Numerical Methods
An Inquiry-Based Approach

Eric Sullivan
Department of Mathematics
Carroll College, Helena, MT
`esullivan@carroll.edu`

Content Last Updated: July 9, 2019

Contents

Chapter 0

To the Student and the Instructor

"Any creative endeavor is built in the ash heap of failure."
–Michael Starbird

This document contains lecture notes, classroom activities, code, examples, and challenge problems specifically designed for an introductory semester of numerical analysis. The content herein is written and maintained by Dr. Eric Sullivan of Carroll College. Problems were either created by Eric Sullivan, the Carroll Mathematics Department faculty, part of NSF Project Mathquest, or come from other sources and are either cited directly or cited in the LaTeX source code for the document.

0.1 An Inquiry Based Approach

Problem 0.1 (Setting The Stage)**.** Let's start the book off right away with a problem designed for groups, discussion, disagreement, and deep critical thinking. This problem is inspired by Dana Ernst's first day IBL activity titled: Setting the Stage.

- Get in groups of size 3-4.
- Group members should introduce themselves.
- For each of the questions that follow I will ask you to:
    1. **Think** about a possible answer on your own
    2. **Discuss** your answers with the rest of the group
    3. **Share** a summary of each group's discussion

Questions:

Question #1: What are the goals of a university education?

Question #2: How does a person learn something new?

Question #3: What do you reasonably expect to remember from your courses in 20 years?

Question #4: What is the value of making mistakes in the learning process?

Question #5: How do we create a safe environment where risk taking is encouraged and productive failure is valued?

▲

This material is written with an Inquiry-Based Learning (IBL) flavor. In that sense, this document could be used as a stand-alone set of materials for the course but these notes are not a *traditional textbook* containing all of the expected theorems, proofs, code, examples, and exposition. The students are encouraged to work through problems and homework, present their findings, and work together when appropriate. You will find that this document contains collections of problems with only minimal interweaving exposition. It is expected that you do every one of the problems and then use other more traditional texts as a backup when you are stuck. Let me say that again: this is not the only set of material for the course. Your brain, your peers, and the books linked in the next section are your best resources when you are stuck.

To learn more about IBL go to http://www.inquirybasedlearning.org/about/. The long and short of it is that the students in the class are the ones that are doing the work; proving theorems, writing code, working problems, leading discussions, and pushing the pace. The instructor acts as a guide who only steps in to redirect conversations or to provide necessary insight. If you are a student using this material you have the following jobs:

1. **Fight!** You will have to fight hard to work through this material. The fight is exactly what we're after since it is ultimately what leads to innovative thinking.

2. **Screw Up!** More accurately, don't be afraid to screw up. You should write code, work problems, and prove theorems then be completely unafraid to scrap what you've done and redo it from scratch. Learning this material is most definitely a non-linear path.

3. **Collaborate!** You should collaborate with your peers with the following caveats:

   (a) When you are done collaborating you should go your separate ways. When you write your solution you should have no written (or digital) record of your collaboration.

   (b) <u>The internet is not a collaborator</u>. Use of the internet to help solve these problems robs you of the most important part of this class; the chance for original thought.

4. **Enjoy!** Part of the fun of IBL is that you get to experience what it is like to think like a true mathematician / scientist. It takes hard work but ultimately this should be fun!

0.2 Online Texts and Other Resources

If you are looking for online textbooks for numerical methods or numerical analysis I can point you to a few of my favorites. Some of the following online resources may be a good place to help you when you're stuck but they will definitely say things a bit differently. Use these resources wisely.

- Holistic Numerical Methods http://nm.mathforcollege.com/
  The Holistic Numerical Methods book is probably the most complete free reference that I've found on the web. This should be your source to look up deeper explanations of problems, algorithms, and code.
- Scientific Computing with MATLAB http://gribblelab.org/scicomp/scicomp.pdf
- Tea Time Numerical Analysis http://lqbrin.github.io/tea-time-numerical/

0.3 To the Instructor

If you are an instructor wishing to use these materials then I only ask that you adhere to the Creative Commons license. You are welcome to use, distribute, and remix these materials for your own purposes. Thanks for considering my materials for your course! Let me know if you have questions, edits, or suggestions: esullivan@carroll.edu.

0.4 Special Thanks

I would first like to thank Dr. Kelly Cline for being brave enough to teach a course that he loves out of a rough draft of my notes. Your time, suggested edits, and thoughts for future directions of the book were, and are, greatly appreciated. You've taught me a lot in the short time that we've worked together. Thanks! Second, I would like to thank my wife, Johnanna, for simply being awesome. I would like to thank the institution of Carroll College for seeing this project as a worthy academic pursuit even though the end result is not a book or publication in the traditional sense. Finally, I would like to thank all of my colleagues and students, both past and present, in the math department at Carroll. The suggestions, questions, struggles, and triumphs of these folks are what have shaped this work into something that I'm proud of and that I hope will be a useful resource for future students and instuctors.

Appendix A

Python Basics

Note to the reader: In future versions of this book we will exclusively be using Python for the programming language of choice. This appendix will eventually be rolled into Chapter 1 as "Introductory" material that is optional for students with prior programming experience.

In this optional Chapter we will walk through some of the basics of using Python3 - the powerful general-purpose programming language that we'll use throughout this class. I'm assuming throughout this Chapter that you're familiar with other programming languages such as R, Java, C, or MATLAB. Hence, I'm assuming that you know the basics about what a programming langue ”is” and ”does”. There are a lot of similarities between several of these languages, and in fact they borrow heavily from each other in syntax, ideas, and implementation.

We are going to be using Python in this class since

- Python is free,
- Python is very widely used,
- Python is flexible,
- Python is relatively easy to learn,
- and Python is quite powerful.

It is important to keep in mind that Python is a general purpose language that we will be using for Scientific Computing. The purpose of Scientific Computing is **not** to build apps, build software, manage databases, or develope user interfaces. Instead, Scientific Computing is the use of a computer programming language (like Python) along with mathematics to solve scientific and mathematical problems. For this reason it is definitely not our purpose to write an all-encompassing guide for how to use Python. We'll only cover what is necessary for our computing needs. You'll learn more as the course progresses so use this chapter as a reference just to get going with the language.

There is a wealth of information available about Python and the suite of tools that we will frequently use.

- Python https://www.python.org/,
- NumPy (numerical python) https://www.numpy.org/,
- SciPy (scientific python) https://www.scipy.org/, and
- SymPy (symbolic python) https://www.sympy.org/en/index.html.

These tools together provide all of the computational power that will need. And they're free!

A.1 Getting Started

Every computer is its own unique flower with its own unique requirements. Hence, we will not spend time here giving you all of the ways that you can install Python and all of the associated packages necessary for this course. We highly recommend that you go to https://www.python.org/downloads/ and follow the appropriate links. The environment in which you code is largely up to personal preference (or perhaps instructor preference). Jupyter Notebooks seem to be a good modern choice for coding environments. For more information and for installation of Jupyter see https://jupyter.org/.

In the rest of this chapter we will assume that you have a working version of Python along with (most likely) a working version of Jupyter Notebooks to work in. Throughout this chapter all code will be highlighted in boxes so you, the reader, can easily tell that it is supposed to be code. The output of the code may also be in a box. Lastly, the code and the output have line numbers so the reader can more easily keep track of the commands, discuss with their peers, and discuss with their instructor. The problems in this appendix are meant to get you going with the Python and you should do every one of them. Remember that this is not a full replacement for a "how to program in Python" resource. We have only included the essential aspects of the Python language in this chapter as they relate to the mathematical goals of the book. There is definitely more to say about Python and we don't intend to cover it all here.

As is tradition for a new programming language, we should create code that prints the words "Hello, world!" to the screen. The code below does just that.

Code:

```
1 print("Hello, world!")
```

Output:

```
1 Hello, world!
```

Problem A.1. Write code to print your name to the screen. ▲

Python is a general purpose programming language that can be used for all sorts of programming tasks. In this book we will focus on uses of Python that are more computational and mathematical in nature. It is expected that you already know a bit a programming from another math class that required some coding, possibly from an introductory computer science class, or from some other exposure to coding. If not then don't fret – not all hope is lost. A good place to start is to ask your instructor for additional programming materials.

A.2 Python Programming Basics

A.2.1 Variables

Variables in Python can contain letters (lower case or capital), numbers 0-9, and some special characters such as the underscore. Variable names should start with a letter. Of course there are a bunch of reserved words (just like in any other language). You should look up what the reserved words are in Python so you don't accidentally use them.

You can do the typical things with variables. Assignment is with an equal sign (be careful R users!).

Warning: When defining numerical variables you don't always get floating point numbers like in MATLAB. In MATLAB if you write `x=1` then automatically `x` is saved as 1.0; a floating point decimal number, not an integer. However, in Python if you assign `x=1` it is defined as an integer (with no decimal digits) but if you assign `x=1.0` it is assigned as a floating point number.

Example A.2 (Number Types in Python)**. Code:**

```
# assign some variables
x = 7 # integer assignment of the integer 7
y = 7.0 # floating point assignment of the decimal number 7.0
print(x, type(x))
print(y, type(y))

# multiplying by a float will convert an integer to a float
print(1.0*x , type(1.0*x))
```

Output:

```
7 <class 'int'>
7.0 <class 'float'>
7.0 <class 'float'>
```

Note that the allowed mathematical operations are:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Integer Division (modular division): // and
- Exponents: **

That's right, the caret key, ^, is NOT an exponent in Python (sigh). Instead we have to get used to `**` for powers.

Example A.3 (Powers in Python)**.** Keep in mind that the caret key is not an exponent in Python.
Code:

```
x = 7.0
y = x**2 # square the value in x
print(y)
```

Output:

```
49.0
```

Problem A.4. What happens if you type 7^2 into Python? What does it give you? Can you figure out what it is doing? ▲

Problem A.5. Write code to define the variables a, b, and c of your own choosing. Then calculate a^2, b^2, and c^2. When you have all three computed, check to see if your three values form a Pythagorean Triple so that $a^2 + b^2 = c^2$ and have Python simply say True or False to verify that you do, or do not, have a Pythagorean Triple defined.
Hint: You will need to use the == Boolean check just like in other programming languages. ▲

A.2.2 Indexing and Lists

Lists are a key component to storing data in Python. Lists are exactly what the name says: lists of things (in our case, usually the entries are floating point numbers).

Warning to MATLAB users: Python indexing starts at 0 whereas MATLAB indexing starts at 1. We just have to keep this in mind.

We can extract a part of a list using the syntax [start:stop] which extracts elements between index start and stop-1.
NOTE: Python stops reading at the second to last index.

Some things to keep in mind with Python lists:

- Python starts indexing at 0
- Python stops reading at the second to last index
- The following blocks of code show this feature in action for several different lists.

Example A.6. Let's look at a few examples of indexing from lists. In this example we will use the list of numbers 1 through 8.

- Create the list of numbers 1 through 8 and then print only the element with index 0.

  Code:

```
MyList = [1,2,3,4,5,6,7,8]
print(MyList[0])
```

Output:

```
1
```

- Print all elements up to, but not including, the third element.

  Code:

  ```
  print(MyList[:2])
  ```

  Output:

  ```
  [1, 2]
  ```

- Print the last element (this is a handy trick!).

  Code:

  ```
  print(MyList[-1])
  ```

  Output:

  ```
  8
  ```

- Print the elements indexed 1 through 4. Beware! This is not the first through fifth element.

  Code:

  ```
  print(MyList[1:5])
  ```

  Output:

  ```
  [2, 3, 4, 5]
  ```

Example A.7. Let's look at another example of indexing in lists. In this one we'll use the range command to build the initial list of numbers. Read the code carefully so you know what each line does.

Code:

```
MySecondList = list(range(4,20)) # range is a handy command for creating a sequence of
print(MySecondList) # notice that it didn't create the last element!
print(MySecondList[0]) # print the first element ... the one with index 0
print(MySecondList[-5]) # print the fifth element from the end
print(MySecondList[-1:0:-1]) # print the last element to the one indexed by 1 counting
print(MySecondList[::2]) # print every other element starting at the beginning
```

Output:

```
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
4
15
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5]
[4, 6, 8, 10, 12, 14, 16, 18]
```

In Python, elements in a list do not need to be the same type. You can mix integers, floats, strings, lists, etc.

Example A.8 (Lists with elements of mixed type)**.** In this example we see a list of several items that have different data types: float, integer, string, and complex. Note that the imaginary number i is represented by j in Python. This is common in many scientific disciplines and is just another thing that we'll need to get used to in Python.
Code:

```
MixedList = [1.0, 7, 'Bob', 1-1j]
print(MixedList)
print(type(MixedList[0]))
print(type(MixedList[1]))
print(type(MixedList[2]))
print(type(MixedList[3])) # Notice that we use 1j for the imaginary number "i".
```

Output:

```
[1.0, 7, 'Bob', (1-1j)]
<class 'float'>
<class 'int'>
<class 'str'>
<class 'complex'>
```

Problem A.9. (a) Create the list of the first several Fibonacci numbers:
$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.$

(b) Print the first four elements of the list.

(c) Print every third element of the list.

(d) Print the last element of the list.

▲

A.2.3 List Operations

Python is awesome about allowing you to do things like appending items to lists, removing items from lists, and inserting items into lists. Note in all of the examples below that we are using the code
`variable.command`
where you put the variable name, a dot, and the thing that you would like to do to that variable. For example, `MyList.append(7)` will append the number 7 to the list `MyList`. This is a common programming feature in Python and we'll use it often.

Example A.10 (Appending To Lists)**.** The .append command can be used to append an element to the end of a list.
Code:

```
MyList = [0,1,2,3]
print(MyList)
MyList.append('a') # append the string 'a' to the end of the list
print(MyList)
MyList.append('a') # do it again ... just for kicks
print(MyList)
MyList.append(15) # append the number 15 to the end of the list
print(MyList)
```

Output:

```
[0, 1, 2, 3]
[0, 1, 2, 3, 'a']
[0, 1, 2, 3, 'a', 'a']
[0, 1, 2, 3, 'a', 'a', 15]
```

Example A.11 (Removing From Lists)**.** The .remove command can be used to remove an element from a list.
Code:

```
MyList.remove('a') # remove the first instance of the string `a` from the list
print(MyList)
MyList.remove(3) # now let's remove the 3
print(MyList)
```

Output:

```
[0, 1, 2, 3, 'a', 15]
[0, 1, 2, 'a', 15]
```

Example A.12 (Inserting Into Lists)**.** The .insert command can be used to insert an element at a location in a list.
Code:

```
MyList.insert(0,'A') # insert the letter `A` at the 0-indexed spot
MyList.insert(3,'B') # insert the letter `B` at the spot with index 3
# remember that index 3 means the fourth spot in the list
print(MyList)
```

Output:

```
['A', 0, 1, 'B', 2, 'a', 15]
```

Problem A.13. (a) Create the list of the first several Lucas Numbers: $1, 3, 4, 7, 11, 18, 29, 47$.

(b) Add the next three Lucas Numbers to the end of the list.

(c) Remove the number 3 from the list.

(d) Insert the 3 back into the list in the correct spot.

(e) Print the list in reverse order (how do you suppose you should do this?)

(f) Do a few other list operations to this list and report your findings.

▲

A.2.4 Tuples

In Python, a ”tuple” is like an ordered pair (or order triple, or order quadruple, ...) in mathematics. We will occasionally see tuples in our work in numerical analysis so for now let's just give a couple of code snippets showing how to store and read them.

Example A.14 (Defining Tuples)**.** We can define the tuple of numbers $(10, 20)$ in Python as follows.
Code:

```
point = 10, 20 # notice that I don't need the parenthesis
print(point, type(point))
```

Output:

```
(10, 20) <class 'tuple'>
```

We can also define the type with parenthesis if we like. Python doesn't care.
Code:

```
point = (10, 20) # now we define the tuple with parenthesis
print(point, type(point))
```

Output:

```
(10, 20) <class 'tuple'>
```

Example A.15 (Unpacking Tuples)**.** The cool thing is that we can then unpack the tuple into components if we wish. **Code:**

```
x, y = point
print("x = ", x)
print("y = ", y)
```

Output:

```
x =  10
y =  20
```

A.2.5 Control Flow: Loops and If Statements

Just like in other programming languages we can do loops and conditional statements in very easy ways. The thing to keep in mind is that Python is very white-space-dependent. This means that your indentations need to be correct in order for a loop to work. You could get away with sloppy indention in other languages (like MATLAB) but not so in Python. Also, in some languages (like R and Java) you need to wrap your loops in curly braces. Again, not so in Python.

Caution: Be really careful of the white space in your code when you write loops.

For Loops

A for loop is designed to do a task a certain number of times and then stop. This is a great tool for automating repetitive tasks, but it also nice numerically for building sequences, summing series, or just checking lots of examples. The following are several examples of Python for loops. Take careful note of the syntax for a for loop as it is the same as for other loops and conditional statements:

- a control statement,
- a colon, a new line,
- indent four spaces,
- some programming statements

When you are done with the loop just back out of the indention. There is no need for an "end" command or a curly brace. All of the control statements in Python are white-space-dependent.

Example A.16. Print the first 6 perfect square.
Code:

```
for x in [1,2,3,4,5,6]:
    print(x**2)
```

Output:

```
1.0
4.0
9.0
16.0
25.0
36.0
```

Example A.17. Print the names in a list.
Code:

```
NamesList = ['Alice','Billy','Charlie','Dom','Enrique','Francisco']
```

```
for name in NamesList:
    print(name)
```

Output:

```
Alice
Billy
Charlie
Dom
Enrique
Francisco
```

You can use a more compact notation sometimes. This takes a bit of getting used to, but is super slick!

Example A.18. Create a list of the perfect squares from 1 to 9.
Code:

```
# create a list of the perfect squares from 1 to 9
CoolList = [x**2 for x in range(1,10)]
print(CoolList)
# Then print the sum of this list
print("The sum of the first 9 perfect squares is",sum(CoolList))
```

Output:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
The sum of the first 9 perfect squares is 285
```

For loops can also be used to build recursive sequences as can be seen in the next couple of examples.

Example A.19. In the following code we write a for loop that outputs a list of the first 10 iterations of the sequence $x_{n+1} = 0.5x_n + 1$ starting with $x_0 = 3$. Notice that we're using `x.append` instead of $x[n+1]$ to append the new term to the list. This allows us to grow the length of the list dynamically as the loop progresses.
Code:

```
x=[3.0]
for n in range(0,9):
    x.append(-0.5*x[n] + 1)
print(x)
```

Output:

```
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375,
    0.67578125, 0.662109375]
```

As an alternative to the code immediately above we can pre-allocate the memory in an array of zeros. This is done with the clever code `x = [0] * 10`. Literally multiplying a list by some number, like 10, says to repeat that list 10 times.

Example A.20. Now we'll build the sequence with pre-allocated memory.
Code:

```
x = [0] * 10
x[0] = 3.0
for n in range(0,9):
    x[n+1] = -0.5*x[n]+1
    print(x) # This print statement shows x at each iteration
```

Output:

```
[3.0, -0.5, 0, 0, 0, 0, 0, 0, 0, 0]
[3.0, -0.5, 1.25, 0, 0, 0, 0, 0, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0, 0, 0, 0, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0, 0, 0, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0, 0, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0.67578125, 0]
[3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0.67578125, 0.662109375
```

Problem A.21. We want to sum the first 100 perfect cubes. Let's do this in two ways.

1. Start off a variable called Total at 0 and write a for loop that adds the next perfect cube to the running total.
2. Write a for loop that builds the sequence of the first 100 perfect cubes. After the list has been built find the sum with the sum command.

The answer is: 25,502,500 so check your work. ▲

Problem A.22. Write a for loop that builds the first 20 terms of the sequence $x_{n+1} = 1x^2$ with $x_0 = 0.1$ Pre-allocate enough memory in your list and then fill it with the terms of the sequence. Only print the list after all of the computations have been completed. ▲

While Loops

A while loop repeats some task (or sequence of tasks) until a logical condition is met. The structure in Python is the same as with for loops.

Example A.23. Print the numbers 0 through 4 and then the word "done". We'll do this by starting a counter variable, `i`, at 0 and incrementing it every time we pass through the loop.
Code:

```
i = 0
while i < 5:
    print(i)
    i += 1 # increment the counter
print("done")
```

Output:

```
0
1
2
3
4
done
```

Example A.24. Now let's use a while loop to build the sequence of Fibonacci numbers and stop when the newest number in the sequence is greater than 1000. Notice that we want to keep looping until the condition that the last term is greater than 1000 – this is the perfect task for a while loop, instead of a for loop, since we don't know how many steps it will take before we start the task

Code:

```python
Fib = [1,1]
while Fib[-1] <= 1000:
    Fib.append(Fib[-1] + Fib[-2])
Fib
```

Output:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Problem A.25. Write a while loop that sums the terms in the Fibonacci sequence until the sum is larger than 1000 ▲

If Statements

Conditional (if) statements allow you to run a piece of code only under certain conditions. This is handy when you have different tasks to perform under different conditions.

Example A.26. Code:

```python
Name = "Alice"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
else:
    print("You're not Alice.  Where is Alice?")
```

Output:

```
Hello, Alice.  Isn't it a lovely day to learn Python?
```

Code:

```python
Name = "Billy"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
```

```
else:
    print("You're not Alice.  Where is Alice?")
```

Output:

```
You're not Alice.  Where is Alice?
```

Example A.27. For another example, if we get a random number between 0 and 1 we could have Python print a different message depending on whether it was above or below 0.5. Run the code below several times and you'll see different results each time.

Note: We had to import the numpy package to get the random number generator in Python. Don't worry about that for now. We'll talk about packages in a moment.
Code:

```
import numpy as np
x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
x = x[0,0] # pull the entry from the first row, first column of the random matrix
if x < 0.5:
    print(x," is less than a half")
else:
    print(x, "is NOT less than a half")
```

Output:

```
0.654697487883 is NOT less than a half
```

(Take note that the output will change every time you run it)

In many programming tasks it is handy to have several different choices between tasks instead of just two choices as in the previous examples. This is a job for the `elif` command.

Example A.28. This is the same code as last time except we will make the decision at 0.33 and 0.67 **Code:**

```
import numpy as np
x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
x = x[0,0] # pull the entry from the first row, first column of the random matrix
if x < 0.33:
    print(x," is less than one third")
elif x < 0.67:
    print(x, "is less than two thirds but greater than or equal to one third")
else:
    print(x, "is greater than or equal to two thirds")
```

Output:

```
0.654697487883 is less than two thirds but greater than or equal to one third
```

(Take note that the output will change every time you run it)

Problem A.29. Write code to give the Collatz Sequence

$$x_{n+1} = \begin{cases} x_n/2, & x_n \text{ is even} \\ 3x_n + 1, & \text{otherwise} \end{cases}$$

starting with a positive integer of your choosing. The sequence will converge to 1 so your code should stop when the sequence reaches 1.

▲

A.2.6 Functions

Mathematicians and programmers talk about functions in very similar ways, but they aren't exactly the same. When we say "function" in a programming sense we are talking about a chunk of code that you can pass parameters and expect an output of some sort. This is not unlike the mathematician's version, but unlike a mathematical function we can have multiple outputs for a programmatic function. We are not going to be talking about symbolic computation on functions in this section. Symbolic computations will have to wait for the 'sympy' tutorial.

In Python, to define a function we start with `def`, followed by the function's name, any input variables in parenthesis, and a colon. The indented code after the colon is what defines the actions of the function.

Example A.30. The following code defines the polynomial $f(x) = x^3 + 3x^2 + 3x + 1$ and then evaluates the function at a point $x = 2.3$.

Code:

```
def f(x):
    return(x**3 + 3*x**2 + 3*x + 1)
f(2.3)
```

Output:

```
35.937
```

Take careful note of several things in the previous example:

- To define the function we can not just type it like we would see it one paper. This is not how Python recognizes functions. We just have to get used to it. Other scientific programming languages will allow you to define mathematical functions in this way, but Python will not.
- Once we have the function defined we can call upon it just like we would on paper.
- We cannot pass symbols into this type of function. See the section on `sympy` in this chapter if you want to do symbolic computation.

Problem A.31. Define the function $g(n) = n^2 + n + 41$ as a Python function. Write a loop that gives the output for this function for integers from $n = 0$ to $n = 39$. It is curious to note that each of these outputs is a prime number (check this on your own). Will the function produce a prime for $n = 40$? For $n = 41$? ▲

One cool thing that you can do with Python functions is call them recursively. That is, you can call the same function from within the function itself. This turns out to be really handy in several mathematical situations.

Example A.32. Now let's define a function for the factorial. This function is naturally going to be recursive in the sense that it calls on itself!
Code:

```
def Fact(n):
    if n==0:
        return(1)
    else:
        return( n*Fact(n-1) ) # we are calling the same function recursively.
```

When you run this code there will be no output. You have just defined the function so you can use it later. So let's use it to make a list of the first several factorials. Note the use of a for loop in the following code.
Code:

```
FactList = [Fact(n) for n in range(0,10)]
FactList
```

Output:

```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

Example A.33. For this next example let's define the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, 0.5] \\ 2x_n - 1, & x_n \in (0.5, 1] \end{cases}$$

as a function and then build a loop to find the first several iterates of the sequence starting at any real number between 0 and 1.
Code:

```
# Define the function
def MySeq(xn):
    if xn <= 0.5:
        return(2*xn)
    else:
        return(2*xn-1)

# Now build a sequence with this function
x = [0.125] # arbitrary starting point
```

```
for n in range(0,5): # Let's only build the first 5 terms
    x.append(MySeq(x[-1]))
print(x)
```

Output:

```
[0.125, 0.25, 0.5, 1.0, 1.0, 1.0]
```

Example A.34. Here is another cool example.

A fun way to approximate the square root of two is to start with any positive real number and iterate over the sequence

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}$$

until we are within any tolerance we like of the square root of two. Write code that defines the sequence as a function and then iterates in a while loop until we are within 10^{-8} of the square root of 2.

Hint: Import the `math` package so that you get the square root. More about packages in the next section.

Code:

```
from math import sqrt
def f(x):
    return(0.5*x + 1/x)
x = 1.1 # arbitrary starting point
print("approximation \t\t exact \t\t abs error")
while abs(x-sqrt(2)) > 10**(-8):
    x = f(x)
    print(x, sqrt(2), abs(x - sqrt(2)))
```

Output:

```
approximation        exact        abs error
1.459090909090909 1.4142135623730951 0.044877346717781385
1.414903709997168 1.4142135623730951 0.0006901476240728233
1.4142137306897584 1.4142135623730951 1.6831663273770 93e-07
1.4142135623731051 1.4142135623730951 9.992007221626409e-15
```

Problem A.35. The previous example is a special case of the Babylonian Algorithm for calculating square roots. If you want the square root of S then iterate the sequence

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

until you are within an appropriate tolerance.

Modify the code given in the previous example to give a list of approximations of the square roots of the natural numbers 2 through 20, each to within 10^{-8}. This problem will require that you build a function, write a 'for' loop (for the integers 2-20), and write a 'while' loop inside your 'for' loop to do the iterations. ▲

A.2.7 Packages

Unlike mathematical programming languages like MATLAB, Maple, or Mathematica, where every package is already installed and ready to use, Python allows you to only load the packages that you might need for a given task.* There are several advantages to this along with a few disadvantages.

Advantages:

1. You can have the same function doing different things in different scenarios. For example, there could be a symbolic differentiation command and a numerical differentiation command comming from different packages that are used in different ways.

2. Housekeeping. It is highly advantageous to have a good understanding of where your functions come from. MATLAB uses the same name for multiple purposes with no indication of how it might behave depending on the inputs. With Python you can avoid that by only importing the appropriate packages for your current use.

3. Your code will be ultimately more readable (more on this later).

Disadvantages:

1. It is often challenging to keep track of which function does which task when they have exactly the same name. For example, you could be working with the `sin()` function numerically from the `numpy` package or symbolically from the `sympy` package, and these functions will behave differently in Python - even though they are exactly the same mathematically.

2. You need to remember which functions live in which packages so that you can load the right ones. It is helpful to keep a list of commonly used packages and functions at least while you're getting started.

Let's start with the `math` package.

Example A.36 (Importing the `math` Package)**.** The code below imports the `math` package into your instance of Python and calculates the cosine of $\pi/4$.

Code:

```
import math
x = math.cos(math.pi / 4)
print(x)
```

Output:

```
0.7071067811865476
```

*If you are familiar with the statistical programming language, R, then you might be used to the idea of importing packages or libraries for certain tasks.

The answer, unsurprisingly, is the decimal form of $\sqrt{2}/2$.

You might already see a potential disadvantage: there is now more typing involved! Let's fix this. When you import a package you could just import all of the functions so they can be used by their proper names.

Example A.37. Here we import the entire math package so we can use every one of the functions therein without having to use the `math` prefix.
Code:

```python
from math import * # read this as: from math import everything
x = cos(pi / 4)
print(x)
```

Output:

```
0.7071067811865476
```

The end result is exactly the same: the decimal form of $\sqrt{2}/2$, but now we had less typing to do.

Now you can freely use the functions that were imported from the math package. There is a disadvantage to this, however. What if we have two packages that import functions with the same name. For example, in the `math` package and in the `numpy` package there is a `cos()` function. In the next block of code we'll import both `math` and `numpy`, but instead we will import them with shortened names so we can type things a bit faster.

Example A.38 (Importing Multiple Packages)**.** Here we import `math` and `numpy` under aliases so we can use the shortened aliases and not mix up which functions belong to which packages.
Code:

```python
import math as ma
import numpy as np
x = ma.cos( ma.pi / 4) # use the math version of the cosine function
y = np.cos( np.pi / 4) # use the numpy version of the cosine function
print(x, y)
```

Output:

```
0.7071067811865476 0.707106781187
```

Both "x" and "y" in the code give the decimal approximation of $\sqrt{2}/2$. This is clearly pretty redundant in this really simple case, but you should be able to see where you might want to use this and where you might run into troubles.

Once you have a package imported you can see what is inside of it using the `dir` command.

Example A.39 (Looking Inside a Package)**.** The following block of code prints a list of all of the functions inside the `math` package.
Code:

```
import math
print(dir(math))
```

Output:

```
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Of course, there will be times when you need help with a function. You can use the `help` command to view the help documentation for any function.

Example A.40 (Help)**.** Here we get help on the arc cosine function in the `math` package.
Code:

```
help(math.acos)
```

Output:

```
Help on built-in function acos in module math:
acos(...)
    acos(x)
    Return the arc cosine (measured in radians) of x.
```

Problem A.41. Import the math package, figure out how the `log` function works, and write code to calculate the logarithm of the number 8.3 in base 10, base 2, base 16, and base e (the natural logarithm). ▲

A.3 Numerical Python with numpy

The base implementation of Python includes the basic programming language, the tools to write loops, check conditions, build and manipulate lists, and all of the other things that we saw in the previous section. In this section we will explore the package numpy that contains optimized numerical routines for doing numerical computations in scientific computing.

Example A.42 (Horrible Numerical Operations)**.** To start with let's look at a really simple example. Say you have a list of real numbers and you want to take the sine every element in the list. If you just try to take the sine of the list you will get an error as you can see below.
Code:

```
from math import pi, sin
MyList = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sin(MyList)
```

Output:

```
TypeError: must be real number, not list
```

You could get around this error using some of the tools from base Python, but none of them are very elegant from a programming perspective.
Code:

```
SineList = [sin(n) for n in MyList]
print(SineList)
```

Output:

```
[0.0, 0.49999999999999994, 0.7071067811865475, 0.8660254037844386,
1.0, 0.8660254037844388, 0.7071067811865476, 0.49999999999999994,
1.2246467991473532e-16]
```

Code:

```
SineList = [ ]
for n in range(0,len(MyList)):
    SineList.append( sin(MyList[n]) )
SineList
```

Output:

```
[0.0,
 0.49999999999999994,
 0.7071067811865475,
 0.8660254037844386,
 1.0,
 0.8660254037844388,
 0.7071067811865476,
 0.49999999999999994,
 1.2246467991473532e-16]
```

The package numpy is used in almost all numerical computations using Python. It provides algorithms for matrix and vector arithmetic. Furthermore, it is optimized to be able to do these computations in the most efficient possible way (both in terms of memory and in terms of speed).

Typically when we import numpy we use `import numpy as np`. This is the standard way to name the numpy package. This means that we will have lots of function with the prefix "np" in order to call on the numpy commands. Let's first see what is inside the

package. A brief glimpse through the following list reveals a huge wealth of mathematical functions that are optimized to work in the best possible way with the Python language. (we are intentionally not showing the output here since it is quite extensive)
Code:

```
import numpy as np
# Uncomment the next line of code and run it.
# print(dir(np))
```

A.3.1 Numpy Arrays, Array Operations, and Matrix Operations

In the previous section you worked with Python lists. As we pointed out, the shortcoming of Python lists is that they don't behave well when we want to apply mathematical functions to the vector as a whole. The "numpy array", `np.array`, is essentially the same as a Python list with the notable exception that

- In a `numpy` array every entry is a floating point number
- In a `numpy` array the memory usage is more efficient (mostly since Python is expecting data of all the same type)
- With a `numpy` array there are ready-made functions that can act directly on the array as a matrix or a vector

Let's just look at a few example. What we're going to do is to define a matrix A and vectors v and w as

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad v = \begin{pmatrix} 5 \\ 6 \end{pmatrix} \quad \text{and} \quad w = v^T = \begin{pmatrix} 5 & 6 \end{pmatrix}.$$

Then we'll do the following

- Get the size and shape of these arrays
- Get individual elements, rows, and columns from these arrays
- Treat these arrays as with linear algebra to
  - do element-wise multiplication
  - do matrix a vector products
  - do scalar multiplication
  - take the transpose of matrices
  - take the inverse of matrices

Example A.43 (Defining Matrices and Vectors)**.** The first thing to note is that a matrix is a list of lists (each row is a list).
Code:

```
A = np.array([[1,2],[3,4]])
print(A)
v = np.array([[5],[6]]) # this creates a column vector
print(v)
w = np.array([5,6]) # this creates a row vector
print(w)
```

Output:

```
[[1 2]
 [3 4]]
[[5]
 [6]]
[5 6]
```

Example A.44 (Shape Matrices and Vectors)**.** The `variable.shape` command can be used to give the shape of a numpy array. Notice that the output is a tuple showing the size (rows, columns). Also notice that the row vector doesn't give (1,2) as expected. Instead it just gives (2,).
Code:

```
print(A.shape) # Shape of the matrix A
print(v.shape) # Shape of the column vector v
print(w.shape) # Shape of the row vector w
```

Output:

```
(2, 2)
(2, 1)
(2,)
```

Example A.45 (Size of Matrices and Vectors)**.** The `variable.size` command can be used to give the size of a numpy array. The size of a matrix or vector will be the total number of elements in the array. You can think of this as the product of the values in the tuple coming from the shape command.
Code:

```
print(A.size) # Size (number of elements) of A
print(v.size) # Size (number of elements) of v
print(w.size) # Size (number of elements) of w
```

Output:

```
4
```

```
2
2
```

Reading individual elements from a numpy array is the same, essentially, as reading elements from a Python list. We will use square brackets to get the row and column. Remember that the indexing all starts from 0, not 1!

Example A.46 (Reading Entries from a Matrix)**.** Let's read the top left and bottom right entries of the matrix A.
Code:

```
print(A)
print(A[0,0]) # top left
print(A[1,1]) # bottom right
```

Output:

```
[[1 2]
 [3 4]]
1
4
```

Example A.47 (Reading a Row from a Matrix)**.** Let's read the first row from that matrix A.
Code:

```
print(A)
print(A[0,:])
```

Output:

```
[[1 2]
 [3 4]]
[1 2]
```

Example A.48 (Reading a Column from a Matrix)**.** Let's read the second column from the matrix A.
Code:

```
print(A)
print(A[:,1])
```

Output:

```
[[1 2]
 [3 4]]
[2 4]
```

Notice when we read the column it was displayed as a column. Be careful.

If we try to multiply either A and v or A and A we will get some funky results. Unlike programming languages like MATLAB, the default notion of multiplication is NOT matrix multiplication. Instead, the default is element-wise multiplication.

Example A.49 (Element-wise Matrix Multiplication)**.** If we write the code `A*A` we do NOT do matrix multiplication. Instead we do element-by-element multiplication. This is a common source of issues when dealing with matrices and linear algebra in Python.
Code:

```
print(A)
A * A # Notice that this is NOT the same as A*A with matrix multiplication
```

Output:

```
[[1 2]
[3 4]]
array([[ 1,  4],
       [ 9, 16]])
```

Example A.50 (Element-wise Matrix-Vector Multiplication)**.** If we write `A * v` Python will do element-wise multiplication across each column since v is a column vector.
Code:

```
print(A)
print(v)
A * v # This will actually do element wise multiplication on each column
```

Output:

```
[[1 2]
 [3 4]]
[[5]
 [6]]
array([[ 5, 10],
       [18, 24]])
```

If, however, we recast these arrays as matrices we can get them to behave as we would expect from Linear Algebra. It is up to you to check that these products are indeed correct from the definitions of matrix multiplication from Linear Algebra.

Example A.51 (Matrix-Matrix and Matrix-Vector Multiplication)**.** Recasting the numpy arrays as matrices allows you to use multiplication as we would expect from linear algebra.
Code:

```
A = np.matrix(A)
v = np.matrix(v)
w = np.matrix(w)
print(A*A)
print(A*v)
print(w*A)
```

Output:

```
[[ 7 10]
 [15 22]]
[[17]
 [39]]
[[23 34]]
```

It remains to show some of the other basic linear algebra operations: inverses, determinants, the trace, and the transpose.

Example A.52 (Transpose of a Matrix)**.** Taking the transpose of a matrix (swapping the rows and columns) is done with the `matrix.T` command. This is just like other array commands we have seen in Python (like `.append`, `.remove`, `.shape`, etc.).
Code:

```
print(A)
A.T # The transpose is relatively simple
```

Output:

```
[[1 2]
 [3 4]]
array([[1, 3],
       [2, 4]])
```

Example A.53 (Inverse of a Matrix)**.** The inverse of a square matrix is done with `A.I`.
Code:

```
print(A)
Ainv = A.I # Taking the inverse is also pretty simple
print(Ainv)
print(A * Ainv) # check that we get the identity matrix back
```

Output:

```
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[  1.00000000e+00   0.00000000e+00]
 [  8.88178420e-16   1.00000000e+00]]
```

Example A.54 (Determinant of a Matrix)**.** The determinant command is hiding under the `linalg` subpackage inside `numpy`. Therefore we need to call it as such.
Code:

```
np.linalg.det(A) # The determinant is hiding inside the linear algebra package under n
```

Output:

```
-2.0000000000000004
```

Example A.55 (Trace of a Matrix)**.** The trace is done with `matrix.trace()`
Code:

```
A.trace() # The trace is pretty darn easy too
```

Output:

```
matrix([[5]])
```

Oddly enough, the trace returns a matrix, not a number. Therefore you'll have to read the first entry (index [0,0]) from the answer to just get the trace.

Problem A.56. Now that we can do some basic linear algebra with 'numpy' it is your turn. Define the matrix B and the vector u as below. Then find

(a) Bu

(b) B^2 (in the traditional linear algebra sense)

(c) The size and shape of B

(d) $B^T u$

(e) The element-by-element product of B with itself

(f) The dot product of u with the first row of B

$$B = \begin{pmatrix} 1 & 4 & 8 \\ 2 & 3 & -1 \\ 0 & 9 & -3 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 6 \\ 3 \\ -7 \end{pmatrix}$$

▲

A.3.2 `arange`, `linspace`, `zeros`, `ones`, and `mgrid`

There are a few built-in ways to build arrays in numpy that save a bit of time in many scientific computing settings.

- `arange` (array range) builds an array of floating point numbers with the arguments `start, stop,` and `step`. Note that you may not actually get to the `stop` point if the distance `stop-start` is not evenly divisible by the 'step'.
- `linspace` (linear space) builds an array of floating point numbers starting at one point, ending at the next point, and have exactly the number of points specified with equal spacing in between: `start, stop, number of points`. In a linear space you are always guaranteed to hit the stop point exactly, but you don't have direct control over the stop size.
- The `zeros` and `ones` commands create matrices of zeros or ones.
- `mgrid` (mesh grid) is the same as the `meshgrid` command in MATLAB. This builds two arrays that when paired make up the ordered pairs for a 2D (or higher D) mesh grid of points.

Note: Just like with all Python lists, the "stop" number is the one immediately after where you intended to stop.

Example A.57 (Arange)**.** The `numpy arange` command is great for building sequences.
Code:

```
x = np.arange(0,5,0.1)
print(x)
```

Output:

```
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3
  1.4  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4  2.5  2.6  2.7
  2.8  2.9    3. 3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4. 4.1
  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9]
```

Example A.58 (Linspace)**.** The `linspace` command builds a list with equal (linear) spacing between the starting and ending values.
Code:

```
y = np.linspace(0,5,10)
print(y)
```

Output:

```
[ 0.          0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
  3.33333333  3.88888889  4.44444444  5.        ]
```

Example A.59 (Zeros)**.** The `zeros` command builds an array of zeros. This is handy for pre-allocating memory.
Code:

```
z = np.zeros((3,5)) # create a 3x5 matrix of zeros
z
```

Output:

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Example A.60 (Ones)**.** The `ones` command builds an array of ones.
Code:

```
u = np.ones((3,5)) # create a 3x5 matrix of ones
u
```

Output:

```
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

Example A.61 (Mgrid)**.** The `mgrid` command creates a mesh grid. This is handy for building 2D (or higher dimensional) arrays of data for multi-variable functions. Notice that the output is defined as a tuple.
Code:

```
x, y = np.mgrid[0:5, 0:5]
print("x = ", x)
print("y = ", y)
```

Output:

```
x =  [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
y =  [[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
```

Problem A.62. (a) Create a `numpy` array of the numbers 1 through 10 and square every entry in the list without using a loop.

(b) Create a 10×10 identity matrix and change the top right corner to a 5. Hint: `np.identity()`

(c) Find the matrix-vector product of the answer to part (a) (as a column) and the answer to part (2).

(d) Change the bottom row of your matrix from part (b) to all 3's, change the third column to all 7's, and then find the 5^{th} power of this matrix.

▲

A.4 Mathematical Plotting with `matplotlib`

A key part of scientific computing is plotting your results or your data. The tool in Python best-suited to this task is the package `matplotlib`. As with all of the other packages in Python, it is best to learn just the basics first and then to dig deeper later. The main advantage to `matplotlib` in Python is that it is modeled off of MATLAB's plotting tools. People coming from a MATLAB background should feel pretty comfortable here, but there are some differences to be aware of.

Note: The reader should note that we will NOT be plotting symbolically defined functions in this section. The `plot` command that we will be using is reserved for numerically defined plots (i.e. plots of data points), not functions that are symbolically defined. If you have a symbolically defined function and need a plot, then pick a domain, build some x data, use the function to build the corresponding y data, and use the plotting tools discussed here. If you need a plot of a symbolic function and for some reason these steps are too much to ask, then look to the section of this Appendix on `sympy`.

A.4.1 Basics with `plt.plot()`

We are going to start right away with an example. In this example, however, we'll walk through each of the code chunks one-by-one so that we understand how to set up a proper plot. Something to keep in mind. The author strongly encourages students and readers to use Jupyter Notebooks for their Python coding. As such, there are some tricks for getting the plots to render that only apply to Jupyter Notebooks.

Example A.63 (Plotting with `plt.plot()`)**.** In the first example we want to simply plot the sine function on the domain $x \in [0, 2\pi]$, color it green, put a grid on it, and give a meaningful legend and axis labels. To do so we first need to take care of a couple of housekeeping items.

- Import `numpy` so we can take advantage of some good numerical routines.
- Import `matplotlib`'s `pyplot` module. The standard way to pull it is in is with the nickname `plt` (just like with `numpy` when we import it as `np`).

Code:

```
import numpy as np
import matplotlib.pyplot as plt
```

In Jupyter Notebooks the plots will not show up unless you tell the notebook to put them "inline". Usually we will use the following command to get the plots to show up.
Code:

```
%matplotlib inline
```

The percent sign is called a *magic* command in Jupyter Notebooks. This is not a Python command, but it is a command for controlling the Jupyter IDE specifically.

Now we'll build a numpy array of x values (using the `linspace` command) and a numpy array of y values for the sine function.
Code:

```
x = np.linspace(0,2*np.pi, 100) # 100 equally spaced points from 0 to 2pi
y = np.sin(x)
```

- Finally, build the plot with `plt.plot()`. This function behaves just like MATLAB's `plot` command:
  `plt.plot(x, y, 'color', ...)`
  where you have several options that you can pass (more on that later).
- Notice that we send the plot legend in directly to the plot command (unlike MATLAB).
- Then we'll add the grid with `plt.grid()`
- Then we'll add the legend to the plot
- Finally we'll add the axis labels
- We end the plotting code with `plt.show()` to tell Python to finally show the plot. You can get by without this line of code, but it is handy to have in there when you're doing complicated plots.

Code:

```
plt.plot(x,y, 'green', label='The Sine Function')
plt.grid()
plt.legend()
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.show()
```

The plot that is produced can be seen in Figure A.1.

Example A.64 (Overlayed Plots with MatPlotLib)**.** Now let's do a second example, but this time we want to show four different plots on top of each other. When you start a figure, `matplotlib` is expecting all of those plots to be layered on top of each

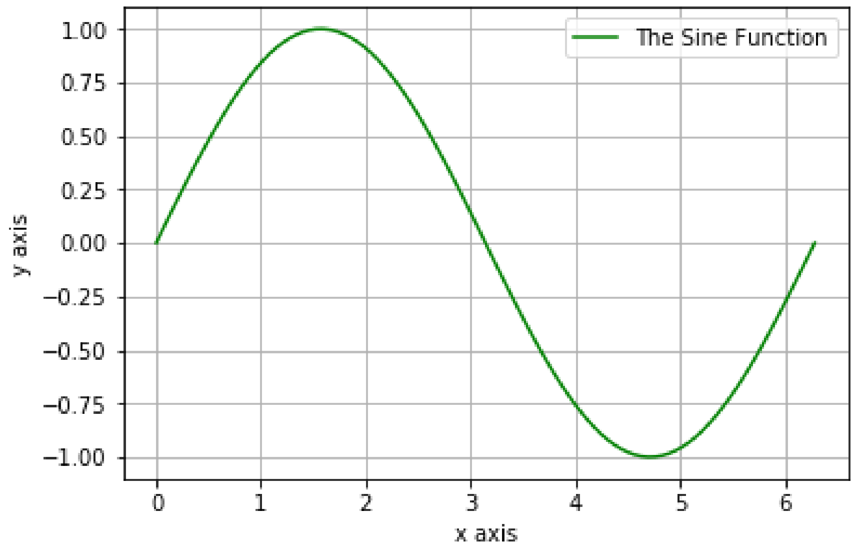

Figure A.1. The sine function on $x \in [0, 2\pi]$, colored greed, with a grid, and with a legend.

other. For MATLAB users, this means that you do not need the `hold on` command since it is automatically "on".

In this example we will plot

$$y_0 = \sin(2\pi x) \quad y_1 = \cos(2\pi x) \quad y_2 = y_0 + y_1 \quad \text{and} \quad y_3 = y_0 - y_1$$

on the domain $x \in [0, 1]$ with 100 equally spaced points. We'll give each of the plots a different line style, built a legend, put a grid on the plot, and give axis labels.

Code:

```
# We really don't need to import the packages again,
# but for completeness of the example let's do it anyway.
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# build the x and y values
x = np.linspace(0,1,100)
y0 = np.sin(2*np.pi*x)
y1 = np.cos(2*np.pi*x)
y2 = y0 + y1
y3 = y0 - y1

# plot each of the functions (notice that they will be on the same axes)
plt.plot(x, y0, 'b-.', label=r"$y_0 = \sin(2\pi x)$")
plt.plot(x, y1, 'r--', label=r"$y_1 = \cos(2\pi x)$")
plt.plot(x, y2, 'g:', label=r"$y_2 = y_0 + y_1$")
plt.plot(x, y3, 'k-', label=r"$y_3 = y_0 - y_1$")

# put in a grid, legend, title, and axis labels
```

```
plt.grid()
plt.legend()
plt.title("Awesome Title")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()
```

The resulting plot can be seen in Figure A.2

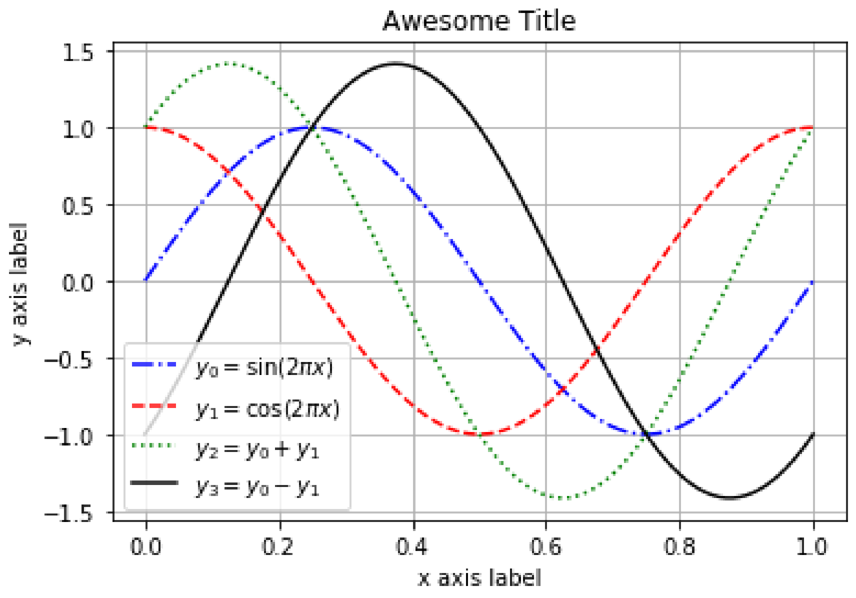

Figure A.2. Several trigonometric functions on the same axes.

Notice in Figure A.2 that the legend was placed automatically. There are ways to control the placement of the legend if you wish, but for now just let Python and `matplotlib` have control over the placement.

Example A.65 (Same Plot, Different Code)**.** Now let's create the same plot as in Figure A.2 with slightly different code. The `plot` command can take several (x,y) pairs in the same line of code. This can really shrink the amount of coding that you have to do when plotting several functions on top of each other.
Code:

```
# The next line of code does all of the plotting of all of the functions.
# Notice the order: x, y, color and line style, repeat
plt.plot(x, y0, 'b-.', x, y1, 'r--', x, y2, 'g:', x, y3, 'k-')

plt.grid()
plt.legend([r"$y_0 = \sin(2\pi x)$",r"$y_1 = \cos(2\pi x)$",\
            r"$y_2 = y_0 + y_1$",r"$y_3 = y_0 - y_1$"])
plt.title("Awesome Title")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()
```

Problem A.66. Plot the functions $f(x) = x^2$, $g(x) = x^3$, and $h(x) = x^4$ on the same axes. Use the domain $x \in [0,1]$ and the range $y \in [0,1]$. Put a grid, a legend, a title, and appropriate labels on the axes. ▲

A.4.2 Subplots

It is often very handy to place plots side-by-side or as some array of plots. The `subplots` command allows us that control. The main idea is that we are setting up a matrix of blank plot and then populating the axes with the plots that we want.

Example A.67 (First Subplot Example)**.** Let's repeat the previous exercise (which produced Figure A.2), but this time we will put each of the plots in its own subplot. There are a few extra coding quirks that come along with building subplots so we'll highlight each block of code separately.

- First we set up the plot area with `plt.subplots()`. The first two inputs to the `subplots` command are the number of rows and the number of columns in your plot array. For the first example we will do 2 rows of plots with 2 columns – so there are four plots total. The last input for the `subplots` command is the size of the figure (this is really just so that it shows up well in Jupyter Notebooks – spend some time playing with the figure size to get it to look right).
- Then we build each plot individually telling `matplotlib` which axes to use for each of the things in the plots.
- Notice the small differences in how we set the titles and labels
- In this example we are setting the y-axis to the interval $[-2,2]$ for consistency across all of the plots.

Code:

```
# set up the blank matrix of plots
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (10,5))

# Build the first plot
axes[0,0].plot(x, y0, 'b-.')
axes[0,0].grid()
axes[0,0].set_title(r"$y_0 = \sin(2\pi x)$")
axes[0,0].set_ylim(-2,2)
axes[0,0].set_xlabel("x")
axes[0,0].set_ylabel("y")

# Build the second plot
```

```
axes[0,1].plot(x, y1, 'r--')
axes[0,1].grid()
axes[0,1].set_title(r"$y_1 = \cos(2\pi x)$")
axes[0,1].set_ylim(-2,2)
axes[0,1].set_xlabel("x")
axes[0,1].set_ylabel("y")

# Build the first plot
axes[1,0].plot(x, y2, 'g:')
axes[1,0].grid()
axes[1,0].set_title(r"$y_2 = y_0 + y_1$")
axes[1,0].set_ylim(-2,2)
axes[1,0].set_xlabel("x")
axes[1,0].set_ylabel("y")

# Build the first plot
axes[1,1].plot(x, y3, 'k-')
axes[1,1].grid()
axes[1,1].set_title(r"$y_3 = y_0 - y_1$")
axes[1,1].set_ylim(-2,2)
axes[1,1].set_xlabel("x")
axes[1,1].set_ylabel("y")

fig.tight_layout()
```

The resulting plot can be seen in Figure A.3. The `fig.tight_layout()` command makes the plot labels a bit more readable in this instance (again, something you can play with).

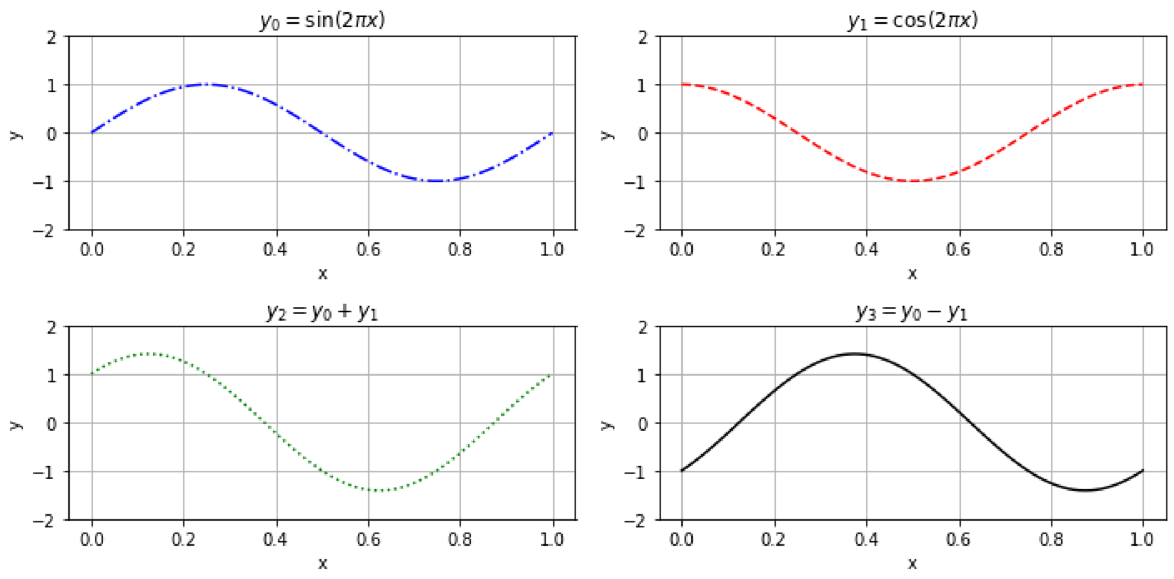

Figure A.3. The trigonometric functions with a 2×2 matrix of subplots.

Problem A.68. Put the functions $f(x) = x^2$, $g(x) = x^3$ and $h(x) = x^4$ in a subplot environment with 1 row and 3 columns of plots. Use the unit interval as the domain and range for all three plot, but sure that each plot has a grid, appropriate labels, an appropriate title, and the overall figure has a title. ▲

A.4.3 Logarithmic Scaling with `semilogy`, `semilogx`, and `loglog`

It is occasionally useful to scale an axis logarithmically. This arises most often when we're examining an exponential function, or some other function, that is close to zero for much of the domain. Scaling logarithmically allows us to see how small the function is getting in orders of magnitude instead of as a raw real number.

Example A.69. In this example we'll plot the function $y = 10^{-0.01x}$ on a regular (linear) scale and on a logarithmic scale on the y axis. Use the interval $[0, 500]$ as a domain.

Code:

```python
x = np.linspace(0,500,1000)
y = 10**(-0.01*x)
fig, axis = plt.subplots(1,2, figsize = (10,5))

axis[0].plot(x,y, 'r')
axis[0].grid()
axis[0].set_title("Linearly scaled y axis")
axis[0].set_xlabel("x")
axis[0].set_ylabel("y")

axis[1].semilogy(x,y, 'k--')
axis[1].grid()
axis[1].set_title("Logarithmically scaled y axis")
axis[1].set_xlabel("x")
axis[1].set_ylabel("Log(y)")
```

The output for this code can be seen in Figure A.4.

It should be noted that the same result can be achieved using the `yscale` command along with the `plot` command instead of using the `semilogy` command. Pay careful attention to the subtle changes in the following code.

Code:

```python
x = np.linspace(0,500,1000)
y = 10**(-0.01*x)
fig, axis = plt.subplots(1,2, figsize = (10,5))

axis[0].plot(x,y, 'r')
axis[0].grid()
axis[0].set_title("Linearly scaled y axis")
axis[0].set_xlabel("x")
axis[0].set_ylabel("y")

axis[1].plot(x,y, 'k--') # <----- Notice the change here
axis[1].set_yscale("log") # <----- And we added this line
axis[1].grid()
```

```
axis[1].set_title("Logarithmically scaled y axis")
axis[1].set_xlabel("x")
axis[1].set_ylabel("Log(y)")
```

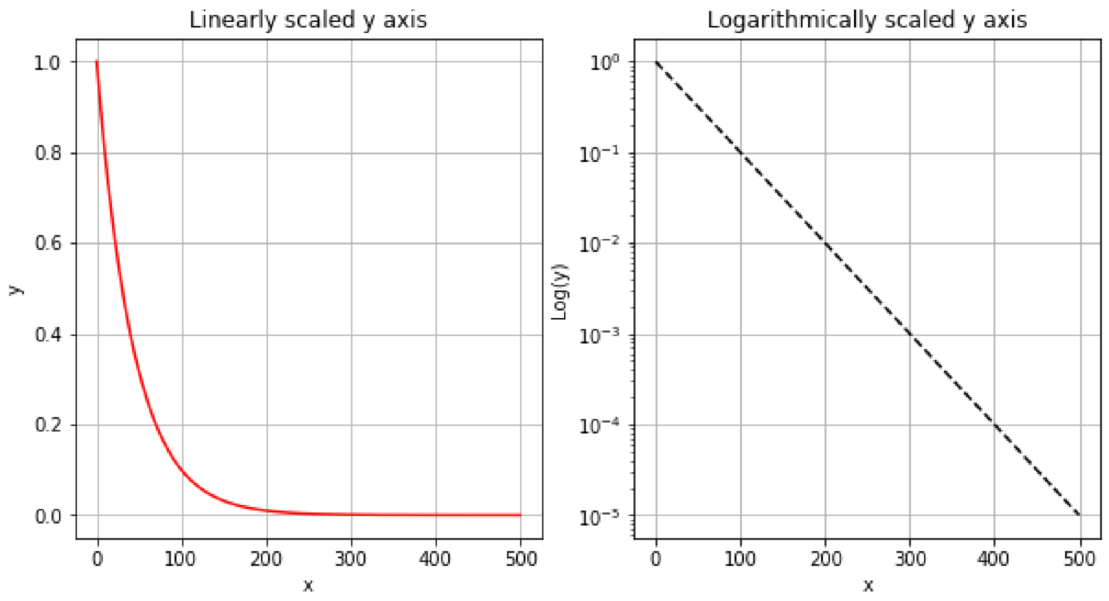

Figure A.4. The left-hand plot shows the exponential function $y = 10^{-0.01x}$ and the right-hand side shows the same plot with the y axis scaled logarithmically.

Problem A.70. Plot the function $f(x) = x^3$ for $x \in [0,1]$ on linearly scaled axes, logarithmic axis in the y direction, logarithmically scaled axes in the x direction, and a log-log plot with logarithmic scaling on both axes. Use `subplots` to put your plots side-by-side. Give appropriate labels, titles, etc. ▲

A.5 Symbolic Algebra and Calculus with `sympy`

In this section we will learn the tools necessary to do symbolic mathematics in Python. The relevant package is `sympy` (symbolic python) and it works much like Mathematica, Maple, or MATLAB's symbolic toolbox. That being said, Mathematica and Maple are designed to do symbolic computation in the fastest and best possible ways, so in some sense, `sympy` is a little step-sibling to these much bigger pieces of software. Remember: Python is free, and this is a book on numerical analysis ...

For the sake of convenience we will load `matplotlib` so we can plot a few things along the way.

Code:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

Let's import sympy in the usual way. We will use the nickname sp (just like we used np for numpy). This is not a standard nickname in the Python literature, but it will suffice for our purposes.
Code:

```
import sympy as sp
# The directory of sympy is full of stuff.
# Uncomment the next line and run the code to see what you're getting.
# print(dir(sp))

sp.init_printing() # This allows for a more LaTeX style output
```

The last line of the previous block of code is again just for Jupyter Notebooks. We want to allow for some beautiful styling of the output of these symbolic calculations and this is the line that allows Jupyter to do it.

A.5.1 Symbolic Variables with `symbols`

When you are working with symbolic variables you have to tell Python that that's what you're doing. In other words, we actually have to type-cast the variables when we name them. Otherwise Python won't know what to do with them.

Let's define the variable x as a symbolic variable. Then we'll define a few symbolic expressions that use x as a variable.
Code:

```
x = sp.Symbol('x') # note the capitalization
```

Now we'll define the function $f(x) = (x+2)^3$ and spend the next few examples playing with it.
Code:

```
f = (x+2)**3 # A symbolic function
f
```

The output of these lines of code is the nicely formatted symbolic expression $(x+2)^3$.

Be careful that you are using symbolically defined function along with your symbols. For example, see the code below:
Code:

```
# g = sin(x) # this line gives an error since it doesn't know which "sine" function to
g = sp.sin(x) # this one works
g
```

The output of these lines of code is the symbolic expression $\sin(x)$.

A.5.2 Symbolic Algebra

One of the primary purposes of doing symbolic programming is to do symbolic algebra (the other is typically symbolic calculus). In this section we'll look at a few of the common algebraic exercises that can be handled with sympy.

Example A.71 (Expand a Polynomial)**.** Expand the function $f(x) = (x+2)^3$. In other words, multiply this out fully so that it is a sum or difference of monomials instead of the cube of a binomial.
Code:

```
f = (x+2)**3
sp.expand(f) # do the multiplication to expand the polynomial
```

The output of these lines of code is the expression $x^3 + 6x^2 + 12x + 8$.

Example A.72 (Factor a Polynomial)**.** We will factor the polynomial $h(x) = x^2 + 4x + 3$.
Code:

```
h = x**2 + 4*x + 3
sp.factor(h) # factor this polynomial
```

The output of this function is the expression $(x+1)(x+3)$.

Example A.73 (Trig Expansion)**.** The sympy package knows how to work with trigonometric identities. In this example we show how sympy expands $\sin(a+b)$.
Code:

```
a, b = sp.symbols('a b')
j = sp.sin(a+b)
sp.expand(j, trig=True) # Trig identities are built in!
```

The output of these linse of code is the expression $\sin(a)\cos(b) + \sin(b)\cos(a)$

Example A.74 (Simplifying Algebraic Expressions)**.** In this example we will simplify the function $g(x) = x^3 + 5x^3 + 12x^2 + 1$.
Code:

```
g = x**3 + 5*x**3 + 12*x**2 + 1
sp.simplify(g) # Simplify some algebraic expression
```

The output of these lines of code is the expression $6x^3 + 12x^2 + 1$.

Example A.75 (Simplifying Trig Expressions)**.** In this example we'll simplify an expression that involves trigonometry.
Code:

```
sp.simplify( sp.sin(x) / sp.cos(x)) # simplify a trig expression.
```

The output of this line of code is the expression $\tan(x)$ as expected.

The primary goal of many algebra problems is to solve an equation. We will dedicate more time to algebraic equation solving later in this section, but the following example gives a simple example of how it works in `sympy`.

Example A.76 (Simple Equation Solving)**.** We want to solve the equation $x^2+4x+3 = 0$ for x.
Code:

```
h = x**2 + 4*x + 3
sp.solve(h,x)
```

Output:

```
[-3, -1]
```

As expected, the roots of the function $h(x)$ are $x = -3$ and $x = 1$ since $h(x)$ factors into $h(x) = (x+3)(x-1)$.

A.5.3 Symbolic Function Evaluation

In `sympy` we cannot simply just evaluate functions as we would on paper. Let's say we have the function $f(x) = (x+2)^3$ and we want to find $f(5)$. We would say that we "substitute 5 into f for x", and that is exactly what we have to tell Python. Unfortunately we cannot just write "f(5)" since that would mean that "f" is a Python function and we are sending the number 5 into that function. This is an unfortunate double-use of the word "function", but stop and think about it for a second: When we write `f = (x+2)**3` we are just telling Python that `f` is a symolic expression in terms of the symbol `x`, but we did not use `def` to define it as a function as we did for all other function.

The following code is what the mathematicians in us would like to do: **Code:**

```
f = (x+2)**3
f(5) # This gives an error!
```

. . . but this is how it should be done: **Code:**

```
f.subs(x,5) # This actually substitutes 5 for x in f
```

Output:

```
343
```

A.5.4 Symbolic Calculus

The `sympy` package has routines to take symbolic derivatives, antiderivatives, limits, and Taylor series just like other computer algebra systems.

Derivatives

The `diff` command is the command for differentiation. Take careful note that `diff` is defined both in `sympy` and in `numpy`. That means that there are symbolic and numerical routines for taking derivatives in Python …and we need to tell our instance of Python which one we're working with every time we use it.

Example A.77 (Symbolic Differentiation)**.** In this example we'll differentiate the function $f(x) = (x+2)^3$.
Code:

```
import sympy as sp # we already did this above, but let's do it again for clarity
x = sp.Symbol('x') # Define the symbol x
f = (x+2)**3 # Define a symbolic function f(x) = (x+2)^3
df = sp.diff(f,x) # Take the derivative of f and call it "df"
print("f(x) = ", f)
print("f'(x) = ",df)
print("f'(x) = ", sp.expand(df))
```

Output:

```
f(x) =  (x + 2)**3
f'(x) =  3*(x + 2)**2
f'(x) =  3*x**2 + 12*x + 12
```

Now let's get the first, second, third, and fourth derivatives of the function `f`.
Code:

```
df = sp.diff(f,x,1) # first derivative
ddf = sp.diff(f,x,2) # second deriative
dddf = sp.diff(f,x,3) # third deriative
ddddf = sp.diff(f,x,4) # fourth deriative
print("f'(x) = ",df)
print("f''(x) = ",sp.simplify(ddf))
print("f'''(x) = ",sp.simplify(dddf))
print("f''''(x) = ",sp.simplify(ddddf))
```

Output:

```
f'(x) =  3*(x + 2)**2
f''(x) =  6*x + 12
f'''(x) =  6
f''''(x) =  0
```

Example A.78 (Partial Derivatives)**.** Now let's do some partial derivatives. The `diff` command is still the right tool. You just have to tell it which variable you're working with.
Code:

```
x, y = sp.symbols('x y') # Define the symbols
f = sp.sin(x*y) + sp.cos(x**2) # Define the function
```

```
fx = sp.diff(f,x)
fy = sp.diff(f,y)
print("f(x,y) = ", f)
print("f_x(x,y) = ", fx)
print("f_y(x,y) = ", fy)
```

Output:

```
f(x,y) =  sin(x*y) + cos(x**2)
f_x(x,y) =  -2*x*sin(x**2) + y*cos(x*y)
f_y(x,y) =  x*cos(x*y)
```

Example A.79 (LaTeX from Symbolic Expressions)**.** It is worth noting that when you have a symbolically defined function you can ask sympy to give you the LaTeX code for the symbolic function so you can use it when you write about it.
Code:

```
sp.latex(f)
```

Output:

```
'\\sin{\\left(x y \\right)} + \\cos{\\left(x^{2} \\right)}'
```

Integrals

For integration, the `integrate` tool is the command for the job.

Limits

Taylor Series

A.5.5 Solving Equations Symbolically

A.5.6 Symbolic Plotting

Bibliography

[1] A. Greenbaum and T. Charier. *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms* Princeton University Press. 2012.

[2] R. Burden, D, Faires, and A. Burden. *Numerical Analysis, 10ed* Cengage Learning. 2016.

[3] D. Kindaid and W. Cheney. *Numerical Analysis, 2ed.* Brooks/Cole Publishing, 1996.

[4] R. Haberman. *Applied Partial Differential Equations, 4ed*. Pearson Education Inc. Upper Saddle River, New Jersey, 2004

[5] D. Lay. *Linear Algebra 4ed.* Pearson Education Inc. Upper Saddle River, New Jersey, 2012.

[6] M. Meerschaert. *Mathematical Modeling 4ed.* Academic Press Publications, 2013.

[7] Holistic Numerical Methods http://nm.mathforcollege.com/
The Holistic Numerical Methods book is probably the most complete free reference that I've found on the web. This should be your source to look up deeper explanations of problems, algorithms, and code.

[8] Scientific Computing with MATLAB http://gribblelab.org/scicomp/scicomp.pdf

[9] Tea Time Numerical Analysis http://lqbrin.github.io/tea-time-numerical/