1) = 0 \Rightarrow 2c_0 = 0$$

and

$$s''(6) = 0 \Rightarrow 2c_2 + 6d_2(6-4) = 0 \Rightarrow 2c_2 + 12d_2 = 0$$

These give us the matrix equation:

$$\begin{bmatrix} 2 & 0 & 0 & 4 & 0 & 0 & 8 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 4 & 0 & 0 & 8 \\ 1 & -1 & 0 & 4 & 0 & 0 & 12 & 0 & 0 \\ 0 & 1 & -1 & 0 & 2 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 & -2 & 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 6 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 12 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ c_0 \\ c_1 \\ c_2 \\ d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solution to Exercise 8.10

x_i	y_i	x_i^2	x_i^3	x_i^4	$x_i y_i$	$x_i^2 y_i$
1	3	1	1	1	3	3
2	6	4	8	16	12	24
4	10	16	64	256	40	160
5	9	25	125	625	45	225
12	28	46	198	898	100	412

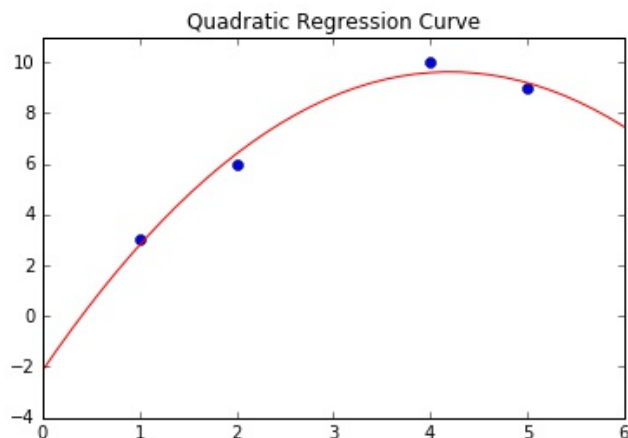
The matrix equation is thus,

$$\begin{bmatrix} 4 & 12 & 46 \\ 12 & 46 & 198 \\ 46 & 198 & 898 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 28 \\ 100 \\ 412 \end{bmatrix}$$

Solving this equation gives $a_0 = -\frac{32}{15}$, $a_1 = \frac{28}{5}$, $a_2 = -\frac{2}{3}$. Thus the least squares regression parabola is:

$$y = -\frac{32}{15} + \frac{28}{5}x - \frac{2}{3}x^2$$

Plotting the data points in blue and the regression parabola in red,



Problem 8.1: This problem will implement the “bad method” for finding the interpolating polynomial that we outlined in class. The method relies on solving a linear system which will become badly conditioned when the number of points gets large.

- a. Write out the equations for the coefficients $\{c_i\}$ of the polynomial

$$p(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4$$

which interpolates the data:

$$S = \{(-2, 3), (1, -1), (4, 8), (5, 6), (7, -10)\}$$

- b. Write this system as a matrix equation, then use Python’s `solve` command to solve this system.
- c. Write a program called `vandersolve` which takes the vectors x and y and returns the coefficients of the interpolating polynomial. Test your program for the information in part(a).

Problem 8.2: Consider the data set where $x = 0, 1, 2 \dots N$ and $y = \cos(0), \cos(1), \cos(2) \dots \cos(N)$.

- a. Compare the results of `vandersolve` with the results provided by the in-class program `NewtonsDD`, by comparing the leading coefficient which should be the same in both methods. How big does N have to be before they are significantly different?
- b. What is the condition number for the Vandermonite matrix for those values of N ?

Problem 8.3:

- a. Write down the system of nine equations for the nine unknowns: $(b_0, b_1, b_2, c_0, c_1, c_2, d_0, d_1, d_2)$ for the **Natural Cubic Spline** through the data points:

$$S = \{(-1, 4), (4, -2), (7, 9), (10, -1)\}$$

Then use Python's `solve` command to solve this system.

- b. Copy our in-class program `spline2ba1`, rename it `spline3`, and modify it so that it takes data vectors x and y , and returns the b_i , c_i , and d_i coefficients of the **Natural Cubic Spline** interpolating that data. It should also produce a red graph of the spline passing through the data points. (The data points should be marked with blue circles.)

Check your program using the data from part(a).

Note that the matrix A will have dimensions $3(n-1) \times 3(n-1)$ and the vector B have a length of $3(n-1)$. When constructing A and B recall that there will be:

- $n - 1$ equations for continuity.
- $n - 2$ equations for smoothness.
- $n - 2$ equations for concavity smoothness.
- 2 equations for the “natural” condition on the endpoints:
 $s''(x_0) = 0 = s''(x_n)$.

- c. Test your program on the data from problem 8.1(a). Include the spline coefficients and the graph.

Problem 8.4: For the data set:

$$S = \{(1, -8), (4, -1), (8, 2), (15, 3)\}$$

- a. Find the least squares regression line.
- b. Find the least squares regression parabola.
- c. For the function:

$$f(\vec{a}, x) = a_0 + a_1 \ln(x)$$

use equation 8.2 to find the general form of the equations for \vec{a} in terms of sums involving x_i and y_i .

- d. Find the least squares regression logarithmic curve for the data given.

Appendix A

Selected Proofs

A.1 Mean Value Theorem for Integrals

Theorem A.1: (Mean Value Theorem for Integrals)

If: G is continuous on $[a, b]$ and $\phi(x) \geq 0$ for every $x \in [a, b]$ or $\phi(x) \leq 0$ for every $x \in [a, b]$, **then** there is a number $\xi \in [a, b]$ so that

$$\int_a^b G(x)\phi(x) dx = G(\xi) \int_a^b \phi(x) dx$$

Before proceeding to the proof, note that this version implies the better known version from elementary calculus. Let $G(x) = f'(x)$ and $\phi(x) \equiv 1$. Then,

$$f(b) - f(a) = \int_a^b f'(x) dx = f'(\xi) \int_a^b dx = f'(\xi)(b - a)$$

proof:

Since G is continuous on $[a, b]$, G achieves a minimum m and maximum M on the interval.

First note that if $\phi(x) \equiv 0$ then the theorem is trivially true. Now consider the case where $\phi(x) \geq 0$ (but not identically zero) on $[a, b]$. Then for every $x \in [a, b]$,

$$m \leq G(x) \leq M \quad \Rightarrow \quad m\phi(x) \leq G(x)\phi(x) \leq M\phi(x)$$

which implies

$$m \int_a^b \phi(x) dx \leq \int_a^b G(x)\phi(x) dx \leq M \int_a^b \phi(x) dx$$

Since $\phi(x)$ is positive somewhere and never negative, $\int_a^b \phi(x) dx > 0$. Thus,

$$m \leq \frac{\int_a^b G(x)\phi(x) dx}{\int_a^b \phi(x) dx} \leq M$$

Again, since G is continuous, by the Intermediate Value Theorem it achieves every value between m and M , so there is a number $\xi \in [a, b]$ such that

$$G(\xi) = \frac{\int_a^b G(x)\phi(x) dx}{\int_a^b \phi(x) dx} \Rightarrow G(\xi) \int_a^b \phi(x) dx = \int_a^b G(x)\phi(x) dx$$

which is our conclusion.

Similarly in the final case where $\phi(x)$ is negative somewhere and never positive, then for every $x \in [a, b]$,

$$m \leq G(x) \leq M \Rightarrow m\phi(x) \geq G(x)\phi(x) \geq M\phi(x)$$

which implies

$$m \int_a^b \phi(x) dx \geq \int_a^b G(x)\phi(x) dx \geq M \int_a^b \phi(x) dx$$

Now since $\int_a^b \phi(x) dx < 0$, we once again have

$$m \leq \frac{\int_a^b G(x)\phi(x) dx}{\int_a^b \phi(x) dx} \leq M$$

and the proof proceeds as in the previous case. \square

A.2 Taylor's Theorem

Theorem 3.1 (Taylor's Theorem)

If: a function $f : \mathbb{R} \rightarrow \mathbb{R}$ has $n + 1$ continuous derivatives on some open interval (a, b) , **then** for any $x, x_0 \in (a, b)$ there exists a number ξ between x and x_0 so that:

$$\begin{aligned} f(x) = & f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \dots \\ & \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1} \end{aligned}$$

proof:

We begin with an intermediate result called a *lemma*.

Lemma:

$$\int_{x_0}^x f^{(n+1)}(t) \frac{(x-t)^n}{n!} dt = f(x) - \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!} (x - x_0)^k$$

The proof of the lemma is by induction.

base: (n=0)

This is just the Fundamental Theorem of Calculus,

$$\int_{x_0}^x f^{(0+1)}(t) \frac{(x-t)^0}{0!} dt = \int_{x_0}^x f'(t) dt = f(x) - f(x_0) = f(x) - \frac{f^{(0)}(x_0)}{0!} (x - x_0)^0$$

inductive:

Integrating by parts,

$$\begin{aligned} \int_{x_0}^x f^{(n+1)}(t) \frac{(x-t)^n}{n!} dt &= f^{(n)}(t) \frac{(x-t)^n}{n!} \Big|_{x_0}^x - \int_{x_0}^x f^{(n)}(t) \frac{(-n)(x-t)^{n-1}}{n!} dt \\ &= -f^{(n)}(x_0) \frac{(x-x_0)^n}{n!} + \int_{x_0}^x f^{(n)}(t) \frac{(x-t)^{n-1}}{(n-1)!} dt \end{aligned}$$

Applying the inductive hypothesis finishes the proof of the lemma.

$$\int_{x_0}^x f^{(n+1)}(t) \frac{(x-t)^n}{n!} dt = -f^{(n)}(x_0) \frac{(x-x_0)^n}{n!} + \left(f(x) - \sum_{k=0}^{n-1} \frac{f^{(k)}(x_0)}{k!} (x-x_0)^k \right)$$

We finish the proof of Taylor's Theorem by applying the Mean Value Theorem for Integrals (Theorem A.1 above).

If $x \geq x_0$ or n is odd, then $(x-t)^{n+1} \geq 0$ for all t between x and x_0 . Otherwise $(x-t)^{n+1} \leq 0$ for all t between x and x_0 , so in either case the Mean Value Theorem applies. Thus there is a number ξ between x and x_0 so that

$$\int_{x_0}^x f^{(n+1)}(t) \frac{(x-t)^n}{n!} dt = f^{(n+1)}(\xi) \int_{x_0}^x \frac{(x-t)^n}{n!} dt = f^{(n+1)}(\xi) \frac{(x-x_0)^{n+1}}{(n+1)!}$$

Substituting this expression into the left side of the lemma completes the proof of Taylor's Theorem. \square

A.3 Improved Trapezoid Rule

Theorem 4.5 (Improved Trapezoid Rule)

If f has four continuous derivatives on $[a, b]$, **then**

$$\int_a^b f(t) dt = \frac{h}{2} (f(a) + f(b)) + h \sum_{i=1}^{n-1} f(x_i) + \left(\frac{f'(a) - f'(b)}{12} \right) h^2 + O(h^4)$$

proof:

We have from Theorem 3.1 that

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + O(h^4)$$

Solving for $f'(x)$ gives the result of Theorem 3.2 with more error terms,

$$f'(x) = \frac{f(x+h) - f(x)}{h} - f''(x)\frac{h}{2} - f'''(x)\frac{h^2}{6} + O(h^3)$$

We also have from Theorem 4.1 that

$$\int_x^{x+h} f(t) dt = f(x)h + f'(x)\frac{h^2}{2} + f''(x)\frac{h^3}{6} + f'''(x)\frac{h^4}{24} + O(h^5)$$

Substituting our expression for f' into our formula for the integral,

$$\begin{aligned} \int_x^{x+h} f(t) dt &= f(x)h + \left[\frac{f(x+h)-f(x)}{h} - f''(x)\frac{h}{2} - f'''(x)\frac{h^2}{6} + O(h^3) \right] \frac{h^2}{2} \\ &\quad + f''(x)\frac{h^3}{6} + f'''(x)\frac{h^4}{24} + O(h^5) \\ &= f(x)h + \left[f(x+h) - f(x) \right] \frac{h}{2} - f''(x)\frac{h^3}{4} - f'''(x)\frac{h^4}{12} \\ &\quad + f''(x)\frac{h^3}{6} + f'''(x)\frac{h^4}{24} + O(h^5) \\ &= \left[f(x+h) + f(x) \right] \frac{h}{2} - f''(x)\frac{h^3}{12} - f'''(x)\frac{h^4}{24} + O(h^5) \end{aligned}$$

We may apply Theorem 3.2 to $f'''(x)$ to express it in terms of $f''(x)$ and $f''(x+h)$,

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

Substituting,

$$\begin{aligned} \int_x^{x+h} f(t) dt &= \left[f(x+h) + f(x) \right] \frac{h}{2} - f''(x)\frac{h^3}{12} - \left[\frac{f''(x+h) - f''(x)}{h} + O(h) \right] \frac{h^4}{24} + O(h^5) \\ &= \left[f(x+h) + f(x) \right] \frac{h}{2} - \left(\left[f''(x+h) + f''(x) \right] \frac{h}{2} \right) \frac{h^2}{12} + O(h^5) \end{aligned}$$

We factor $h^3/24$ in the second term to (hopefully) make clear that the f'' terms just constitute the area of another trapezoid. After we sum the trapezoids,

$$\begin{aligned} \int_a^b f(t) dt &= \sum_{k=0}^{n-1} \int_x^{x+h} f(t) dt \\ &= (f(a) + f(b))\frac{h}{2} + h \sum_{k=1}^{n-1} f(x_k) - \left((f''(a) + f''(b))\frac{h}{2} + h \sum_{k=1}^{n-1} f''(x_k) \right) \frac{h^2}{12} + O(h^4) \end{aligned}$$

But the f'' terms are just the Trapezoid Rule applied to $\int_a^b f''(t)dt$. Thus,

$$\begin{aligned} (f''(a) + f''(b))\frac{h}{2} + h \sum_{k=1}^{n-1} f''(x_k) &= \int_a^b f''(t) dt + O(h^2) \\ &= f'(b) - f'(a) + O(h^2) \end{aligned}$$

Substituting for the f'' terms gives us our conclusion. \square

A.4 Minimal Energy of the Natural Cubic Spline

We may model the energy in the tension of a bent metal ribbon to be a quadratic function of the “bending”, as measured by the second derivative, summed over the length of the curve. That is, if we have a parametrized curve, $\langle x, r(x) \rangle$, for $x_0 \leq x \leq x_{n-1}$, the energy should be:

$$E[r] = \alpha \int_{x_0}^{x_{n-1}} [r''(x)]^2 dx$$

where the positive constant α depends on the flexibility of the material. Note that a straight ribbon would have $r''(x) = 0$ and so zero energy from tension.

Theorem: If s is the natural cubic spline through the points $\{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$, **then** $E[s] \leq E[r]$ for any other twice differentiable curve r that also passes through those points.

proof:

Consider a curve $r = s + t$ which passes through the points, where t is a twice differentiable variation to the natural cubic spline s . Then,

$$r(x_i) = s(x_i) + t(x_i) \Rightarrow y_i = y_i + t(x_i) \Rightarrow t(x_i) = 0$$

for every i . We may find the energy of r ,

$$\begin{aligned} E[r] &= \alpha \int_{x_0}^{x_{n-1}} [(s + t)''(x)]^2 dx \\ &= \alpha \int_{x_0}^{x_{n-1}} [s''(x)]^2 + 2s''(x)t''(x) + [t''(x)]^2 dx \\ &= E[s] + E[t] + 2\alpha \int_{x_0}^{x_{n-1}} s''(x)t''(x) dx \end{aligned}$$

Now for the far right-hand term, we may consider it as a sum, then integrate by-parts:

$$\begin{aligned} \int_{x_0}^{x_{n-1}} s''(x)t''(x) dx &= \sum_{i=1}^{n-1} \int_{x_{i-1}}^{x_i} s''(x)t''(x) dx \\ &= \sum_{i=1}^{n-1} \left[s''(x)t'(x) \Big|_{x_{i-1}}^{x_i} - \int_{x_{i-1}}^{x_i} s'''(x)t'(x) dx \right] \\ &= \sum_{i=1}^{n-1} [s''(x_i)t'(x_i) - s''(x_{i-1})t'(x_{i-1})] - \sum_{i=1}^{n-1} \int_{x_{i-1}}^{x_i} 6d_{i-1}t'(x) dx \end{aligned}$$

The first sum is a telescoping series, while the second may be evaluated using the fundamental theorem. If we now apply the natural boundary conditions and recall that t is zero at the endpoints,

$$\begin{aligned} \int_{x_0}^{x_{n-1}} s''(x)t''(x) dx &= s''(x_{n-1})t'(x_{n-1}) - s''(x_0)t'(x_0) - \sum_{i=1}^{n-1} 6d_{i-1}[t(x_i) - t(x_{i-1})] \\ &= 0 \cdot t'(x_{n-1}) - 0 \cdot t'(x_0) - \sum_{i=1}^{n-1} 6d_{i-1}[0 - 0] \\ &= 0 \end{aligned}$$

Therefore,

$$E[r] = E[s] + E[t] \geq E[s]$$

since $E[t] \geq 0$, \square