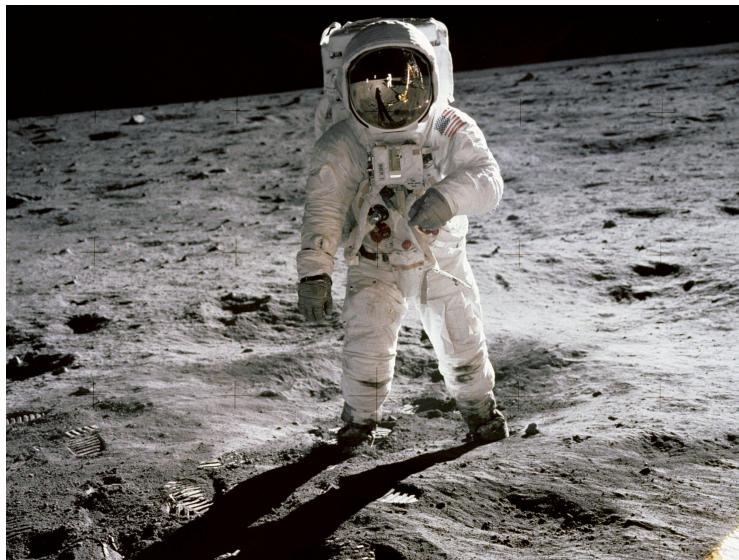


# Labs for Foundations of Applied Mathematics

Volume 2  
Algorithm Design and Optimization

Jeffrey Humpherys & Tyler J. Jarvis, managing editors





# List of Contributors

B. Barker <i>Brigham Young University</i>	T. Christensen <i>Brigham Young University</i>
E. Evans <i>Brigham Young University</i>	M. Cook <i>Brigham Young University</i>
R. Evans <i>Brigham Young University</i>	M. Cutler <i>Brigham Young University</i>
J. Grout <i>Drake University</i>	R. Dorff <i>Brigham Young University</i>
J. Humpherys <i>Brigham Young University</i>	B. Ehler <i>Brigham Young University</i>
T. Jarvis <i>Brigham Young University</i>	M. Fabiano <i>Brigham Young University</i>
J. Whitehead <i>Brigham Young University</i>	K. Finlinson <i>Brigham Young University</i>
J. Adams <i>Brigham Young University</i>	J. Fisher <i>Brigham Young University</i>
K. Baldwin <i>Brigham Young University</i>	R. Flores <i>Brigham Young University</i>
J. Bejarano <i>Brigham Young University</i>	R. Fowers <i>Brigham Young University</i>
J. Bennett <i>Brigham Young University</i>	A. Frandsen <i>Brigham Young University</i>
A. Berry <i>Brigham Young University</i>	R. Fuhriman <i>Brigham Young University</i>
Z. Boyd <i>Brigham Young University</i>	T. Gledhill <i>Brigham Young University</i>
M. Brown <i>Brigham Young University</i>	S. Giddens <i>Brigham Young University</i>
A. Carr <i>Brigham Young University</i>	C. Gigena <i>Brigham Young University</i>
C. Carter <i>Brigham Young University</i>	M. Graham <i>Brigham Young University</i>
S. Carter <i>Brigham Young University</i>	F. Glines <i>Brigham Young University</i>

C. Glover <i>Brigham Young University</i>	E. Manner <i>Brigham Young University</i>
M. Goodwin <i>Brigham Young University</i>	M. Matsushita <i>Brigham Young University</i>
R. Grout <i>Brigham Young University</i>	R. McMurray <i>Brigham Young University</i>
D. Grundvig <i>Brigham Young University</i>	S. McQuarrie <i>Brigham Young University</i>
S. Halverson <i>Brigham Young University</i>	E. Mercer <i>Brigham Young University</i>
E. Hannesson <i>Brigham Young University</i>	D. Miller <i>Brigham Young University</i>
K. Harmer <i>Brigham Young University</i>	J. Morrise <i>Brigham Young University</i>
J. Henderson <i>Brigham Young University</i>	M. Morrise <i>Brigham Young University</i>
J. Hendricks <i>Brigham Young University</i>	A. Morrow <i>Brigham Young University</i>
A. Henriksen <i>Brigham Young University</i>	R. Murray <i>Brigham Young University</i>
I. Henriksen <i>Brigham Young University</i>	J. Nelson <i>Brigham Young University</i>
B. Hepner <i>Brigham Young University</i>	C. Noorda <i>Brigham Young University</i>
C. Hettinger <i>Brigham Young University</i>	A. Oldroyd <i>Brigham Young University</i>
S. Horst <i>Brigham Young University</i>	A. Oveson <i>Brigham Young University</i>
R. Howell <i>Brigham Young University</i>	E. Parkinson <i>Brigham Young University</i>
E. Ibarra-Campos <i>Brigham Young University</i>	M. Probst <i>Brigham Young University</i>
K. Jacobson <i>Brigham Young University</i>	M. Proudfoot <i>Brigham Young University</i>
R. Jenkins <i>Brigham Young University</i>	D. Reber <i>Brigham Young University</i>
J. Larsen <i>Brigham Young University</i>	H. Ringer <i>Brigham Young University</i>
J. Leete <i>Brigham Young University</i>	C. Robertson <i>Brigham Young University</i>
Q. Leishman <i>Brigham Young University</i>	M. Russell <i>Brigham Young University</i>
J. Lytle <i>Brigham Young University</i>	R. Sandberg <i>Brigham Young University</i>

- C. Sawyer  
*Brigham Young University*
- N. Sill  
*Brigham Young University*
- D. Smith  
*Brigham Young University*
- J. Smith  
*Brigham Young University*
- P. Smith  
*Brigham Young University*
- M. Stauffer  
*Brigham Young University*
- E. Steadman  
*Brigham Young University*
- J. Stewart  
*Brigham Young University*
- S. Suggs  
*Brigham Young University*
- A. Tate  
*Brigham Young University*
- T. Thompson  
*Brigham Young University*
- B. Trendler  
*Brigham Young University*
- M. Victors  
*Brigham Young University*
- E. Walker  
*Brigham Young University*
- J. Webb  
*Brigham Young University*
- R. Webb  
*Brigham Young University*
- J. West  
*Brigham Young University*
- R. Wonnacott  
*Brigham Young University*
- A. Zaitzeff  
*Brigham Young University*



# Preface

This lab manual is designed to accompany the textbooks *Foundations of Applied Mathematics Volume 2: Algorithms, Approximation, and Optimization* by Humpherys and Jarvis. The labs focus mainly on data structures, signal transforms, and numerical optimization, including applications to data science, signal processing, and machine learning. The reader should be familiar with Python [VD10] and its NumPy [Oli06, ADH<sup>+</sup>01, Oli07] and Matplotlib [Hun07] packages before attempting these labs. See the Python Essentials manual for introductions to these topics.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

<https://github.com/Foundations-of-Applied-Mathematics/Labs>  
as the original source of this work.

To view a copy of the Creative Commons Attribution 3.0 License, visit

<http://creativecommons.org/licenses/by/3.0/us/>  
or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.





# Contents

Preface	v
<b>I    Labs</b>	<b>1</b>
1    Introduction to Python	3
2    Introduction to NumPy	25
3    Introduction to Matplotlib	43
4    Unit Testing	59
5    Linked Lists	73
6    Binary Search Trees	83
7    Nearest Neighbor Search	99
8    Breadth-first Search	113
9    Markov Chains	125
10   Unix Shell 2	137
11   Sampling	151
12   The Discrete Fourier Transform	157
13   Introduction to Wavelets	167
14   Polynomial Interpolation	185
15   Gaussian Quadrature	197
16   One-dimensional Optimization	203
17   Regular Expressions	211

<b>18</b>	<b>Gradient Descent Methods</b>	<b>225</b>
<b>19</b>	<b>The Simplex Method</b>	<b>235</b>
<b>20</b>	<b>Gymnasium</b>	<b>245</b>
<b>21</b>	<b>CVXPY</b>	<b>253</b>
<b>22</b>	<b>Non-negative Matrix Factorization</b>	<b>261</b>
<b>23</b>	<b>Interior Point 1: Linear Programs</b>	<b>271</b>
<b>24</b>	<b>Dynamic Programming</b>	<b>281</b>
<b>25</b>	<b>Policy Function Iteration</b>	<b>291</b>
<b>II</b>	<b>Appendices</b>	<b>301</b>
<b>A</b>	<b>NumPy Visual Guide</b>	<b>303</b>
<b>B</b>	<b>Matplotlib Customization</b>	<b>307</b>
	<b>Bibliography</b>	<b>323</b>

# Part I

# Labs



# 1

# Introduction to Python

**Lab Objective:** *Python is a powerful general-purpose programming language. It can be used interactively, allowing for very rapid development. Python has many powerful scientific computing tools, making it an ideal language for applied and computational mathematics. In this introductory lab we introduce Python syntax, data types, functions, and control flow tools. These Python basics are an essential part of almost every problem you will solve and almost every program you will write.*

## Getting Started

Python is quickly gaining momentum as a fundamental tool in scientific computing. To install Python, see the Getting Started document.

## Running Python

Python files are saved with a .py extension. For beginners, we strongly recommend using a simple text editor for writing Python files, though many free IDEs (Integrated Development Environments—large applications that facilitate code development with some sophisticated tools) are also compatible with Python. For now, the simpler the coding environment, the better.

A plain Python file looks similar to the following code.

```
# filename.py
"""This is the file header.
The header contains basic information about the file.
"""

if __name__ == "__main__":
    pass # 'pass' is a temporary placeholder.
```

The `#` character creates a single-line *comment*. Comments are ignored by the interpreter and serve as annotations for the accompanying source code. A pair of three quotes, `""" """` or `''' '''`, creates a multi-line string literal, which may also be used as a multi-line comment. A triple-quoted string literal at the top of the file serves as the *header* for the file. The header typically identifies the author and includes instructions on using the file. Executable Python code comes after the header.

**Problem 1.** Open the file named `python_intro.py` (or create the file in a text editor if you don't have it). Add your information to the header at the top, then add the following code.

```
if __name__ == "__main__":
    print("Hello, world!") # Indent with four spaces (NOT a tab).
```

Be sure to save your edited file. Open a command prompt (*Terminal* on Linux or Mac and *Command Prompt* or *GitBash* on Windows) and navigate to the directory where the new file is saved. Use the command `ls` (or `DIR` on Windows) to list the files and folders in the current directory, `pwd` (CD , on Windows) to print the working directory, and `cd` to change directories.

```
$ pwd # Print the working directory.
/Users/Guest
$ ls # List the files and folders here.
Desktop Documents Downloads Pictures Music
$ cd Documents # Navigate to a different folder.
$ pwd
/Users/Guest/Documents
$ ls # Check to see that the file is here.
python_intro.py
```

Now the Python file can be executed with the following command:

```
$ python python_intro.py
```

If `Hello, world!` is displayed on the screen, you have just successfully executed your first Python program!

### ACHTUNG!

The `if __name__ == "__main__"` clause is incredibly helpful to create test functions and debug your code. In order to use `if __name__ == "__main__"` to test declared functions, **it must be placed at the end of your file** with the code you desire to run directly following. If you attempt to run that block of code at the beginning of the file with functions that are declared afterwards, you'll get an error of having undefined functions.

## IPython

Python can be run interactively using several interfaces. The most basic of these is the Python interpreter. In this and subsequent labs, the triple brackets `>>>` indicate that the given code is being executed one line at a time via the Python interpreter.

```
$ python # Start the Python interpreter.
>>> print("This is plain Python.") # Execute some code.
This is plain Python.
```

There are, however, more useful interfaces. Chief among these is *IPython*,<sup>1</sup> [PG07, jup]. To execute a script in IPython, use the `%run` command.

```
>>> exit()                                # Exit the Python interpreter.
$ ipython                                 # Start IPython.

In [1]: print("This is IPython!")      # Execute some code.
This is IPython!

In [2]: %run python_intro.py            # Run a particular Python script.
Hello, world!
```

Python is an **object-oriented** programming language. As a reminder:

- An **object** refers to a particular instance of a class.
- **Attributes** are the characteristics or properties of an object that store data.
- **Methods** define the actions or behaviors that an object can perform.

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection quickly reveals all methods and attributes associated with an object. IPython also has a built-in `help()` function that provides interactive help.

```
# A list is a basic Python data structure. To see the methods associated with
# a list, type the object name (list), followed by a period, and press tab.
In [1]: list.  # Press 'tab'.
        append()  count()  insert()  remove()
        clear()   extend()  mro()    reverse()
        copy()    index()  pop()    sort()

# To learn more about a specific method, use a '?' and hit 'Enter'.
In [1]: list.append?
Signature: list.append(self, object, /)
Docstring: Append object to the end of the list.
Type:      method_descriptor

In [2]: help()                           # Start IPython's interactive help utility.

help> list                            # Get documentation on the list class.
Help on class list in module builtins:

class list(object)
|  list(iterable=(),/)
|  # ...                                # Press 'q' to exit the info screen.

help> quit                            # End the interactive help session.
```

---

<sup>1</sup>See <https://ipython.org/> and <https://jupyter.org/>.

## NOTE

Use IPython side-by-side with a text editor to test syntax and small code snippets quickly. Testing small pieces of code in IPython **before** putting them into a program reveals errors and greatly speeds up the coding process. Consult the internet with questions; [stack overflow .com](http://stackoverflow.com) is a particularly valuable resource for answering common programming questions.

The best way to learn a new coding language is by actually writing code. Follow along with the examples in the yellow code boxes in this lab by executing them in an IPython console. Avoid copy and paste for now; your fingers need to learn the language as well.

# Python Basics

## Arithmetic

Python can be used as a calculator with the regular +, -, \*, and / operators. Use \*\* for exponentiation and % for modular division.

```
>>> 3**2 + 2*5          # Python obeys the order of operations.
19

>>> 13 % 3              # The modulo operator % calculates the
1                      # remainder: 13 = (3*4) + 1.
```

In most Python interpreters, the underscore character \_ is a variable with the value of the previous command's output, like the ANS button on many calculators.

```
>>> 12 * 3
36
>>> _ / 4
9.0
```

Data comparisons like < and > act as expected. The == operator checks for numerical equality and the <= and >= operators correspond to  $\leq$  and  $\geq$ , respectively. To connect multiple boolean expressions, use the operators `and`, `or`, and `not`.<sup>2</sup>

```
>>> 3 > 2.99
True
>>> 1.0 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True

>>> True and True and True and True and True and False
False
>>> False or False or False or False or False or True
```

---

<sup>2</sup>In many other programming languages, the `and`, `or`, and `not` operators are written as `&&`, `||`, and `!`, respectively. Python's convention is much more readable and does not require parentheses.

```
True
>>> True or not True
True
```

## Variables

Variables are used to temporarily store data. A **single** equals sign `=` assigns one or more values (on the right) to one or more variable names (on the left). A **double** equals sign `==` is a comparison operator that returns `True` or `False`, as in the previous code block.

Unlike many programming languages, Python does not require a variable's data type to be specified upon initialization. Because of this, Python is called a *dynamically typed* language.

```
>>> x = 12                      # Initialize x with the integer 12.
>>> y = 2 * 6                    # Initialize y with the integer 2*6 = 12.
>>> x == y                      # Compare the two variable values.
True

>>> x, y = 2, 4                 # Give both x and y new values in one line.
>>> x == y
False
```

## Functions

To define a function, use the `def` keyword followed by the function name, a parenthesized list of parameters, and a colon. Then indent the function body using exactly **four** spaces.

```
>>> def add(x, y):
...     return x + y               # Indent with four spaces.
```

### ACHTUNG!

Many other languages use the curly braces `{}` to delimit blocks, but Python uses whitespace indentation. In fact, whitespace is essentially the only thing that Python is particularly picky about compared to other languages: **mixing tabs and spaces confuses the interpreter and causes problems**. Most text editors have a setting to set the indentation type to spaces so you can use the tab key on your keyboard to insert four spaces (sometimes called *soft tabs*). For consistency, **never** use tabs; **always** use spaces.

Functions are defined with *parameters* and called with *arguments*, though the terms are often used interchangeably. Below, `width` and `height` are parameters for the function `area()`. The values 2 and 5 are the arguments that are passed when calling the function.

```
>>> def area(width, height):      # Define the function.
...     return width * height
...
```

```
>>> area(2, 5)                      # Call the function.
10
```

Python functions can also return multiple values.

```
>>> def arithmetic(a, b):
...     return a - b, a * b           # Separate return values with commas.
...
>>> x, y = arithmetic(5, 2)        # Unpack the returns into two variables.
>>> print(x, y)
3 10
```

The keyword `lambda` is a shortcut for creating one-line functions. For example, the polynomials  $f(x) = 6x^3 + 4x^2 - x + 3$  and  $g(x, y, z) = x + y^2 - z^3$  can be defined as functions in one line each.

```
# Define the polynomials the usual way using 'def'.
>>> def f(x):
...     return 6*x***3 + 4*x**2 - x + 3
>>> def g(x, y, z):
...     return x + y**2 - z***3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> f = lambda x: 6*x***3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z***3
```

### NOTE

Documentation is important in every programming language. Every function should have a *docstring*—a string literal in triple quotes just under the function declaration—that describes the purpose of the function, the expected inputs and return values, and any other notes that are important to the user. Short docstrings are acceptable for very simple functions, but more complicated functions require careful and detailed explanations.

```
>>> def add(x, y):
...     """Return the sum of the two inputs."""
...     return x + y

>>> def area(width, height):
...     """Return the area of the rectangle with the specified width
...     and height.
...
...     """
...     return width * height
...
>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Lambda functions cannot have custom docstrings, so the `lambda` keyword should be only be used as a shortcut for very simple or intuitive functions that need no additional labeling.

**Problem 2.** The volume of a sphere with radius  $r$  is  $V = \frac{4}{3}\pi r^3$ . In your Python file from Problem 1, define a function called `sphere_volume()` that accepts a single parameter  $r$ . Return the volume of the sphere of radius  $r$ , using 3.14159 as an approximation for  $\pi$  (for now). Also write an appropriate docstring for your function.

To test your function, call it under the `if __name__ == "__main__"` clause and print the returned value. Run your file to see if your answer is what you expect it to be.

### ACHTUNG!

The `return` statement instantly ends the function call and passes the return value to the function caller. However, functions are not required to have a return statement. A function without a return statement implicitly returns the Python constant `None`, which is similar to the special value `null` of many other languages. Calling `print()` at the end of a function does **not** cause a function to return any values.

```
>>> def oops(i):
...     """Increment i (but forget to return anything)."""
...     print(i + 1)
...
>>> def increment(i):
...     """Increment i."""
...     return i + 1
...
>>> x = oops(1999)          # x contains 'None' since oops()
2000                         # doesn't have a return statement.
>>> y = increment(1999)      # However, y contains a value.
>>> print(x, y)
None 2000
```

If you have any intention of using the results of a function, use a `return` statement.

It is also possible to specify *default values* for a function's parameters. In the following example, the function `pad()` has three parameters, and the value of `c` defaults to 0. If it is not specified in the function call, the variable `c` will contain the value 0 when the function is executed.

```
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
...
```

```
>>> pad(1, 2, 3)          # Specify each parameter.  
1 2 3  
>>> pad(1, 2)           # Specify only non-default parameters.  
1 2 0
```

It's important to note that positional arguments must precede named arguments in a function call. Additionally, parameters without default values must precede parameters with default values in a function definition. For example, `a` and `b` must come before `c` in the function definition of `pad()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

```
# Try defining pad() with a named argument before a positional argument.  
>>> def pad(c=0, a, b):  
...     print(a, b, c)  
...  
SyntaxError: non-default argument follows default argument
```

```
# Correctly define pad() with the named argument after positional arguments.  
>>> def pad(a, b, c=0):  
...     """Print the arguments, plus a zero if c is not specified."""  
...     print(a, b, c)  
  
# Call pad() with 3 positional arguments.  
>>> pad(2, 4, 6)  
2 4 6  
  
# Call pad() with 3 named arguments. Note the change in order.  
>>> pad(b=3, c=5, a=7)  
7 3 5  
  
# Call pad() with 2 named arguments, excluding c.  
>>> pad(b=1, a=2)  
2 1 0  
  
# Call pad() with 1 positional argument and 2 named arguments.  
>>> pad(1, c=2, b=3)  
1 3 2
```

**Problem 3.** The built-in `print()` function has the useful keyword arguments `sep` and `end`. It accepts any number of positional arguments and prints them out with `sep` inserted between values (defaulting to a space), then prints `end` (defaulting to the *newline character* '`\n`').

Write a function called `isolate()` that accepts five arguments. The function should print the first three arguments separated by 5 spaces and then print the last two arguments with a single space separating the last three arguments. For example,

```
>>> isolate(1, 2, 3, 4, 5)
1     2     3 4 5
```

### ACHTUNG!

In previous versions of Python, `print()` was a *statement* (like `return`), not a function, and could therefore be executed without parentheses. However, it lacked keyword arguments like `sep` and `end`. If you are using Python 2.7, include the following line at the top of the file to turn the `print` statement into the new `print()` function.

```
>>> from __future__ import print_function
```

## Data Types and Structures

### Numerical Types

Python has four numerical data types: `int`, `long`, `float`, and `complex`. Each stores a different kind of number. The built-in function `type()` identifies an object's data type.

```
>>> type(3)                                # Numbers without periods are integers.
int

>>> type(3.0)                             # Floats have periods (3. is also a float).
float
```

Python has two types of division: integer and float. The `/` operator performs float division (true fractional division), and the `//` operator performs integer division, which rounds the result down to the next integer. If both operands for `//` are integers, the result will be an `int`. If one or both operands are floats, the result will be a `float`. Regular division with `/` always returns a `float`.

```
>>> 15 / 4                                # Float division performs as expected.
3.75
>>> 15 // 4                               # Integer division rounds the result down.
3
>>> 15. // 4
3.0
```

### ACHTUNG!

In previous versions of Python, using `/` with two integers performed integer division, even in cases where the division was not even. This can result in some incredibly subtle and frustrating errors. If you are using Python 2.7, always include a `.` on the operands or cast at least one as a float when you want float division.

```
# PYTHON 2.7
>>> 15 / 4                                # The answer should be 3.75, but the
3                                         # interpreter does integer division!

>>> 15. / float(4)                         # 15. and float(4) are both floats, so
3.75                                       # the interpreter does float division.
```

Alternatively, including the following line at the top of the file redefines the `/` and `//` operators so they are handled the same way as in Python 3.

```
>>> from __future__ import division
```

Python also supports complex numbers computations by pairing two numbers as the real and imaginary parts. Use the letter *j*, not *i*, for the imaginary part.

```
>>> x = complex(2,3)                      # Create a complex number this way...
>>> y = 4 + 5j                            # ...or this way, using j (not i).
>>> x.real                               # Access the real part of x.
2.0
>>> y.imag                               # Access the imaginary part of y.
5.0
```

## Strings

In Python, strings are created with either single or double quotes. To concatenate two or more strings, use the `+` operator between string variables or literals.

```
>>> str1 = "Hello"
>>> str2 = 'world'
>>> my_string = str1 + " " + str2 + '!'
>>> my_string
'Hello world!'
```

Parts of a string can be accessed using *slicing*, indicated by square brackets `[ ]`. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 1.

```
>>> my_string = "Hello world!"
>>> my_string[4]                           # Indexing begins at 0.
'o'
>>> my_string[-1]                          # Negative indices count backward from the end.
```

```
'!'

# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
'Hello'

# Slice from the 6th character to the end.
>>> my_string[6:]
'world!'

# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'

# Get every other character in the string.
>>> my_string[::-2]
'Hlowrd'
```

**Problem 4.** Write two new functions, called `first_half()` and `backward()`.

1. `first_half()` should accept a parameter and return the first half of it, excluding the middle character if there is an odd number of characters.  
(Hint: the built-in function `len()` returns the length of the input.)
2. The `backward()` function should accept a parameter and reverse the order of its characters using slicing, then return the reversed parameter.  
(Hint: The `step` parameter used in slicing can be negative.)

Use IPython to quickly test your syntax for each function.

## Lists

A Python `list` is created by enclosing comma-separated values with square brackets `[ ]`. Entries of a list do **not** have to be of the same type. Access entries in a list with the same indexing or slicing operations used with strings.

```
>>> my_list = ["Hello", 93.8, "world", 10]
>>> my_list[0]
'Hello'
>>> my_list[-2]
'world'
>>> my_list[:2]
['Hello', 93.8]
```

Common list methods (functions) include `append()`, `insert()`, `remove()`, and `pop()`. Consult IPython for details on each of these methods using object introspection.

```
>>> my_list = [1, 2]                      # Create a simple list of two integers.
>>> my_list.append(4)                     # Append the integer 4 to the end.
>>> my_list.insert(2, 3)                  # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)                   # Remove 3 from the list.
>>> my_list.pop()                       # Remove (and return) the last entry.
4
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"           # 'in' also works on strings.
False
```

## Tuples

A Python `tuple` is an ordered collection of elements, created by enclosing comma-separated values with parentheses ( and ). Tuples are similar to lists, but they are much more rigid, have fewer built-in operations, and cannot be altered after creation. Lists are therefore preferable for managing dynamic ordered collections of objects.

When multiple objects are returned by a function, they are returned as a tuple. For example, recall that the `arithmetic()` function returns two values.

```
>>> x, y = arithmetic(5,2)                # Get each value individually,
>>> print(x, y)
3 10
>>> both = arithmetic(5,2)                 # or get them both as a tuple.
>>> print(both)
(3, 10)
```

**Problem 5.** Write a function called `list_ops()`. Define a list with the entries "bear", "ant", "cat", and "dog", in that order. Then perform the following operations on the list:

1. Append "eagle".
2. Replace the entry at index 2 with "fox".
3. Remove (or pop) the entry at index 1.
4. Sort the list in reverse alphabetical order.
5. Replace "eagle" with "hawk".  
(Hint: the list's `index()` method may be helpful.)
6. Add the string "hunter" to the last entry in the list.

Return the resulting list of strings.

Work out (on paper) what the result should be, then check that your function returns the correct list. Consider printing the list at each step to see the intermediate results.

## Sets

A Python `set` is an unordered collection of distinct objects. Objects can be added to or removed from a set after its creation. Initialize a set with curly braces {}, separating the values by commas, or use `set()` to create an empty set. Like mathematical sets, Python sets have operations like union, intersection, difference, and symmetric difference.

```
# Initialize some sets. Note that repeats are not added.
>>> gym_members = {"Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"}
>>> print(gym_members)
{'Doe, John', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.add("Lytle, Josh")      # Add an object to the set.
>>> gym_members.discard("Doe, John")    # Delete an object from the set.
>>> print(gym_members)
{'Lytle, Josh', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Ian", "Webb, Jared"})
{'Lytle, Josh'}
>>> gym_members.difference({"Brown, Bob", "Sharp, Sarah"})
{'Lytle, Josh', 'Smith, Jane'}
```

## Dictionaries

Like a set, a Python `dict` (dictionary) is an unordered data type. A dictionary stores key-value pairs, called *items*. The values of a dictionary are indexed by its keys. Dictionaries are initialized with curly braces, colons, and commas. Use `dict()` or {} to create an empty dictionary.

```
>>> my_dictionary = {"business": 4121, "math": 2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
2061

# Add a value indexed by 'science' and delete the 'business' keypair.
>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")           # Use 'pop' or 'popitem' to remove.
4121
>>> print(my_dictionary)
{'math': 2061, 'visual arts': 7321, 'science': 6284}

# Display the keys and values.
>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])
```

As far as data access goes, lists are like dictionaries whose keys are the integers  $0, 1, \dots, n - 1$ , where  $n$  is the number of items in the list. The keys of a dictionary need not be integers, but they must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the Standard Library lab.

## Type Casting

The names of each of Python's data types can be used as functions to cast a value as that type. This is particularly useful for converting between integers and floats.

```
# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)
>>> z = complex(3)
>>> print(x, y, z)
3 3.0 (3+0j)

# Cast a list as a set and vice versa.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
>>> str(list(set([complex(float(3))])))
'[(3+0j)]'
```

## Control Flow Tools

Control flow blocks dictate the order in which code is executed. Python supports the usual control flow statements including `if` statements, `while` loops and `for` loops.

### The If Statement

An `if` statement executes the indented code `if` (and only if) the given condition holds. The `elif` statement is short for “else if” and can be used multiple times following an if statement, or not at all. The `else` keyword may be used at most once at the end of a series of `if/elif` statements.

```
>>> food = "bagel"
>>> if food == "apple":           # As with functions, the colon denotes
...     print("72 calories")      # the start of each code block.
... elif food == "banana" or food == "carrot":
...     print("105 calories")
... else:
...     print("calorie count unavailable")
...
calorie count unavailable
```

**Problem 6.** Write a function called `pig_latin()`. Accept a string parameter `word`, translate it into Pig Latin, then return the translation. Specifically, if `word` starts with a vowel, add “hay” to the end; if `word` starts with a consonant, take the first character of `word`, move it to the end, and add “ay”.

(Hint: use the `in` operator to check if the first letter is a vowel.)

### The While Loop

A `while` loop executes an indented block of code `while` the given condition holds.

```
>>> i = 0
>>> while i < 10:
...     print(i, end=' ')
...     i += 1
...
0 1 2 3 4 5 6 7 8 9
```

There are two additional useful statements to use inside of loops:

1. `break` manually exits the loop, regardless of which iteration the loop is on or if the termination condition is met.
2. `continue` skips the current iteration and returns to the top of the loop block if the termination condition is still not met.

```
>>> i = 0
>>> while True:
...     print(i, end=' ')
...     i += 1
...     if i >= 10:
...         break                  # Exit the loop.
...
0 1 2 3 4 5 6 7 8 9

>>> i = 0
>>> while i < 10:
...     i += 1
...     if i % 3 == 0:
...         continue            # Skip multiples of 3.
...     print(i, end=' ')
1 2 4 5 7 8 10
```

## The For Loop

A `for` loop iterates over the items in any *iterable*. Iterables include (but are not limited to) strings, lists, sets, and dictionaries.

```
>>> colors = ["red", "green", "blue", "yellow"]
>>> for entry in colors:
...     print(entry + "!")
...
red!
green!
blue!
yellow!
```

The `break` and `continue` statements also work in for loops, but a `continue` in a for loop will automatically increment the index or item, whereas a `continue` in a while loop makes no automatic changes to any variable.

```
>>> for word in ["It", "definitely", "looks", "pretty", "bad", "today"]:
...     if word == "definitely":
...         continue
...     elif word == "bad":
...         break
...     print(word, end=' ')
...
It looks pretty
```

In addition, Python has some very useful built-in functions that can be used in conjunction with the `for` statement:

1. `range(start, stop, step)`: Produces a sequence of integers, following slicing syntax. If only one argument is specified, it produces a sequence of integers from 0 up to (but not including) the argument, incrementing by one. This function is used **very** often.
2. `zip()`: Joins multiple sequences in parallel so they can be iterated over simultaneously.
3. `enumerate()`: Yields both a count and a value from the sequence. Typically used to get both the index of an item and the actual item simultaneously.
4. `reversed()`: Reverses the order of the iteration.
5. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

Each of these functions except for `sorted()` returns an *iterator*, an object that is built specifically for looping but not for creating actual lists. To put the items of the sequence in a collection, use `list()`, `set()`, or `tuple()`.

```
# Strings and lists are both iterables.
>>> vowels = "aeiou"
>>> colors = ["red", "yellow", "white", "blue", "purple"]

# Iterate by index.
>>> for i in range(5):
...     print(i, vowels[i], colors[i])
...
0 a red
1 e yellow
2 i white
3 o blue
4 u purple

# Iterate through both sequences at once.
>>> for letter, word in zip(vowels, colors):
...     print(letter, word)
...
a red
e yellow
i white
o blue
u purple

# Get the index and the item simultaneously.
>>> for i, color in enumerate(colors): #
...     print(i, color)
...
0 red
1 yellow
2 white
3 blue
4 purple
```

```
# Iterate through the list in sorted (alphabetical) order.
>>> for item in sorted(colors):
...     print(item, end=' ')
...
blue purple red white yellow

# Iterate through the list backward.
>>> for item in reversed(colors):
...     print(item, end=' ')
...
purple blue white yellow red

# range() arguments follow slicing syntax.
>>> list(range(10))                  # Integers from 0 to 10, exclusive.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(4, 8))                # Integers from 4 to 8, exclusive.
[4, 5, 6, 7]

>>> set(range(2, 20, 3))            # Every third integer from 2 to 20.
{2, 5, 8, 11, 14, 17}
```

**Problem 7.** This problem originates from <https://projecteuler.net>, an excellent resource for math-related coding problems.

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is  $9009 = 91 \times 99$ . Write a function called `palindrome()` that finds and returns the largest palindromic number made from the product of two 3-digit numbers.

## List Comprehension

A *list comprehension* uses for loop syntax between square brackets to create a list. This is a powerful, efficient way to build lists. The code is concise and runs quickly.

```
>>> [float(n) for n in range(5)]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

List comprehensions can be thought of as “inverted loops”, meaning that the body of the loop comes before the looping condition. The following loop and list comprehension produce the same list, but the list comprehension takes only about two-thirds the time to execute.

```
>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]
```

Tuple, set, and dictionary comprehensions can be done in the same way as list comprehensions by using the appropriate style of brackets on the end.

```
>>> colors = ["red", "blue", "yellow"]
>>> {c[0]:c for c in colors}
{'y': 'yellow', 'r': 'red', 'b': 'blue'}
```

```
>>> {"bright " + c for c in colors}
{'bright blue', 'bright red', 'bright yellow'}
```

**Problem 8.** The alternating harmonic series is defined as follows.

$$\sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots = \ln(2)$$

Write a function called `alt_harmonic()` that accepts an integer  $n$ . Use a list comprehension to quickly compute and sum the first  $n$  terms of this series (be careful not to sum only  $n - 1$  terms). The sum of the first 500,000 terms of this series approximates  $\ln(2)$  to five decimal places.

(Hint: consider using Python's built-in `sum()` function.)

## Additional Material

### Further Reading

Refer back to this and other introductory labs often as you continue getting used to Python syntax and data types. As you continue your study of Python, we strongly recommend the following readings.

- The official Python tutorial: <https://docs.python.org/3/tutorial/introduction.html> (especially chapters 3, 4, and 5).
- Section 1.2 of the SciPy lecture notes: <http://scipy-lectures.github.io/>.
- PEP8 - Python style guide: <http://www.python.org/dev/peps/pep-0008/>.

### Generalized Function Input

On rare occasion, it is necessary to define a function without knowing exactly what the parameters will be like or how many there will be. This is usually done by defining the function with the parameters `*args` and `**kwargs`. Here `*args` is a list of the positional arguments and `**kwargs` is a dictionary mapping the keywords to their argument. This is the most general form of a function definition.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print("Argument " + str(i) + ":", arg)
...     for key in kwargs:
...         print("Keyword", key, "-->", kwargs[key])
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword missing --> True
Keyword exceptional --> False
```

See <https://docs.python.org/3/tutorial/controlflow.html> for more on this topic.

### Function Decorators

A *function decorator* is a special function that “wraps” other functions. It takes in a function as input and returns a new function that pre-processes the inputs or post-processes the outputs of the original function.

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs)      # Call the decorated function.
...         print("output type:", type(output)) # Process before finishing.
...         return output                     # Return the function output.
...     return wrapper
```

The outer function, `typewriter()`, returns the new function `wrapper()`. Since `wrapper()` accepts `*args` and `**kwargs` as arguments, the input function `func()` could accept any number of positional or keyword arguments.

Apply a decorator to a function by tagging the function's definition with an @ symbol and the decorator name.

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b // c
```

Placing the tag above the definition is equivalent to adding the following line of code after the function definition:

```
>>> combine = typewriter(combine)
```

Now calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Function decorators can also be customized with arguments. This requires another level of nesting: the outermost function must define and return a decorator that defines and returns a wrapper.

```
>>> def repeat(times):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(3)
... def hello_world():
...     print("Hello, world!")
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

See <https://www.python.org/dev/peps/pep-0318/> for more details.



# 2

# Introduction to NumPy

**Lab Objective:** *NumPy is a powerful Python package for manipulating data with multi-dimensional vectors. Its versatility and speed makes Python an ideal language for applied and computational mathematics. In this lab we introduce basic NumPy data structures and operations as a first step to numerical computing in Python.*

## Arrays

In many algorithms, data can be represented mathematically as a *vector* or a *matrix*. Conceptually, a vector is just a list of numbers and a matrix is a two-dimensional list of numbers (a list of lists). However, even basic linear algebra operations like matrix multiplication are cumbersome to implement and slow to execute when data is stored this way. The *NumPy* module<sup>1</sup> [Oli06, ADH<sup>+</sup>01, Oli07] offers a much better solution.

The basic object in NumPy is the *array*, which is conceptually similar to a matrix. The NumPy array class is called `ndarray` (for “*n*-dimensional array”). The simplest way to explicitly create a 1-D `ndarray` is to define a list, then cast that list as an `ndarray` with NumPy’s `array()` function.

```
>>> import numpy as np

# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])

# The string representation has no commas or an array() label.
>>> print(np.array([1, 3, 5, 7, 9]))
[1 3 5 7 9]
```

The alias “`np`” is standard in the Python community.

An `ndarray` can have arbitrarily many dimensions. A 2-D array is a 1-D array of 1-D arrays (like a list of lists), a 3-D array is a 1-D array of 2-D arrays (a list of lists of lists), and, more generally, an *n*-dimensional array is a 1-D array of (*n* – 1)-dimensional arrays (a list of lists of lists of lists...). Each dimension is called an *axis*. For a 2-D array, the 0-axis indexes the rows and the 1-axis indexes the columns. Elements are accessed using brackets and indices, with the axes separated by commas.

---

<sup>1</sup>NumPy is *not* part of the standard library, but it is included in most Python distributions.

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array( [ [1, 2, 3], [4, 5, 6] ] )
>>> print(A)
[[1 2 3]
 [4 5 6]]

# Access elements of the array with brackets.
>>> print(A[0, 1], A[1, 2])
2 6

# The elements of a 2-D array are 1-D arrays.
>>> A[0]
array([1, 2, 3])
```

**Problem 1.** There are two main ways to perform matrix multiplication in NumPy: with NumPy's `dot()` function (`np.dot(A, B)`), or with the `@` operator (`A @ B`). Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 1 & 5 & -9 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 6 & -5 & 3 \\ 5 & -8 & 9 & 7 \\ 9 & -3 & -2 & -3 \end{bmatrix}$$

Return the matrix product  $AB$ .

For examples of array initialization and matrix multiplication, use object introspection in IPython to look up the documentation for `np.ndarray`, `np.array()` and `np.dot()`.

```
In [1]: import numpy as np

In [2]: np.array? # press 'enter'
```

### ACHTUNG!

The `@` operator was not introduced until Python 3.5. It triggers the `__matmul__()` magic method,<sup>a</sup> which for the `ndarray` is essentially a wrapper around `np.dot()`. If you are using a previous version of Python, always use `np.dot()` to perform basic matrix multiplication.

<sup>a</sup>See the lab on Object Oriented Programming for an overview of magic methods.

## Basic Array Operations

NumPy arrays behave differently with respect to the binary arithmetic operators `+` and `*` than Python lists do. For lists, `+` concatenates two lists and `*` replicates a list by a scalar amount (strings also behave this way).

```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

# Multiplication concatenates a list with itself a given number of times.
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

NumPy arrays act like mathematical vectors and matrices: `+` and `*` perform component-wise addition or multiplication.

```
>>> x, y = np.array([1, 2, 3]), np.array([4, 5, 6])

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10                                # Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4                                # Multiply each entry of x by 4.
array([ 4,  8, 12])

# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])

# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

**Problem 2.** Write a function that defines the following matrix as a NumPy array.

$$A = \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ -5 & 3 & 1 \end{bmatrix}$$

Return the matrix  $-A^3 + 9A^2 - 15A$ .

In this context,  $A^2 = AA$  (the matrix product, not the component-wise square). The somewhat surprising result is a demonstration of the Cayley-Hamilton theorem.

## Array Attributes

An `ndarray` object has several attributes, some of which are listed below.

Attribute	Description
<code>dtype</code>	The type of the elements in the array.
<code>ndim</code>	The number of axes (dimensions) of the array.
<code>shape</code>	A tuple of integers indicating the size in each dimension.
<code>size</code>	The total number of elements in the array.

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])

# 'A' is a 2-D array with 2 rows, 3 columns, and 6 entries.
>>> print(A.ndim, A.shape, A.size)
2 (2, 3) 6
```

Note that `ndim` is the number of entries in `shape`, and that the `size` of the array is the product of the entries of `shape`.

## Array Creation Routines

In addition to casting other structures as arrays via `np.array()`, NumPy provides efficient ways to create certain commonly-used arrays.

Function	Returns
<code>arange()</code>	Array of sequential integers (like <code>list(range())</code> ).
<code>eye()</code>	2-D array with ones on the diagonal and zeros elsewhere.
<code>ones()</code>	Array of given shape and type, filled with ones.
<code>ones_like()</code>	Array of ones with the same shape and type as a given array.
<code>zeros()</code>	Array of given shape and type, filled with zeros.
<code>zeros_like()</code>	Array of zeros with the same shape and type as a given array.
<code>full()</code>	Array of given shape and type, filled with a specified value.
<code>full_like()</code>	Full array with the same shape and type as a given array.

Each of these functions accepts the keyword argument `dtype` to specify the data type. Common types include `np.bool_`, `np.int64`, `np.float64`, and `np.complex128`.

```
# A 1-D array of 5 zeros.
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# A 2x5 matrix (2-D array) of integer ones.
>>> np.ones((2,5), dtype=np.int)      # The shape is specified as a tuple.
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

# The 2x2 identity matrix.
>>> I = np.eye(2)
>>> print(I)
[[ 1.  0.]
 [ 0.  1.]]

# Array of 3s the same size as 'I'.
>>> np.full_like(I, 3)              # Equivalent to np.full(I.shape, 3).
array([[ 3.,  3.],
       [ 3.,  3.]])
```

Unlike native Python data structures, **all elements of a NumPy array must be of the same data type**. To change an existing array's data type, use the array's `astype()` method.

```
# A list of integers becomes an array of integers.
>>> x = np.array([0, 1, 2, 3, 4])
>>> print(x)
[0 1 2 3 4]
>>> x.dtype
dtype('int64')

# Change the data type to one of NumPy's float types.
>>> x = x.astype(np.float64)          # Equivalent to x = np.float64(x).
>>> print(x)                        # Floats are displayed with periods.
[ 0.  1.  2.  3.  4.]
>>> x.dtype
dtype('float64')
```

The following functions are for dealing with the diagonal, upper, or lower portion of an array.

Function	Description
<code>diag()</code>	Extract a diagonal or construct a diagonal array.
<code>tril()</code>	Get the lower-triangular portion of an array by replacing entries above the diagonal with zeros.
<code>triu()</code>	Get the upper-triangular portion of an array by replacing entries below the diagonal with zeros.

```
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Get only the upper triangular entries of 'A'.
>>> np.triu(A)
array([[1, 2, 3],
       [0, 5, 6],
       [0, 0, 9]])

# Get the diagonal entries of 'A' as a 1-D array.
>>> np.diag(A)
array([1, 5, 9])

# diag() can also be used to create a diagonal matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[ 1,  0,  0],
       [ 0, 11,  0],
       [ 0,  0, 111]])
```

See <http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html> for the official documentation on NumPy's array creation routines.

**Problem 3.** Write a function that defines the following matrices as NumPy arrays using the functions presented in this section (not `np.array()`). Calculate the matrix product  $ABA$ . Change the data type of the resulting matrix to `np.int64`, then return it.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} -1 & 5 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

## Data Access

### Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access elements of the array with slicing syntax.
>>> x[3]                                # The element at index 3.
3
>>> x[:3]                               # Everything up to index 3 (exclusive).
array([0, 1, 2])
>>> x[3:]                               # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                             # The elements from index 3 to 8.
array([3, 4, 5, 6, 7])

>>> A = np.array([[0,1,2,3,4],[5,6,7,8,9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> A[1, 2]                            # The element at row 1, column 2.
7
>>> A[:, 2:]                           # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

### NOTE

Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. Changes to a copy of an array does not affect the original array, but copying an array uses more time and memory than getting a view.

## Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)          # The integers from 0 to 50 by tens.
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])        # Get the 3rd, 1st, and 4th elements.
>>> x[index]                         # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]                           # Get the 0th and 3rd entries.
array([ 0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like `<` and `==` to create masks.

```
>>> y = np.arange(10, 20, 2)          # Every other integer from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                  # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False,  True,  True], dtype=bool)
>>> y[mask]                         # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

**Problem 4.** Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the resulting array.

## Array Manipulation

### Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)                      # The integers from 0 to 12 (exclusive).
>>> print(A)
[ 0  1  2  3  4  5  6  7  8  9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3,4))                      # The new shape is specified as a tuple.
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2,-1))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> A = np.arange(12).reshape((3,4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                         # Equivalent to A.reshape(A.size)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                                # Equivalent to np.transpose(A).
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

## NOTE

By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into “flat” 1-D arrays. For example, by default an array will have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented to purposefully work well with 1-D “row arrays”.

```
>>> A = np.arange(10).reshape((2,5))
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Slicing out a column of A still produces a "flat" 1-D array.
>>> x = A[:,1]                      # All of the rows, column 1.
>>> x
array([1, 6])                      # Not array([[1],
                                         #           [6]])
>>> x.shape
(2,)
>>> x.ndim
1
```

However, it is occasionally necessary to change a 1-D array into a “column array”. Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((-1,1))              # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
       [1],
       [2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

## Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
<code>concatenate()</code>	Join a sequence of arrays along an existing axis
<code>hstack()</code>	Stack arrays in sequence horizontally (column wise).
<code>vstack()</code>	Stack arrays in sequence vertically (row wise).
<code>column_stack()</code>	Stack 1-D arrays as columns into a 2-D array.

```

>>> A = np.arange(6).reshape((2,3))
>>> B = np.zeros((4,3))

# vstack() stacks arrays vertically (row-wise).
>>> np.vstack((A,B,A))
array([[ 0.,  1.,  2.],                  # A
       [ 3.,  4.,  5.],
       [ 0.,  0.,  0.],                  # B
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  2.],                  # A
       [ 3.,  4.,  5.]])
```

```

>>> A = A.T
>>> B = np.ones((3,4))

# hstack() stacks arrays horizontally (column-wise).
>>> np.hstack((A,B,A))
array([[ 0.,  3.,  1.,  1.,  1.,  0.,  3.],
       [ 1.,  4.,  1.,  1.,  1.,  1.,  4.],
       [ 2.,  5.,  1.,  1.,  1.,  2.,  5.]])
```

```

# column_stack() stacks arrays horizontally, including 1-D arrays.
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[ 0.,  3.,  0.,  1.,  2.],
       [ 1.,  4.,  0.,  1.,  2.],
       [ 2.,  5.,  0.,  1.,  2.]])
```

See <http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html> for more array manipulation routines and documentation.

**Problem 5.** Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 0 \\ 3 & 3 & 0 \\ 3 & 3 & 3 \end{bmatrix} \quad C = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

Use NumPy's stacking functions to create and return the block matrix:

$$\begin{bmatrix} \mathbf{0} & A^T & I \\ A & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & C \end{bmatrix},$$

where  $I$  is the  $3 \times 3$  identity matrix and each  $\mathbf{0}$  is a matrix of all zeros of appropriate size.

A block matrix of this form is used in the interior point method for linear optimization.

## Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape, such as element-wise addition. *Array broadcasting* extends such operations to accept some (but not all) operands with different shapes, and occurs automatically whenever possible.

Suppose, for example, that we would like to add different values to the columns of an  $m \times n$  matrix  $A$ . Adding a 1-D array  $x$  with the  $n$  entries to  $A$  will automatically do this correctly. To add different values to the different rows of  $A$ , first reshape a 1-D array of  $m$  values into a column array. Broadcasting then correctly takes care of the operation.

Broadcasting can also occur between two 1-D arrays, once they are reshaped appropriately.

```
>>> A = np.arange(12).reshape((4,3))
>>> x = np.arange(3)
>>> A
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> x
array([0, 1, 2])

# Add the entries of 'x' to the corresponding columns of 'A'.
>>> A + x
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])

>>> y = np.arange(0, 40, 10).reshape((4,1))
>>> y
array([[ 0],
       [10],
       [20],
       [30]])

# Add the entries of 'y' to the corresponding rows of 'A'.
>>> A + y
array([[ 0,  1,  2],
       [13, 14, 15],
       [26, 27, 28],
       [39, 40, 41]])

# Add 'x' and 'y' together with array broadcasting.
>>> x + y
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

## Numerical Computing with NumPy

### Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an array.

Function	Description
<code>abs()</code> or <code>absolute()</code>	Calculate the absolute value element-wise.
<code>exp()</code> / <code>log()</code>	Exponential ( $e^x$ ) / natural log element-wise.
<code>maximum()</code> / <code>minimum()</code>	Element-wise maximum / minimum of two arrays.
<code>sqrt()</code>	The positive square-root, element-wise.
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , etc.	Element-wise trigonometric operations.

```
>>> x = np.arange(-2,3)
>>> print(x, np.abs(x))           # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                  # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.          ,  0.84147098,  0.90929743])
```

See <http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs> for a more comprehensive list of universal functions.

### ACHTUNG!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.0855369232

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.          ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

## Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

Method	Returns
<code>all()</code>	<code>True</code> if all elements evaluate to <code>True</code> .
<code>any()</code>	<code>True</code> if any elements evaluate to <code>True</code> .
<code>argmax()</code>	Index of the maximum value.
<code>argmin()</code>	Index of the minimum value.
<code>argsort()</code>	Indices that would sort the array.
<code>clip()</code>	restrict values in an array to fit within a given range
<code>max()</code>	The maximum element of the array.
<code>mean()</code>	The average value of the array.
<code>min()</code>	The minimum element of the array.
<code>sort()</code>	Return nothing; sort the array in-place.
<code>std()</code>	The standard deviation of the array.
<code>sum()</code>	The sum of the elements of the array.
<code>var()</code>	The variance of the array.

Each of these `np.ndarray` methods has an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed can operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an  $n$ -D array, the return value is an  $(n - 1)$ -D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar. Refer to the NumPy Visual Guide in the appendix for more visual examples.

```
>>> A = np.arange(9).reshape((3,3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
>>> A.min(axis=0)                  # np.array([min(A[:,i]) for i in range(3)])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)                  # np.array([sum(A[i,:]) for i in range(3)])
array([3, 12, 21])
```

See <http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html> for a more comprehensive list of array methods.

**Problem 6.** A matrix is called *row-stochastic*<sup>a</sup> if its rows each sum to 1. Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function that accepts a matrix (as a 2-D NumPy array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

<sup>a</sup>Similarly, a matrix is called *column-stochastic* if its columns each sum to 1.

**Problem 7.** This problem comes from <https://projecteuler.net>.

In the  $20 \times 20$  grid below, four numbers along a diagonal line have been marked in red.

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

```

The product of these numbers is  $26 \times 63 \times 78 \times 14 = 1788696$ . Write a function that returns the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the grid.

For convenience, this array has been saved in the file `grid.npy`. Use the following syntax to extract the array:

```
>>> grid = np.load("grid.npy")
```

One way to approach this problem is to iterate through the rows and columns of the array, checking small slices of the array at each iteration and updating the current largest product. Array slicing, however, provides a much more efficient solution.

The naïve method for computing the greatest product of four adjacent numbers in a horizontal row might be as follows:

```
>>> winner = 0
>>> for i in range(20):
...     for j in range(17):
...         winner = max(np.prod(grid[i,j:j+4]), winner)
...
>>> winner
48477312
```

Instead, use array slicing to construct a single array where the  $(i, j)$ th entry is the product of the four numbers to the right of the  $(i, j)$ th entry in the original grid. Then find the largest element in the new array.

```
>>> np.max(grid[:, :-3] * grid[:, 1:-2] * grid[:, 2:-1] * grid[:, 3:])
48477312
```

Use slicing to similarly find the greatest products of four vertical, right diagonal, and left diagonal adjacent numbers.

(Hint: Consider drawing the portions of the grid that each slice in the above code covers, like the examples in the visual guide. Then draw the slices that produce vertical, right diagonal, or left diagonal sequences, and translate the pictures into slicing syntax.)

### ACHTUNG!

All of the examples in this lab use NumPy arrays, objects of type `np.ndarray`. NumPy also has a “matrix” data structure called `np.matrix` that was built specifically for MATLAB users who are transitioning to Python and NumPy. It behaves slightly differently than the regular array class, and can cause some unexpected and subtle problems.

For consistency (and your sanity), **never** use a NumPy matrix; **always** use NumPy arrays. If necessary, cast a matrix object as an array with `np.array()`.

## Additional Material

### Random Sampling

The submodule `np.random` holds many functions for creating arrays of random values chosen from probability distributions such as the uniform, normal, and multinomial distributions. It also contains some utility functions for getting non-distributional random samples, such as random integers or random samples from a given array.

Function	Description
<code>choice()</code>	Take random samples from a 1-D array.
<code>random()</code>	Uniformly distributed floats over [0, 1).
<code>randint()</code>	Random integers over a half-open interval.
<code>randn()</code>	Sample from the standard normal distribution.
<code>permutation()</code>	Randomly permute a sequence / generate a random sequence.
Function	Distribution
<code>beta()</code>	Beta distribution over [0, 1].
<code>binomial()</code>	Binomial distribution.
<code>exponential()</code>	Exponential distribution.
<code>gamma()</code>	Gamma distribution.
<code>geometric()</code>	Geometric distribution.
<code>multinomial()</code>	Multivariate generalization of the binomial distribution.
<code>multivariate_normal()</code>	Multivariate generalization of the normal distribution.
<code>normal()</code>	Normal / Gaussian distribution.
<code>poisson()</code>	Poisson distribution.
<code>uniform()</code>	Uniform distribution.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 5 uniformly distributed values in the interval [0, 1].
>>> np.random.random(5)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])

# A 2x5 matrix (2-D array) of integers in the interval [10, 20].
>>> np.random.randint(10, 20, (2,5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

### Saving and Loading Arrays

It is often useful to save an array as a file for later use. NumPy provides several easy methods for saving and loading array data.

Function	Description
<code>save()</code>	Save a single array to a <code>.npy</code> file.
<code>savez()</code>	Save multiple arrays to a <code>.npz</code> file.
<code>savetxt()</code>	Save a single array to a <code>.txt</code> file.
<code>load()</code>	Load and return an array or arrays from a <code>.npy</code> or <code>.npz</code> file.
<code>loadtxt()</code>	Load and return an array from a text file.

```
# Save a 100x100 matrix of uniformly distributed random values.
>>> x = np.random.random((100,100))
>>> np.save("uniform.npy", x)          # Or np.savetxt("uniform.txt", x).

# Read the array from the file and check that it matches the original.
>>> y = np.load("uniform.npy")        # Or np.loadtxt("uniform.txt").
>>> np.allclose(x, y)                # Check that x and y are close entry-wise.
True
```

To save several arrays to a single file, specify a keyword argument for each array in `np.savez()`. Then `np.load()` will return a dictionary-like object with the keyword parameter names from the save command as the keys.

```
# Save two 100x100 matrices of normally distributed random values.
>>> x = np.random.randn(100,100)
>>> y = np.random.randn(100,100)
>>> np.savez("normal.npz", first=x, second=y)

# Read the arrays from the file and check that they match the original.
>>> arrays = np.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```



# 3

# Introduction to Matplotlib

**Lab Objective:** *Matplotlib is the most commonly used data visualization library in Python. Being able to visualize data helps to determine patterns and communicate results and is a key component of applied and computational mathematics. In this lab we introduce techniques for visualizing data in 1, 2, and 3 dimensions. The plotting techniques presented here will be used in the remainder of the labs in the manual.*

## Line Plots

Raw numerical data is rarely helpful unless it can be visualized. The quickest way to visualize a simple 1-dimensional array is with a *line plot*. The following code creates an array of outputs of the function  $f(x) = x^2$ , then visualizes the array using the `matplotlib` module<sup>1</sup> [Hun07].

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

>>> y = np.arange(-5,6)**2
>>> y
array([25, 16,  9,  4,  1,  0,  1,  4,  9, 16, 25])

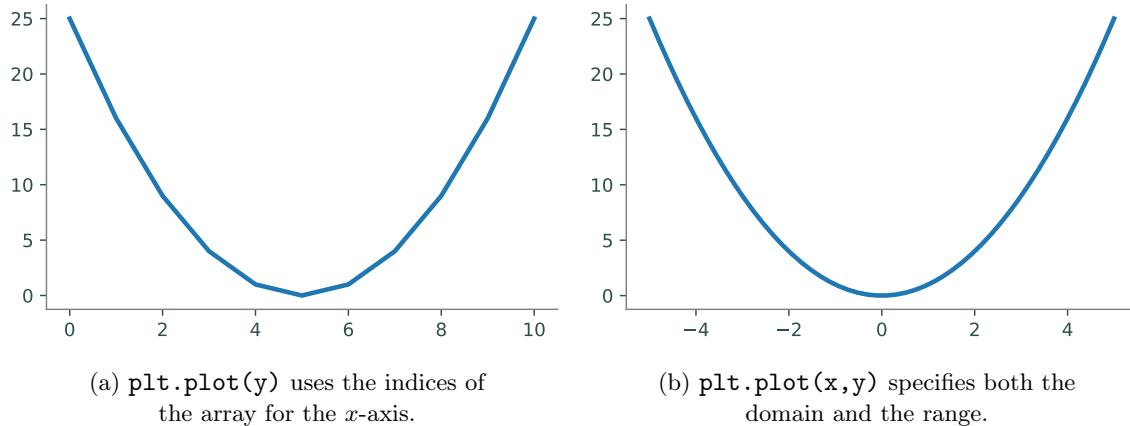
# Visualize the plot.
>>> plt.plot(y)                      # Draw the line plot.
[<matplotlib.lines.Line2D object at 0x1084762d0>]
>>> plt.show()                      # Reveal the resulting plot.
```

The result is shown in Figure 3.1a. Just as `np` is a standard alias for NumPy, `plt` is a standard alias for `matplotlib.pyplot` in the Python community.

The call `plt.plot(y)` creates a figure and draws straight lines connecting the entries of `y` relative to the  $y$ -axis. The  $x$ -axis is (by default) the index of the array, which in this case is the integers from 0 to 10. Calling `plt.show()` then displays the figure.

---

<sup>1</sup>Like NumPy, Matplotlib is *not* part of the Python standard library, but it is included in most Python distributions. See <https://matplotlib.org/> for the complete Matplotlib documentation.

Figure 3.1: Plots of  $f(x) = x^2$  over the interval  $[-5, 5]$ .

**Problem 1.** NumPy's `random` module has tools for sampling from probability distributions. For instance, `np.random.normal()` draws samples from the normal (Gaussian) distribution. The `size` parameter specifies the shape of the resulting array.

```
>>> np.random.normal(size=(2,3))      # Get a 2x3 array of samples.
array([[ 1.65896515, -0.43236783, -0.99390897],
       [-0.35753688, -0.76738306,  1.29683025]])
```

Write a function that accepts an integer  $n$  as input.

1. Use `np.random.normal()` to create an  $n \times n$  array of values randomly sampled from the standard normal distribution.
2. Compute the mean of each row of the array.  
(Hint: Use `np.mean()` and specify the `axis` keyword argument.)
3. Return the variance of these means.  
(Hint: Use `np.var()` to calculate the variance).

Define another function that creates an array of the results of the first function with inputs  $n = 100, 200, \dots, 1000$ . Plot (and show) the resulting array.

## Specifying a Domain

An obvious problem with Figure 3.1a is that the  $x$ -axis does not correspond correctly to the  $y$ -axis for the function  $f(x) = x^2$  that is being drawn. To correct this, define an array `x` for the domain, then use it to calculate the image `y = f(x)`. The command `plt.plot(x,y)` plots `x` against `y` by drawing a line between the consecutive points `(x[i], y[i])`.

Another problem with Figure 3.1a is its poor resolution: the curve is visibly bumpy, especially near the bottom of the curve. NumPy's `linspace()` function makes it easy to get a higher-resolution domain. Recall that `np.arange()` returns an array of evenly-spaced values in a given interval, where

the **spacing** between the entries is specified. In contrast, `np.linspace()` creates an array of evenly-spaced values in a given interval where the **number of elements** is specified.

```
# Get 4 evenly-spaced values between 0 and 32 (including endpoints).
>>> np.linspace(0, 32, 4)
array([ 0.          , 10.66666667, 21.33333333, 32.          ])

# Get 50 evenly-spaced values from -5 to 5 (including endpoints).
>>> x = np.linspace(-5, 5, 50)
>>> y = x**2                                # Calculate the image of f(x) = x**2.
>>> plt.plot(x, y)
>>> plt.show()
```

The resulting plot is shown in Figure 3.1b. This time, the  $x$ -axis correctly matches up with the  $y$ -axis. The resolution is also much better because `x` and `y` have 50 entries each instead of only 10.

Subsequent calls to `plt.plot()` modify the same figure until `plt.show()` is executed, which displays the current figure and resets the system. This behavior can be altered by specifying separate figures or axes, which we will discuss shortly.

#### NOTE

Plotting can seem a little mystical because the actual plot doesn't appear until `plt.show()` is executed. Matplotlib's *interactive mode* allows the user to see the plot be constructed one piece at a time. Use `plt.ion()` to turn interactive mode on and `plt.ioff()` to turn it off. This is very useful for quick experimentation. Try executing the following commands in IPython:

```
In [1]: import numpy as np
In [2]: from matplotlib import pyplot as plt

# Turn interactive mode on and make some plots.
In [3]: plt.ion()
In [4]: x = np.linspace(1, 4, 100)
In [5]: plt.plot(x, np.log(x))
In [6]: plt.plot(x, np.exp(x))

# Clear the figure, then turn interactive mode off.
In [7]: plt.clf()
In [8]: plt.ioff()
```

Use interactive mode **only** with IPython. Using interactive mode in a non-interactive setting may freeze the window or cause other problems.

**Problem 2.** Write a function that plots the functions  $\sin(x)$ ,  $\cos(x)$ , and  $\arctan(x)$  on the domain  $[-2\pi, 2\pi]$  (use `np.pi` for  $\pi$ ). Make sure the domain is refined enough to produce a figure with good resolution.

## Plot Customization

`plt.plot()` receives several keyword arguments for customizing the drawing. For example, the color and style of the line are specified by the following string arguments.

Key	Color	Key	Style
'b'	blue	'-'	solid line
'g'	green	'--'	dashed line
'r'	red	'-. '	dash-dot line
'c'	cyan	': '	dotted line
'k'	black	'o'	circle marker

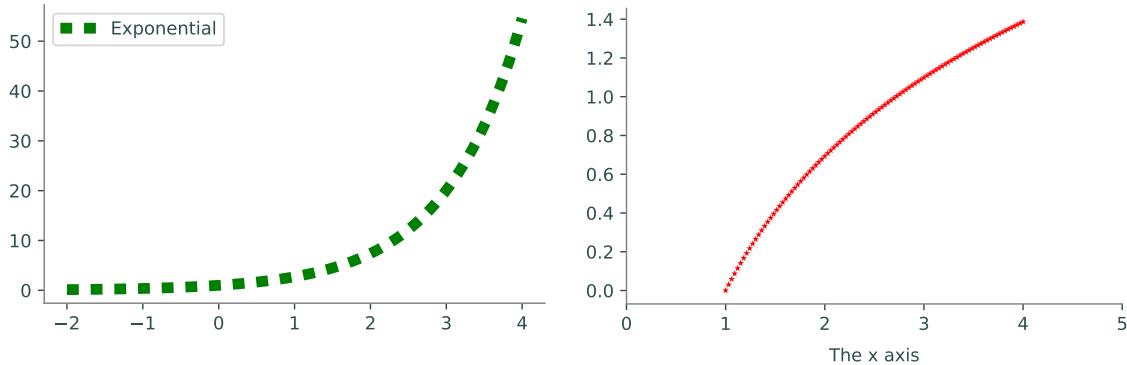
Specify one or both of these string codes as the third argument to `plt.plot()` to change from the default color and style. Other `plt` functions further customize a figure.

Function	Description
<code>legend()</code>	Place a legend in the plot
<code>title()</code>	Add a title to the plot
<code>xlim() / ylim()</code>	Set the limits of the <i>x</i> - or <i>y</i> -axis
<code>xlabel() / ylabel()</code>	Add a label to the <i>x</i> - or <i>y</i> -axis

```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', linewidth=6, label="Exponential")
>>> plt.title("This is the title.", fontsize=18)
>>> plt.legend(loc="upper left")      # plt.legend() uses the 'label' argument of
>>> plt.show()                      # plt.plot() to create a legend.

>>> x2 = np.linspace(1, 4, 100)
>>> plt.plot(x2, np.log(x2), 'r*', markersize=4)
>>> plt.xlim(0, 5)                  # Set the visible limits of the x axis.
>>> plt.xlabel("The x axis")        # Give the x axis a label.
>>> plt.show()
```

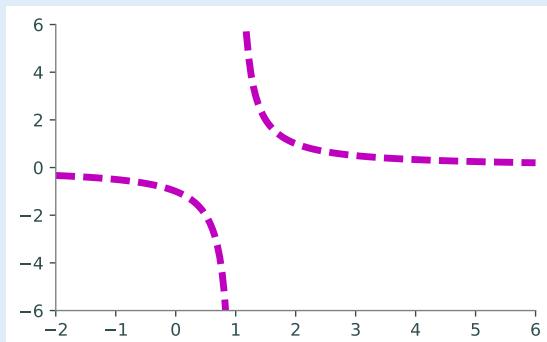
This is the title.



**Problem 3.** Write a function to plot the curve  $f(x) = \frac{1}{x-1}$  on the domain  $[-2, 6]$ .

1. Although  $f(x)$  has a discontinuity at  $x = 1$ , a single call to `plt.plot()` in the usual way will make the curve look continuous. Split up the domain into  $[-2, 1)$  and  $(1, 6]$ . Plot the two sides of the curve separately so that the graph looks discontinuous at  $x = 1$ .
2. Plot both curves with a dashed magenta line. Set the keyword argument `linewidth` (or `lw`) of `plt.plot()` to 4 to make the line a little thicker than the default setting.
3. Use `plt.xlim()` and `plt.ylim()` to change the range of the  $x$ -axis to  $[-2, 6]$  and the range of the  $y$ -axis to  $[-6, 6]$ .

The plot should resemble the figure below.



## Figures, Axes, and Subplots

The window that `plt.show()` reveals is called a *figure*, stored in Python as a `plt.Figure` object. A space on a figure where a plot is drawn is called an *axes*, a `plt.Axes` object. A figure can have multiple axes, and a single program may create several figures. There are several ways to create or grab figures and axes with `plt` functions.

Function	Description
<code>axes()</code>	Add an axes to the current figure
<code>figure()</code>	Create a new figure or grab an existing figure
<code>gca()</code>	Get the current axes
<code>gcf()</code>	Get the current figure
<code>subplot()</code>	Add a single subplot to the current figure
<code>subplots()</code>	Create a figure and add several subplots to it

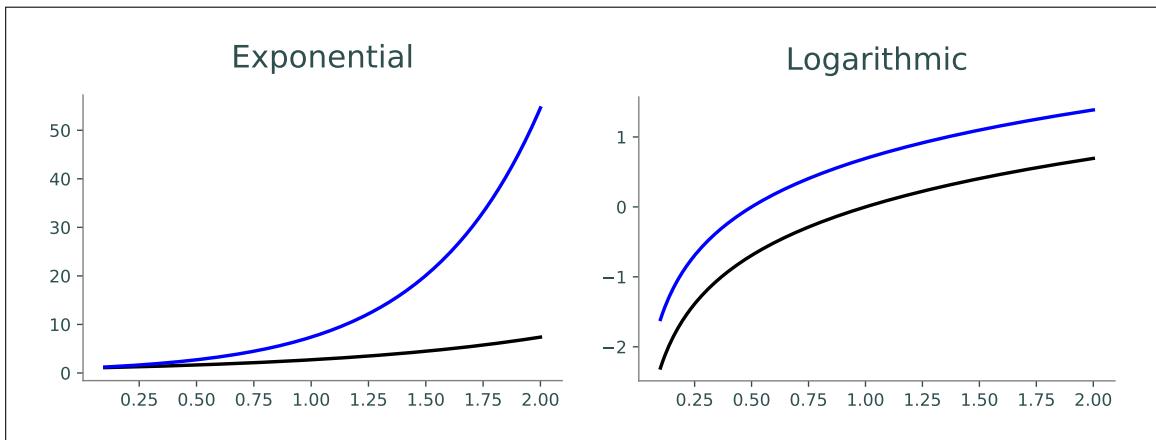
Usually when a figure has multiple axes, they are organized into non-overlapping *subplots*. The command `plt.subplot(nrows, ncols, plot_number)` creates an axes in a subplot grid where `nrows` is the number of rows of subplots in the figure, `ncols` is the number of columns, and `plot_number` specifies which subplot to modify. If the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.



Figure 3.3: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

```
>>> x = np.linspace(.1, 2, 200)
# Create a subplot to cover the left half of the figure.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, np.exp(x), 'k', lw=2)
>>> ax1.plot(x, np.exp(2*x), 'b', lw=2)
>>> plt.title("Exponential", fontsize=18)

# Create another subplot to cover the right half of the figure.
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, np.log(x), 'k', lw=2)
>>> ax2.plot(x, np.log(2*x), 'b', lw=2)
>>> ax2.set_title("Logarithmic", fontsize=18)
>>> plt.show()
```



## NOTE

Plotting functions such as `plt.plot()` are shortcuts for accessing the current axes on the current figure and calling a method on that `Axes` object. Calling `plt.subplot()` changes the current axis, and calling `plt.figure()` changes the current figure. Use `plt.gca()` to get the current axes and `plt.gcf()` to get the current figure. Compare the following equivalent strategies for producing a figure with two subplots.

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

**Problem 4.** Write a function that plots the functions  $\sin(x)$ ,  $\sin(2x)$ ,  $2\sin(x)$ , and  $2\sin(2x)$  on the domain  $[0, 2\pi]$ , each in a separate subplot of a single figure.

1. Arrange the plots in a  $2 \times 2$  grid of subplots.
2. Set the limits of each subplot to  $[0, 2\pi] \times [-2, 2]$ .  
(Hint: Consider using `plt.axis([xmin, xmax, ymin, ymax])` instead of `plt.xlim()` and `plt.ylim()` to set all boundaries simultaneously.)
3. Use `plt.title()` or `ax.set_title()` to give each subplot an appropriate title.
4. Use `plt.suptitle()` or `fig.suptitle()` to give the overall figure a title.
5. Use the following colors and line styles.

$\sin(x)$ : green solid line.       $\sin(2x)$ : red dashed line.

$2\sin(x)$ : blue dashed line.       $2\sin(2x)$ : magenta dotted line.

### ACHTUNG!

Be careful not to mix up the following functions.

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` (or `ax.axis()`) sets properties of the  $x$ - and  $y$ -axis in the current axes, such as the  $x$  and  $y$  limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

## Other Kinds of Plots

Line plots are not always the most illuminating choice of graph to describe a set of data. Matplotlib provides several other easy ways to visualize data.

- A *scatter plot* plots two 1-dimensional arrays against each other without drawing lines between the points. Scatter plots are particularly useful for data that is not correlated or ordered.

To create a scatter plot, use `plt.plot()` and specify a point marker (such as '`'o'` or '`'*'`') for the line style, or use `plt.scatter()` (or `ax.scatter()`). Beware that `plt.scatter()` has slightly different arguments and syntax than `plt.plot()`.

- A *histogram* groups entries of a 1-dimensional data set into a given number of intervals, called *bins*. Each bin has a bar whose height indicates the number of values that fall in the range of the bin. Histograms are best for displaying distributions, relating data values to frequency.

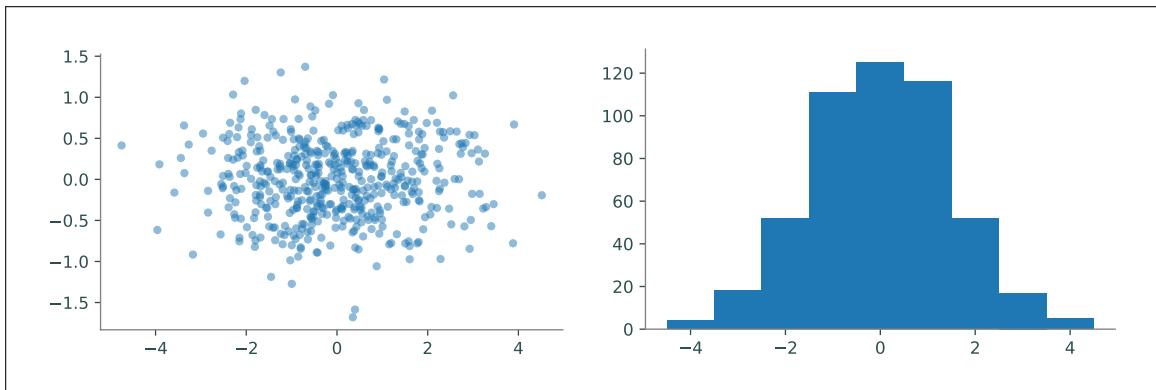
To create a histogram, use `plt.hist()` (or `ax.hist()`). Use the argument `bins` to specify the edges of the bins or to choose a number of bins. The `range` argument specifies the outer limits of the first and last bins.

```
# Get 500 random samples from two normal distributions.
>>> x = np.random.normal(scale=1.5, size=500)
>>> y = np.random.normal(scale=0.5, size=500)

# Draw a scatter plot of x against y, using transparent circle markers.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, y, 'o', markersize=5, alpha=.5)

# Draw a histogram to display the distribution of the data in x.
>>> ax2 = plt.subplot(122)
>>> ax2.hist(x, bins=np.arange(-4.5, 5.5))      # Or, equivalently,
#   ax2.hist(x, bins=9, range=[-4.5, 4.5])

>>> plt.show()
```



**Problem 5.** The Fatality Analysis Reporting System (FARS) is a nationwide census that provides yearly data regarding fatal injuries suffered in motor vehicle traffic crashes.<sup>a</sup> The array contained in `FARS.npy` is a small subset of the FARS database from 2010–2014. Each of the 148,206 rows in the array represents a different car crash; the columns represent the hour (in military time, as an integer), the longitude, and the latitude, in that order.

Write a function to visualize the data in `FARS.npy`. Use `np.load()` to load the data, then create a single figure with two subplots:

1. A scatter plot of longitudes against latitudes. Because of the large number of data points, use black pixel markers (use `"k"` as the third argument to `plt.plot()`). Label both axes using `plt.xlabel()` and `plt.ylabel()` (or `ax.set_xlabel()` and `ax.set_ylabel()`). (Hint: Use `plt.axis("equal")` or `ax.set_aspect("equal")` so that the  $x$ - and  $y$ -axis are scaled the same way.)
2. A histogram of the hours of the day, with one bin per hour. Set the limits of the  $x$ -axis appropriately. Label the  $x$ -axis. You should be able to clearly see which hours of the day experience more traffic.

<sup>a</sup>See <http://www.nhtsa.gov/FARS>.

Matplotlib also has tools for creating other kinds of plots for visualizing 1-dimensional data, including bar plots and box plots. See the Matplotlib Appendix for examples and syntax.

## Visualizing 3-D Surfaces

Line plots, histograms, and scatter plots are good for visualizing 1- and 2-dimensional data, including the domain and range of a function  $f : \mathbb{R} \rightarrow \mathbb{R}$ . However, visualizing 3-dimensional data or a function  $g : \mathbb{R}^2 \rightarrow \mathbb{R}$  (two inputs, one output) requires a different kind of plot. The process is similar to creating a line plot but requires slightly more setup: first construct an appropriate domain, then calculate the image of the function on that domain.

NumPy's `np.meshgrid()` function is the standard tool for creating a 2-dimensional domain in the Cartesian plane. Given two 1-dimensional coordinate arrays, `np.meshgrid()` creates two corresponding coordinate matrices. See Figure 3.6.

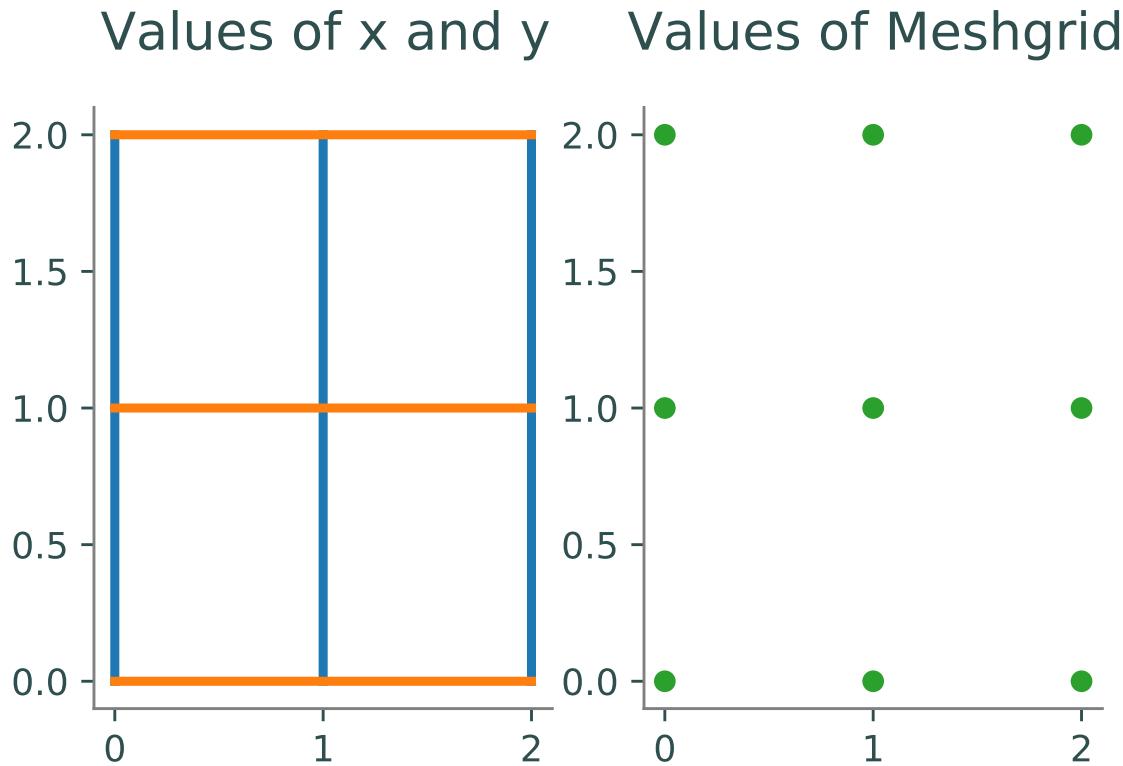


Figure 3.6: In the left plot, we have two arrays where  $x$  and  $y$  have the values  $x = y = [0, 1, 2]$ .

The command `np.meshgrid(x, y)` returns the arrays  $X = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$  and  $Y = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$ .

These give the  $x$ - and  $y$ -coordinates of the points in the grid formed by  $x$  and  $y$  as seen in the right plot and satisfy the relation  $(X[i, j], Y[i, j]) = (x[j], y[i])$ .

```
>>> x, y = [0, 1, 2], [3, 4, 5]      # A rough domain over [0,2]x[3,5].
>>> X, Y = np.meshgrid(x, y)        # Combine the 1-D data into 2-D data.
>>> for xrow, yrow in zip(X,Y):
...     print(xrow, yrow, sep='\t')
...
[0 1 2]    [3 3 3]
[0 1 2]    [4 4 4]
[0 1 2]    [5 5 5]
```

With a 2-dimensional domain,  $g(x, y)$  is usually visualized with two kinds of plots.

- A *heat map* assigns a color to each point in the domain, producing a 2-dimensional colored picture describing a 3-dimensional shape. Darker colors typically correspond to lower values while lighter colors typically correspond to higher values.

Use `plt.pcolormesh()` to create a heat map. You can add an optional argument for the shading type; this determines the layout and fill style of the heat map. This argument defaults to `shading='auto'`, and will automatically choose a fill method suited to the data being graphed.

- A *contour map* draws several *level curves* of  $g$  on the 2-dimensional domain. A level curve corresponding to the constant  $c$  is the collection of points  $\{(x, y) \mid c = g(x, y)\}$ . Coloring the space between the level curves produces a discretized version of a heat map. Including more and more level curves makes a filled contour plot look more and more like the complete, blended heat map.

Use `plt.contour()` to create a contour plot and `plt.contourf()` to create a filled contour plot. Specify either the number of level curves to draw, or a list of constants corresponding to specific level curves.

These functions each receive the keyword argument `cmap` to specify a color scheme (some of the better schemes are `"viridis"`, `"magma"`, and `"coolwarm"`). For the list of all Matplotlib color schemes, see [http://matplotlib.org/examples/color/colormaps\\_reference.html](http://matplotlib.org/examples/color/colormaps_reference.html).

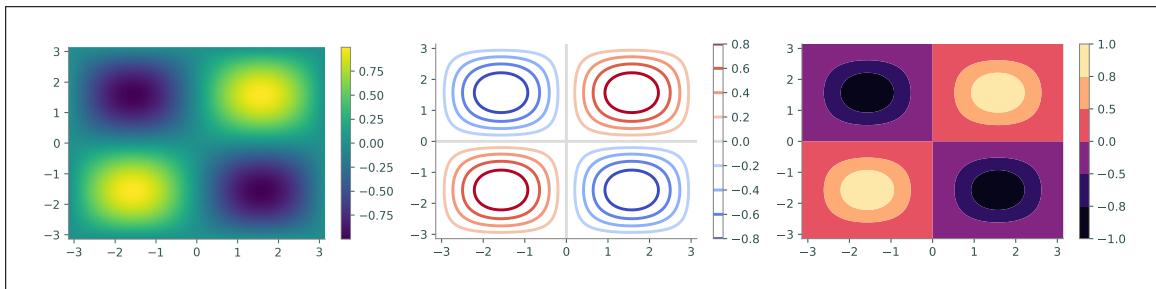
Finally, `plt.colorbar()` draws the color scale beside the plot to indicate how the colors relate to the values of the function.

```
# Create a 2-D domain with np.meshgrid().
>>> x = np.linspace(-np.pi, np.pi, 100)
>>> y = x.copy()
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)      # Calculate g(x,y) = sin(x)sin(y).

# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis", shading="auto")
>>> plt.colorbar()
>>> plt.xlim(-np.pi, np.pi)
>>> plt.ylim(-np.pi, np.pi)

# Plot a contour map of f with 10 level curves.
>>> plt.subplot(132)
>>> plt.contour(X, Y, Z, 10, cmap="coolwarm")
>>> plt.colorbar()

# Plot a filled contour map, specifying the level curves.
>>> plt.subplot(133)
>>> plt.contourf(X, Y, Z, [-1, -.8, -.5, 0, .5, .8, 1], cmap="magma")
>>> plt.colorbar()
>>> plt.show()
```



**Problem 6.** Write a function to plot  $g(x, y) = \frac{\sin(x)\sin(y)}{xy}$  on the domain  $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$ .

1. Create 2 subplots: one with a heat map of  $g$ , and one with a contour map of  $g$ . Choose an appropriate number of level curves, or specify the curves yourself.
2. Set the limits of each subplot to  $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$ .
3. Choose a non-default color scheme.
4. Include a color scale bar for each subplot.

## Additional Material

### Further Reading and Tutorials

Plotting takes some getting used to. See the following materials for more examples.

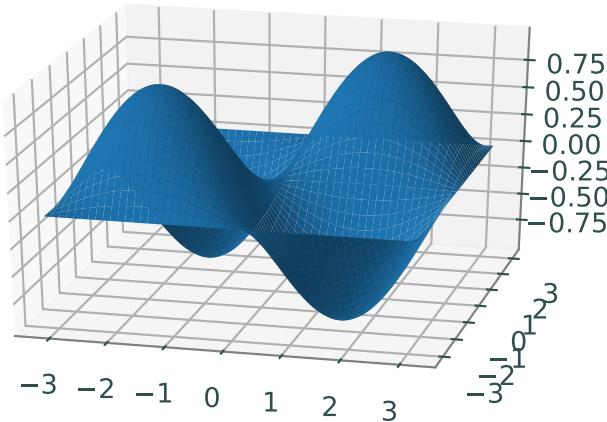
- <https://www.labri.fr/perso/nrougier/teaching/matplotlib/>.
- <https://matplotlib.org/stable/tutorials/introductory/pyplot.html>.
- <http://scipy-lectures.org/intro/matplotlib/>.
- The Matplotlib Appendix in this manual.

## 3-D Plotting

Matplotlib can also be used to plot 3-dimensional surfaces. The following code produces the surface corresponding to  $g(x, y) = \sin(x) \sin(y)$ .

```
# Create the domain and calculate the range like usual.
>>> x = np.linspace(-np.pi, np.pi, 200)
>>> y = np.copy(x)
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)

# Draw the corresponding 3-D plot using some extra tools.
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1, projection='3d')
>>> ax.plot_surface(X, Y, Z)
>>> plt.show()
```



## Animations

Lines and other graphs can be altered dynamically to produce animations. Follow these steps to create a Matplotlib animation:

1. Calculate all data that is needed for the animation.
2. Define a figure explicitly with `plt.figure()` and set its window boundaries.
3. Draw empty objects that can be altered dynamically.
4. Define a function to update the drawing objects.
5. Use `matplotlib.animation.FuncAnimation()`.

The submodule `matplotlib.animation` contains the tools for putting together and managing animations. The function `matplotlib.animation.FuncAnimation()` accepts the figure to animate, the function that updates the figure, the number of frames to show before repeating, and how fast to run the animation (lower numbers mean faster animations).

```
from matplotlib.animation import FuncAnimation

def sine_animation():
    # Calculate the data to be animated.
    x = np.linspace(0, 2*np.pi, 200)[-1]
    y = np.sin(x)

    # Create a figure and set the window boundaries of the axes.
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    # Draw an empty line. The comma after 'drawing' is crucial.
    drawing, = plt.plot([], [])

    # Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing,           # Note the comma!

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Try using the following function in place of `update()`. Can you explain why this animation is different from the original?

```
def wave(index):
    drawing.set_data(x, np.roll(y, index))
    return drawing,
```

To animate multiple objects at once, define the objects separately and make sure the update function returns both objects.

```

def sine_cosine_animation():
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y1, y2 = np.sin(x), np.cos(x)

    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    sin_drawing, = plt.plot([],[])
    cos_drawing, = plt.plot([],[])

    def update(index):
        sin_drawing.set_data(x[:index], y1[:index])
        cos_drawing.set_data(x[:index], y2[:index])
        return sin_drawing, cos_drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()

```

Animations can also be 3-dimensional. The only major difference is an extra operation to set the 3-dimensional component of the drawn object. The code below animates the space curve parametrized by the following equations:

$$x(\theta) = \cos(\theta) \cos(6\theta), \quad y(\theta) = \sin(\theta) \cos(6\theta), \quad z(\theta) = \frac{\theta}{10}$$

```

def rose_animation_3D():
    theta = np.linspace(0, 2*np.pi, 200)
    x = np.cos(theta) * np.cos(6*theta)
    y = np.sin(theta) * np.cos(6*theta)
    z = theta / 10

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')           # Make the figure 3-D.
    ax.set_xlim3d(-1.2, 1.2)                      # Use ax instead of plt.
    ax.set_ylim3d(-1.2, 1.2)
    ax.set_aspect("equal")

    drawing, = ax.plot([],[],[])                  # Provide 3 empty lists.

    # Update the first 2 dimensions like usual, then update the 3-D component.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        drawing.set_3d_properties(z[:index])
        return drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10, repeat=False)
    plt.show()

```



# 4

# Unit Testing

**Lab Objective:** *Finding and fixing programming errors can be difficult and time consuming, especially in large or complex programs. Unit testing is a formal strategy for finding and eliminating errors quickly as a program is constructed and for ensuring that the program still works whenever it is modified. A single unit test checks a small piece of code (usually a function or class method) for correctness, independent of the rest of the program. A well-written collection of unit tests can help make sure that every unit of code functions as intended, thereby decreasing the chances for errors in the program. In this lab we learn to write unit tests in Python and practice test-driven development. Applying these principles will greatly speed up the coding process and improve your code quality.*

## Unit Tests

A *unit test* verifies a piece of code by running a series of test cases and comparing actual outputs with expected outputs. Each test case is usually checked with an `assert` statement, a shortcut for raising an `AssertionError` with an optional error message if a boolean statement is false.

```
# Store the result of a boolean expression in a variable.  
>>> result = str(5)=='5'  
  
# Check the result, raising an error if it is false.  
>>> if result is False:  
...     raise AssertionError("incorrect result")  
  
# Do the same check in one line with an assert statement.  
>>> assert result, "incorrect result"  
  
# Asserting a false statement raises an AssertionError.  
>>> assert 5=='5', "5 is not a string"  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
AssertionError: 5 is not a string
```

Now suppose we wanted to test a simple `add()` function, located in the file `specs.py`.

```
# specs.py

def add(a, b):
    """Add two numbers."""
    return a + b
```

In a corresponding file called `test_specs.py`, which should contain all of the unit tests for the code in `specs.py`, we write a unit test called `test_add()` to verify the `add()` function.

```
# test_specs.py
import specs

def test_add():
    assert specs.add(1, 3) == 4, "failed on positive integers"
    assert specs.add(-5, -7) == -12, "failed on negative integers"
    assert specs.add(-6, 14) == 8
```

In this case, running `test_add()` raises no errors since all three test cases pass. Unit test functions don't need to return anything, but they should raise an exception if a test case fails.

#### NOTE

This style of external testing—checking that certain inputs result in certain outputs—is called *black box testing*. The actual structure of the code is not considered, but what it produces is thoroughly examined. In fact, the author of a black box test doesn't even need to be the person who eventually writes the program: having one person write tests and another write the code helps detect problems that one developer or the other may not have caught individually.

## PyTest

Python's `pytest` module<sup>1</sup> provides tools for building tests, running tests, and providing detailed information about the results. To begin, run `pytest` in the current directory. Without any test files, the output should be similar to the following.

```
$ pytest
===== test session starts =====
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
rootdir: /Users/Student, inifile:
collected 0 items

===== no tests ran in 0.02 seconds =====
```

Given some test files, say `test_calendar.py` and `test_google.py`, the output of `pytest` identifies failed tests and provides details on why they failed.

---

<sup>1</sup>Pytest is not part of the standard library. Install pytest with `pip install pytest` if needed. The standard library's `unittest` module also provides a testing framework, but is less popular and straightforward than PyTest.

```
$ pytest
=====
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
rootdir: /Users/Student/example_tests, inifile:
collected 12 items

test_calendar.py .....
test_google.py .F..

=====
 FAILURES =====
----- test_subtract -----
def test_subtract():
>     assert google.subtract(42, 17)==25, "subtract() failed for a > b > 0"
E     AssertionError: subtract() failed for a > b > 0
E     assert 35 == 25
E         +  where 35 = <function subtract at 0x102d4eb90>(42, 17)
E         +      where <function subtract at 0x102d4eb90> = google.subtract

test_google.py:11: AssertionError
=====
 1 failed, 11 passed in 0.02 seconds =====
```

Each dot represents a passed test and each F represents a failed test. They show up in order, so in the example above, only the second of four tests in `test_google.py` failed.

### ACHTUNG!

PyTest will not find or run tests if they are not contained in files named `test_*.py` or `*_test.py`, where \* represents any number of characters. In addition, the unit tests themselves must be named `test_*`() or `*_test()`. If you need to change this behavior, consult the documentation at <http://pytest.org/latest/example/pythoncollection.html>.

**Problem 1.** The following function contains a subtle but important error.

```
def smallest_factor(n):
    """Return the smallest prime factor of the positive integer n."""
    if n == 1: return 1
    for i in range(2, int(n**.5)):
        if n % i == 0: return i
    return n
```

Write a unit test for this function, including test cases that you suspect might uncover the error (what are the edge cases for this function?). Use `pytest` to run your unit test and discover a test case that fails, then use this information to correct the function.

## Coverage

Successful unit tests include enough test cases to test the entire program. *Coverage* refers to the number of lines of code that are executed by at least one test case. One tool for measuring coverage is called `pytest-cov`, an extension of `pytest`. This tool must be installed separately. To install, run the following code in a terminal.

```
$ pip install pytest-cov
```

Add the flag `--cov` to the `pytest` command to print out code coverage information. Running `pytest --cov` in the same directory as `specs.py` and `test_specs.py` yields the following output.

```
$ pytest --cov
===== test session starts =====
platform linux -- Python 3.10.6, pytest-7.4.0, py-1.11.0, pluggy-1.2.0
rootdir: /Users/Student/Testing, inifile:
plugins: cov-2.3.1
collected 7 items

test_specs.py ......

----- coverage: platform darwin, python 3.6.6-final-0 -----
Name          Stmts    Miss   Cover
-----
specs.py        73      38    48%
test_specs.py   42       0   100%
-----
TOTAL          115     38    67%

===== 7 passed in 0.03 seconds =====
```

Here, `Stmts` refers to the number of lines of code covered by a unit test, while `Miss` is the number of lines that are not currently covered. Notice that the file `test_specs.py` has 100% coverage while `specs.py` does not. Test files generally have 100% coverage, since `pytest` is designed to run these files in their entirety. However, `specs.py` does not have full coverage and requires additional unit tests. To find out which lines are not yet covered, `pytest-cov` has a useful feature called `cov-report` that creates an HTML file for visualizing the current line coverage.

```
$ pytest --cov-report html --cov
===== test session starts =====
# ...
----- coverage: platform linux, python 3.10.6-final-0 -----
Coverage HTML written to dir htmlcov
```

Instead of printing coverage statistics, this command creates various files with coverage details in a new directory called `htmlcov/`. The file `htmlcov/specs_py.html`, which can be viewed in an internet browser, highlights in red the lines of `specs.py` that are not yet covered by any unit tests.

### NOTE

Statement coverage is categorized as *white box testing* because it requires an understanding of the code's structure. While most black box tests can be written before a program is actually implemented, white box tests should be added to the collection of unit tests after the program is completed. By designing unit tests so that they cover every statement in a program, you may discover that some lines of code are unreachable, find that a conditional statement isn't functioning as intended, or uncover problems that accompany edge cases.

**Problem 2.** With `pytest-cov` installed, check your coverage of `smallest_factor()` from Problem 1. Write additional test cases if necessary to get complete coverage. Then, write a comprehensive unit test for the following (correctly written) function.

```
def month_length(month, leap_year=False):
    """Return the number of days in the given month."""
    if month in {"September", "April", "June", "November"}:
        return 30
    elif month in {"January", "March", "May", "July",
                   "August", "October", "December"}:
        return 31
    if month == "February":
        if not leap_year:
            return 28
        else:
            return 29
    else:
        return None
```

## Testing Exceptions

Many programs are designed to raise exceptions in response to bad input or an unexpected error. A good unit test makes sure that the program raises the exceptions that it is expected to raise, but also that it doesn't raise any unexpected exceptions. The `raises()` method in `pytest` is a clean, formal way of asserting that a program raises a desired exception. For example, the following code should raise a `ZeroDivisionError` if the divisor is 0.

```
# specs.py

def divide(a, b):
    """Divide two numbers, raising an error if the second number is zero."""
    if b == 0:
        raise ZeroDivisionError("second input cannot be zero")
    return a / b
```

The corresponding unit test checks that the function raises the `ZeroDivisionError` correctly.

```
# test_specs.py
import pytest

def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    pytest.raises(ZeroDivisionError, specs.divide, a=4, b=0)
```

If calling `divide(a=4, b=0)` results in a `ZeroDivisionError`, `pytest.raises()` catches the exception and the test case passes. On the other hand, if `divide(a=4, b=0)` does not raise a `ZeroDivisionError`, or if it raises a different kind of exception, the test fails.

To ensure that the `ZeroDivisionError` is coming from the written `raise` statement, combine `pytest.raises()` and the `with` statement to check the exception's error message.

```
def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    with pytest.raises(ZeroDivisionError) as excinfo:
        specs.divide(4, 0)
    assert excinfo.value.args[0] == "second input cannot be zero"
```

Here `excinfo` is an object containing information about the exception; the actual exception object is stored in `excinfo.value`, and hence `excinfo.value.args[0]` is the error message.

**Problem 3.** Write a comprehensive unit test for the following function. Make sure that each exception is raised properly by explicitly checking the exception message. Use `pytest-cov` and its `cov-report` tool to confirm that you have full coverage for this function.

```
def operate(a, b, oper):
    """Apply an arithmetic operation to a and b."""
    if type(oper) is not str:
        raise TypeError("oper must be a string")
    elif oper == '+':
        return a + b
    elif oper == '-':
        return a - b
    elif oper == '*':
        return a * b
    elif oper == '/':
        if b == 0:
            raise ZeroDivisionError("division by zero is undefined")
        return a / b
    raise ValueError("oper must be one of '+', '/', '-', or '*'")
```

## Fixtures

Consider the following class for representing rational numbers as reduced fractions.

```
class Fraction(object):
    """Reduced fraction class with integer numerator and denominator."""
    def __init__(self, numerator, denominator):
        if denominator == 0:
            raise ZeroDivisionError("denominator cannot be zero")
        elif type(numerator) is not int or type(denominator) is not int:
            raise TypeError("numerator and denominator must be integers")

        def gcd(a,b):
            while b != 0:
                a, b = b, a % b
            return a
        common_factor = gcd(numerator, denominator)
        self.numer = numerator // common_factor
        self.denom = denominator // common_factor

    def __str__(self):
        if self.denom != 1:
            return "{} / {}".format(self.numer, self.denom)
        else:
            return str(self.numer)

    def __float__(self):
        return self.numer / self.denom

    def __eq__(self, other):
        if type(other) is Fraction:
            return self.numer==other.numer and self.denom==other.denom
        else:
            return float(self) == other

    def __add__(self, other):
        return Fraction(self.numer*other.numer + self.denom*other.denom,
                       self.denom*other.denom)
    def __sub__(self, other):
        return Fraction(self.numer*other.numer - self.denom*other.denom,
                       self.denom*other.denom)
    def __mul__(self, other):
        return Fraction(self.numer*other.numer, self.denom*other.denom)

    def __truediv__(self, other):
        if self.denom*other.numer == 0:
            raise ZeroDivisionError("cannot divide by zero")
        return Fraction(self.numer*other.denom, self.denom*other.numer)
```

```
>>> from specs import Fraction
>>> print(Fraction(8, 12))                                # 8/12 reduces to 2/3.
2 / 3
>>> Fraction(1, 5) == Fraction(3, 15)                      # 3/15 reduces to 1/5.
True
>>> print(Fraction(1, 3) * Fraction(1, 4))
1 / 12
```

To test this class, it would be nice to have some ready-made `Fraction` objects to use in each unit test. A *fixture*, a function marked with the `@pytest.fixture` decorator, sets up variables that can be used as mock data for multiple unit tests. The individual unit tests take the fixture function in as input and unpack the constructed tests. Below, we define a fixture that instantiates three `Fraction` objects. The unit tests for the `Fraction` class use these objects as test cases.

```
@pytest.fixture
def set_up_fractions():
    frac_1_3 = specs.Fraction(1, 3)
    frac_1_2 = specs.Fraction(1, 2)
    frac_n2_3 = specs.Fraction(-2, 3)
    return frac_1_3, frac_1_2, frac_n2_3

def test_fraction_init(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_3.numer == 1
    assert frac_1_2.denom == 2
    assert frac_n2_3.numer == -2
    frac = specs.Fraction(30, 42)                         # 30/42 reduces to 5/7.
    assert frac.numer == 5
    assert frac.denom == 7

def test_fraction_str(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert str(frac_1_3) == "1 / 3"
    assert str(frac_1_2) == "1 / 2"
    assert str(frac_n2_3) == "-2 / 3"

def test_fraction_float(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert float(frac_1_3) == 1 / 3.
    assert float(frac_1_2) == .5
    assert float(frac_n2_3) == -2 / 3.

def test_fraction_eq(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_2 == specs.Fraction(1, 2)
    assert frac_1_3 == specs.Fraction(2, 6)
    assert frac_n2_3 == specs.Fraction(8, -12)
```

**Problem 4.** Add test cases to the unit tests provided above to get full coverage for the `__init__()`, `__str__()`, `__float__()`, and `__eq__()` methods. You may modify the fixture function if it helps. Also add unit tests for the magic methods `__add__()`, `__sub__()`, `__mul__()`, and `__truediv__()`. Verify that you have full coverage with `pytest-cov`.

Additionally, **two** of the `Fraction` class's methods are implemented incorrectly. Use your tests to find the issues, then correct the methods so that your tests pass.

See <http://doc.pytest.org/en/latest/index.html> for complete documentation on `pytest`.

## Test-driven Development

*Test-driven development* (TDD) is the programming style of writing tests **before** implementing the actual code. It may sound tedious at first, but TDD incentivizes simple design and implementation, speeds up the actual coding, and gives quantifiable checkpoints for the development process. TDD can be summarized in the following steps:

1. Define with great detail the program specifications. Write function declarations, class definitions, and especially docstrings, determining exactly what each function or class method should accept and return.
2. Write a unit test for each unit of the program, usually black box tests.
3. Implement the program code, making changes until all tests pass.

For adding new features or cleaning existing code, the process is similar.

1. Redefine program specifications to account for planned modifications.
2. Add or modify tests to match the new specifications.
3. Change the code until all tests pass.

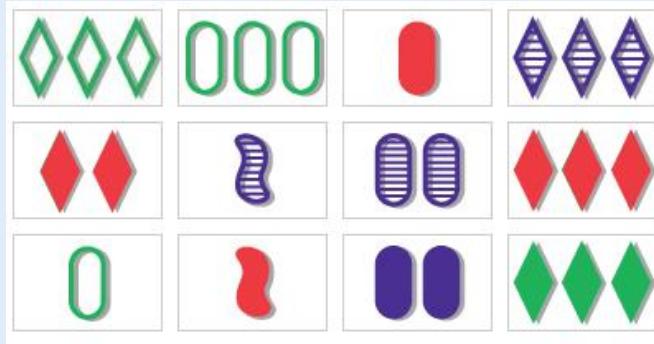


If the test cases are sufficiently thorough, then when the tests all pass, the program can be considered complete. Remember, however, that it is not sufficient to just have tests, but to have tests that accurately and rigorously test the code. To check that the test cases are sufficient, examine the test coverage and add additional tests if necessary.

See [https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development) for more discussion on TDD and [https://en.wikipedia.org/wiki/Behavior-driven\\_development](https://en.wikipedia.org/wiki/Behavior-driven_development) for an overview of Behavior-driven development (BDD), a close relative of TDD.

**Problem 5.** *Set* is a card game about finding patterns. Each card contains a design with 4 different properties: color (red, green or purple), shape (diamond, oval or squiggly), quantity (one, two, or three) and pattern (solid, striped or outlined). A *set* is a group of three cards which are either all the same or all different for each property. You can try playing Set online at <http://smart-games.org/en/set/start>.

Here is a group of twelve Set cards.

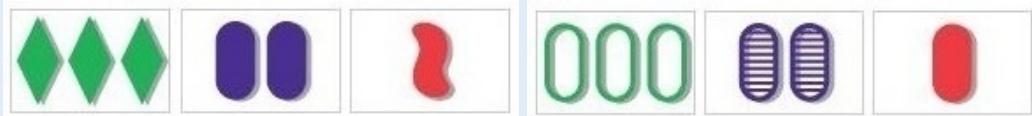


This collection of cards contains six unique sets:



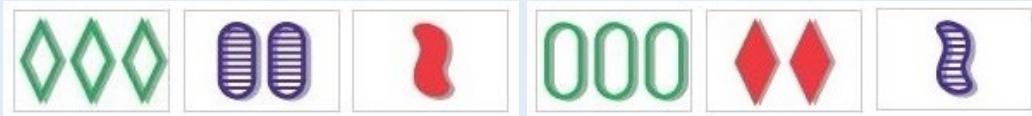
(a) Same in quantity and shape; different in pattern and color

(b) Same in color and pattern; different in shape and quantity



(c) Same in pattern; different in shape, quantity and color

(d) Same in shape; different in quantity, pattern and color



(e) Different in all aspects

(f) Different in all aspects

Each Set card can be uniquely represented by a 4-bit integer in base 3,<sup>a</sup> where each digit represents a different property and each property has three possible values. A full hand in Set is a group of twelve unique cards, so a hand can be represented by a list of twelve 4-digit integers in base 3. For example, the hand shown above could be represented by the following list.

```
hand1 = ["1022", "1122", "0100", "2021",
         "0010", "2201", "2111", "0020",
         "1102", "0200", "2110", "1020"]
```

The following function definitions provide a framework for partially implementing Set by calculating the number of sets in a given hand.

```

def count_sets(cards):
    """Return the number of sets in the provided Set hand.

    Parameters:
        cards (list(str)) a list of twelve cards as 4-bit integers in
            base 3 as strings, such as ["1022", "1122", ..., "1020"].

    Returns:
        (int) The number of sets in the hand.

    Raises:
        ValueError: if the list does not contain a valid Set hand, meaning
            - there are not exactly 12 cards,
            - the cards are not all unique,
            - one or more cards does not have exactly 4 digits, or
            - one or more cards has a character other than 0, 1, or 2.
    """
    pass

def is_set(a, b, c):
    """Determine if the cards a, b, and c constitute a set.

    Parameters:
        a, b, c (str): string representations of 4-bit integers in base 3.
            For example, "1022", "1122", and "1020" (which is not a set).

    Returns:
        True if a, b, and c form a set, meaning the ith digit of a, b,
        and c are either the same or all different for i=1,2,3,4.
        False if a, b, and c do not form a set.
    """
    pass

```

Write unit tests for these functions, but **do not** implement them yet. Focus on *what* the functions should do rather than on *how* they will be implemented.

(Hint: if three cards form a set, then the first digits of the cards are either all the same or all different. Then the sums of these digits can only be 0, 3, or 6. Thus, a group of cards forms a set only if for each set of digits—first digits, second digits, etc.—the sum is a multiple of 3.)

---

<sup>a</sup>A 4-bit integer in base 3 contains four digits that are either 0, 1 or 2. For example, 0000 and 1201 are 4-bit integers in base 3, whereas 000 is not because it has only three digits, and 0123 is not because it contains the number 3.

**Problem 6.** After you have written unit tests for the functions in Problem 5, implement the actual functions. If needed, add additional test cases to get full coverage.

(Hint: The `combinations()` function from the standard library module `itertools` may be useful in implementing `count_sets()`.)

## Additional Material

### The Python Debugger

Python has a built in debugger called `pdb` to aid in finding mistakes in code during execution. The debugger can be run either in a terminal or in a Jupyter Notebook.

A *break point*, set with `pdb.set_trace()`, is a spot where the program pauses execution. Once the program is paused, use the following commands to tell the program what to do next.

Command	Description
<code>n</code>	next: executes the next line
<code>p &lt;var&gt;</code>	print: display the value of the specified variable.
<code>c</code>	continue: stop debugging and run the program normally to the end.
<code>q</code>	quit: terminate the program.
<code>l</code>	list: show several lines of code around the current line.
<code>r</code>	return: return to the end of a subroutine.
<code>&lt;Enter&gt;</code>	Execute the most recent command again.

For example, suppose we have a long loop where the value of a variable changes unpredictably.

```
# pdb_example.py
import pdb
from random import randint

i = 0
pdb.set_trace()                                     # Set a break point.
while i < 1000000000:
    i += randint(1, 10)
print("DONE")
```

Run the file in the terminal to begin a debugging session.

```
$ python pdb_example.py
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) 1                                         # Show where we are.
2     import pdb
3     from random import randint
4
5     i = 0
6     pdb.set_trace()
7 -> while i < 1000000000:
8         i += randint(1, 10)
9     print("DONE")
[EOF]
```

We can check the value of the variable `i` at any step with `p i`, and we can even change the value of `i` mid-program.

```
(Pdb) n                                         # Execute a few lines.
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) n
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 10000000000:
(Pdb) n
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) p i                                     # Check the value of i.
8
(Pdb) n                                         # Execute another line.
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 10000000000:
(Pdb) p i                                     # Check i again.
14
(Pdb) i = 9999999999                         # Change the value of i.
(Pdb) c                                         # Continue the program.
DONE
```

See <https://docs.python.org/3/library/pdb.html> for documentation and examples for the Python debugger.

## Other Testing Suites

There are several frameworks other than `pytest` for writing unit tests. Each shares the same basic structure, but the setup, syntax, and particular features vary. For more unit testing practice, try out the standard library's `unittest` (<https://docs.python.org/3/library/unittest.html>) or `doctest` (<https://docs.python.org/3/library/doctest.html>), or the third-party `nose` module (<https://nose.readthedocs.io/en/latest/>). For a much larger list of unit testing tools, see <https://wiki.python.org/moin/PythonTestingToolsTaxonomy>.

## The Fractions Module

The standard library's `fractions` module (<https://docs.python.org/3/library/fractions.html>) has a `Fraction` class that is similar to the `Fraction` class presented in this lab. Its structure and syntax is a little different from this lab's class, but it is a little more robust in that it can take in floats, decimals, integers, and strings to its constructor. See also the `decimals` module (<https://docs.python.org/3/library/decimal.html>) for tools relating to decimal arithmetic.



# 5

# Linked Lists

**Lab Objective:** *One of the fundamental problems in programming is knowing which data structures to use to optimize code. The type of data structure used determines how quickly data is accessed and modified, which affects the overall speed of a program. In this lab we introduce a basic data structure called a linked list and create a class to implement it.*

A *linked list* is a data structure that chains data together. Every linked list needs a reference to the first item in the chain, called the **head**. A reference to the last item in the chain, called the **tail**, is also often included. Each item in the list stores a piece of data, plus at least one reference to another item in the list. The items in the list are called **nodes**.

## Nodes

Think of data as several types of objects that need to be stored in a warehouse. A *node* is like a standard size box that can hold all the different types of objects. For example, suppose a particular warehouse stores lamps of various sizes. Rather than trying to carefully stack lamps of different shapes on top of each other, it is preferable to first put them in boxes of standard size. Then adding new boxes and retrieving stored ones becomes much easier. A *data structure* is like the warehouse, which specifies where and how the different boxes are stored.

A node class is usually simple. The data in the node is stored as an attribute. Other attributes may be added (or inherited) specific to a particular data structure.

**Problem 1.** Consider the following generic node class.

```
class Node:  
    """A basic node class for storing data."""  
    def __init__(self, data):  
        """Store the data in the value attribute."""  
        self.value = data
```

Modify the constructor so that it only accepts data of type `int`, `float`, or `str`. If another type of data is given, raise a `TypeError` with an appropriate error message. Modify the constructor docstring to document these restrictions.

The nodes of a *singly linked list* have a single reference to the next node in the list (see Figure 5.1), while the nodes of a *doubly linked list* have two references: one for the previous node, and one for the next node (see Figure 5.2). This allows for a doubly linked list to be traversed in both directions, whereas a singly linked list can only be traversed in one direction.

```
class LinkedListNode(Node):
    """A node class for doubly linked lists. Inherits from the Node class.
    Contains references to the next and previous nodes in the linked list.
    """
    def __init__(self, data):
        """Store the data in the value attribute and initialize
        attributes for the next and previous nodes in the list.
        """
        Node.__init__(self, data) # Use inheritance to set self.value.
        self.next = None          # Reference to the next node.
        self.prev = None          # Reference to the previous node.
```

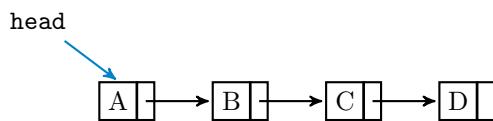


Figure 5.1: A singly linked list. Each node has a reference to the next node in the list. The `head` attribute is always assigned to the first node.

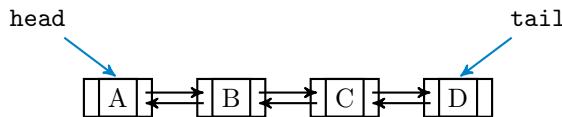


Figure 5.2: A doubly linked list. Each node has a reference to the node before it and a reference to the node after it. In addition to the `head` attribute, this list has a `tail` attribute that is always assigned to the last node.

The following `LinkedList` class chains `LinkedListNode` instances together by modifying each node's `next` and `prev` attributes. The list is empty initially, so the `head` and `tail` attributes are assigned the placeholder value `None` in the constructor. The `append()` method makes a new node and adds it to the very end of the list (see Figure 5.3). There are two cases for appending that must be considered separately in the implementation: either the list is empty, or the list is nonempty.

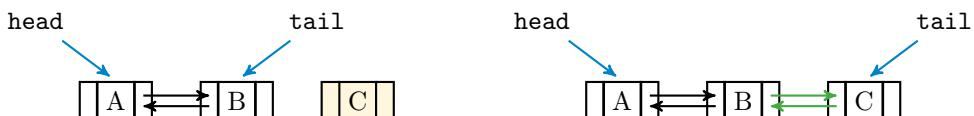


Figure 5.3: Appending a new node to the end of a nonempty doubly linked list. The green arrows are the new connections. Note that the `tail` attribute is reassigned from `B` to `C`.

```

class LinkedList:
    """Doubly linked list data structure class.

    Attributes:
        head (LinkedListNode): the first node in the list.
        tail (LinkedListNode): the last node in the list.
    """

    def __init__(self):
        """Initialize the head and tail attributes by setting them to None, since the list is empty initially."""
        self.head = None
        self.tail = None

    def append(self, data):
        """Append a new node containing the data to the end of the list."""
        # Create a new node to store the input data.
        new_node = LinkedListNode(data)
        if self.head is None:
            # If the list is empty, assign the head and tail attributes to new_node, since it becomes the first and last node in the list.
            self.head = new_node
            self.tail = new_node
        else:
            # If the list is not empty, place new_node after the tail.
            self.tail.next = new_node           # tail --> new_node
            new_node.prev = self.tail          # tail <-- new_node
            # Now the last node in the list is new_node, so reassign the tail.
            self.tail = new_node

```

## ACHTUNG!

The `is` operator is **not** the same as the `==` operator. While `==` checks for numerical equality, `is` evaluates whether or not two objects are the same by checking their location in memory.

```

>>> 7 == 7.0          # True since the numerical values are the same.
True

# 7 is an int and 7.0 is a float, so they cannot be stored at the same
# location in memory. Therefore 7 "is not" 7.0.
>>> 7 is 7.0
False

```

For numerical comparisons, always use `==`. When comparing to built-in Python constants such as `None`, `True`, `False`, or `NotImplemented`, use `is` instead.

## Locating Nodes

The `LinkedList` class only explicitly keeps track of the first and last nodes in the list via the `head` and `tail` attributes. To access any other node, use each successive node's `next` and `prev` attributes.

```
>>> my_list = LinkedList()
>>> for data in (2, 4, 6):
...     my_list.append(data)
...
# To access each value, use the head attribute of the LinkedList
# and the next and value attributes of each node in the list.
>>> my_list.head.value
2
>>> my_list.head.next.value          # 2 --> 4
4
>>> my_list.head.next.next is my_list.tail    # 2 --> 4 --> 6
True
```

**Problem 2.** Add the following methods to the `LinkedList` class.

1. `find()`: Accept a piece of data and return the first node in the list containing that data (return the actual `LinkedListNode` object, not its `value`). If no such node exists, or if the list is empty, raise a `ValueError` with an appropriate error message.  
(Hint: if `n` is assigned to one of the nodes the list, what does `n = n.next` do?)
2. `get()`: Accept an integer  $i$  and return the  $i$ th node in the list. If  $i$  is negative or greater than or equal to the number of nodes in the list, raise an `IndexError`.  
(Hint: add an attribute that tracks the current size of the list. Update it every time a node is successfully added or removed, such as at the end of the `append()` method.)

## Magic Methods

Endowing data structures with magic methods makes them much more intuitive to use. Consider, for example, how a Python list responds to built-in functions like `len()` and `print()`. At the bare minimum, the `LinkedList` class should have the same functionality.

**Problem 3.** Add the following magic methods to the `LinkedList` class.

1. Write the `__len__()` method so that the length of a `LinkedList` instance is equal to the number of nodes in the list.
2. Write the `__str__()` method so that when a `LinkedList` instance is printed, its output matches that of a Python list. Entries are separated by a comma and one space; strings are surrounded by single quotes, or by double quotes if the string itself has a single quote.  
(Hint: use `repr()` to deal with quotes easily.)

## UNIT TEST

Write unit tests for Problem 3 in `test_linked_lists.py`. Make sure you write tests for both the `__len__()` and `__str__()` functions. These tests will be graded. There are example unit tests for Problem 2 to help guide you.

## Removal

To delete a node, all references to the node must be removed. Python automatically deletes the object once there is no way for the user to access it. Naïvely, this might be done by finding the previous node to the one being removed, and setting its `next` attribute to `None`. However, there is a problem with this approach.

```
class LinkedList:
    ...
    def remove(self, data):
        """Attempt to remove the first node containing the specified data.
        This method incorrectly removes additional nodes.
        """
        # Find the target node and sever the links pointing to it.
        target = self.find(data)
        target.prev.next = None          # -/-> target
        target.next.prev = None          # target <-/-
```

Removing all references to the target node deletes the node (see Figure 5.4). Unfortunately, the nodes before and after the target node are no longer linked.

```
>>> my_list = LinkedList()
>>> for i in range(10):
...     my_list.append(i)
...
>>> print(my_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> my_list.remove(4)           # Removing a node improperly results in
>>> print(my_list)            # the rest of the chain being lost.
[0, 1, 2, 3]                  # Should be [0, 1, 2, 3, 5, 6, 7, 8, 9].
```

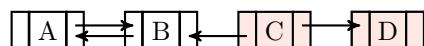


Figure 5.4: Naïve removal for doubly linked Lists. Deleting all references pointing to C deletes the node, but it also separates nodes A and B from node D.

This can be remedied by pointing the previous node's `next` attribute to the node after the deleted node, and similarly changing that node's `prev` attribute. Then there will be no reference to the removed node and it will be deleted, but the chain will still be connected.



Figure 5.5: Correct removal for doubly linked Lists. To avoid gaps in the chain, nodes B and D must be linked together.

**Problem 4.** Modify the `remove()` method given above so that it correctly removes the first node in the list containing the specified data. Also account for the special cases of removing the first, last, or only node, in which `head` and/or `tail` must be reassigned. Raise a `ValueError` if there is no node in the list that contains the data.

(Hint: use the `find()` method from Problem 2 to locate the target node.)

### ACHTUNG!

Python keeps track of the variables in use and automatically deletes a variable (freeing up the memory that stored the object) if there is no access to it. This feature is called *garbage collection*. In many other languages, leaving a reference to an object without explicitly deleting it can lead to a serious memory leak. See <https://docs.python.org/3/library/gc.html> for more information on Python's garbage collection system.

## Insertion

The `append()` method can add new nodes to the end of the list, but not to the middle. To do this, get references to the nodes before and after where the new node should be, then adjust their `next` and `prev` attributes. Be careful not to disconnect the nodes in a way that accidentally deletes nodes like in Figure 5.4.

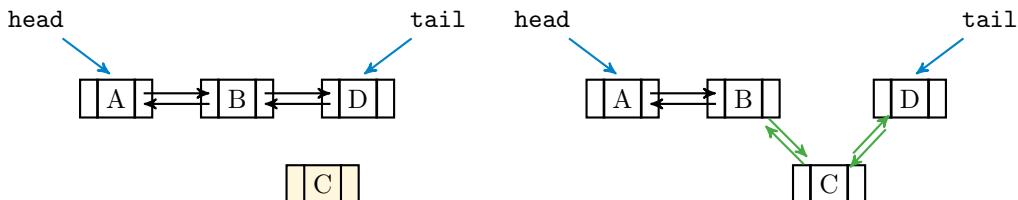


Figure 5.6: Insertion for doubly linked lists.

**Problem 5.** Add an `insert()` method to the `LinkedList` class that accepts an integer `index` and data to add to the list. Insert a new node containing the data immediately **before** the node in the list currently at position `index`. After the insertion, the new node should be at position `index`. For example, Figure 5.6 places a new node containing C at index 2. Carefully account for the special case of inserting before the first node, which requires `head` to be reassigned. (Hint: except when inserting before the head, get references to the nodes that should be immediately before and after the new node following the insertion. Consider using the `get()` method from Problem 2 to locate one of these nodes.)

If `index` is equal to the number of nodes in the list, append the node to the end of the list by calling `append()`. If `index` is negative or strictly greater than the number of nodes in the list, raise an `IndexError`.

#### NOTE

The temporal complexity for inserting to the beginning or end of a linked list is  $O(1)$ , but inserting anywhere else is  $O(n)$ , where  $n$  is the number of nodes in the list. This is quite slow compared other data structures. In the next lab we turn our attention to *trees*, special kinds of linked lists that allow for much quicker sorting and data retrieval.

## Restricted-Access Lists

It is sometimes wise to restrict the user's access to some of the data within a structure. In particular, because insertion, removal, and lookup are  $O(n)$  for data in the middle of a linked list, cutting off access to the middle of the list forces the user to only use  $O(1)$  operations at the front and end of the list. The three most common and basic restricted-access structures that implement this idea are *stacks*, *queues*, and *deques*. Each structure restricts the user's access differently, making them ideal for different situations.

- **Stack:** *Last In, First Out* (LIFO). Only the last item that was inserted can be accessed. A stack is like a pile of plates: the last plate put on the pile is (or should be) the first one to be taken off. Stacks usually have two main methods: `push()`, to insert new data, and `pop()`, to remove and return the last piece of data inserted.
- **Queue** (pronounced “cue”): *First In, First Out* (FIFO). New nodes are added to the end of the queue, but an existing node can only be removed or accessed if it is at the front of the queue. A queue is like a polite line at the bank: the person at the front of the line is served first, while newcomers add themselves to the back of the line. Queues also usually have a `push()` and a `pop()` method, but `push()` inserts data to the end of the queue while `pop()` removes and returns the data at the front of the queue. The `push()` and `pop()` operations are sometimes called `enqueue()` and `dequeue()`, respectively.
- **Deque** (pronounced “deck”): a double-ended queue. Data can be inserted or removed from either end, but data in the middle is inaccessible. A deque is like a deck of cards, where only the top and bottom cards are readily accessible. A deque has two methods for insertion and two for removal, usually called `append()`, `appendleft()`, `pop()`, and `popleft()`.

A deque can act as a queue by using only `append()` and `popleft()` (or `appendleft()` and `pop()`), or as a stack by using only `append()` and `pop()` (or `appendleft()` and `popleft()`).

**Problem 6.** Write a `Deque` class that inherits from `LinkedList`.

1. Write the following methods. Since they all involve data at the endpoints, avoid iterating through the list so the resulting operations are  $O(1)$ .

- `pop()`: Remove the last node in the list and return its data. Account for the special case of removing the only node in the list. Raise a `ValueError` if the list is empty.
- `popleft()`: Remove the first node in the list and return its data. Raise a `ValueError` if the list is empty.  
(Hint: use inheritance and the `remove()` method of `LinkedList`.)
- `appendleft()`: Insert a new node at the beginning of the list.  
(Hint: use inheritance and the `insert()` method of `LinkedList`.)

Note that the `LinkedList` class already implements `append()`.

2. Override the `remove()` method with the following code.

```
def remove(*args, **kwargs):
    raise NotImplementedError("Use pop() or popleft() for removal")
```

This effectively disables `remove()` for the `Deque` class, preventing the user from removing a node from the middle of the list.

3. Disable `insert()` as well.

**NOTE**

The `*args` argument allows the `remove()` method to receive any number of positional arguments without raising a `TypeError`, and the `**kwargs` argument allows it to receive any number of keyword arguments. This is the most general form of a function signature.

Python lists have `append()` and `pop()` methods, so they can be used as stacks. However, data access and removal from the front is much slower than from the end, as Python lists are implemented as dynamic arrays and not linked lists.

The `collections` module in the standard library has a `deque` object that is implemented as a doubly linked list. This is an excellent object to use in practice instead of a Python list when speed is of the essence and data only needs to be accessed from the ends of the list. Both lists and deques are slow to modify elements in the middle, but lists can access middle elements quickly. Table 5.1 describes the complexity for common operations on lists v. deques in Python.

Operation	List Complexity	Deque Complexity
Append/Remove from the end	$O(1)$	$O(1)$
Append/Remove from the start	$O(n)$	$O(1)$
Insert/Delete in the middle	$O(n)$	$O(n)$
Access element at the start/end	$O(1)$	$O(1)$
Access element in the middle	$O(1)$	$O(n)$

Table 5.1: Complexity of operations on lists and deques.

**Problem 7.** Write a function that accepts the name of a file to be read and a file to write to. Read the first file, adding each line of text to a stack. After reading the entire file, pop each entry off of the stack one at a time, writing the result to the second file.

For example, if the file to be read has the following list of words on the left, the resulting file should have the list of words on the right.

My homework is too hard for me.	I am a mathematician.
I do not believe that	Programming is hard, but
I can solve these problems.	I can solve these problems.
Programming is hard, but	I do not believe that
I am a mathematician.	My homework is too hard for me.

You may use a Python list, your `Deque` class, or `collections.deque` for the stack. Test your function on the file `english.txt`, which contains a list of over 58,000 English words in alphabetical order.

## Additional Material

### Possible Improvements to the `LinkedList` Class

The following are some ideas for expanding the functionality of the `LinkedList` class.

1. Add a keyword argument to the constructor so that if an iterable is provided, each element of the iterable is immediately added to the list. This makes it possible to cast an iterable as a `LinkedList` the same way that an iterable can be cast as one of Python's standard data structures.

```
>>> my_list = [1, 2, 3, 4, 5]
>>> my_linked_list = LinkedList(my_list) # Cast my_list as a LinkedList.
>>> print(my_linked_list)
[1, 2, 3, 4, 5]
>>> type(my_linked_list)
LinkedList
```

2. Add the following methods.

- `count()`: return the number of occurrences of a specified value.
- `reverse()`: reverse the ordering of the nodes (in place).
- `roll()`: shift the nodes a given number of steps to the right or left (in place).
- `sort()`: sort the nodes by their data (in place).

3. Implement more magic methods.

- `__add__()`: concatenate two lists.
- `__getitem__()` and `__setitem__()`: enable standard bracket indexing. Try to allow for negative indexing as well.
- `__iter__()`: support `for` loop iteration, the `iter()` built-in function, and the `in` statement.

## Other Kinds of Linked Lists

The `LinkedList` class can also be used as the backbone for more specialized data structures.

1. A *sorted list* adds new nodes strategically so that the data is always kept in order. Therefore, a `SortedLinkedList` class should have an `add()` method that receives some `data` and inserts a new node containing `data` before the first node in the list that has a `value` that is greater or equal to `data` (thereby preserving the ordering). Other methods for adding nodes should be disabled. Note however, that a linked list is **not** an ideal implementation for a sorted list because each insertion is  $O(n)$  (try sorting `english.txt`).
2. In a *circular linked list*, the “last” node connects back to the “first” node. Thus a reference to the tail is unnecessary. The `roll()` method mentioned above is used often so the `head` attribute is at an “active” part of the list where nodes are inserted, removed, or accessed often. This data structure can therefore decrease the average insertion or removal time for certain data sets.

# 6

# Binary Search Trees

**Lab Objective:** A tree is link-based data structure where each node may refer to more than one other node. This structure makes trees more useful and efficient than regular linked lists in many applications. Many trees are constructed recursively, so we begin with an overview of recursion. We then implement a recursively structured doubly linked binary search tree (BST). Finally, we compare the standard linked list, our BST, and an AVL tree to illustrate the relative strengths and weaknesses of each data structure.

## Recursion

A recursive function is one that calls itself. When the function is executed, it continues calling itself until reaching a *base case* where the value of the function is known. The function then exits without calling itself again, and each previous function call is resolved. The idea is to solve large problems by first solving smaller problems, then combining their results.

As a simple example, consider the function  $f : \mathbb{N} \rightarrow \mathbb{N}$  that sums all positive integers from 1 to some integer  $n$ .

$$f(n) = \sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i = n + f(n-1)$$

Since  $f(n-1)$  appears in the formula for  $f(n)$ ,  $f$  can be implemented recursively. Calculating  $f(n)$  requires the value of  $f(n-1)$ , which requires  $f(n-2)$ , and so on. The base case is  $f(1) = 1$ , at which point the recursion halts and unwinds. For example,  $f(4)$  is calculated as follows.

$$\begin{aligned} f(4) &= 4 + f(3) \\ &= 4 + (3 + f(2)) \\ &= 4 + (3 + (2 + f(1))) \\ &= 4 + (3 + (2 + 1)) \\ &= 4 + (3 + 3) \\ &= 4 + 6 \\ &= 10 \end{aligned}$$

The implementation accounts separately for the base case and the recursive case.

```
def recursive_sum(n):
    """Calculate the sum of all positive integers in [1, n] recursively."""
    if n <= 1:           # Base case: f(1) = 1.
        return 1
    else:                 # Recursive case: f(n) = n + f(n-1).
        return n + recursive_sum(n-1)
```

Many problems that can be solved iteratively can also be solved with a recursive approach. Consider the function  $g : \mathbb{N} \rightarrow \mathbb{N}$  that calculates the  $n$ th Fibonacci number.

$$g(n) = g(n - 1) + g(n - 2), \quad g(0) = 0, \quad g(1) = 1.$$

This function is doubly recursive since  $g(n)$  calls itself twice, and there are two different base cases to deal with. On the other hand,  $g(n)$  could be computed iteratively by calculating  $g(0), g(1), \dots, g(n)$  in that order. Compare the iterative and recursive implementations for  $g$  given below.

```
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    if n <= 0:           # Special case: g(0) = 0.
        return 0
    g0, g1 = 0, 1          # Initialize g(0) and g(1).
    for i in range(1, n):
        g0, g1 = g1, g0 + g1
    return g1

def recursive_fib(n):
    """Calculate the nth Fibonacci number recursively."""
    if n <= 0:           # Base case 1: g(0) = 0.
        return 0
    elif n == 1:           # Base case 2: g(1) = 1.
        return 1
    else:                  # Recursive case: g(n) = g(n-1) + g(n-2).
        return recursive_fib(n-1) + recursive_fib(n-2)
```

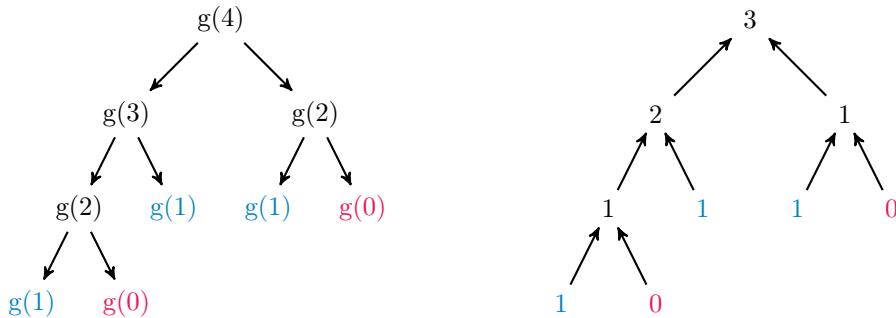


Figure 6.1: To calculate  $g(n)$  recursively, call  $g(n - 1)$  and  $g(n - 2)$ , down to the base cases  $g(0)$  and  $g(1)$ . As the recursion unwinds, the values from the base cases are passed up to previous calls and combined, eventually giving the value for  $g(n)$ .

**Problem 1.** Consider the following class for singly linked lists.

```
class SinglyLinkedListNode:
    """A node with a value and a reference to the next node."""
    def __init__(self, data):
        self.value, self.next = data, None

class SinglyLinkedList:
    """A singly linked list with a head and a tail."""
    def __init__(self):
        self.head, self.tail = None, None

    def append(self, data):
        """Add a node containing the data to the end of the list."""
        n = SinglyLinkedListNode(data)
        if self.head is None:
            self.head, self.tail = n, n
        else:
            self.tail.next = n
            self.tail = n

    def iterative_find(self, data):
        """Search iteratively for a node containing the data."""
        current = self.head
        while current is not None:
            if current.value == data:
                return current
            current = current.next
        raise ValueError(str(data) + " is not in the list")
```

Write a method that does the same task as `iterative_find()`, but with the following recursive approach. Define a function within the method that checks a single node for the data. There are two base cases: if the node is `None`, meaning the data could not be found, raise a `ValueError`; if the node contains the data, return the node. Otherwise, call the function on the next node in the list. Start the recursion by calling this inner function on the head node.

(Hint: see `BST.find()` in the next section for a similar idea.)

### ACHTUNG!

It is usually **not** better to rewrite an iterative method recursively, partly because recursion results in an increased number of function calls. Each call requires a small amount of memory so the program remembers where to return to in the program. By default, Python raises a `RuntimeError` after 1000 calls to prevent a stack overflow. On the other hand, recursion lends itself well to some problems; in this lab, we use a recursive approach to construct a few data structures, but it is possible to implement the same structures with iterative strategies.

## Binary Search Trees

Mathematically, a *tree* is a directed graph with no cycles. Trees can be implemented with link-based data structures that are similar to a linked list. The first node in a tree is called the *root*, like the *head* of a linked list. The root node points to other nodes, which are called its children. A node with no children is called a *leaf node*.

A *binary search tree* (BST) is a tree that allows each node to have up to two children, usually called *left* and *right*. The left child of a node contains a value that is less than its parent node's value; the right child's value is greater than its parent's value. This specific structure makes it easy to search a BST: while the computational complexity of finding a value in a linked list is  $O(n)$  where  $n$  is the number of nodes, a well-built tree finds values in  $O(\log n)$  time.

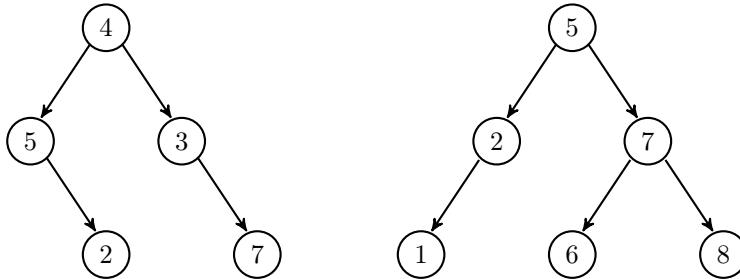


Figure 6.2: Both of these graphs are trees, but the tree on the left is not a binary search tree because 5 is to the left of 4. Swapping 5 and 3 in the graph on the left would result in a BST.

Binary search tree nodes have attributes that keep track of their value, their children, and (in doubly linked trees) their parent. The actual binary search tree has an attribute to keep track of its root node.

```

class BSTNode:
    """A node class for binary search trees. Contains a value, a
    reference to the parent node, and references to two child nodes.
    """
    def __init__(self, data):
        """Construct a new node and set the value attribute. The other
        attributes will be set when the node is added to a tree.
        """
        self.value = data
        self.prev = None      # A reference to this node's parent node.
        self.left = None      # self.left.value < self.value
        self.right = None     # self.value < self.right.value

class BST:
    """Binary search tree data structure class.
    The root attribute references the first node in the tree.
    """
    def __init__(self):
        """Initialize the root attribute."""
        self.root = None

```

## NOTE

Conceptually, each node of a BST partitions the data of its subtree into two halves: the data that is less than the parent, and the data that is greater. We will extend this concept to higher dimensions in the next lab.

## Locating Nodes

Finding a node in a binary search tree can be done recursively. Starting at the root, check if the target data matches the current node. If it does not, then if the data is less than the current node's value, search again on the left child; if the data is greater, search on the right child. Continue the process until the data is found or until hitting a dead end. This method illustrates the advantage of the binary structure—if a value is in a tree, then we know where it ought to be based on the other values in the tree.

```
class BST:
    # ...
    def find(self, data):
        """Return the node containing the data. If there is no such node
        in the tree, including if the tree is empty, raise a ValueError.
        """
        # Define a recursive function to traverse the tree.
        def _step(current):
            """Recursively step through the tree until the node containing
            the data is found. If there is no such node, raise a Value Error.
            """
            if current is None:                      # Base case 1: dead end.
                raise ValueError(str(data) + " is not in the tree.")
            if data == current.value:                 # Base case 2: data found!
                return current
            if data < current.value:                  # Recursively search left.
                return _step(current.left)
            else:                                    # Recursively search right.
                return _step(current.right)

        # Start the recursion on the root of the tree.
        return _step(self.root)
```

## Insertion

New elements are always added to a BST as leaf nodes. To insert a new value, recursively step through the tree as if searching for the value until locating an empty slot. The node with the empty child slot becomes the parent of the new node; connect it to the new node by modifying the parent's `left` or `right` attribute (depending on which side the child should be on) and the child's `prev` attribute.

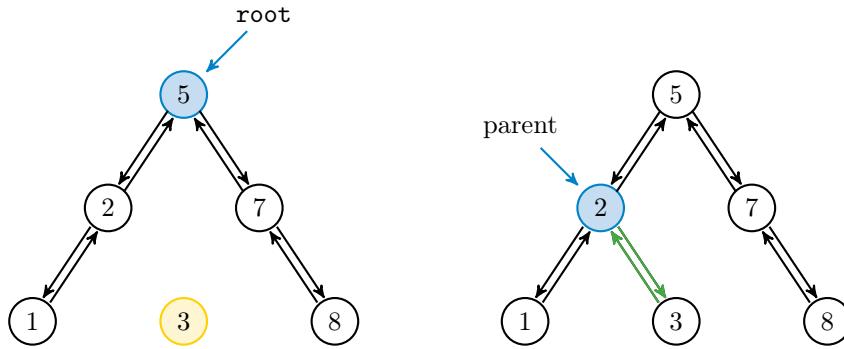


Figure 6.3: To insert 3 to the BST on the left, start at the root and recurse down the tree as if searching for 3: since  $3 < 5$ , step left to 2; since  $2 < 3$ , step right. However, 2 has no right child, so 2 becomes the parent of a new node containing 3.

**Problem 2.** Write an `insert()` method for the `BST` class that accepts some data.

1. If the tree is empty, assign the `root` attribute to a new `BSTNode` containing the data.
2. If the tree is nonempty, create a new `BSTNode` containing the data and find the existing node that should become its parent. Determine whether the new node will be the parent's `left` or `right` child, then double link the parent to the new node accordingly.  
(Hint: write a recursive function like `_step()` to find and link the parent).
3. Do not allow duplicates in the tree: if there is already a node in the tree containing the insertion data, raise a `ValueError`.

To test your method, use the `__str__()` and `draw()` methods provided in the Additional Materials section. Try constructing the binary search trees in Figures 6.2 and 6.3.

## Removal

Node removal is much more delicate than node insertion. While insertion always creates a new leaf node, a remove command may target the root node, a leaf node, or anything in between. There are three main requirements for a successful removal.

1. The target node is no longer in the tree.
2. The former children of the removed node are still accessible from the root. In other words, if the target node has children, those children must be adopted by other nodes in the tree.
3. The tree still has an ordered binary structure.

When removing a node from a linked list, there are three possible cases that must each be accounted for separately: the target node is the head, the target node is the tail, or the target node is in the middle of the list. For BST node removal, we must similarly account separately for the removal of a leaf node, a node with one child, a node with two children, and the root node.

### Removing a Leaf Node

Recall that Python's garbage collector automatically deletes objects that cannot be accessed by the user. If the node to be removed—called the *target node*—is a leaf node, then the only way to access it is via the target's parent. Locate the target with `find()`, get a reference to the parent node (using the `prev` attribute of the target), and set the parent's `right` or `left` attribute to `None`.

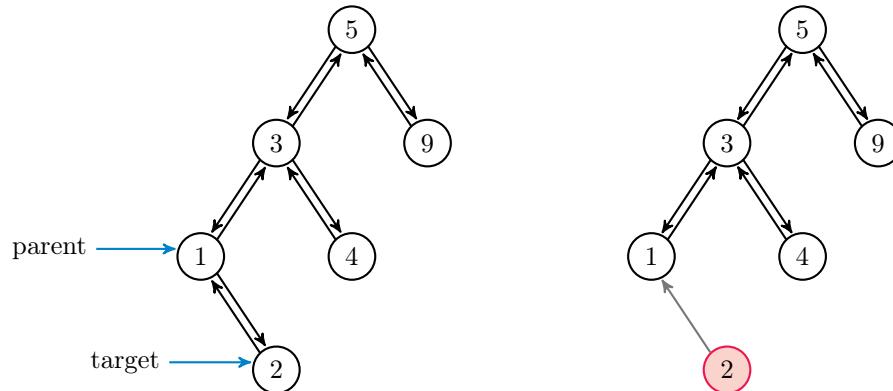


Figure 6.4: To remove 2, get a reference to its parent. Then set the parent's `right` attribute to `None`. Even though 2 still points to 1, 2 is deleted since nothing in the tree points to it.

### Removing a Node with One Child

If the target node has one child, the child must be adopted by the target's parent in order to remain in the tree. That is, the parent's `left` or `right` attribute should be set to the child, and the child's `prev` attribute should be set to the parent. This requires checking which side of the target the child is on and which side of the parent the target is on.

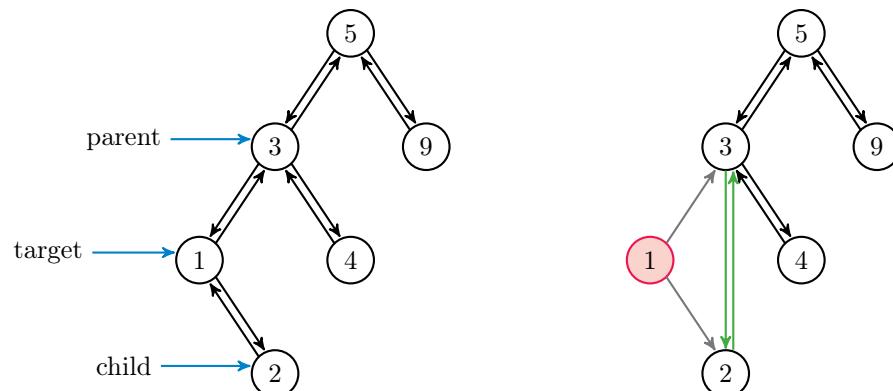


Figure 6.5: To remove 1, locate its parent (3) and its child (2). Set the parent's `left` attribute to the child and the child's `prev` attribute to the parent. Even though 1 still points to other nodes, it is deleted since nothing in the tree points to it.

### Removing a Node with Two Children

Removing a node with two children requires a slightly different approach in order to preserve the ordering in the tree. The *immediate predecessor* of a node with value  $x$  is the node in the tree with the largest value that is still smaller than  $x$ . Replacing a target node with its immediate predecessor preserves the order of the tree because the predecessor's value is greater than the values in the target's left branch, but less than the values in the target's right branch. Note that because of how the predecessor is chosen, any immediate predecessor can only have at most one child.

To remove a target with two children, find its immediate predecessor by stepping to the left of the target (so that it's value is less than the target's value), and then to the right for as long as possible (so that it has the largest such value). Remove the predecessor, recording its value. Then overwrite the value of the target with the predecessor's value.

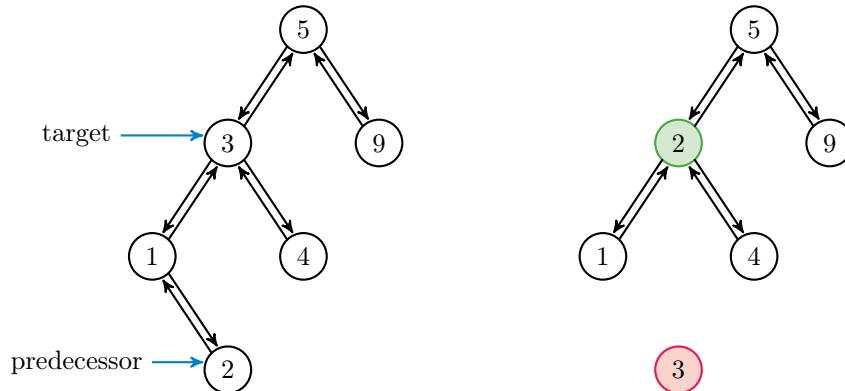


Figure 6.6: To remove 3, locate its immediate predecessor 2 by stepping left to 1, then right as far as possible. Since it is a leaf node, the predecessor can be deleted using the process in Figure 6.4. Delete the predecessor, and replace the value of the target with the predecessor's value. If the predecessor has a left child, it can be deleted with the procedure from Figure 6.5.

### Removing the Root Node

If the target is the root node, the `root` attribute may need to be reassigned after the target is removed. This adds two extra cases to consider:

1. If the root has no children, meaning it is the only node in the tree, set the root to `None`.
2. If the root has one child, that child becomes the new root of the tree. The new root's `prev` attribute should be set to `None` so the garbage collector deletes the target.

When the targeted root has two children, the node stays where it is (only its value is changed), so `root` does not need to be reassigned.

**Problem 3.** Write a `remove()` method for the `BST` class that accepts some data. If the tree is empty, or if there is no node in the tree containing the data, raise a `ValueError`. Otherwise, remove the node containing the specified data using the strategies described in Figures 6.4–6.6. Test your solutions thoroughly.

(Hint: **Before coding anything**, outline the entire method with comments and `if-else` blocks. Consider using the following control flow to account for all possible cases.)

1. The target is a leaf node.
  - (a) The target is the root.
  - (b) The target is to the left of its parent.
  - (c) The target is to the right of its parent.
2. The target has two children.  
   (Hint: use `remove()` on the predecessor's value).
3. The target has one child.  
   (Hint: start by getting a reference to the child.)
  - (a) The target is the root.
  - (b) The target is to the left of its parent.
  - (c) The target is to the right of its parent.

## UNIT TEST

Write a unit test for Problem 3: creating a remove method for your BST class. The unit test is found in the file `test_binary_trees.py` and the function is called `test_remove`.

There is an example unit test for Problem 2, your insert method, to help you structure your unit test.

## AVL Trees

The advantage of a BST is that it organizes its data so that values can be located, inserted, or removed in  $O(\log n)$  time. However, this efficiency is dependent on the *balance* of the tree. In a well-balanced tree, the number of descendants in the left and right subtrees of each node is about the same. An unbalanced tree has some branches with many more nodes than others. Finding a node at the end of a long branch is closer to  $O(n)$  than  $O(\log n)$ . This is a common problem; inserting ordered data, for example, results in a “linear” tree, since new nodes always become the right child of the previously inserted node (see Figure 6.7). The resulting structure is essentially a linked list without a `tail` attribute.

An *Adelson-Velsky Landis tree* (AVL) is a BST that prevents any one branch from getting longer than the others by recursively “balancing” the branches as nodes are added or removed. Insertion and removal thus become more expensive, but the tree is guaranteed to retain its  $O(\log n)$  search efficiency. The AVL’s balancing algorithm is beyond the scope of this lab, but the Volume 2 text includes details and exercises on the algorithm.

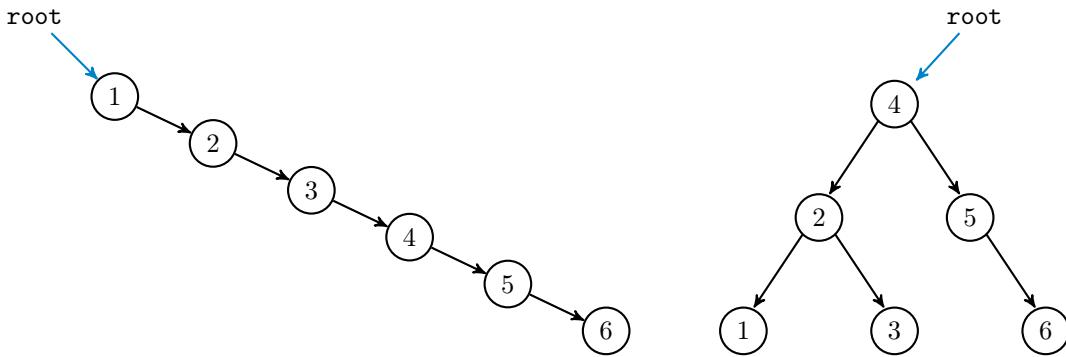


Figure 6.7: On the left, the unbalanced BST resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. On the left, the balanced AVL tree that results from the same insertion. After each insertion, the AVL tree rebalances if necessary.

**Problem 4.** Write a function to compare the build and search times of the `SinglyLinkedList` from Problem 1, the `BST` from Problems 2 and 3, and the `AVL` provided in the Additional Materials section. Begin by reading the file `english.txt`, storing the contents of each line in a list. For  $n = 2^3, 2^4, \dots, 2^{10}$ , repeat the following experiment.

1. Get a subset of  $n$  **random** items from the data set.  
(Hint: use a function from the `random` or `np.random` modules.)
2. Time (separately) how long it takes to load a new `SinglyLinkedList`, a `BST`, and an `AVL` with the  $n$  items.
3. Choose 5 **random** items from the subset, and time how long it takes to find all 5 items in each data structure. Use the `find()` method for the trees, but to avoid exceeding the maximum recursion depth, use the provided `iterative_find()` method from Problem 1 to search the `SinglyLinkedList`.

Report your findings in a single figure with two subplots: one for build times, and one for search times. Use log scales where appropriate.

## Additional Material

### Possible Improvements to the BST Class

The following are a few ideas for expanding the functionality of the `BST` class.

1. Add a keyword argument to the constructor so that if an iterable is provided, each element of the iterable is immediately added to the tree. This makes it possible to cast other iterables as a `BST` the same way that an iterable can be cast as one of Python's standard data structures.
2. Add an attribute that keeps track of the number of items in the tree. Use this attribute to implement the `__len__()` magic method.
3. Add a method for translating the `BST` into a sorted Python list.  
(Hint: examine the provided `__str__()` method carefully.)
4. Add methods `min()` and `max()` that return the smallest or largest value in the tree, respectively. Consider adding `head` and `tail` attributes that point to the minimal and maximal elements; this would make inserting new minima and maxima  $O(1)$ .

### Other Kinds of Binary Trees

In addition to the AVL tree, there are many other variations on the binary search tree, each with its own advantages and disadvantages. Consider writing classes for the following structures.

1. A *B-tree* is a tree whose nodes can contain more than one piece of data and point to more than one other node. See the Volume 2 text for details.
2. The nodes of a *red-black tree* are labeled either red or black. The tree satisfies the following rules to maintain a balanced structure.
  - (a) Every leaf node is black.
  - (b) Red nodes only have black children.
  - (c) Every (directed) path from a node to any of its descendent leaf nodes contains the same number of black nodes.

When a node is added that violates one of these constraints, the tree is rebalanced and recolored.

3. A *Splay Tree* includes an additional operation, called splaying, that makes a specified node the root of the tree. Splaying several nodes of interest makes them easier to access because they are placed close to the root.
4. A *heap* is similar to a `BST` but uses a different binary sorting rule: the value of every parent node is greater than each of the values of its children. This data structure is particularly useful for sorting algorithms; see the Volume 2 text for more details.

### Additional Code: Tree Visualization

The following methods may be helpful for visualizing instances of the `BST` and `AVL` classes. Note that the `draw()` method uses NetworkX's `graphviz_layout`, which requires the `pygraphviz` module (install it with `pip install pygraphviz`).

```

import networkx as nx
from matplotlib import pyplot as plt
from networkx.drawing.nx_agraph import graphviz_layout

class BST:
    # ...
    def __str__(self):
        """String representation: a hierarchical view of the BST.

        Example: (3)
                  / \
                (2) (5)   [2, 5]      The nodes of the BST are printed
                  / \   [1, 4, 6] by depth levels. Edges and empty
                (1) (4) (6)  nodes are not printed.

        """
        if self.root is None:
            return "[]"
        out, current_level = [], [self.root]
        while current_level:
            next_level, values = [], []
            for node in current_level:
                values.append(node.value)
                for child in [node.left, node.right]:
                    if child is not None:
                        next_level.append(child)
            out.append(values)
            current_level = next_level
        return "\n".join([str(x) for x in out])

    def draw(self):
        """Use NetworkX and Matplotlib to visualize the tree."""
        if self.root is None:
            return
        # Build the directed graph.
        G = nx.DiGraph()
        G.add_node(self.root.value)
        nodes = [self.root]
        while nodes:
            current = nodes.pop(0)
            for child in [current.left, current.right]:
                if child is not None:
                    G.add_edge(current.value, child.value)
                    nodes.append(child)
        # Plot the graph. This requires graphviz_layout (pygraphviz).
        nx.draw(G, pos=graphviz_layout(G, prog="dot"), arrows=True,
                with_labels=True, node_color="C1", font_size=8)
        plt.show()

```

## Additional Code: AVL Tree

Use the following class for Problem 4. Note that it inherits from the BST class, so its functionality is dependent on the `insert()` method from Problem 2. Note that the `remove()` method is disabled, though it is possible for an AVL tree to rebalance itself after removing a node.

```

class AVL(BST):
    """Adelson-Velsky Landis binary search tree data structure class.
    Rebalances after insertion when needed.
    """
    def insert(self, data):
        """Insert a node containing the data into the tree, then rebalance."""
        BST.insert(self, data)      # Insert the data like usual.
        n = self.find(data)
        while n:                  # Rebalance from the bottom up.
            n = self._rebalance(n).prev

    def remove(*args, **kwargs):
        """Disable remove() to keep the tree in balance."""
        raise NotImplementedError("remove() is disabled for this class")

    def _rebalance(self, n):
        """Rebalance the subtree starting at the specified node."""
        balance = AVL._balance_factor(n)
        if balance == -2:          # Left heavy
            if AVL._height(n.left.left) > AVL._height(n.left.right):
                n = self._rotate_left_left(n)           # Left Left
            else:
                n = self._rotate_left_right(n)         # Left Right
        elif balance == 2:          # Right heavy
            if AVL._height(n.right.right) > AVL._height(n.right.left):
                n = self._rotate_right_right(n)        # Right Right
            else:
                n = self._rotate_right_left(n)         # Right Left
        return n

    @staticmethod
    def _height(current):
        """Calculate the height of a given node by descending recursively until
        there are no further child nodes. Return the number of children in the
        longest chain down.
        """
        if current is None:      # Base case: the end of a branch.
            return -1            # Otherwise, descend down both branches.
        return 1 + max(AVL._height(current.right), AVL._height(current.left))

    @staticmethod
    def _balance_factor(n):
        return AVL._height(n.right) - AVL._height(n.left)

```

```
def _rotate_left_left(self, n):
    temp = n.left
    n.left = temp.right
    if temp.right:
        temp.right.prev = n
    temp.right = n
    temp.prev = n.prev
    n.prev = temp
    if temp.prev:
        if temp.prev.value > temp.value:
            temp.prev.left = temp
        else:
            temp.prev.right = temp
    if n is self.root:
        self.root = temp
    return temp

def _rotate_right_right(self, n):
    temp = n.right
    n.right = temp.left
    if temp.left:
        temp.left.prev = n
    temp.left = n
    temp.prev = n.prev
    n.prev = temp
    if temp.prev:
        if temp.prev.value > temp.value:
            temp.prev.left = temp
        else:
            temp.prev.right = temp
    if n is self.root:
        self.root = temp
    return temp

def _rotate_left_right(self, n):
    temp1 = n.left
    temp2 = temp1.right
    temp1.right = temp2.left
    if temp2.left:
        temp2.left.prev = temp1
    temp2.prev = n
    temp2.left = temp1
    temp1.prev = temp2
    n.left = temp2
    return self._rotate_left_left(n)

def _rotate_right_left(self, n):
    temp1 = n.right
    temp2 = temp1.left
```

```
temp1.left = temp2.right
if temp2.right:
    temp2.right.prev = temp1
temp2.prev = n
temp2.right = temp1
temp1.prev = temp2
n.right = temp2
return self._rotate_right(n)
```



# 7

# Nearest Neighbor Search

**Lab Objective:** *The nearest neighbor problem is an optimization problem that arises in applications such as computer vision, internet marketing, and data compression. The problem can be solved efficiently with a k-d tree, a generalization of the binary search tree. In this lab we implement a k-d tree, use it to solve the nearest neighbor problem, then use that solution as the basis of an elementary machine learning algorithm.*

## The Nearest Neighbor Problem

Let  $X \subset \mathbb{R}^k$  be a collection of data, called the *training set*, and let  $\mathbf{z} \in \mathbb{R}^k$ , called the *target*. The *nearest neighbor search problem* is determining the point  $\mathbf{x}^* \in X$  that is “closest” to  $\mathbf{z}$ .

For example, suppose you move into a new city with several post offices. Since your time is valuable, you wish to know which post office is closest to your home. The set  $X$  could be addresses or latitude and longitude data for each post office in the city;  $\mathbf{z}$  would be the data that represents your new home. The task is to find the closest post office in  $\mathbf{x} \in X$  to your home  $\mathbf{z}$ .

## Metrics and Distance

Solving the nearest neighbor problem requires a definition for distance between  $\mathbf{z}$  and elements of  $X$ . In  $\mathbb{R}^k$ , distance is typically defined by the *Euclidean metric*.

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\| = \sqrt{\sum_{i=1}^k (x_i - z_i)^2} \quad (7.1)$$

Here  $\|\cdot\|$  is the standard *Euclidean norm*, which computes vector length. In other words,  $d(\mathbf{x}, \mathbf{z})$  is the length of the straight line from  $\mathbf{x}$  to  $\mathbf{z}$ . With this notation, the nearest neighbor search problem can be written as follows.

$$\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \quad d^* = \min_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \quad (7.2)$$

NumPy and SciPy implement the Euclidean norm (and other norms) in `linalg.norm()`. This function accepts vectors or matrices. Use the `axis` argument to compute the norm along the rows or columns of a matrix: `axis=0` computes the norm of each column, and `axis=1` computes the norm of each row (see the NumPy Visual Guide).

```

>>> import numpy as np
>>> from scipy import linalg as la

>>> x0 = np.array([1, 2, 3])
>>> x1 = np.array([6, 5, 4])

# Calculate the length of the vectors x0 and x1 using the Euclidean norm.
>>> la.norm(x0)
3.7416573867739413
>>> la.norm(x1)
8.7749643873921226

# Calculate the distance between x0 and x1 using the Euclidean metric.
>>> la.norm(x0 - x1)
5.9160797830996161

>>> A = np.array([[1, 2, 3],           # or A = np.vstack((x0,x1)).
...                  [6, 5, 4]])
>>> la.norm(A, axis=0)             # Calculate the norm of each column of A.
array([ 6.08276253,  5.38516481,  5.          ])
>>> la.norm(A, axis=1)             # Calculate the norm of each row of A.
array([ 3.74165739,  8.77496439])  # This is ||x0|| and ||x1||.

```

## Exhaustive Search

Consider again the post office example. One way to find out which post office is closest is to drive from home to each post office, measuring the distance travelled in each trip. That is, we solve (7.2) by computing  $\|\mathbf{x} - \mathbf{z}\|$  for every point  $\mathbf{x} \in X$ . This strategy is called a *brute force* or *exhaustive search*.

**Problem 1.** Write a function that accepts a  $m \times k$  NumPy array  $X$  (the training set) and a 1-dimensional NumPy array  $\mathbf{z}$  with  $k$  entries (the target). Each of the  $m$  rows of  $X$  represents a point in  $\mathbb{R}^k$  that is an element of the training set.

Solve (7.2) with an exhaustive search. Return the nearest neighbor  $\mathbf{x}^*$  and its Euclidean distance  $d^*$  from the target  $\mathbf{z}$ .

(Hint: use array broadcasting and the `axis` argument to avoid using a loop.)

The complexity of an exhaustive search for  $X \subset \mathbb{R}^k$  with  $m$  points is  $O(km)$ , since (7.1) is  $O(k)$  and there are  $m$  norms to compute. This method works, but it is only feasible for relatively small training sets. Solving the problem with greater efficiency requires the use of a specialized data structure.

## K-D Trees

A *k-d tree* is a generalized binary search tree where each node in the tree contains  $k$ -dimensional data. Just as a BST makes searching easy in  $\mathbb{R}$ , a *k-d tree* provides a way to efficiently search  $\mathbb{R}^k$ .

A BST creates a partition of  $\mathbb{R}$ : if a node contains the value  $x$ , all of the nodes in its left subtree contain values that are less than  $x$ , and the nodes of its right subtree have values that are greater than  $x$ . Similarly, a  $k$ -d tree partitions  $\mathbb{R}^k$ . Each node is assigned a *pivot* value  $i \in \{0, 1, \dots, k - 1\}$  corresponding to the depth of the node: the root has  $i = 0$ , its children have  $i = 1$ , their children have  $i = 2$ , and so on. If a node has  $i = k - 1$ , its children have  $i = 0$ , their children have  $i = 1$ , and so on. The tree is constructed such that for a node containing  $\mathbf{x} = [x_0, x_1, \dots, x_{k-1}]^\top \in \mathbb{R}^k$ , if a node in the left subtree contains  $\mathbf{y}$ , then  $y_i < x_i$ . Conversely, if a node in the right subtree contains  $\mathbf{z}$ , then  $x_i \leq z_i$ . See Figure 7.1 for an example where  $k = 3$ .

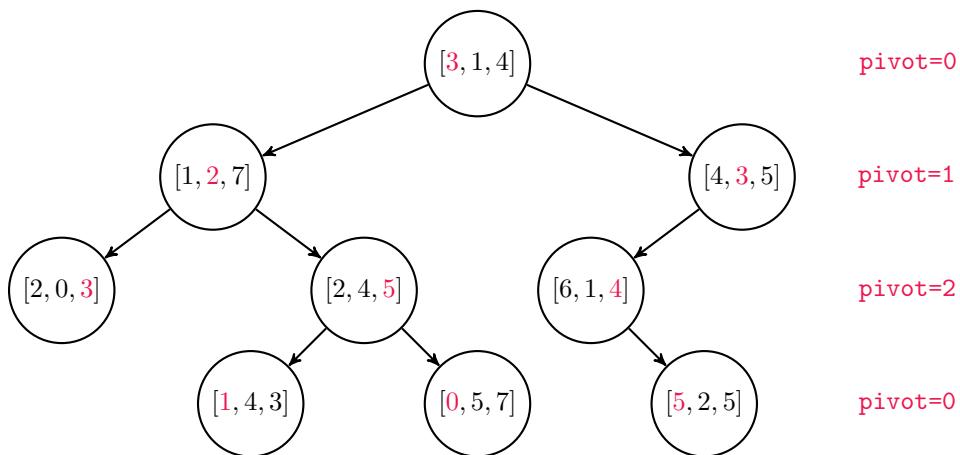


Figure 7.1: A  $k$ -d tree with  $k = 3$ . The root  $[3, 1, 4]$  has an pivot of 0, so  $[1, 2, 7]$  is to the left of the root because  $1 < 3$ , and  $[4, 3, 5]$  is to the right since  $3 \leq 4$ . Similarly, the node  $[2, 4, 5]$  has an pivot of 2, so  $[1, 4, 3]$  is to its left since  $4 < 5$  and  $[0, 5, 7]$  is to its right because  $5 \leq 7$ . The nodes that are furthest from the root have an pivot of 0 because their parents have an pivot of  $2 = k - 1$ .

**Problem 2.** Write a `KDTNode` class whose constructor accepts a single parameter  $\mathbf{x} \in \mathbb{R}^k$ . If  $\mathbf{x}$  is not a NumPy array (of type `np.ndarray`), raise a `TypeError`. Save  $\mathbf{x}$  as an attribute called `value`, and initialize attributes `left`, `right`, and `pivot` as `None`. The `pivot` will be assigned when the node is inserted into the tree, and `left` and `right` will refer to child nodes.

### UNIT TEST

The file `test_nearest_neighbor.py` contains some prewritten unit tests for Problem 3. You need to write at least one unit test for Problem 2 which will be graded.

## Constructing the Tree

### Locating Nodes

The `find()` methods for  $k$ -d trees and binary search trees are very similar. Both recursively compare the values of a target and nodes in the tree, but in a  $k$ -d tree, these values must be compared according to their `pivot` attribute. Every comparison in the recursive `_step()` function, implemented below, compares the data of `target` and `current` based on the `pivot` attribute of `current`. See Figure 7.2.

```
class KDT:
    """A k-dimensional tree for solving the nearest neighbor problem.

    Attributes:
        root (KDTNode): the root node of the tree. Like all other nodes in
            the tree, the root has a NumPy array of shape (k,) as its value.
        k (int): the dimension of the data in the tree.
    """

    def __init__(self):
        """Initialize the root and k attributes."""
        self.root = None
        self.k = None

    def find(self, data):
        """Return the node containing the data. If there is no such node in
        the tree, or if the tree is empty, raise a ValueError.
        """
        def _step(current):
            """Recursively step through the tree until finding the node
            containing the data. If there is no such node, raise a ValueError.
            """
            if current is None:                      # Base case 1: dead end.
                raise ValueError(str(data) + " is not in the tree")
            elif np.allclose(data, current.value):
                return current                      # Base case 2: data found!
            elif data[current.pivot] < current.value[current.pivot]:
                return _step(current.left)          # Recursively search left.
            else:
                return _step(current.right)         # Recursively search right.

        # Start the recursive search at the root of the tree.
        return _step(self.root)
```

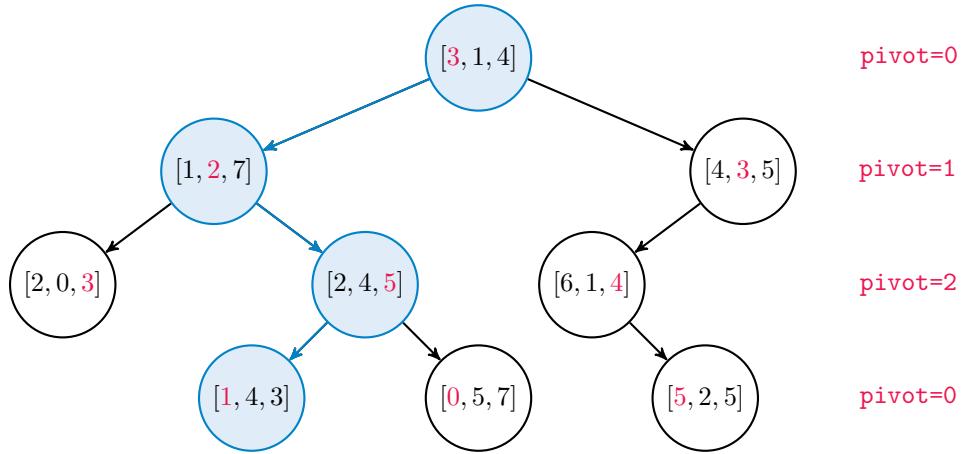
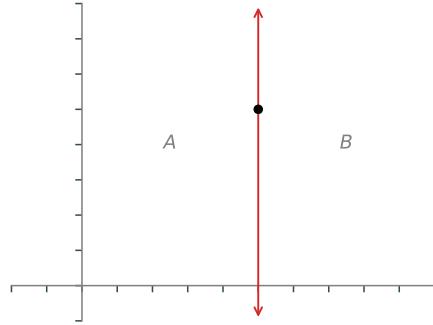
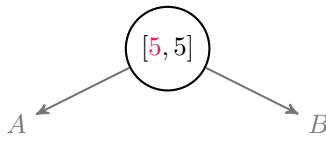
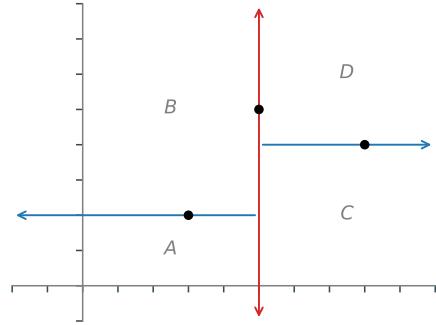
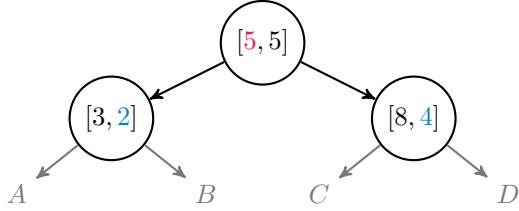


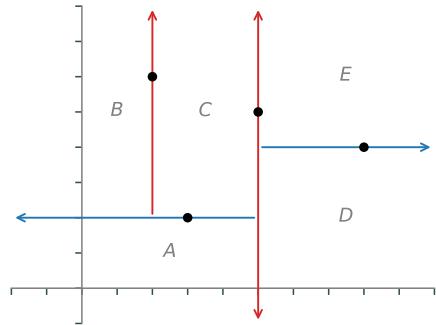
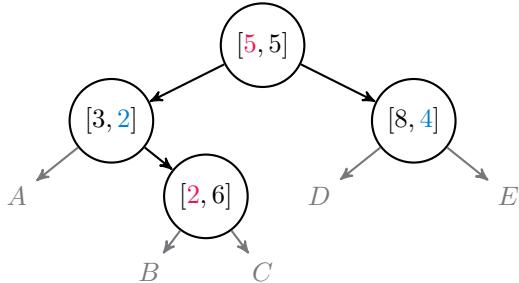
Figure 7.2: To locate the node containing  $[1, 4, 3]$ , start by comparing  $[1, 4, 3]$  to the root  $[3, 1, 4]$ . The root has a **pivot** of 0, so compare the first component of the data to the first component of the root: since  $1 < 3$ , step left. Next,  $[1, 4, 3]$  must be to the right of  $[1, 2, 7]$  because  $2 \leq 4$ . Similarly,  $[1, 4, 3]$  must be to the left of  $[2, 4, 5]$  as  $3 < 5$ .



(a) Insert  $[5, 5]$  as the root. The root always has a **pivot** of 0, so nodes to the left of the root contain points from  $A = \{(x, y) \in \mathbb{R}^2 : x < 5\}$ , and nodes on the right branch have points in  $B = \{(x, y) \in \mathbb{R}^2 : 5 \leq x\}$ .



(b) Insert  $[3, 2]$ , then  $[8, 4]$ . Since  $3 < 5$ ,  $[3, 2]$  becomes the left child of  $[5, 5]$ . Likewise, as  $5 \leq 8$ ,  $[8, 4]$  becomes the right child of  $[5, 5]$ . These new nodes have an **pivot** of 1, so they partition the space vertically: nodes to the right of  $[3, 2]$  contain points from  $B = \{(x, y) \in \mathbb{R}^2 : x < 5, 2 \leq y\}$ ; nodes to the left of  $[8, 4]$  hold points from  $C = \{(x, y) \in \mathbb{R}^2 : 5 \leq x, y < 8\}$ .



(c) Insert  $[2, 6]$ . The **pivot** cycles back to 0 since  $k = 2$ , so nodes to the left of  $[2, 6]$  have points that lie in  $B = \{(x, y) \in \mathbb{R}^2 : x < 2, 2 \leq y\}$  and nodes to the right store points in  $C = \{(x, y) \in \mathbb{R}^2 : 2 \leq x < 5, 2 \leq y\}$ .

Figure 7.3: As a  $k$ -d tree is constructed (left), it creates a partition of  $\mathbb{R}^k$  (right) by defining separating hyperplanes that pass through the points. The more points, the finer the partition.

## Inserting Nodes

To add a new node to a  $k$ -d tree, determine which existing node should be the parent of the new node by recursively stepping down the tree as in the `find()` method. Next, assign the new node as the `left` or `right` child of the parent, and set its `pivot` based on its parent's `pivot`: if the parent's `pivot` is  $i$ , the new node's `pivot` should be  $i + 1$ , or 0 if  $i = k - 1$ .

Consider again the  $k$ -d tree in Figure 7.2. To insert  $[2, 3, 4]$ , search the tree for  $[2, 3, 4]$  until hitting an empty slot. In this case, the search steps from the root down to  $[1, 4, 3]$ , which has an `pivot` of 0. Then since  $1 \leq 2$ , the new node should be to the right of  $[1, 4, 3]$ . However,  $[1, 4, 3]$  has no right child, so it becomes the parent of  $[2, 3, 4]$ . The `pivot` of the new node should therefore be 1. See Figure 7.3 for another example.

**Problem 3.** Write an `insert()` method for the `KDT` class that accepts a point  $\mathbf{x} \in \mathbb{R}^k$ .

1. If the tree is empty, create a new `KDTreeNode` containing  $\mathbf{x}$  and set its `pivot` to 0. Assign the `root` attribute to the new node and set the `k` attribute as the length of  $\mathbf{x}$ . Thereafter, raise a `ValueError` if data to be inserted is not in  $\mathbb{R}^k$ .
2. If the tree is nonempty, create a new `KDTreeNode` containing  $\mathbf{x}$  and find the existing node that should become its parent. Determine whether the new node will be the parent's `left` or `right` child, then link the parent to the new node accordingly. Set the `pivot` of the new node based on its parent's `pivot`.  
(Hint: write a recursive function like `_step()` to find and link the parent.)
3. Do not allow duplicates in the tree: if there is already a node in the tree containing  $\mathbf{x}$ , raise a `ValueError`.

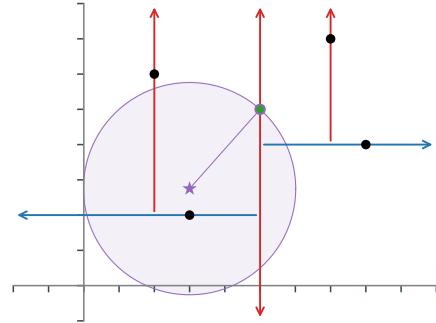
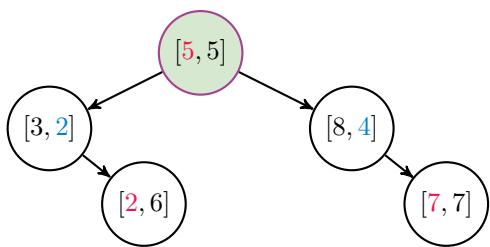
To test your method, use the `__str__()` method provided in the Additional Materials section. Try constructing the trees in Figures 7.1 and 7.3. Also check that the provided `find()` method works as expected.

## Nearest Neighbor Search with K-D Trees

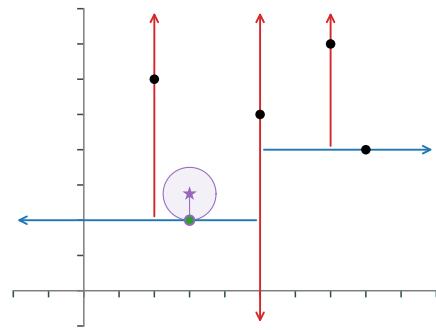
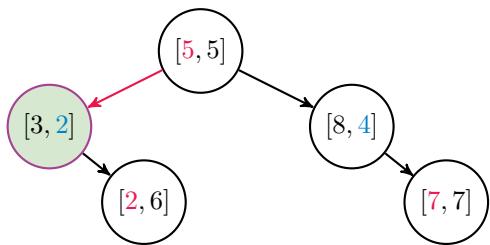
Given a target  $\mathbf{z} \in \mathbb{R}^k$  and a  $k$ -d tree containing a set  $X \subset \mathbb{R}^k$  of  $m$  points, the nearest neighbor problem can be solved by traversing the tree in a manner that is similar to the `find()` or `insert()` methods from the previous section. The advantage of this strategy over an exhaustive search is that not every  $\mathbf{x} \in X$  has to be compared to  $\mathbf{z}$  via (7.1); the tree structure makes it possible to rule out some elements of  $X$  without actually computing their distances to  $\mathbf{z}$ . The complexity is  $O(k \log(m))$ , a significant improvement over the  $O(km)$  complexity of an exhaustive search.

To begin, set  $\mathbf{x}^*$  as the value of the root and compute  $d^* = d(\mathbf{x}^*, \mathbf{z})$ . Starting at the root, step down through the tree as if searching for the target  $\mathbf{z}$ . At each step, determine if the value  $\mathbf{x}$  of the current node is closer to  $\mathbf{z}$  than  $\mathbf{x}^*$ . If it is, assign  $\mathbf{x}^* = \mathbf{x}$  and recompute  $d^* = d(\mathbf{x}^*, \mathbf{z})$ . Continue this process until reaching a leaf node.

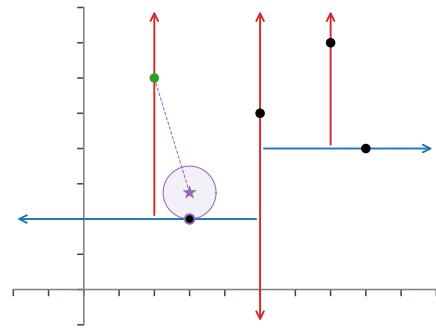
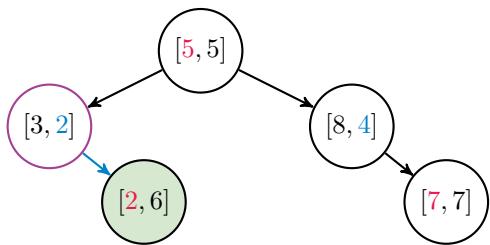
Next, backtrack along the search path and determine if the non-explored branch needs to be searched. To do this, check that the sphere of radius  $d^*$  centered at  $\mathbf{z}$  does not intersect with the separating hyperplane defined by the current node. That is, if the separating hyperplane is further than  $d^*$  from  $\mathbf{z}$ , then no points on the other side of the hyperplane can possibly be the nearest neighbor. See Figure 7.4 for an example and Algorithm 1 for the details of the procedure.



(a) Start at the root, setting  $\mathbf{x}^* = [5, 5]$ . The sphere of radius  $d^* = d(\mathbf{x}^*, \mathbf{z})$  centered at  $\mathbf{z}$  intersects the hyperplane  $x = 5$ , so (at this point) it is possible that a nearer neighbor lies to the right of the root.



(b) If the target  $\mathbf{z} = [3, 2.75]$  were in the tree, it would be to the left of the root, so step left and examine  $\mathbf{x} = [3, 2]$ . Since  $d(\mathbf{x}, \mathbf{z}) < d(\mathbf{x}^*, \mathbf{z})$ , reassign  $\mathbf{x}^* = \mathbf{x}$  and recompute  $d^*$ . Now the sphere of radius  $d^*$  centered at  $\mathbf{z}$  no longer intersects the root's hyperplane, so the nearest neighbor cannot be in the root's right subtree.



(c) Continuing the search, step right to check the point  $\mathbf{x} = [2, 6]$ . In this case  $d(\mathbf{x}, \mathbf{z}) > d(\mathbf{x}^*, \mathbf{z})$ , meaning  $\mathbf{x}$  is **not** nearer to  $\mathbf{z}$  than  $\mathbf{x}^*$ . Since  $[2, 6]$  is a leaf node, retrace the search steps up the tree to check the non-searched branches. However, the sphere around  $\mathbf{z}$  does not intersect any splitting hyperplanes defined by the tree, so  $\mathbf{x}^*$  is guaranteed to be the nearest neighbor.

Figure 7.4: Nearest neighbor search of a  $k$ -d tree with  $k = 2$ . The target is  $\mathbf{z} = [3, 2.75]$  and the nearest neighbor is  $\mathbf{x}^* = [3, 2]$  with minimal distance  $d^* = 0.75$ . The tree structure allows the algorithm to eliminate  $[8, 4]$  and  $[7, 7]$  from consideration without computing their distance from  $\mathbf{z}$ .

**Algorithm 1** *k*-d tree nearest neighbor search

---

```

1: procedure NEAREST NEIGHBOR SEARCH(z, root)
2:   procedure KDSEARCH(current, nearest,  $d^*$ )
3:     if current is None then                                 $\triangleright$  Base case: dead end.
4:       return nearest,  $d^*$ 
5:     x  $\leftarrow$  current.value
6:     i  $\leftarrow$  current.pivot
7:     if  $d(\mathbf{x}, \mathbf{z}) < d^*$  then                       $\triangleright$  Check if current is closer to z than nearest.
8:       nearest  $\leftarrow$  current
9:        $d^* \leftarrow d(\mathbf{x}, \mathbf{z})$ 
10:    if  $z_i < x_i$  then                                $\triangleright$  Search to the left.
11:      nearest,  $d^* \leftarrow KDSearch(current.left, nearest,  $d^*$ )
12:      if  $z_i + d^* \geq x_i$  then                   $\triangleright$  Search to the right if needed.
13:        nearest,  $d^* \leftarrow KDSearch(current.right, nearest,  $d^*$ )
14:    else                                          $\triangleright$  Search to the right.
15:      nearest,  $d^* \leftarrow KDSearch(current.right, nearest,  $d^*$ )
16:      if  $z_i - d^* \leq x_i$  then                   $\triangleright$  Search to the left if needed.
17:        nearest,  $d^* \leftarrow KDSearch(current.left, nearest,  $d^*$ )
18:    return nearest,  $d^*$ 
19:  node,  $d^* \leftarrow KDSearch(root, root,  $d(\mathbf{root.value}, \mathbf{z})$ )
20:  return node.value,  $d^*$$$$$$ 
```

---

**Problem 4.** Write a method for the KDT class that accepts a target point  $\mathbf{z} \in \mathbb{R}^k$ . Use Algorithm 1 to solve (7.2). Return the nearest neighbor  $\mathbf{x}^*$  (the actual NumPy array, not the KDTNode) and its distance  $d^*$  from  $\mathbf{z}$ .

Compare your method to the exhaustive search in Problem 1 and to SciPy’s built-in `KDTree` class. This structure is essentially a heavily optimized version of the KDT class. To solve the nearest neighbor problem, initialize the tree with data, then “query” the tree with the target point. The `query()` method returns a tuple of the minimum distance and the index of the nearest neighbor in the data.

```

>>> from scipy.spatial import KDTree

# Initialize the tree with data (in this example, use random data).
>>> data = np.random.random((100,5))      # 100 5-dimensional points.
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree for the nearest neighbor and its distance from 'target'.
>>> min_distance, index = tree.query(target)
>>> print(min_distance)
0.24929868807
>>> tree.data[index]                      # Get the actual nearest neighbor.
array([ 0.26927057,  0.03160271,  0.46830759,  0.26766863,  0.63073275])
```

### ACHTUNG!

There are a few caveats to using a  $k$ -d tree for the nearest neighbor search problem.

- Constructing the tree takes time. For small enough data sets, an exhaustive search may be faster than the combined time of constructing and searching a tree. On the other hand, once the tree is constructed, it can be used for multiple nearest-neighbor queries.
- In the worst case—when the tree is completely unbalanced—the search complexity is  $O(km)$  instead of  $O(k \log(m))$ . Fortunately, there are algorithms for constructing the tree intelligently so that it is mostly balanced, and a random insertion order usually results in a somewhat balanced tree.

## K-Nearest Neighbors

The nearest neighbor algorithm provides one way to solve a common machine learning problem. In *supervised learning*, a *training set*  $X \subset D$  has a corresponding set of *labels*  $Y$  that specifies a category for each element of  $X$ . For instance,  $X$  could contain financial data on  $m$  individuals, and  $Y$  could be a set of  $m$  booleans indicating which individuals have filed for bankruptcy. Supervised learning algorithms use the training data to construct a function  $f : D \rightarrow Y$  that maps points to their corresponding label. In other words, the algorithm “learns” enough about the relationship between  $X$  and  $Y$  to intelligently label arbitrary elements of  $D$ . In the bankruptcy example, a person could then use their own financial data to learn whether or not they look more like someone who files for bankruptcy or someone who does not.

A  *$k$ -nearest neighbors* classifier uses a simple strategy to label an arbitrary  $\mathbf{z} \in D$ : find the  $k$  elements of  $X$  that are nearest to  $\mathbf{z}$  (usually in terms of the Euclidean metric) and choose the most common label from those  $k$  elements as the label of  $\mathbf{z}$ . That is, the points in the  $k$  labeled points that are most like  $\mathbf{z}$  are allowed to “vote” on how  $\mathbf{z}$  should be labeled. See Figure 7.5.

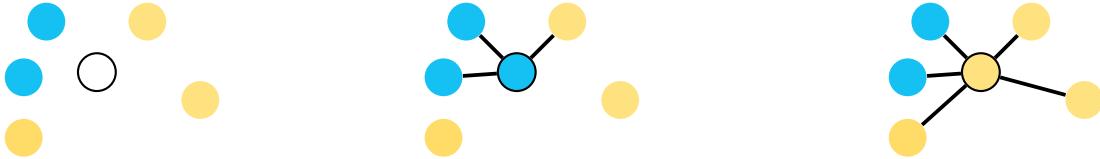


Figure 7.5: To classify the center node, determine its  $k$ -nearest neighbors and pick the most common label of the neighbors. If  $k = 3$ , the  $k$  nearest points are two blues and a yellow, so the center node is labeled blue. For  $k = 5$ , the  $k$  nearest points consists of two blues and three yellows, so the center node is labeled yellow.

### ACHTUNG!

The  $k$  in  $k$ -d tree refers to the **dimension** of the data housed in the tree, but the  $k$  in  $k$ -nearest neighbors refers to the **number of neighbors** to use in the voting scheme. Unfortunately, both names are standard.

**Problem 5.** Write a `KNeighborsClassifier` class with the following methods.

1. The constructor should accept an integer `n_neighbors`, the number of neighbors to include in the vote (the  $k$  in  $k$ -nearest neighbors). Save this value as an attribute.
2. `fit()`: accept an  $m \times k$  NumPy array  $X$  (the training set) and a 1-dimensional NumPy array  $y$  with  $m$  entries (the training labels). As in Problems 1 and 4, each of the  $m$  rows of  $X$  represents a point in  $\mathbb{R}^k$ . Here  $y_i$  is the label corresponding to row  $i$  of  $X$ . Load a SciPy `KDTree` with the data in  $X$ . Save the tree and the labels as attributes.
3. `predict()`: accept a 1-dimensional NumPy array  $z$  with  $k$  entries. Query the `KDTree` for the `n_neighbors` elements of  $X$  that are nearest to  $z$  and return the most common label of those neighbors. If there is a tie for the most common label (such as if  $k = 2$  in Figure 7.5), choose the alphanumerically smallest label.  
(Hint: use `scipy.stats.mode()`. The default behavior splits ties correctly.)

To get several nearest neighbors from the tree, specify `k` in `KDTree.query()`.

```
>>> data = np.random.random((100,5))      # 100 5-dimensional points.
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree for the 3 nearest neighbors.
>>> distances, indices = tree.query(target, k=3)
>>> print(indices)
[26 30 32]
```

#### NOTE

The format of the `KNeighborsClassifier` in Problem 5 conforms to the style of *scikit-learn* (`sklearn`), a large machine learning library in Python. In fact, scikit-learn has a class called `sklearn.neighbors.KNeighborsClassifier` that is a more robust version of the class from Problem 5. See <http://scikit-learn.org/stable/modules/neighbors.html> for more tools from scikit-learn for solving the nearest neighbor problem in the context of machine learning.

## Handwriting Recognition

*Computer vision* is a challenging area of artificial intelligence that focuses on autonomously interpreting images. Perhaps the simplest computer vision problem is that of translating images into text. Roughly speaking, computers store grayscale images as  $M \times N$  arrays of pixel brightness values: 0 corresponds to black, and 255 to white. Flattening out such an array yields a vector in  $\mathbb{R}^{MN}$ . Given some images of characters with labels (assigned by humans), a  $k$ -nearest neighbor classifier can intelligently decide what character the image represents.

**Problem 6.** The file `mnist_subset.npz` contains part of the MNIST dataset,<sup>a</sup> a collection of  $28 \times 28$  images of handwritten digits and their labels. The data is split into four parts.

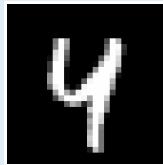
- `X_train`: A  $3000 \times 728$  matrix, the training set. Each of the 3000 rows is a flattened  $28 \times 28$  image to be used in training the classifier.
- `y_train`: A 1-dimensional NumPy array with 3000 entries. The entries are integers from 0 to 9, the labels corresponding to the images in `X_train`.
- `X_test`: A  $500 \times 728$  matrix of 500 images to classify.
- `y_test`: A 1-dimensional NumPy array with 500 entries. These are the labels corresponding to `X_test`, the “right answers” that the classifier will try to guess.

The following code uses `np.load()` to extract the data.

```
>>> data = np.load("mnist_subset.npz")
>>> X_train = data["X_train"].astype(np.float)           # Training data
>>> y_train = data["y_train"]                           # Training labels
>>> X_test = data["X_test"].astype(np.float)            # Test data
>>> y_test = data["y_test"]                            # Test labels
```

To visualize one of the images, reshape it as a  $28 \times 28$  array and use `plt.imshow()`.

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(X_test[0].reshape((28,28)), cmap="gray")
>>> plt.show()
```



Write a function than accepts an integer `n_neighbors`. Load a classifier from Problem 5 with the data `X_train` and the corresponding labels `y_train`. Use the classifier to predict the labels of each image in `X_test`. Return the classification accuracy, the percentage of predictions that match `y_test`. The accuracy should be at least 90% using 4 nearest neighbors.

---

<sup>a</sup>See <http://yann.lecun.com/exdb/mnist/>.

### NOTE

The  $k$ -nearest neighbors algorithm is **not** the best machine learning algorithm for this problem, but it is a good starting point because of its simplicity. In fact,  $k$ -nearest neighbors is often used as a baseline to compare against more complicated machine learning techniques.

## Additional Material

### Ball Trees

The nearest neighbor problem can also be solved efficiently with a *ball tree*, another space-partitioning data structure. Instead of separating  $\mathbb{R}^k$  by hyperplanes, a ball tree uses nested hyperspheres to split up the space. Since the partitioning scheme is different, a nearest neighbor search through a ball tree is more efficient than the  $k$ -d tree search for some data sets. See [https://en.wikipedia.org/wiki/Ball\\_tree](https://en.wikipedia.org/wiki/Ball_tree) for more details.

### The Curse of Dimensionality

The *curse of dimensionality* refers to a phenomena that occurs when dealing with high-dimensional data: the computational cost of an algorithm increases much more rapidly as the dimension increases than it does when the number of points increases. This problem occurs in many other areas involving multi-dimensional data, but it is quite apparent in a nearest neighbor search.

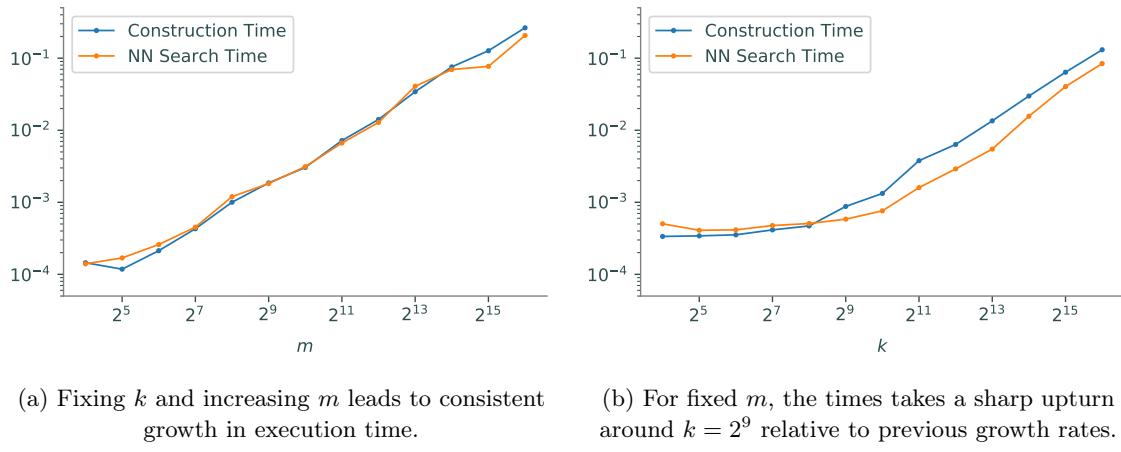


Figure 7.6: Construction and nearest neighbor search times for a  $k$ -d tree with a  $m \times k$  training set.

See [https://en.wikipedia.org/wiki/Curse\\_of\\_dimensionality](https://en.wikipedia.org/wiki/Curse_of_dimensionality) for more examples. One way to avoid the curse of dimensionality is via *dimension reduction*, a process usually based on the singular value decomposition (SVD) that projects data into a lower-dimensional space.

### Tiebreaker Strategies

As mentioned in Problem 5, the majority voting scheme in the  $k$ -nearest neighbor algorithm can often result in a tie. Breaking the tie intelligently is a science unto itself, but here are a few common strategies.

1. For binary classification (meaning there are only two labels), choose an odd  $k$  to avoid a tie in the first place.
2. Redo the search with  $k - 1$  neighbors, repeating as needed until  $k = 1$ .
3. Choose the label that appears more frequently in the test set.
4. Choose randomly among the labels that are tied for most common.

## Additional Code

The following code creates a string representation for the KDT class. Use this to test Problem 3.

```
class KDT:
    # ...
    def __str__(self):
        """String representation: a hierarchical list of nodes and their axes.

        Example:
            [5,5]                                'KDT(k=2)
            /   \
        [3,2]  [8,4]                            [5 5]  pivot = 0
                                                [3 2]  pivot = 1
                                                \   \
                                                [2,6]  [7,5]  [8 4]  pivot = 1
                                                [2 6]  pivot = 0
                                                [7 5]  pivot = 0
        """
        if self.root is None:
            return "Empty KDT"
        nodes, strs = [self.root], []
        while nodes:
            current = nodes.pop(0)
            strs.append("{}\tpivot = {}".format(current.value, current.pivot))
            for child in [current.left, current.right]:
                if child:
                    nodes.append(child)
        return "KDT(k={})\n".format(self.k) + "\n".join(strs)
```

# 8

# Breadth-first Search

**Lab Objective:** *Shortest path problems are an important part of graph theory and network analysis. Applications include finding the fastest way to drive between two points on a map, network routing, genealogy, automated circuit layout, and a variety of other important problems. In this lab we represent graphs as adjacency dictionaries, implement a shortest path algorithm based on a breadth-first search, and use the NetworkX package to solve a shortest path problem on a large network of movies and actors.*

## Adjacency Dictionaries

Computers can represent mathematical graphs in various ways. Graphs with very specific structures are often stored with specialized data structures, such as binary search trees. More general graphs without structural constraints are usually represented with an *adjacency matrix*, where each row and column of the matrix corresponds to a node in the graph, and the entries indicate connections between nodes. Adjacency matrices are usually implemented in a sparse matrix format since only the entries corresponding to node connections are nonzero.

Another common graph data structure is an *adjacency dictionary*, a dictionary with a key for each node in the graph. The dictionary values are the set of nodes connected to the key node. Adjacency dictionaries automatically gain the advantages of a sparse matrix format since they only store information on the actual node connections (the nonzero entries of the adjacency matrix).

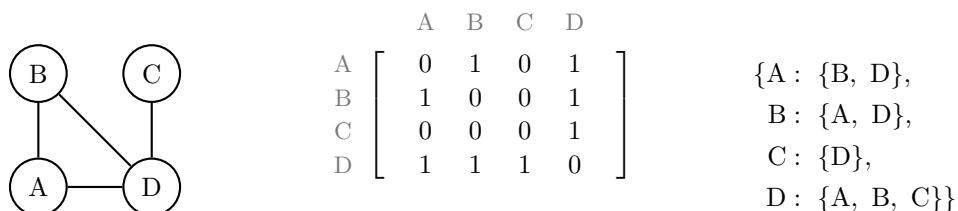


Figure 8.1: A simple unweighted graph (left), its adjacency matrix (middle), and its adjacency dictionary (right). The graph is undirected, so the adjacency matrix is symmetric. Note that the adjacency dictionary also encodes this behavior: since A and B are connected, B is in the set of values corresponding to the key A, and A is in the set of values corresponding to the key B.

## Hash-based Data Structures

A Python `set` is an unordered data type with no repeated elements. The set class is implemented as a *hash table*, meaning it uses *hash values*—integers that uniquely identify an object—to organize its elements. Roughly speaking, in order to access, add, or remove an object `x` to a set, Python computes the hash value of `x`, and that value indicates where `x` is (or should be) in memory. In other words, there is only one place in memory that `x` could be; if it isn't in that place, it isn't in the set. This implementation results in  $O(1)$  lookup, insertion, and removal operations, an enormous improvement over the  $O(n)$  search time for lists and the  $O(\log n)$  search time for sorted structures like binary search trees. It is also why set elements are unique.

Method	Description
<code>add()</code>	Add an element to the set. This has no effect if the element is already present.
<code>remove()</code>	Remove an element from the set, raising a <code>KeyError</code> if it is not a member of the set.
<code>discard()</code>	Remove an element from the set without raising an exception if it is not a member of the set.
<code>pop()</code>	Remove and return an arbitrary set element.
<code>union()</code>	Return all elements that are in either set as a new set.
<code>intersection()</code>	Return all elements that are in both sets as a new set.
<code>update()</code>	Add all elements of another set in-place.

Table 8.1: Basic methods of the `set` class.

```
# Initialize a set. Note that repeats are not added.
>>> animals = {"cow", "cat", "dog", "mouse", "cow"}
>>> print(animals)
{'cow', 'dog', 'mouse', 'cat'}

>>> animals.add("horse")      # Add an object to the set.
>>> "horse" in animals
True
>>> animals.remove("emu")     # Attempt to delete an object from the set,
                             # resulting in an exception.
KeyError: 'emu'
>>> animals.pop()            # Delete and return a random object from the set.
'mouse'
>>> print(animals)
{'cat', 'horse', 'dog', 'cow'}

# Add all of the elements of another set to this one.
>>> animals.update({"dog", "velociraptor"})
>>> print(animals)
{'velociraptor', 'cat', 'horse', 'dog', 'cow'}

# Intersect this set with another one.
>>> animals.intersection({"cat", "cow", "cheetah"})
{'cat', 'cow'}
```

Sets are extremely fast, but they do not support indexing because the elements are unordered. A Python `dict`, on the other hand, is a hash-based data structure that stores key-value pairs: the keys of a dictionary act like a set (unique and unordered, with  $O(1)$  lookup), but each key corresponds to another object, called its value. The keys index the dictionary and allow  $O(1)$  lookup of the values.

Method	Description
<code>keys()</code>	Return a set-like iterator for the dictionary's keys.
<code>values()</code>	Return a set-like iterator for the dictionary's values.
<code>items()</code>	Return an iterator for the dictionary's key-value pairs.
<code>pop()</code>	Remove a specified key and return the corresponding value, raising a <code>KeyError</code> if the key is not a member of the dictionary.
<code>update()</code>	Add or overwrite key-value pairs in-place with those from another dictionary.

Table 8.2: Basic methods of the `dict` class.

```
# Initialize a dictionary.
>>> grades = {"business": "A", "math": "A+", "visual arts": "B"}
>>> grades["math"]
'A+'                                         # The key "math" maps to the value "A+".

# Add a "science" key with corresponding value "A".
>>> grades["science"] = "A"

# Remove the "business" key.
>>> grades.pop("business")
'A'
>>> print(grades)
{'math': 'A+', 'visual arts': 'B', 'science': 'A'}

# Display the keys, values, and items.
>>> list(grades.keys()), list(grades.values())
(['math', 'visual arts', 'science'], ['A+', 'B', 'A'])
>>> for key, value in grades.items():
...     print(key, "=>", value)
...
math => A+
visual arts => B
science => A

# Add key-value pairs from another dictionary.
>>> grades.update({"cooking": "A+", "math": "C"})
>>> print(grades)
{'math': 'C', 'visual arts': 'B', 'science': 'A', 'cooking': 'A+'}
```

Dictionaries are ideal for storing values that need to be accessed often and for representing one-to-one or one-to-many relationships. Thus, the `dict` class is a natural choice for implementing adjacency dictionaries. For example, the following code defines the adjacency dictionary for the graph in Figure 8.1. Note that the dictionary values are sets.

```
>>> adjacency = {'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}}

# The nodes of the graph are the dictionary keys.
>>> set(adjacency.keys())
{'B', 'D', 'A', 'C'}

# The values are the nodes that the key node is adjacent to.
>>> adjacency['A']
{'B', 'D'}                                # A is adjacent to B and D.
>>> 'C' in adjacency['B']
False                                         # B and C are not adjacent.
>>> 'C' in adjacency['D']
True                                          # C and D are adjacent.
```

### ACHTUNG!

Elements of a `set` and keys of a `dict` must be *hashable*. Mutable objects—lists, sets and dictionaries—are not hashable, so they are not allowed as set elements or dictionary keys. Thus, in order to represent a graph with an adjacency dictionary, each of the node labels should a string, a number, or some other hashable type.

**Problem 1.** Consider the following `Graph` class.

```
class Graph:
    """A graph object, stored as an adjacency dictionary. Each node in the
    graph is a key in the dictionary. The value of each key is a set of
    the corresponding node's neighbors.

    Attributes:
        d (dict): the adjacency dictionary of the graph.
    """
    def __init__(self, adjacency={}):
        """Store the adjacency dictionary as a class attribute"""
        self.d = dict(adjacency)

    def __str__(self):
        """String representation: a view of the adjacency dictionary."""
        return str(self.d)
```

Add the following methods to this class.

1. `add_node()`: Add a node (with no initial edges) if it is not already present.  
(Hint: use `set()` to create an empty set.)
2. `add_edge()`: Add an edge between two nodes. Add the nodes to the graph if they are not already present.
3. `remove_node()`: Remove a node, including all edges adjacent to it. This method should raise a `KeyError` if the node is not in the graph.
4. `remove_edge()`: Remove the edge between two nodes. This method should raise a `KeyError` if either node is not in the graph, or if there is no edge between the nodes.

## Breadth-first Search

Many common problems that arise in graph theory require finding the shortest path between two nodes in a graph. For some highly structured graphs, such as binary search trees, this is a fairly straightforward problem (in the case of a tree, the shortest path is also the only path). Finding a path between nodes in a graph of arbitrary structure, however, requires a careful and methodical approach. The two most common graph search algorithms are *depth-first search* (DFS) and *breadth-first search* (BFS). The breadth-first strategy is almost always better at finding shortest paths than the depth-first strategy,<sup>1</sup> though a DFS can be useful for path problems in certain graphs.

To traverse a graph with a BFS, choose a node to start at, called the *source* node. First, visit each of the source node's neighbors. Next, visit each of the source node's neighbors' neighbors. Then visit each of their neighbors, continuing the process until all nodes have been visited. This strategy explores all of the nodes closest to the source node before incrementally moving “deeper” (further from the source node) into the tree.

The implementation of a BFS requires the following data structures to keep track of which nodes have already been visited and the order in which to visit nodes in future steps.

- A list  $V$ : The nodes that have been **visited**, in visitation order.
- A **queue**  $Q$ : The nodes to be visited, in the order that they were discovered. Recall that a *queue* is a limited-access list where data is inserted to one end, but removed from the other (first-in, first-out).
- A set  $M$ : The nodes that have been visited, or that are **marked** to be visited. This is the union of the nodes in  $V$  and  $Q$ .

To begin the search, add the source node to  $Q$  and  $M$ . Then, until  $Q$  is empty, repeat the following:

1. Pop a node off of  $Q$ ; call it the *current* node.
2. “Visit” the current node by appending it to  $V$ .
3. Add the neighbors of the current node that are not in  $M$  to  $Q$  and  $M$ .

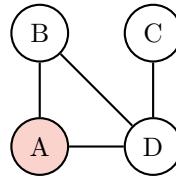
The “that are not in  $M$ ” clause of step 3 prevents nodes from being added to  $Q$  more than once. Note that step 3 could be replaced with “Add the neighbors of the current node that are not in  $V \cup Q$  to  $Q$ .” However, lookup in  $M$  (a set) is much faster than lookup in  $V$  and  $Q$  (arrays or linked lists), so including  $M$  greatly speeds up the algorithm.

---

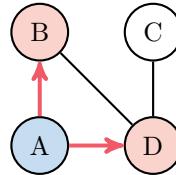
<sup>1</sup>See <https://xkcd.com/761/>.

## NOTE

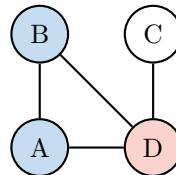
The first-in, first-out (FIFO) structure of  $Q$  enforces the “breadth-first” nature of the BFS: nodes that are marked first are visited first. Using a last-in, first-out (LIFO) stack for  $Q$  changes the search to a DFS: the next node to visit is the one that was marked last.



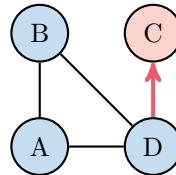
$V$	
$Q$	A
$M$	



$V$	A
$Q$	B D
$M$	A B D



$V$	A B
$Q$	D
$M$	A B D



$V$	A B D
$Q$	C
$M$	A B D C

Figure 8.2: To start a BFS from node A to node C, put A in the visit queue  $Q$  and mark it by adding it to the set  $M$ . Pop A off the queue and “visit” it by adding A to the visited list  $V$  and the neighboring nodes B and D to  $Q$ . Then visit B, but do not add anything to  $Q$  because all of the neighbors of B are already marked. Finally, visit D, at which point the target node C is located because it is adjacent to D.

**Problem 2.** Write a method for the `Graph` class that accepts a source node. Traverse the graph with a breadth-first search until all nodes have been visited. Return the list of nodes in the order that they were visited. If the source node is not in the graph, raise a `KeyError`.

(Hint: for  $Q$ , use a `deque` from the `collections` module, and make sure that nodes are added to one end but popped off of the other.)

## Shortest Paths via BFS

Consider the problem of locating a path between two nodes with a BFS. The nodes that are directly connected to the source node are all visited before any other nodes; more generally, the nodes that are  $n$  nodes away from the source node are all visited before nodes that are  $n+1$  or more nodes from the source point. Therefore, the search path taken to discover the target with a BFS must be the shortest path from the source node to the target node.

Examine again the graph in Figures 8.1 and 8.2. The shortest path from A to C starts at A, goes to D, and ends at C. During a BFS originating at A, D is placed on the visit queue because it is one of A's neighbors, and C is placed on the queue because it is one of D's neighbors. Given that A was the node that visited D, and that D was the node that visited C, the shortest path from A to C can be constructed by stepping backward along the search path.

To implement this idea, initialize a dictionary before starting the BFS. When a node is marked and added to the visit queue, add a key-value pair mapping the `visited` node to the `visiting` node (for example,  $B \mapsto A$  means B was marked while visiting A). When the target node is found, step through the dictionary until arriving at the source node, recording each step.

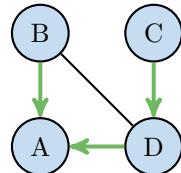


Figure 8.3: In the BFS from Figure 8.2, nodes B and D were marked while visiting node A, and node C was marked while visiting node D (this is same as reversing the red arrows in Figure 8.2). Thus the “visit path” from C to A is  $C \rightarrow D \rightarrow A$ , so the shortest path from A to C is  $[A, D, C]$ .

**Problem 3.** Add a method to the `Graph` class that accepts source and target nodes. Begin a BFS at the source node and proceed until the target is found. Return a list containing the node values in the shortest path from the source to the target (including the endpoints). If either of the input nodes are not in the graph, raise a `KeyError`.

## Shortest Paths via NetworkX

*NetworkX* is a Python package for creating, manipulating, and exploring graphs. Its `Graph` object represents a graph with an adjacency dictionary, similar to the class from Problems 1–3, and has many methods for interpreting information about the graph and its structure. As before, the nodes must be hashable (a number, string, or another immutable object).

Method	Description
<code>add_node()</code>	Add a single node to the graph.
<code>add_nodes_from()</code>	Add a list of nodes to the graph.
<code>add_edge()</code>	Add an edge between two nodes.
<code>add_edges_from()</code>	Add a list of edges to the graph.

Table 8.3: Methods of the `nx.Graph` class for adding nodes and edges.

```
>>> import networkx as nx

# Initialize a NetworkX graph from an adjacency dictionary.
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

>>> print(G.nodes())          # Print the nodes.
['A', 'B', 'C', 'D']
>>> print(G.edges())         # Print the edges as tuples.
[('A', 'D'), ('A', 'B'), ('B', 'D'), ('C', 'D')]

>>> G.add_node('E')          # Add a new node.
>>> G.add_edge('A', 'F')      # Add an edge, which also adds a new node 'F'.
>>> G.add_edges_from([('A', 'C'), ('F', 'G')]) # Add several edges at once.

>>> set(G['A'])              # Get the set of nodes neighboring node 'A'.
{'B', 'C', 'D', 'F'}
```

## The Kevin Bacon Problem

The vintage parlor game *Six Degrees of Kevin Bacon* is played by naming an actor, then trying to find the shortest chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Pulp Fiction (1994)* with Frank Whaley, who was in *JFK (1991)* with Kevin Bacon. In other words, the goal of the game is to solve a shortest path problem on a graph that connects actors to the movies that they have been in.

**Problem 4.** The file `movie_data.txt` contains IMDb data for about 137,000 movies. Each line of the file represents one movie: the title is listed first, then the cast members, with entries separated by a / character. For example, the line for *The Dark Knight (2008)* starts with

The Dark Knight (2008)/Christian Bale/Heath Ledger/Aaron Eckhart/...

Any / characters in movie titles have been replaced with the vertical pipe character | (for example, *Frost|Nixon (2008)*).

Write a class whose constructor accepts the name of a file to read. Initialize a set for movie titles, a set for actor names, and an empty NetworkX Graph, and store them as attributes. Read the file line by line, adding the title to the set of movies and the cast members to the set of actors. Add an edge to the graph between the movie and each cast member.  
(Hint: Use the `split()` method for strings to parse each line.)

It should take no more than 20 seconds to construct the entire graph. Check that there are 137,018 movies and 930,717 actors. Compare parts of your graph to Figure 8.4.

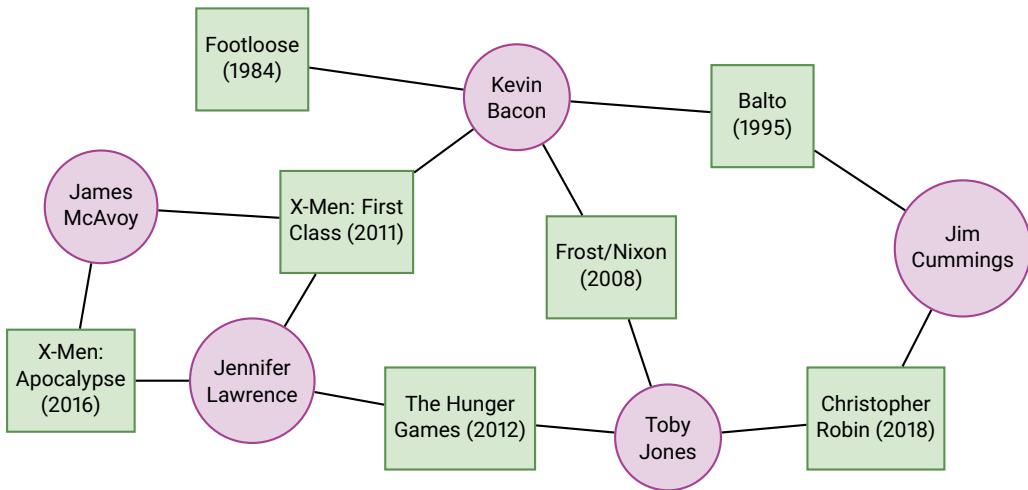


Figure 8.4: A subset of the graph in `movie_data.txt`. Each of these actors have a Bacon number of 1 because they have all been in a movie with Kevin Bacon. Every actor in *The Hunger Games* has a Bacon number of at most 2 because of the paths through Jennifer Lawrence or Toby Jones.

#### NOTE

The movie/actor graph of Problem 4 and Figure 8.4 has an interesting property: actors are only directly connected to movies, and movies are only directly connected to actors. This kind of graph is called *bipartite* because there are two types of nodes, and no node has an edge connecting it to another node of its type.

#### ACHTUNG!

NetworkX Graph objects can be visualized with `nx.draw()` (followed by `plt.show()`). However, this visualization tool is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research. Because of the size of the dataset, **do not** attempt to visualize the graph in Problem 4 with `nx.draw()`.

The Six Degrees of Kevin Bacon game poses an interesting question: can any actor be linked to Kevin Bacon, and if so, in how many steps? The game hypothesizes, “Yes, within 6 steps” (hence the title). More precisely, let the *Bacon number* of an actor be the number of steps from that actor to Kevin Bacon, only counting actors. For example, since Samuel L. Jackson was in a film with Frank Whaley, who was in a film with Kevin Bacon, Samuel L. Jackson has a Bacon number of 2. Actors who have been in a movie with Kevin Bacon have a Bacon number of 1, and actors with no path to Kevin Bacon have a Bacon number of  $\infty$ . The game asserts that the largest Bacon number is 6.

NetworkX is equipped with a variety of graph analysis tools, including a few for computing paths between nodes (and, therefore, Bacon numbers). To compute a shortest path between nodes  $u$  and  $v$ , `nx.shortest_path()` starts one BFS from  $u$  and another from  $v$ , switching off between the two searches until they both discover a common node. This approach is called a *bidirectional BFS* and is typically faster than a regular, one-sided BFS.

Function	Description
<code>has_path()</code>	Return <code>True</code> if there is a path between two specified nodes.
<code>shortest_path()</code>	Return <code>one</code> shortest path between nodes.
<code>shortest_path_length()</code>	Return the length of the shortest path between nodes.
<code>all_shortest_paths()</code>	Yield <code>all</code> shortest paths between nodes.

Table 8.4: NetworkX functions for path problems. Each accepts a `Graph`, then a pair of nodes.

```
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

# Compute the shortest path between 'A' and 'D'.
>>> nx.has_path(G, 'A', 'C')
True
>>> nx.shortest_path(G, 'A', 'C')
['A', 'D', 'C']
>>> nx.shortest_path_length(G, 'A', 'C')
2

# Compute all possible shortest paths between two nodes.
>>> G.add_edge('B', 'C')
>>> list(nx.all_shortest_paths(G, 'A', 'C'))
[['A', 'D', 'C'], ['A', 'B', 'C']]

# When the second node is omitted from these functions, the shortest paths
# from the given node to EVERY node are computed and returned as a dictionary.
>>> nx.shortest_path(G, 'A')
{'A': ['A'], 'D': ['A', 'D'], 'B': ['A', 'B'], 'C': ['A', 'D', 'C']}
>>> nx.shortest_path_length(G, 'A')      # Path lengths are defined by the
{'A': 0, 'D': 1, 'B': 1, 'C': 2}          # number of edges, not nodes.
```

**Problem 5.** Write a method for your class from Problem 4 that accepts two actors' names. Use NetworkX to compute the shortest path between the actors and the degrees of separation between the two actors (if one of the actors is "**Kevin Bacon**", this is the Bacon number of the other actor). Note that this number is different than the number of entries in the actual shortest path list, since the movies are just intermediate steps between actors and are not counted when calculating the degrees of separation.

The idea of a Bacon number provides a few ways to analyze the connectivity of the Hollywood network. For example, the distribution of all Bacon numbers describes how close Kevin Bacon is to actually knowing all of the actors in Hollywood. Someone with a lower average number—for instance, the average *Jackson number*, for Samuel L. Jackson—is, on average, “more connected with Hollywood” than Kevin Bacon. The actor with the lowest average number is sometimes called *the center of the Hollywood universe*.

**Problem 6.** Write a method for your class from Problem 4 that accepts one actor's name. Calculate the shortest path lengths of every actor in the collection to the specified actor (not including movies). Use `plt.hist()` to plot the distribution of path lengths and return the average path length.

(Hint: Use a NetworkX function to compute all path lengths simultaneously; this is significantly faster than calling your method from Problem 5 repeatedly. Also, use the keyword argument `bins=[i-.5 for i in range(8)]` in `plt.hist()` to get the histogram bins to correspond to integers nicely.)

As an aside, the prolific Paul Erdős is the Kevin Bacon equivalent in the mathematical community. Someone with an *Erdős number* of 2 co-authored a paper with someone who co-authored a paper with Paul Erdős. Having an Erdős number of 1 or 2 is considered quite an achievement (see <https://xkcd.com/599/>).

## Additional Material

### Other Hash-based Structures

The standard library has a few specialized alternatives to regular sets and dictionaries.

- `frozenset`: an immutable version of the usual set class. Frozen sets cannot be altered after creation and therefore lack methods like `add()`, `pop()`, and `remove()`, but they can be placed in other sets and used as dictionary keys.
- `collections.defaultdict`: a dictionary with default values. For instance, `defaultdict(set)` creates a dictionary that automatically uses an empty set as the value whenever a non-present key is used for indexing. See <https://docs.python.org/3/library/collections.html> for examples.
- `collections.OrderedDict`: a dictionary that remembers insertion order. For example, the `popitem()` method returns the most recently added key-value pair.

### Depth-first Search

A *depth-first search* (DFS) takes the opposite approach of a BFS. Instead of checking all neighbors of a single node before moving on, it checks the first neighbor, then their first neighbor, then their first neighbor, and so on until reaching a leaf node. The algorithm then backtracks to the previous node and checks its second neighbor. While a DFS is rarely useful for finding shortest paths, it is a common strategy for solving recursively structured problems, such as mazes or Sudoku puzzles.

Consider adding a keyword argument to your method from Problem 2 that specifies whether to use a BFS (the default) or a DFS. To change from a BFS to a DFS, change the visit queue  $Q$  to a stack. You may be able to implement the change in a single line of code.

### The Center of the Hollywood Universe

Computing the center of the universe in a graph amounts to solving Problem 6 for every node in the graph. This is computationally expensive, but since each average number is independent of the others, the problem is a good candidate for *parallel programming*, which divides the computational workload between multiple processes. Even with parallelism, however, computing the center of the Hollywood universe may require significant computational time and resources.

### Shortest Paths on Weighted Graphs

The graphs presented in this lab are *unweighted*, meaning all edges have the same importance. A *weighted graph* assigns a weight to each edge, which can usually be thought of as the distance between the two adjacent nodes. The shortest path problem becomes much more complicated on weighted graphs, and requires additions to the plain BFS. The standard approach is *Dijkstra's algorithm*, which is implemented as `nx.dijkstra_path()`. Another approach, the *Bellman-Ford algorithm*, is implemented as `nx.bellman_ford_path()`.

# 9

# Markov Chains

**Lab Objective:** A Markov chain is a collection of states with specified probabilities for transitioning from one state to another. They are characterized by the fact that the future behavior of the system depends only on its current state. In this lab we learn to construct, analyze, and interact with Markov chains, then use a Markov-based approach to simulate natural language.

## State Space Models

Many systems can be described by a finite number of *states*. For example, a board game where players move around the board based on dice rolls can be modeled by a Markov chain. Each space represents a state, and a player is said to be in a state if their piece is currently on the corresponding space. In this case, the probability of moving from one space to another only depends on the player's current location; where the player was on a previous turn does not affect their current turn.

Markov chains with a finite number of states have an associated *transition matrix* that stores the information about the possible transitions between the states in the chain. The  $(i, j)$ th entry of the matrix gives the probability of moving **from state  $j$  to state  $i$** . Thus, each of the columns of the transition matrix sum to 1.

### NOTE

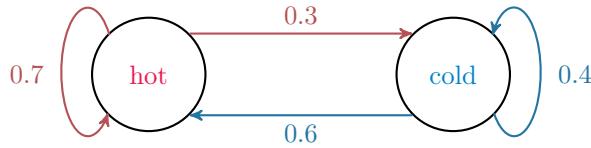
A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the  $(i, j)$ th entry of the matrix is the probability of moving from state  $i$  to state  $j$ . Both representations are common, but in this lab we exclusively use column stochastic transition matrices for consistency.

Consider a very simple weather model in which the weather tomorrow depends only on the weather today. For now, we consider only two possible weather states: hot and cold. Suppose that if today is hot, then the probability that tomorrow is also hot is 0.7, and that if today is cold, the probability that tomorrow is also cold is 0.4. By assigning "hot" to the 0th row and column, and "cold" to the 1st row and column, this Markov chain has the following transition matrix.

$$\begin{array}{cc} & \text{hot today} \quad \text{cold today} \\ \text{hot tomorrow} & \left[ \begin{array}{cc} 0.7 & 0.6 \\ 0.3 & 0.4 \end{array} \right] \\ \text{cold tomorrow} & \end{array}$$

The 0th column of the matrix says that if it is hot today, there is a 70% chance that tomorrow will be hot (0th row) and a 30% chance that tomorrow will be cold (1st row). The 1st column says if it is cold today, then there is a 60% chance of heat and a 40% chance of cold tomorrow.

Markov chains can be represented by a *state diagram*, a type of directed graph. The nodes in the graph are the states, and the edges indicate the state transition probabilities. The Markov chain described above has the following state diagram.



**Problem 1.** Define a `MarkovChain` class whose constructor accepts an  $n \times n$  transition matrix  $A$  and, optionally, a list of state labels. If  $A$  is not column stochastic, raise a `ValueError`. Construct a dictionary mapping the state labels to the row/column index that they correspond to in  $A$  (given by order of the labels in the list), and save  $A$ , the list of labels, and this dictionary as attributes. If there are no state labels given, use the labels  $[0 \ 1 \ \dots \ n - 1]$ .

For example, for the weather model described above, the transition matrix is

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix},$$

the list of state labels is `["hot", "cold"]`, and the dictionary mapping labels to indices is `{"hot":0, "cold":1}`. This Markov chain could be also represented by the transition matrix

$$\tilde{A} = \begin{bmatrix} 0.4 & 0.3 \\ 0.6 & 0.7 \end{bmatrix},$$

the labels `["cold", "hot"]`, and the resulting dictionary `{"cold":0, "hot":1}`.

## Simulating State Transitions

Simulating the weather model described above requires a programmatic way of choosing between the outgoing transition probabilities of each state. For example, if it is cold today, we could flip a weighted coin that lands on tails 60% of the time (guess tomorrow is hot) and heads 40% of the time (guess tomorrow is cold) to predict the weather tomorrow. The *Bernoulli distribution* with parameter  $p = 0.4$  simulates this behavior: 60% of draws are 0, and 40% of draws are a 1.

A *binomial distribution* is the sum several Bernoulli draws: one binomial draw with parameters  $n$  and  $p$  indicates the number of successes out of  $n$  independent experiments, each with probability  $p$  of success. In other words,  $n$  is the number of times to flip the coin, and  $p$  is the probability that the coin lands on heads. Thus, a binomial draw with  $n = 1$  is a Bernoulli draw.

NumPy does not have a function dedicated to drawing from a Bernoulli distribution; instead, use the more general `np.random.binomial()` with  $n = 1$  to make a Bernoulli draw.

```
>>> import numpy as np

# Draw from the Bernoulli distribution with p = .5 (flip one fair coin).
>>> np.random.binomial(n=1, p=.5)
0                                         # The coin flip resulted in tails.

# Draw from the Bernoulli distribution with p = .3 (flip one weighted coin).
>>> np.random.binomial(n=1, p=.3)
0                                         # Also tails.
```

For the weather model, if the “cold” state corresponds to row and column 1 in the transition matrix,  $p$  should be the probability that tomorrow is cold. So, if today is cold, select  $p = 0.4$ ; if today is hot, set  $p = 0.3$ . Then draw from the binomial distribution with  $n = 1$  and the selected  $p$ . If the result is 0, transition to the “hot” state; if the result is 1, stay in the “cold” state.

Using Bernoulli draws to determine state transitions works for Markov chains with two states, but larger Markov chains require draws from a *categorical distribution*, a multivariate generalization of the Bernoulli distribution. A draw from a categorical distribution with parameters  $(p_1, p_2, \dots, p_k)$  satisfying  $\sum_{i=1}^k p_i = 1$  indicates which of  $k$  outcomes occurs. If  $k = 2$ , a draw simulates a coin flip (a Bernoulli draw); if  $k = 6$ , a draw simulates rolling a six-sided die. Just as the Bernoulli distribution is a special case of the binomial distribution, the categorical distribution is a special case of the *multinomial distribution* which indicates how many times each of the  $k$  outcomes occurs in  $n$  repeated experiments. Use `np.random.multinomial()` with  $n = 1$  to make a categorical draw.

```
# Draw from the categorical distribution (roll a fair four-sided die).
>>> np.random.multinomial(1, np.array([1./4, 1./4, 1./4, 1./4]))
array([0, 0, 0, 1])      # The roll resulted in a 3.

# Draw from another categorical distribution (roll a weighted four-sided die).
>>> np.random.multinomial(1, np.array([.5, .3, .2, 0]))
array([0, 1, 0, 0])      # The roll resulted in a 1.
```

Consider a four-state weather model with the transition matrix

$$\begin{array}{ccccc} & \text{hot} & \text{mild} & \text{cold} & \text{freezing} \\ \text{hot} & \left[ \begin{matrix} 0.5 & 0.3 & 0.1 & 0 \end{matrix} \right] \\ \text{mild} & \left[ \begin{matrix} 0.3 & 0.3 & 0.3 & 0.3 \end{matrix} \right] \\ \text{cold} & \left[ \begin{matrix} 0.2 & 0.3 & 0.4 & 0.5 \end{matrix} \right] \\ \text{freezing} & \left[ \begin{matrix} 0 & 0.1 & 0.2 & 0.2 \end{matrix} \right] \end{array}.$$

If today is hot, the probabilities of transitioning to each state are given by the “hot” column of the transition matrix. Therefore, to choose a new state, draw from the categorical distribution with parameters  $(0.5, 0.3, 0.2, 0)$ . The result  $[0 \ 1 \ 0 \ 0]$  indicates a transition to the state corresponding to the 1st row and column (tomorrow is mild), while the result  $[0 \ 0 \ 1 \ 0]$  indicates a transition to the state corresponding to the 2nd row and column (tomorrow is cold). In other words, the position of the 1 tells which column of the matrix to use as the parameters for the next categorical draw.

**Problem 2.** Write a method for the `MarkovChain` class that accepts a single state label. Use the label-to-index dictionary to determine the column of  $A$  that corresponds to the provided state label, then draw from the corresponding categorical distribution to choose a state to transition to. Return the corresponding label of the new state (not its index).

(Hint: `np.argmax()` may be useful.)

**Problem 3.** Add the following methods to the `MarkovChain` class.

- `walk()`: Accept a state label and an integer  $N$ . Starting at the specified state, use your method from Problem 2 to transition from state to state  $N - 1$  times, recording the state label at each step. Return the list of  $N$  state labels, including the initial state.
- `path()`: Accept labels for an initial state and an end state. Beginning at the initial state, transition from state to state until arriving at the specified end state, recording the state label at each step. Return the list of state labels, including the initial and final states.

Test your methods on the two-state and four-state weather models described previously.

## General State Distributions

For a Markov chain with  $n$  states, the probability of being in each state can be encoded by a  $n$ -vector  $\mathbf{x}$ , called a *state distribution vector*. The entries of  $\mathbf{x}$  must be nonnegative and sum to 1, and the  $i$ th entry  $x_i$  of  $\mathbf{x}$  is the probability of being in state  $i$ . For example, the state distribution vector  $\mathbf{x} = [0.8 \ 0.2]^\top$  corresponding to the 2-state weather model indicates an 80% chance that today is hot and a 20% chance that today is cold. On the other hand, the vector  $\mathbf{x} = [0 \ 1]^\top$  implies that today is, with 100% certainty, cold.

If  $A$  is a transition matrix for a Markov chain with  $n$  states and  $\mathbf{x}$  is a corresponding state distribution vector, then  $A\mathbf{x}$  is also a state distribution vector. In fact, if  $\mathbf{x}_k$  is the state distribution vector corresponding to a certain time  $k$ , then  $\mathbf{x}_{k+1} = A\mathbf{x}_k$  contains the probabilities of being in each state after allowing the system to transition again. For the weather model, this means that if there is an 80% chance that it will be hot 5 days from now, written  $\mathbf{x}_5 = [0.8 \ 0.2]^\top$ , then since

$$\mathbf{x}_6 = A\mathbf{x}_5 = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix} \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.68 \\ 0.32 \end{bmatrix},$$

there is a 68% chance that 6 days from now will be a hot day.

## Convergent Transition Matrices

Given an initial state distribution vector  $\mathbf{x}_0$ , defining  $\mathbf{x}_{k+1} = A\mathbf{x}_k$  yields the significant relation

$$\mathbf{x}_k = A\mathbf{x}_{k-1} = A(A\mathbf{x}_{k-2}) = A(A(A\mathbf{x}_{k-3})) = \cdots = A^k\mathbf{x}_0.$$

This indicates that the  $(i, j)$ th entry of  $A^k$  is the probability of transition from state  $j$  to state  $i$  in  $k$  steps. For the transition matrix of the 2-state weather model, a pattern emerges in  $A^k$  for even small values of  $k$ :

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0.67 & 0.66 \\ 0.33 & 0.34 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0.667 & 0.666 \\ 0.333 & 0.334 \end{bmatrix}.$$

As  $k \rightarrow \infty$ , the entries of  $A^k$  converge, written

$$\lim_{k \rightarrow \infty} A^k = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix}. \quad (9.1)$$

In addition, for any initial state distribution vector  $\mathbf{x}_0 = [a, b]^\top$  (meaning  $a, b \geq 0$  and  $a + b = 1$ ),

$$\lim_{k \rightarrow \infty} \mathbf{x}_k = \lim_{k \rightarrow \infty} A^k \mathbf{x}_0 = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 2(a+b)/3 \\ (a+b)/3 \end{bmatrix} = \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix}.$$

Thus,  $\mathbf{x}_k \rightarrow \mathbf{x} = [2/3 \ 1/3]^\top$  as  $k \rightarrow \infty$ , regardless of the initial state distribution  $\mathbf{x}_0$ . So, according to this model, no matter the weather today, the probability that it is hot a week from now is approximately 66.67%. In fact, approximately 2 out of 3 days in the year should be hot.

## Steady State Distributions

The state distribution  $\mathbf{x} = [2/3 \ 1/3]^\top$  has another important property:

$$A\mathbf{x} = \begin{bmatrix} 7/10 & 3/5 \\ 3/10 & 2/5 \end{bmatrix} \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 14/30 + 3/15 \\ 6/30 + 2/15 \end{bmatrix} = \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix} = \mathbf{x}.$$

Any  $\mathbf{x}$  satisfying  $A\mathbf{x} = \mathbf{x}$  is called a *steady state distribution* or a *stable fixed point* of  $A$ . In other words, a steady state distribution is an eigenvector of  $A$  corresponding to the eigenvalue  $\lambda = 1$ .

Every finite Markov chain has at least one steady state distribution. If some power  $A^k$  of  $A$  has all positive (nonzero) entries, then the steady state distribution is unique.<sup>1</sup> In this case,  $\lim_{k \rightarrow \infty} A^k$  is the matrix whose columns are all equal to the unique steady state distribution, as in (9.1). Under these circumstances, the steady state distribution  $\mathbf{x}$  can be found by iteratively calculating  $\mathbf{x}_{k+1} = A\mathbf{x}_k$ , as long as the initial vector  $\mathbf{x}_0$  is a state distribution vector.

### ACHTUNG!

Though every Markov chain has at least one steady state distribution, the procedure described above fails if  $A^k$  fails to converge. For instance, consider the transition matrix

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad A^k = \begin{cases} A & \text{if } k \text{ is odd} \\ I & \text{if } k \text{ is even.} \end{cases}$$

In this case as  $k \rightarrow \infty$ ,  $A^k$  oscillates between two different matrices.

Furthermore, the steady state distribution is not always unique; the transition matrix defined above, for example, has infinitely many.

<sup>1</sup>This is a consequence of the *Perron-Frobenius theorem*, which is presented in detail in Volume 1.

**Problem 4.** Write a method for the `MarkovChain` class that accepts a convergence tolerance `tol` and a maximum number of iterations `maxiter`. Generate a random state distribution vector  $\mathbf{x}_0$  and calculate  $\mathbf{x}_{k+1} = A\mathbf{x}_k$  until  $\|\mathbf{x}_{k-1} - \mathbf{x}_k\|_1 < \text{tol}$ , where  $A$  is the transition matrix saved in the constructor. If  $k$  exceeds `maxiter`, raise a `ValueError` to indicate that  $A^k$  does not converge. Return the approximate steady state distribution  $\mathbf{x}$  of  $A$ .

To test your function, generate a random transition matrix  $A$ . Verify that  $A\mathbf{x} = \mathbf{x}$  and that the columns of  $A^k$  approach  $\mathbf{x}$  as  $k \rightarrow \infty$ . To compute  $A^k$ , use NumPy's (very efficient) algorithm for computing matrix powers.

```
>>> A = np.array([[.7, .6], [.3, .4]])
>>> np.linalg.matrix_power(A, 10)           # Compute A^10.
array([[ 0.66666667,  0.66666667],
       [ 0.33333333,  0.33333333]])
```

Finally, use your method to validate the results of Problem 3: for the two-state and four-state weather models,

1. Calculate the steady state distribution corresponding to the transition matrix.
2. Run a weather simulation for a large number of days using `walk()` and verify that the results match the steady state distribution (for example, approximately 2/3 of the days should be hot for the two-state model).

#### NOTE

Problem 4 is a special case of the *power method*, an algorithm for calculating an eigenvector of a matrix corresponding to the eigenvalue of largest magnitude. The general power method, together with a discussion of its convergence conditions, is discussed in Volume 1.

## Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study *natural languages*, meaning spoken or written languages like English or Russian [VHL06]. In the early 20th century, Markov used his chains to model how Russian switched from vowels to consonants. By mid-century, they had been used as an attempt to model English. It turns out that plain Markov chains are, by themselves, insufficient to model or produce very good English. However, they can approach a fairly good model of bad English, with sometimes amusing results.

By nature, a Markov chain is only concerned with its current state, not with previous states. A Markov chain simulating transitions between English words is therefore completely unaware of context or even of previous words in a sentence. For example, if a chain's current state is the word "continuous," the chain may say that the next word in a sentence is more likely to be "function" rather than "raccoon." However the phrase "continuous function" may be gibberish in the context of the rest of the sentence.

## Generating Random Sentences

Consider the problem of generating English sentences that are similar to the text contained in a specific file, called the *training set*. The goal is to construct a Markov chain whose states and transition probabilities represent the vocabulary and—hopefully—the style of the source material. There are several ways to approach this problem, but one simple strategy is to assign each unique word in the training set to a state, then construct the transition probabilities between the states based on the ordering of the words in the training set. To indicate the beginning and end of a sentence requires two extra states: a *start state*, `$start`, marking the beginning of a sentence; and a *stop state*, `$stop`, marking the end. The start state should only transitions to words that appear at the beginning of a sentence in the training set, and only words that appear at the end a sentence in the training set should transition to the stop state.

Consider the following small training set, paraphrased from Dr. Seuss [Gei60].

```
I am Sam Sam I am.  
Do you like green eggs and ham?  
I do not like them, Sam I am.  
I do not like green eggs and ham.
```

There are 15 unique words in this training set, including punctuation (so "ham?" and "ham." are counted as distinct words) and capitalization (so "Do" and "do" are also different):

```
I am Sam am. Do you like green  
eggs and ham? do not them, ham.
```

With start and stop states, the transition matrix should be  $17 \times 17$ . Each state must be assigned a row and column index in the transition matrix, for example,

<code>\$start</code>	I	am	Sam	...	ham.	<code>\$stop</code>
0	1	2	3	...	15	16

The  $(i, j)$ th entry of the transition matrix  $A$  should be the probability that word  $j$  is followed by word  $i$ . For instance, the word "Sam" is followed by the words "Sam" once and "I" twice in the training set, so the state corresponding to "Sam" (index 3) should transition to the state for "Sam" with probability  $1/3$ , and to the state for "I" (index 1) with probability  $2/3$ . That is,  $A_{3,3} = 1/3$ ,  $A_{1,3} = 2/3$ , and  $A_{i,3} = 0$  for  $i \notin \{1, 3\}$ . Similarly, the start state should transition to the state for "I" with probability  $3/4$ , and to the state for "Do" with probability  $1/4$ ; the states for "am.", "ham?", and "ham." should each transition to the stop state.

To construct the transition matrix, parse the training set and add 1 to  $A_{i,j}$  every time word  $j$  is followed by word  $i$ , in this case arriving at the matrix

$$\begin{array}{ccccccccc} & \text{\$start} & \text{I} & \text{am} & \text{Sam} & & \text{ham.} & & \text{\$stop} \\ \text{\$start} & \left[ \begin{array}{ccccccccc} 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3 & 0 & 0 & 2 & \dots & 0 & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 \end{array} \right] \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \vdots \\ \text{ham.} \\ \text{\$stop} \end{array}.$$

To avoid a column of zeros, set  $A_{j,j} = 1$  where  $j$  is the index of the stop state (so the stop state always transitions to itself). Next, divide each column by its sum so that each column sums to 1:

$$\begin{array}{ccccccccc} & \text{\$start} & \text{I} & \text{am} & \text{Sam} & & \text{ham.} & & \text{\$stop} \\ \text{\$start} & \left[ \begin{array}{ccccccccc} 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3/4 & 0 & 0 & 2/3 & \dots & 0 & 0 \\ 0 & 1/5 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1/3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \end{array} \right] \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \vdots \\ \text{ham.} \\ \text{\$stop} \end{array}.$$

The  $3/4$  indicates that 3 out of 4 times, the sentences in the training set start with the word “T”. Similarly, the  $2/3$  and  $1/3$  says that “Sam” is followed by “T” twice and by “Sam” once in the training set. Note that “am” (without a period) always transitions to “Sam” and that “ham.” (with a period) always transitions the stop state.

The entire procedure of creating the transition matrix for the Markov chain with words from a file as states is summarized below.

---

**Algorithm 1** Convert a training set of sentences into a Markov chain.

---

- 1: **procedure** MAKETRANSITIONMATRIX(*filename*)
  - 2:     Read the training set from the file *filename*.
  - 3:     Get the set of unique words in the training set (the state labels).
  - 4:     Add labels “**\$start**” and “**\$stop**” to the set of states labels.
  - 5:     Initialize an appropriately sized square array of zeros to be the transition matrix.
  - 6:     **for** each sentence in the training set **do**
  - 7:         Split the sentence into a list of words.
  - 8:         Prepend “**\$start**” and append “**\$stop**” to the list of words.
  - 9:         **for** each consecutive pair  $(x,y)$  of words in the list of words **do**
  - 10:             Add 1 to the entry of the transition matrix that corresponds to transitioning from state  $x$  to state  $y$ .
  - 11:     Make sure the stop state transitions to itself.
  - 12:     Normalize each column by dividing by the column sums.
-

**Problem 5.** Write a class called `SentenceGenerator` that inherits from the `MarkovChain` class. The constructor should accept a filename (the training set). Read the file and build a transition matrix from its contents as described in Algorithm 1. Save the same attributes as the constructor of `MarkovChain` does so that inherited methods work correctly. Assume that the training set has one complete sentence written on each line.  
 (Hint: if the contents of the file are in the string `s`, then `s.split()` is the list of words and `s.split('n')` is the list of sentences.)

#### NOTE

The Markov chains that result from the procedure in Problem 5 have a few interesting structural characteristics. The stop state is a *sink*, meaning it only transitions to itself. Because of this, and since every node has a path to the stop state, any traversal of the chain will end up in the stop state forever. The stop state is therefore called an *absorbing state*, and the chain as a whole is called an *absorbing Markov chain*. Furthermore, the steady state is the vector with a 1 in the entry corresponding to the stop state and 0s everywhere else.

**Problem 6.** Add a method to the `SentenceGenerator` class called `babble()`. Use the `path()` method from Problem 3 to generate a random sentence based on the training document. That is, generate a path from the start state to the stop state, remove the `"$start"` and `"$stop"` labels from the path, and join the resulting list together into a single, space-separated string.

For example, your `SentenceGenerator` class should be able to create random sentences that sound somewhat like Yoda speaking.

```
>>> yoda = SentenceGenerator("yoda.txt")
>>> for _ in range(3):
...     print(yoda.babble())
...
Impossible to my size, do not!
For eight hundred years old to enter the dark side of Congress there is.
But beware of the Wookiees, I have.
```

## Additional Material

### Other Applications of Markov Chains

Markov chains are a useful way to study many probabilistic phenomena, so they have a wide variety of applications. The following are just a few that are covered in other parts of this lab manual series.

- **PageRank:** Google’s PageRank algorithm uses a Markov chain-based approach to rank web pages. The main idea is to use the entries of the steady state vector as a measure of importance for the corresponding states. For example, the steady state  $\mathbf{x} = [2/3 \ 1/3]^T$  for the two-state weather model means that the hot state is “more important” (occurs more frequently) than the cold state. See the PageRank lab in Volume 1.
- **MCMC Sampling:** A *Monte Carlo Markov Chain* (MCMC) method constructs a Markov chain whose steady state is a probability distribution that is difficult to sample from directly. This provides a way to sample from nontrivial or abstract distributions. Many MCMC methods are used in various fields, from machine learning to physics. See the Volume 3 lab on the Metropolis-Hastings algorithm.
- **Hidden Markov Models:** The Markov chain simulations in this lab use an initial condition (a state distribution vector  $\mathbf{x}_0$ ) and known transition probabilities to make predictions forward in time. Conversely, a *hidden Markov model* (HMM) assumes that a given set of observations are the result of a Markov process, then uses those observations to infer the corresponding transition probabilities. Hidden Markov models are used extensively in modern machine learning, especially for speech and language processing. See the Volume 3 lab on Speech Recognition.

### Large Training Sets

The approach in Problems 5 and 6 begins to fail as the training set grows larger. For example, a single Shakespearean play may not be large enough to cause memory problems, but *The Complete Works of William Shakespeare* certainly will.

To accommodate larger data sets, consider use a sparse matrix from `scipy.sparse` for the transition matrix instead of a regular NumPy array. Specifically, construct the transition matrix as a `lil_matrix` (which is easy to build incrementally), then convert it to the `csc_matrix` format (which supports fast column operations). Ensure that the process still works on small training sets, then proceed to larger training sets. How are the resulting sentences different if a very large training set is used instead of a small training set?

### Variations on the English Model

Choosing a different state space for the English Markov model produces different results. Consider modifying the `SentenceGenerator` class so that it can determine the state space in a few different ways. The following ideas are just a few possibilities.

- Let each punctuation mark have its own state. In the Dr. Seuss training set, instead of having two states for the words “ham?” and “ham.”, there would be three states: “ham”, “?”, and “.”, with “ham” transitioning to both punctuation states.
- Model paragraphs instead of sentences. Add a `$startParagraph` state that always transitions to `$startSentence` and a `$stopParagraph` state that is sometimes transitioned to by `$stopSentence`.

- Let the states be individual letters instead of individual words. Be sure to include a state for the spaces between words.
- Construct the state space so that the next state depends on both the current and previous states. This kind of Markov chain is called a *Markov chain of order 2*. This way, every set of three consecutive words in a randomly generated sentence should be part of the training set, as opposed to only every consecutive pair of words coming from the set.
- Instead of generating random sentences from a single source, simulate a random conversation between  $n$  people. Construct a Markov chain  $M_i$ , for each person,  $i = 1, \dots, n$ , then create a Markov chain  $C$  describing the conversation transitions from person to person; in other words, the states of  $C$  are the  $M_i$ . To create the conversation, generate a random sentence from the first person using  $M_1$ . Then use  $C$  to determine the next speaker, generate a random sentence using their Markov chain, and so on.

## Natural Language Processing Tools

The Markov model of Problems 5 and 6 is a *natural language processing* application. Python's `nltk` module (natural language toolkit) has many tools for parsing and analyzing text for these kinds of problems [BL04]. For example, `nltk.sent_tokenize()` reads a single string and splits it up into sentences. This could be useful, for example, in making the `SentenceGenerator` class compatible with files that do not have one sentence per line.

```
>>> from nltk import sent_tokenize
>>> with open("yoda.txt", 'r') as yoda:
...     sentences = sent_tokenize(yoda.read())
...
>>> print(sentences)
['Away with your weapon!',
 'I mean you no harm.',
 'I am wondering - why are you here?',
 ...]
```

The `nltk` module is **not** part of the Python standard library. For instructions on downloading, installing, and using `nltk`, visit <http://www.nltk.org/>.



# 10 Unix Shell 2

**Lab Objective:** *Introduce system management, calling Unix Shell commands within Python, and other advanced topics. As in the last Unix lab, the majority of learning will not be had in finishing the problems, but in following the examples.*

## Archiving and Compression

In file management, the terms archiving and compressing are commonly used interchangeably. However, these are quite different. Archiving is combining a certain number of files into one file. The resulting file will be the same size as the group of files that were archived. Compressing takes a file or group of files and shrinks the file size as much as possible. The resulting compressed file will need to be extracted before being used.

The ZIP file format is common for archiving and compressing files. If the zip Unix command is not installed on your system, you can download it by running

```
>>> sudo apt-get install zip
```

Note that you will need to have administrative rights to download this package. To unzip a file, use `unzip`.

### NOTE

To begin this lab, unzip the `Shell2.zip` file into your `UnixShell2/` directory using a terminal command.

```
# Unzip a zipped file using the unzip command.  
$ unzip Shell2.zip  
Archive Shell2.zip  
  creating: Shell2/  
  creating: Shell2/Test/  
  inflating: Shell2/.DS_Store  
  creating: Shell2/Scripts/
```

```
extracting: Shell2/Scripts/fifteen_secs
extracting: Shell2/Scripts/script3
extracting: Shell2/Scripts/hello.sh...
```

While the **zip** file format is more popular on the Windows platform, the **tar** utility is more common in the Unix environment.

#### NOTE

When submitting this lab, you will need to archive and compress your entire **Shell2/** directory into a file called **Shell2.tar.gz** and push **Shell2.tar.gz** as well as **shell2.py** to your online repository.

If you are doing multiple submissions, make sure to delete your previous **Shell2.tar.gz** file before creating a new one from your modified **Shell2/** directory. Refer to **Unix1** for more information on deleting files.

As a final note, please do not push the entire directory to your online repository. *Only* push **ShellFinal.tar.gz** and **shell2.py**.

The example below demonstrates how to archive and compress our **Shell2/** directory. The **-z** flag calls for the **gzip** compression tool, the **-v** flag calls for a verbose output, the **-p** flag tells the tool to preserve file permission, and the **-f** flag indicates the next parameter will be the name of the archive file. Note that the **-f** flag must always come last.

```
# Remove your archive tar.gz file if you already have one.
$ rm -v Shell2.tar.gz
removed 'Shell2.tar.gz'

# Create a new one from your update Shell2 directory content.
# Remember that the * is the wildcard that represents all strings.
$ tar -zcpf Shell2.tar.gz Shell2/*
```

## Working with Files

### Displaying File Contents

The unix file system presents many opportunities for the manipulation, viewing, and editing of files. Before moving on to more complex commands, we will look at some of the commands available to view the content of a file.

The **cat** command, followed by the filename, will display all the contents of a file on the terminal screen. This can be problematic if you are dealing with a large file. There are a few available commands to control the output of **cat** in the terminal. See Table 10.1.

As an example, use **less <filename>** to restrict the number of lines that are shown. With this command, use the arrow keys to navigate up and down and press **q** to exit.

Command	Description
<code>cat</code>	Print all of the file contents
<code>more</code>	Print the file contents one page at a time, navigating forwards
<code>less</code>	Like more, but you navigate forward and backwards
<code>head</code>	Print the first 10 lines of a file
<code>head -nk</code>	Print the first $k$ lines of a file
<code>tail</code>	Print the last 10 lines of a file
<code>tail -nk</code>	Print the last $k$ lines of a file

Table 10.1: Commands for printing file contents

## Pipes and redirects

To combine terminal commands, we can use *pipes*. When we combine or *pipe* commands, the output of one command is passed to the other. We pipe commands together using the `| (bar)` operator. In the example directly below, the `cat` command output is piped to `wc -l` (`wc` stands for word count, and the `-l` flag tells the `wc` command to count lines).

In the second part of the example, `ls -s` is piped to `sort -nr`. Refer to the *Unix 1* lab for explanations of `ls` and `sort`. Recall that the `man` command followed by an additional command will output details on the additional command's possible flags and what they mean (for example `man sort`).

```
$ cd Shell2/Files/Feb
# Output the number of lines in assignments.txt.
$ cat assignments.txt | wc -l
9
# Sort the files by file size and output file names and their size.
$ls -s | sort -nr
4 project3.py
4 project2.py
4 assignments.txt
4 pics
total 16
```

In addition to *piping* commands together, when working with files specifically, we can use *redirects*. A *redirect*, represented as `<` in the terminal, passes the file to a terminal command.

To save a command's output to a file, we can use `>` or `>>`. The `>` operator will overwrite anything that may exist in the output file whereas `>>` will append the output to the end of the output file. Examples of *redirects* and writing to a file are given below.

```
# Gets the same result as the first command in the above example.
$ wc -l < assignments.txt
9
# Writes the number of lines in the assignments.txt file to word_count.txt.
$ wc -l < assignments.txt >> word_count.txt
```

**Problem 1.** The `words.txt` file in the `Documents/` directory contains a list of words that are not in alphabetical order. Write an alphabetically sorted list of words in `words.txt` to a new file in your `Documents/` called `sortedwords.txt` using pipes and redirects. After you write the alphabetized words to the designated file, also write the number of words in `words.txt` to the end of `sortedwords.txt`. Save this file in the `Documents/` directory. Try to accomplish this with a total of two commands or fewer.

## Resource Management

To be able to optimize performance, it is valuable to be aware of the resources, specifically hard drive space and computer memory, being used.

### Job Control

One way to monitor and optimize performance is in job control. Any time you start a program in the terminal (you could be running a script, opening ipython, etc.,) that program is called a **job**. You can run a job in the foreground and also in the background. When we run a program in the foreground, we see and interact with it. Running a script in the foreground means that we will not be able to enter any other commands in the terminal while the script is running. However, if we choose to run it in the background, we can enter other commands and continue interacting with other programs while the script runs.

Consider the scenario where we have multiple scripts that we want to run. If we know that these scripts will take awhile, we can run them all in the background while we are working on something else. Table 10.2 lists some common commands that are used in job control. We strongly encourage you to experiment with some of these commands.

Command	Description
<code>COMMAND &amp;</code>	Adding an ampersand to the end of a command runs the command in the background
<code>bg %N</code>	Restarts the Nth interrupted job in the background
<code>fg %N</code>	Brings the Nth job into the foreground
<code>jobs</code>	Lists all the jobs currently running
<code>kill %N</code>	Terminates the Nth job
<code>ps</code>	Lists all the current processes
<code>Ctrl-C</code>	Terminates current job
<code>Ctrl-Z</code>	Interrupts current job
<code>nohup</code>	Run a command that will not be killed if the user logs out

Table 10.2: Job control commands

The `fifteen_secs` and `five_secs` scripts in the `Scripts/` directory take fifteen seconds and five seconds to execute respectively. The python file `fifteen_secs.py` in the `Python/` directory takes fifteen seconds to execute, this file counts to fifteen and then outputs "Success!". These will be particularly useful as you are experimenting with these commands.

Remember, that when you use the `./` command in place of other commands you will probably need to change permissions. For more information on changing permissions, review *Unix 1*. Run the following command sequence from the `Shell2` directory.

```

# Remember to add executing permissions to the user.
$ ./Scripts/fifteen_secs &
$ python Python/fifteen_secs.py &
$ jobs
[1]+  Running      ./Scripts/fifteen_secs &
[2]-  Running      python Python/fifteen_secs.py &
$ kill %1
[1]-  Terminated  ./Scripts/fifteen_secs &
$ jobs
[1]+  Running      python Python/fifteen_secs.py &
# After the python script finishes it outputs the results.
$ Success!
# To move on, click enter after "Success!" appears in the terminal.

# List all current processes
$ ps
  PID TTY      TIME CMD
    6 tty1    00:00:00 bash
   44 tty1    00:00:00 ps

$ ./Scripts/fifteen_secs &
$ ps
  PID TTY      TIME CMD
    6 tty1    00:00:00 bash
   59 tty1    00:00:00 fifteen_secs
   60 tty1    00:00:00 sleep
   61 tty1    00:00:00 ps

# Stop fifteen_secs
$ kill 59
$ ps
  PID TTY      TIME CMD
    6 tty1    00:00:00 bash
   60 tty1    00:00:00 sleep
   61 tty1    00:00:00 ps
[1]+  Terminated      ./fifteen_secs

```

**Problem 2.** In addition to the `five_secs` and `fifteen_secs` scripts, the `Scripts/` folder contains three scripts (named `script1`, `script2`, and `script3`) that each take about forty-five seconds to execute. From the `Scripts` directory, execute each of these commands in the background in the following order; `script1`, `script2`, and `script3`. Do this so all three are running at the same time. While they are all running, write the output of `jobs` to a new file `log.txt` saved in the `Scripts/` directory.

(Hint: In order to get the same output as the solutions file, you need to run the `./` command and not the `bash` command.)

## Using Python for File Management

### OS and Glob

Bash has control flow tools like if-else blocks and loops, but most of the syntax is highly unintuitive. Python, on the other hand, has extremely intuitive syntax for these control flow tools, so using Python to do shell-like tasks can result in some powerful but specific file management programs. Table 10.3 relates some of the common shell commands to Python functions, most of which come from the `os` module in the standard library.

Shell Command	Python Function
<code>ls</code>	<code>os.listdir()</code>
<code>cd</code>	<code>os.chdir()</code>
<code>pwd</code>	<code>os.getcwd()</code>
<code>mkdir</code>	<code>os.mkdir(), os.mkdirs()</code>
<code>cp</code>	<code>shutil.copy()</code>
<code>mv</code>	<code>os.rename(), os.replace()</code>
<code>rm</code>	<code>os.remove(), shutil.rmtree()</code>
<code>du</code>	<code>os.path.getsize()</code>
<code>chmod</code>	<code>os.chmod()</code>

Table 10.3: Shell-Python compatibility

In addition to these, Python has a few extra functions that are useful for file management and shell commands. See Table 10.4. The two functions `os.walk()` and `glob.glob()` are especially useful for doing searches like `find` and `grep`. Look at the example below and then try out a few things on your own to try to get a feel for them.

Function	Description
<code>os.walk()</code>	Iterate through the subfolders and subfolder files of a given directory.
<code>os.path.isdir()</code>	Return <code>True</code> if the input is a directory.
<code>os.path.isfile()</code>	Return <code>True</code> if the input is a file.
<code>os.path.join()</code>	Join several folder names or file names into one path.
<code>glob.glob()</code>	Return a list of file names that match a pattern.
<code>subprocess.call()</code>	Execute a shell command.
<code>subprocess.check_output()</code>	Execute a shell command and return its output as a string.

Table 10.4: Other useful Python functions for shell operations.

```
# Your output may differ from the example's output.
>>> import os
>>> from glob import glob

# Get the names of all Python files in the Python/ directory.
>>> glob("Python/*.py")
['Python/calc.py',
 'Python/count_files.py',
 'Python/fifteen_secs.py',
 'Python/mult.py',
```

```
'Python/project.py']

# Get the names of all .jpg files in any subdirectory.
# The recursive parameter lets '**' match more than one directory.
>> glob("/**/*.jpg", recursive=True)
['Photos/IMG_1501.jpg',
 'Photos/img_1879.jpg',
 'Photos/IMG_2164.jpg',
 'Photos/IMG_2379.jpg',
 'Photos/IMG_2182.jpg',
 'Photos/IMG_1510.jpg',
 'Photos/IMG_2746.jpg',
 'Photos/IMG_2679.jpg',
 'Photos/IMG_1595.jpg',
 'Photos/IMG_2044.jpg',
 'Photos/img_1796.jpg',
 'Photos/IMG_2464.jpg',
 'Photos/img_1987.jpg',
 'Photos/img_1842.jpg']

# Walk through the directory, looking for .sh files.
>>> for directory, subdirectories, files in os.walk('.'):
...     for filename in files:
...         if filename.endswith(".sh"):
...             print(os.path.join(directory, filename))
...
./Scripts/hello.sh
./Scripts/organize_photos.sh
```

**Problem 3.** Write a Python function `grep()` that accepts the name of a target string and a file pattern. Find all files in the current directory or its subdirectories that match the file pattern. Next, check within the contents of the matched file for the target string. For example, `grep("range", "*.py")` should search Python files for the command `range`. Return a list of the file paths that matched the file pattern *and* the target string. For example, if you're in the `Shell2` directory and your grep function matches the '`calc.py`' file then your grep should return '`Python/calc.py`'

## The Subprocess module

The `subprocess` module allows Python to execute actual shell commands in the current working directory. Some important commands for executing shell commands from the `subprocess` module are listed in Table 10.5.

```
$ cd Shell2/Scripts
$ python
```

Function	Description
<code>subprocess.call()</code>	run a Unix command
<code>subprocess.check_output()</code>	run a Unix command and record its output
<code>subprocess.check_output.decode()</code>	this translates Unix command output to a string
<code>subprocess.Popen()</code>	use this to pipe together Unix commands

Table 10.5: Python subprocess module important commands

```
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
total 40
-rw-r--r-- 1 username groupname 20 Aug 26 2016 five_secs
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script1
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script2
-rw-r--r-- 1 username groupname 21 Aug 26 2016 script3
-rw-r--r-- 1 username groupname 21 Aug 26 2016 fifteen_secs
0
# Decode() translates the result to a string.
>>> file_info = subprocess.check_output(["ls", "-l"]).decode()
>>> file_info.split('\n')
['total 40',
 '-rw-r--r-- 1 username groupname 20 Aug 26 2016 five_secs',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script1',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script2',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 script3',
 '-rw-r--r-- 1 username groupname 21 Aug 26 2016 fifteen_secs',
 '']
```

`Popen` is a class of the subprocess module, with its own attributes and commands. It pipes together a few commands, similar to what we did at the beginning of the lab. This allows for more versatility in the shell input commands. If you wish to know more about the `Popen` class, go to the subprocess documentation on the internet.

```
$ cd Shell12
$ python
>>> import subprocess
>>> args = ["cat Files/Feb/assignments.txt | wc -l"]
# shell = True indicates to open a new shell process
# note that task is now an object of the Popen class
>>> task = subprocess.Popen(args, shell=True)
>>> 9
```

ACHTUNG!

If shell commands depend on user input, the program is vulnerable to a *shell injection attack*. This applies to Unix Shell commands as well as other situations like web browser interaction with web servers. Be extremely careful when creating a shell process from Python. There are specific functions, like `shlex.quote()`, that quote specific strings that are used to construct shell commands. But, when possible, it is often better to avoid user input altogether. For example, consider the following function.

```
>>> def inspect_file(filename):
...     """Return information about the specified file from the shell."""
...     return subprocess.check_output(["ls", "-l", filename]).decode()
```

If `inspect_file()` is given the input `".; rm -rf /"`, then `ls -l .` is executed innocently, and then `rm -rf /` destroys the computer by force deleting everything in the root directory.<sup>a</sup> Be careful not to execute a shell command from within Python in a way that a malicious user could potentially take advantage of.

<sup>a</sup>See [https://en.wikipedia.org/wiki/Code\\_injection#Shell\\_injection](https://en.wikipedia.org/wiki/Code_injection#Shell_injection) for more example attacks.

**Problem 4.** Using `os.path` and `Glob`, write a Python function that accepts an integer  $n$ . Search the current directory and all subdirectories for the  $n$  largest files. Then sort the list of filenames from the largest to the smallest files. Next, write the line count of the smallest file to a file called `smallest.txt` into the current directory. Finally, return the list of filenames, including the file path, in order from largest to smallest.

(Hint: the shell commands `ls -s` shows the file size.)

As a note, same as in problem 3, to get this problem correct, you need to return the entire file path **starting from the directory that was searched and continuing to the name of the file**. Do not return just the filenames, or the complete file path. For example, if you are currently in the `UnixShell2/` directory, meaning the next directory down to be searched will be `Shell2/`, then `Shell2/` should be the first part in the names of largest files returned by your function. More concretely, if '`data.txt`' is one files your function will return then instead of returning just '`data.txt`' or all of

'`YourComputerSpecificFilePath/UnixShell2/Shell2/Files/Mar/docs/data.txt`' as part of your list, you would return only '`Shell2/Files/Mar/docs/data.txt`'. Notice the paths returned will vary based on both the current working directory you're in and what directories were searched. However, also make sure that your file paths do not begin with '`./`' (Hint: To avoid the additional '`./`' in your file path, use `Glob` instead of `os.walk`.)

## Downloading Files

The Unix shell has tools for downloading files from the internet. The most popular are `wget` and `curl`. At its most basic, `curl` is the more robust of the two while `wget` can download recursively. This means that `wget` is capable of following links and directory structure when downloading content.

When we want to download a single file, we just need the URL for the file we want to download. This works for PDF files, HTML files, and other content simply by providing the right URL.

```
$ wget https://github.com/Foundations-of-Applied-Mathematics/Data/blob/master/←
  Volume1/dream.png
```

The following are also useful commands using `wget`.

```
# Download files from URLs listed in urls.txt.
$ wget -i list_of_urls.txt

# Download in the background.
$ wget -b URL

# Download something recursively.
$ wget -r --no-parent URL
```

### ACHTUNG!

If you're using Git Bash for Windows, `wget` is not automatically included. In order to use it, you have to find a download for `wget.exe` online. After installing it, you need to include it in the correct directory for git. If git is installed in the default location, you'll need to add it to `C:\Program Files\Git\mingw64\bin`.

**Problem 5.** The file `urls.txt` in the `Documents/` directory contains a list of URLs. Download the files in this list using `wget` and move them to the `Photos/` directory.

## sed and awk

`sed` and `awk` are two different scripting languages in their own right. `sed` is a stream editor; it performs basic transformations on input text. `Awk` is a text processing language that manipulates and reports data. Like Unix, these languages are easy to learn but difficult to master. It is very common to combine Unix commands and `sed` and `awk` commands.

### Printing Specific Lines Using `sed`

We have already used the `head` and `tail` commands to print the beginning and end of a file respectively. What if we wanted to print lines 30 to 40, for example? We can accomplish this using `sed`. In the `Documents/` folder, you will find the `lines.txt` file. We will use this file for the following examples.

```
# Same output as head -n3.
$ sed -n 1,3p lines.txt
line 1
line 2
line 3
```

```
# Same output as tail -n3.
$ sed -n 3,5p lines.txt
line 3
line 4
line 5

# Print lines 1,3,5.
$ sed -n -e 1p -e 3p -e 5p lines.txt
line 1
line 3
line 5
```

## Find and Replace Using sed

Using `sed`, we can also find and replace. We can perform this function on the output of another command, or we can perform this function in place on other files. The basic syntax of this `sed` command is the following.

```
sed s/str1/str2/g
```

This command will replace every instance of `str1` with `str2`. More specific examples follow.

```
$ sed s/line/LINE/g lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5

# Notice the file didn't change at all
$ cat lines.txt
line 1
line 2
line 3
line 4
line 5

# To save the changes, add the -i flag
$ sed -i s/line/LINE/g lines.txt
$ cat lines.txt
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
```

## Formatting output using awk

Earlier in this lab we mentioned `ls -l`, and as we have seen, this outputs lots of information. Using `awk`, we can select which fields we wish to print. Suppose we only cared about the file name and the permissions. We can get this output by running the following command.

```
$ cd Shell2/Documents
$ ls -l | awk ' {print $1, $9} '
total
-rw-r--r--. assignments.txt
-rw-r--r--. doc1.txt
-rw-r--r--. doc2.txt
-rw-r--r--. doc3.txt
-rw-r--r--. doc4.txt
-rw-r--r--. files.txt
-rw-r--r--. lines.txt
-rw-r--r--. newfiles.txt
-rw-r--r--. people.txt
-rw-r--r--. review.txt
-rw-r--r--. urls.txt
-rw-r--r--. words.txt
```

Notice we pipe the output of `ls -l` to `awk`. When calling a command using `awk`, we have to use quotation marks. It is a common mistake to forget to add these quotation marks. Inside these quotation marks, commands always take the same format.

```
awk ' <options> {<actions>} '
```

In the remaining examples we will not be using any of the options, but we will address various actions.

In the `Documents/` directory, you will find a `people.txt` file that we will use for the following examples. In our first example, we use the `print` action. The `$1` and `$9` mean that we are going to print the first and ninth fields.

Beyond specifying which fields we wish to print, we can also choose how many characters to allocate for each field. This is done using the `%` command within the `printf` command, which allows us to edit how the relevant data is printed. Look at the last part of the example below to see how it is done.

```
# contents of people.txt
$ cat people.txt
male,John,23
female,Mary,31
female,Sally,37
male,Ted,19
male,Jeff,41
female,Cindy,25

# Change the field separator (FS) to space at the beginning of run using BEGIN
# Printing each field individually proves we have successfully separated the ←
# fields
```

```
$ awk ' BEGIN{ FS = "," }; {print $1,$2,$3} ' < people.txt
male John 23
female Mary 31
female Sally 37
male Ted 19
male Jeff 41
female Cindy 25

# Format columns using printf so everything is in neat columns in different ←
# order
$ awk ' BEGIN{ FS = "," }; {printf "%-6s %2s %s\n", $1,$3,$2} ' < people.txt
male    23 John
female  31 Mary
female  37 Sally
male    19 Ted
male    41 Jeff
female  25 Cindy
```

The statement `"%-6s %2s %s\n"` formats the columns of the output. This says to set aside six characters left justified, then two characters right justified, then print the last field to its full length.

**Problem 6.** Inside the `Documents/` directory, you should find a file named `files.txt`. This file contains details on approximately one hundred files. The different fields in the file are separated by tabs. Using `awk`, `sort`, pipes, and redirects, write it to a new file in the current directory named `date_modified.txt` with the following specifications:

- in the first column, print the date the file was modified
- in the second column, print the name of the file
- sort the file from newest to oldest based on the date last modified

All of this can be accomplished using one command.

(Hint: change the field separator to account for tab-delimited files by setting `FS = "\t"` in the `BEGIN` command)

We have barely scratched the surface of what `awk` can do. Performing an internet search for *awk one-liners* will give you many additional examples of useful commands you can run using `awk`.

#### NOTE

Remember to archive and compress your `Shell2` directory before pushing it to your online repository for grading.

## Additional Material

### Customizing the Shell

Though there are multiple Unix shells, one of the most popular is the *bash* shell. The bash shell is highly customizable. In your home directory, you will find a hidden file named `.bashrc`. All customization changes are saved in this file. If you are interested in customizing your shell, you can customize the prompt using the `PS1` environment variable. As you become more and more familiar with the Unix shell, you will come to find there are commands you run over and over again. You can save commands you use frequently with `alias`. If you would like more information on these and other ways to customize the shell, you can find many quality reference guides and tutorials on the internet.

### System Management

In this section, we will address some of the basics of system management. As an introduction, the commands in Table 10.6 are used to learn more about the computer system.

Command	Description
<code>passwd</code>	Change user password
<code>uname</code>	View operating system name
<code>uname -a</code>	Print all system information
<code>uname -m</code>	Print machine hardware
<code>w</code>	Show who is logged in and what they are doing
<code>whoami</code>	Print userID of current user

Table 10.6: Commands for system administration.

# 11

# Sampling

**Lab Objective:** *Sampling is an important and fundamental tool in statistical modeling. In this lab we will learn to use PyMC for Bayesian modeling and statistical sampling.*

## Sampling

When seeking to understand a group or a phenomenon, we, as data scientists, will often look to find some sort of sample. A good sample can tell us a lot, and while we will not examine what makes a good sample in this lab, we will examine how much a good sample can tell us. One goal of Bayesian statistics is to be able to quantify, with degrees of certainty, how much we can learn from a sample, given what we already know about its source. This quantification of certainty allows us to extract more information and nuance from a sample than would otherwise be possible, and in turn allow us to better predict and describe events.

## Parameter Estimation

### Maximum Likelihood Estimation

*Maximum Likelihood Estimation* is a frequentist approach to parameter estimation, viewing probability as the limit of the frequency of success generated by several repeated trials. This contrasts with the Bayesian way of thinking about probability as the chance of success occurring. We will first examine the derivation of the maximum likelihood estimate (MLE) from the frequentist point of view, then examine how it is a special case of the Bayesian method of finding the *maximum a posteriori estimate* (MAP).

### Likelihood

Finding the maximum likelihood involves, unsurprisingly, maximizing the likelihood function, which is defined as follows

$$\mathcal{L}(\theta) = \mathcal{L}(\theta|\mathbf{x}) = f(\mathbf{x}|\theta),$$

where  $\mathbf{x}$  is a sample,  $\theta$  is the parameter we are estimating and  $f$  is the pdf. Determining the MLE  $\hat{\theta}$  is as simple as finding the argmax of  $\mathcal{L}$

$$\hat{\theta} = \operatorname{argmax}_{\theta \in \Theta} \mathcal{L}(\theta|\mathbf{x}).$$

## Maximum A Posteriori Estimate

If we examine closely, we can see a similarity between the likelihood function and Bayes' rule. Bayes' rule gives the following relation

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

To apply this rule to the problem of parameter estimation we get the following relation

$$f(\theta|\mathbf{x}) = \frac{f(\mathbf{x}|\theta)g(\theta)}{\int_{\Theta} f(\mathbf{x}|\vartheta)g(\vartheta)d\vartheta}. \quad (11.1)$$

We call  $f(\theta|\mathbf{x})$  the posterior distribution and  $g(\theta)$  the prior distribution. The MAP estimate is then the argmax of the posterior

$$\theta_{MAP} = \operatorname{argmax}_{\theta \in \Theta} f(\theta|\mathbf{x}). \quad (11.2)$$

When finding the MAP estimate, the exact posterior is often left uncalculated because it is difficult to compute. Instead we approximate it by finding its value at grid points of  $\theta$ . Similarly, because of the complexity of the denominator, we often don't find it until the end of the process. After we calculate  $f(\mathbf{x}|\theta)g(\theta)$  for each relevant  $\theta$  we can then approximate the integral in the denominator with a finite sum. First we find  $f(\mathbf{x}|\theta_i)g(\theta_i)$  for a grid of  $\theta$  values. Then, we can approximate the integral in denominator with the following sum  $\sum_i f(\mathbf{x}|\theta_i)g(\theta_i)$ .

Here we can see that the likelihood function is similar to Bayes' rule as long as we take  $g(\theta)$  to be a constant, i.e.  $\theta \sim \mathcal{U}(a, b)$ . This means that the MAP estimate is the MLE if we assume a uniform prior distribution.

**Problem 1.** Write a function called `bernoulli_sampling()` that takes the following parameters: `p` a float that is the "fairness" of a coin and `n` the size of the sample to be generated. In this function simulate `n` tosses of a coin which gives heads with probability `p`. Then use that sample to calculate the posterior distribution on  $p$  given a uniform prior using Equation 11.2.

For `p=.2` and `n=100` plot the posterior distribution and return the MAP estimate of  $p$ , which is also the MLE in this case.

Hint: In this case  $f$  is the Binomial pmf  $f(x) = p^x(1-p)^{n(1-x)}$ . You do not need to calculate the integral in the denominator exactly; since you are using a finite approximation of the distributions, you may use a finite approximation of the integral. You may simulate the tosses of a coin by using `np.random.binomial()` or `scipy.stats.binom()`.

## Non-Uniform Priors

While we are able to get good estimates, we leave a lot of the power of Bayesian statistics on the table when we only use a uniform prior. While the uniform prior is free from any preconceptions or biases, it also imparts the least amount of information. Using a non-uniform prior allows us to actually incorporate prior knowledge or assumptions into our model. If we have good reason to believe something about a parameter we are exploring before we even draw a sample, we can learn a lot more by accounting for those beliefs.

**Problem 2.** Suppose you choose a coin from a bag that produces coins of many weights. However, the bag seems to be more likely to produce coins that are strongly biased in favor of heads. You're unsure of which kind of coin you've drawn so in order to find out you perform 20 flips.

Write a function called `non_uniform_prior()` that takes the following parameters: `p` a float that is the "fairness" of a coin, `n` the size of the sample to be generated, and `prior` a SciPy distribution object which will act as the prior on  $p$ .

Similar to Problem 1, simulate `n` flips and calculate and plot the posterior distribution. Return the MAP estimate.

Examine the difference in confidence we can have in estimating the bias of the coin if the coin we draw gives heads 90% of the time as opposed to 40% of the time.

Because we think that coins biased in favor of heads are likely, we can choose a prior distribution that matches that assumption. In this case we will choose `Beta(5, 1.5)` as the prior distribution because it gives much more weight to parameters larger than .5. This is most easily achieved with `scipy.stats.beta(5, 1.5)` and using the `pdf()` method to calculate  $g(\theta)$

## Sampling from a Markov Chain

A Markov chain is a way to model sequences of states or events. Markov chains make a few assumptions, one of those being that the probability of each state occurring is dependent only on the previous state. The relationship between the states are described by what is called a transition matrix.

Markov chains and sampling are like peanut butter and jelly: neither one really lives up to their full potential without the other. Given the transition matrix of a Markov chain, we can use sampling to better understand what that chain looks like or to get a well-informed idea of what the future may hold.

Sampling from a simple (row stochastic) transition matrix like the one below is as simple as picking a starting state  $s_0$ , and then using the corresponding row to sample randomly using the probabilities in the row.

	a	b	c
a	0.7	0.1	0.2
b	0.5	0.4	0.1
c	0.1	0.8	0.1

For example, using the above transition matrix, let  $s_0 = a$ , so we will randomly sample from the array  $[a, b, c]$  using the respective probabilities  $[0.7, 0.1, 0.2]$ . If the sample gives us  $c$ , we can set  $s_1 = c$  and can continue the process to find  $s_2, \dots, s_n$ .

**Problem 3.** Given the transition matrix below and assuming the 0th day is sunny, sample from the markov chain to give a possible forecast of the 10 following days. Return a list of strings, not including the 0th day.

	sun	rain	wind
sun	0.6	0.1	0.3
rain	0.2	0.6	0.2
wind	0.3	0.4	0.3

Hint: `np.random.choice()` may be helpful here.

## PyMC

Python has many powerful sampling tools including PyMC, an efficient implementation of a method known as Monte Carlo Markov Chain (MCMC) Sampling. This is a useful technique as it constructs a Markov Chain whose steady state is a probability distribution that is difficult to sample from directly. Unlike our simple Markov Chain from the last problem, certain Markov Chains are abstract. PyMC gives us a way to work with these more complex scenarios.

### Single Variable PyMC

Consider the following: owners of a restaurant are trying to decide if they should keep selling nachos. They gather the data for several months about how many people order nachos each day. One of the owners happened to take a class in Bayesian statistics in college, so she decides test her knowledge. She assumes the data are distributed as  $\text{Poisson}(\lambda)$  for some unknown value of  $\lambda$ , where  $\lambda$  has a prior of  $\text{Gamma}(2,2)$ . She wishes to solve for  $\lambda$  and sets up a PyMC Model for the situation as follows:

```
import numpy as np
import pymc as pm
import arviz as az    # visualization package

with pm.Model() as model:
    # define the prior of lambda as a Gamma(2, 2) distribution
    lam = pm.Gamma('lambda', alpha=2, beta=2)

    # define the likelihood of the data (called nacho_data) to be distributed
    # as Poisson where the expected value of the outcome (mu) is lam
    y = pm.Poisson('y', mu=lam, observed=nacho_data)

    # sample from the posterior
    trace = pm.sample(n)    # n is the desired number of samples
    az.plot_trace(trace)    # plot the posterior and trace plot for lambda

    new_lambda = trace.posterior['lambda'] # trace values of lambda as a list
    mean = float(new_lambda.mean()) # expected value of lambda
```

This code generates a model for the prior distribution of  $\lambda$ , and then incorporates that prior into a model for the Poisson likelihood. It then samples from the posterior of  $\lambda$   $n$  times, from which we can estimate its expected value. The function `az.plot_trace()` plots both the posterior (on the left) as well as a *trace plot* (on the right), which indicates how well the sampling converged. The rule of thumb is: the closer the trace plot resembles a fuzzy caterpillar, the better the Markov Chain converged to the posterior.

We will now reconsider the initial problem of the coin flip.

**Problem 4.** Write a function that accepts the coin flip data in array form and an integer  $n$  for the desired number of samples. Given data that flips a coin 100 times, assume the data are distributed as Bernoulli( $p$ ) for some unknown value of  $p$ , where  $p$  has a prior of Beta(1,1). Set up a PyMC model for this situation and sample from the posterior  $n$  times. Plot the trace plot and return the expected value of the posterior as a float, *not an array*.

Run the function with data generated by the following code

```
from scipy.stats import bernoulli
data = bernoulli.rvs(0.2, size=100)
```

## Multivariate PyMC

Unlike the Poisson and Bernoulli distributions, many other distributions (including the Normal, Beta, Gamma, and Binomial distributions) have two or more parameters. These problems can really showcase the usefulness and ease of PyMC. Multivariate PyMC problems are coded up exactly the same way as the single variable example above, except that now there will be multiple priors defined separately, all of which will be called by the likelihood.

**Problem 5.** Write a function that accepts height data in array form and an integer  $n$  for the desired number of samples. Given a dataset of the measured heights of 100 men, assume the data are distributed as  $\text{Normal}(\mu, 1/\tau)$  where  $\mu$  has a prior of  $\text{Normal}(m, s)$ , and  $\tau$  has a prior of  $\text{Gamma}(\alpha, \beta)$ . Your function should have default values  $m=180$ ,  $s=10$ ,  $\text{alpha}=2$ , and  $\text{beta}=10$ . Set up a PyMC model for this situation and sample from the posterior  $n$  times. Plot the trace plots for  $\mu$  and  $\tau$ , and return the expected value of the posterior of  $\mu$  as a float, *not as an array*.

Run the function with data generated by the following code

```
heights = np.random.normal(180, 10, 100)
```

Hint: `pm.Normal()` uses parameters `mu` and either `sigma` or `tau`, where the variance of the distribution is given by `sigma2` or `1/tau` respectively.



# 12

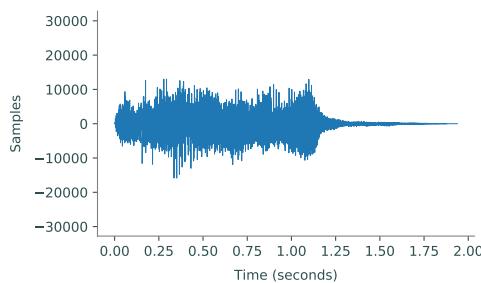
## The Discrete Fourier Transform

**Lab Objective:** *The analysis of periodic functions has many applications in pure and applied mathematics, especially in settings dealing with sound waves. The Fourier transform provides a way to analyze such periodic functions. In this lab, we introduce how to work with digital audio signals in Python, implement the discrete Fourier transform, and use the Fourier transform to detect the frequencies present in a given sound wave. We strongly recommend completing the exercises in a Jupyter Notebook.*

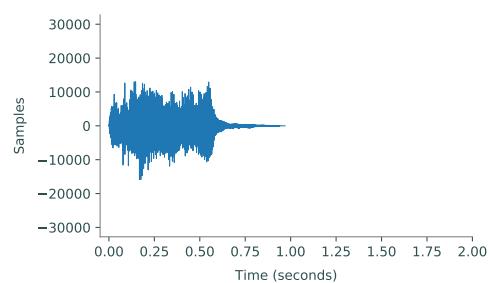
### Digital Audio Signals

Sound waves have two important characteristics: *frequency*, which determines the pitch of the sound, and *intensity* or *amplitude*, which determines the volume of the sound. Computers use *digital audio signals* to approximate sound waves. These signals have two key components: *sample rate*, which relates to the frequency of sound waves, and *samples*, which measure the amplitude of sound waves at a specific instant in time.

To see why the sample rate is necessary, consider an array with samples from a sound wave. The sound wave can be arbitrarily stretched or compressed to make a variety of sounds. If compressed, the sound becomes shorter and has a higher pitch. Similarly, the same set of samples with a lower sample rate becomes stretched and has a lower pitch.



(a) The plot of `tada.wav`.



(b) Compressed plot of `tada.wav`.

Figure 12.1: Plots of the same set of samples from a sound wave with varying sample rates. The plot on the left is the plot of the samples with the original sample rate. The sample rate of the plot on the right has been doubled, resulting in a compression of the actual sound when played back.

Given the rate at which a set of samples is taken, the wave can be reconstructed exactly as it was recorded. In most applications, this sample rate is measured in *Hertz* (Hz), the number of samples taken per second. The standard rate for high quality audio is 44100 equally spaced samples per second, or 44.1 kHz.

## Wave File Format

One of the most common audio file formats across operating systems is the *wave* format, also called `wav` after its file extension. SciPy has built-in tools to read and create `wav` files. To read a `wav` file, use `scipy.io.wavfile.read()`. This function returns the signal's sample rate and its samples.

```
# Read from the sound file.
>>> from scipy.io import wavfile
>>> rate, samples = wavfile.read("tada.wav")
```

Sound waves can be visualized by plotting time against the amplitude of the sound, as in Figure 12.1. The amplitude of the sound at a given time is just the value of the sample at that time. Since the sample rate is given in samples per second, the length of the sound wave in seconds is found by dividing the number of samples by the sample rate:

$$\frac{\text{num samples}}{\text{sample rate}} = \frac{\text{num samples}}{\text{num samples/second}} = \text{second.} \quad (12.1)$$

**Problem 1.** Write a `SoundWave` class for storing digital audio signals.

1. The constructor should accept an integer sample rate and an array of samples. Store each input as an attribute.
2. Write a method that plots the stored sound wave. Use (12.1) to correctly label the *x*-axis in terms of seconds, and set the *y*-axis limits to  $[-32768, 32767]$  (the reason for this is discussed in the next section).

Use SciPy to read `tada.wav`, then instantiate a corresponding `SoundWave` object and display its plot. Compare your plot to Figure 12.1a.

## Scaling

To write to a `wav` file, use `scipy.io.wavfile.write()`. This function accepts the name of the file to write to, the sample rate, and the array of samples as parameters.

```
>>> import numpy as np

# Write a 2-second random sound wave sampled at a rate of 44100 Hz.
>>> samples = np.random.randint(-32768, 32767, 88200, dtype=np.int16)
>>> wavfile.write("white_noise.wav", 44100, samples)
```

For `scipy.io.wavfile.write()` to correctly create a `wav` file, the samples must be one of four numerical datatypes: 32-bit floating point (`np.float32`), 32-bit integers (`np.int32`), 16-bit integers (`np.int16`), or 8-bit unsigned integers (`np.uint8`). If samples of a different type are passed into the function, it may still write a file, but the sound will likely be distorted in some way. In this lab, we only work with 16-bit integer samples, unless otherwise specified.

A 16-bit integer is an integer between  $-32768$  and  $32767$ , inclusive. If the elements of an array of samples are not all within this range, the samples must be scaled before writing to a file: multiply the samples by  $32767$  (the largest number in the 16-bit range) and divide by the largest sample magnitude. This ensures the most accurate representation of the sound and sets it to full volume.

$$\text{np.int16} \left( \left( \frac{\text{original samples}}{\max(|\text{original samples}|)} \right) \times 32767 \right) = \text{scaled samples} \quad (12.2)$$

Because 16-bit integers can only store numbers within a certain range, it is important to multiply the original samples by the largest number in the 16-bit range *after* dividing by the largest sample magnitude. Otherwise, the results of the multiplication may be outside the range of integers that can be represented, causing overflow errors. Also, samples may sometimes contain complex values, especially after some processing. Make sure to scale and export only the real part (use the `real` attribute of the array).

#### NOTE

The IPython API includes a tool for embedding sounds in a Jupyter Notebook. The function `IPython.display.Audio()` accepts either a file name or a sample rate (`rate`) and an array of samples (`data`); calling the function generates an interactive music player in the Notebook.

```
In [1]: import IPython
from scipy.io import wavfile

# Embed tada.wav straight from the file.
IPython.display.Audio(filename="tada.wav")

# Alternatively, embed tada.wav using the raw data.
# rate, samples = wavfile.read("tada.wav")
# IPython.display.Audio(rate=rate, data=samples)
```

Out[1]: 

#### ACHTUNG!

Turn the volume down before listening to any of the sounds in this lab.

**Problem 2.** Add a method to the `SoundWave` class that accepts a file name and a boolean `force`. Write to the specified file using the stored sample rate and the array of samples. If the array of samples does not have `np.int16` as its data type, or if `force` is `True`, scale the samples as in (12.2) before writing the file.

Use your method to create two new files that contains the same sound as `tada.wav`: one without scaling, and one with scaling (use `force=True`). Use `IPython.display.Audio()` to display `tada.wav` and the new files. All three files should sound identical, except the scaled file should be louder than the other two.

## Generating Sounds

Sinusoidal waves correspond to pure frequencies, like a single note on the piano. Recall that the function  $\sin(x)$  has a period of  $2\pi$ . To create a specific tone for 1 second, we sample from the sinusoid with period 1,

$$f(x) = \sin(2\pi xk),$$

where  $k$  is the desired frequency. According to (12.1), generating a sound that lasts for  $s$  seconds at a sample rate  $r$  requires  $rs$  equally spaced samples in the interval  $[0, s]$ .

**Problem 3.** Write a function that accepts floats  $k$  and  $s$ . Create a `SoundWave` instance containing a tone with frequency  $k$  that lasts for  $s$  seconds. Use a sample rate of  $r = 44100$ .

The following table shows some frequencies that correspond to common notes. Octaves of these notes are obtained by doubling or halving these frequencies.

Note	Frequency (Hz)
A	440
B	493.88
C	523.25
D	587.33
E	659.25
F	698.46
G	783.99
A	880

Use your function to generate an A tone lasting for 2 seconds.

**Problem 4.** Digital audio signals can be combined by addition or concatenation. Adding samples overlays tones so they play simultaneously; concatenated samples plays one set of samples after the other with no overlap.

1. Implement the `__add__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A + B` creates a new `SoundWave` object whose samples are the element-wise sum of the samples from `A` and `B`. Raise a `ValueError` if the sample arrays from `A` and `B` are not the same length.

Use your method to generate a three-second A minor chord (A, C, and E together).

2. Implement the `__rshift__()` magic method<sup>a</sup> for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A >> B` creates a new `SoundWave` object whose samples are the concatenation of the samples from `A`, then the samples from `B`. Raise a `ValueError` if the sample rates from the two objects are not equal.

(Hint: `np.concatenate()`, `np.hstack()`, and/or `np.append()` may be useful.)

Use your method to generate the arpeggio A → C → E, where each pitch lasts one second.

Consider using these two methods to produce elementary versions of some simple tunes.

<sup>a</sup>The `>>` operator is a *bitwise shift operator* and is usually reserved for operating on binary numbers.

## The Discrete Fourier Transform

As with the chords generated above, all sound waves are sums of varying amounts of different frequencies (pitches). In the case of the discrete samples  $\mathbf{f} = [f_0 \ f_1 \ \dots \ f_{n-1}]^\top$  that we have worked with thus far, each  $f_i$  gives information about the amplitude of the sound wave at a specific instant in time. However, sometimes it is useful to find out what frequencies are present in the sound wave and in what amount.

We can write the sound wave sample as a sum

$$\mathbf{f} = \sum_{k=0}^{n-1} c_k \mathbf{w}_n^{(k)}, \quad (12.3)$$

where  $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$ , called the *discrete Fourier basis*, represents various frequencies. The coefficients  $c_k$  represent the amount of each frequency present in the sound wave.

The *discrete Fourier transform (DFT)* is a linear transformation that takes  $\mathbf{f}$  and finds the coefficients  $\mathbf{c} = [c_0 \ c_1 \ \dots \ c_{n-1}]^\top$  needed to write  $\mathbf{f}$  in this frequency basis. Later in the lab, we will convert the index  $k$  to a value in Hertz to find out what frequency  $c_k$  corresponds to.

Because the sample  $\mathbf{f}$  was generated by taking  $n$  evenly spaced samples of the sound wave, we generate the basis  $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$  by taking  $n$  evenly spaced samples of the frequencies represented by the oscillating functions  $\{e^{-2\pi i k t/n}\}_{k=0}^{n-1}$ . (Note that  $i = \sqrt{-1}$ , the imaginary unit, is represented as `1j` in Python). This yields

$$\mathbf{w}_n^{(k)} = [\omega_n^0 \ \omega_n^{-k} \ \dots \ \omega_n^{-(n-1)k}]^\top, \quad (12.4)$$

where  $\omega_n = e^{2\pi i / n}$ .

The DFT is then represented by the change of basis matrix

$$F_n = \frac{1}{n} [\mathbf{w}_n^0 \ \mathbf{w}_n^1 \ \mathbf{w}_n^2 \ \dots \ \mathbf{w}_n^{n-1}] = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \dots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \dots & \omega_n^{-(n-1)^2} \end{bmatrix}, \quad (12.5)$$

and we can take the DFT of  $f$  by calculating

$$\mathbf{c} = F_n \mathbf{f}. \quad (12.6)$$

Note that the DFT depends on the number of samples  $n$ , since the discrete Fourier basis we use depends on the number of samples taken. The larger  $n$  is, the closer the frequencies approximated by the DFT will be to the actual frequencies present in the sound wave.

### ACHTUNG!

There are several different conventions for defining the DFT. For example, instead of (12.6), `scipy.fftpack.fft()` uses the formula

$$\mathbf{c} = nF_n\mathbf{f},$$

where  $F_n$  is as given (12.5). Denoting this version of the DFT as  $\hat{F}_n\mathbf{f} = \hat{\mathbf{c}}$ , we have  $nF_n = \hat{F}_n$  and  $n\mathbf{c} = \hat{\mathbf{c}}$ . The conversion is easy, but it is very important to be aware of which convention a particular implementation of the DFT uses.

**Problem 5.** Write a function that accepts an array  $\mathbf{f}$  of samples. Use 12.6 to calculate the coefficients  $\mathbf{c}$  of the DFT of  $\mathbf{f}$ . Include the  $1/n$  scaling in front of the sum.

Test your implementation on small, random arrays against `scipy.fftpack.fft()`, scaling your output  $\mathbf{c}$  to match SciPy's output  $\hat{\mathbf{c}}$ . Once your function is working, try to optimize it so that the entire array of coefficients is calculated in the one line.

(Hint: Use array broadcasting.)

## The Fast Fourier Transform

Calculating the DFT of a vector of  $n$  samples using only (12.6) is at least  $O(n^2)$ , which is incredibly slow for realistic sound waves. Fortunately, due to its inherent symmetry, the DFT can be implemented as a recursive algorithm by separating the computation into even and odd indices. This method of calculating the DFT is called the *fast Fourier transform* (FFT) and runs in  $O(n \log n)$  time.

---

**Algorithm 1** The fast Fourier transform for arrays with  $2^a$  entries for some  $a \in \mathbb{N}$ .

---

```

1: procedure SIMPLE_FFT( $\mathbf{f}$ ,  $N$ )
2:   procedure SPLIT( $\mathbf{g}$ )
3:      $n \leftarrow \text{size}(\mathbf{g})$ 
4:     if  $n \leq N$  then
5:       return  $nF_n\mathbf{g}$             $\triangleright$  Use the function from Problem 5 for small enough  $\mathbf{g}$ .
6:     else
7:        $\mathbf{even} \leftarrow \text{SPLIT}(\mathbf{g}_{::2})$        $\triangleright$  Get the DFT of every other entry of  $\mathbf{g}$ , starting from 0.
8:        $\mathbf{odd} \leftarrow \text{SPLIT}(\mathbf{g}_{1::2})$        $\triangleright$  Get the DFT of every other entry of  $\mathbf{g}$ , starting from 1.
9:        $\mathbf{z} \leftarrow \text{zeros}(n)$ 
10:      for  $k = 0, 1, \dots, n - 1$  do           $\triangleright$  Calculate the exponential parts of the sum.
11:         $z_k \leftarrow e^{-2\pi ik/n}$ 
12:         $m \leftarrow n // 2$                    $\triangleright$  Get the middle index for  $\mathbf{z}$  ( $//$  is integer division).
13:        return [ $\mathbf{even} + \mathbf{z}_{::m} \odot \mathbf{odd}$ ,  $\mathbf{even} + \mathbf{z}_{m::} \odot \mathbf{odd}$ ]  $\triangleright$  Concatenate two arrays of length  $m$ .
14:      return SPLIT( $\mathbf{f}$ ) / size( $\mathbf{f}$ )

```

---

Note that the base case in lines 4–5 of Algorithm 1 results from setting  $n = 1$  in (12.6), yielding the single coefficient  $c_0 = g_0$ . The  $\odot$  in line 13 indicates the component-wise product

$$\mathbf{f} \odot \mathbf{g} = [f_0g_0 \quad f_1g_1 \quad \cdots \quad f_{n-1}g_{n-1}]^T,$$

which is also called the *Hadamard product* of  $\mathbf{f}$  and  $\mathbf{g}$ .

This algorithm performs significantly better than the naïve implementation of the DFT, but the simple version described in Algorithm 1 only works if the number of original samples is exactly a power of 2. SciPy’s FFT routines avoid this problem by padding the sample array with zeros until the size is a power of 2, then executing the remainder of the algorithm from there. Of course, SciPy also uses various other tricks to further speed up the computation.

**Problem 6.** Write a function that accepts an array  $\mathbf{f}$  of  $n$  samples where  $n$  is a power of 2. Use Algorithm 1 to calculate the DFT of  $\mathbf{f}$ .

(Hint: eliminate the loop in lines 10–11 with `np.arange()` and array broadcasting, and use `np.concatenate()` or `np.hstack()` for the concatenation in line 13.)

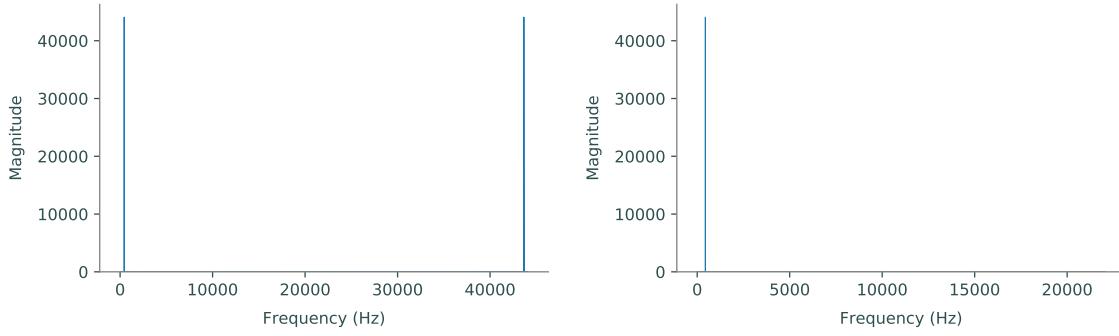
Test your implementation on random arrays against `scipy.fftpack.fft()`, scaling your output  $\mathbf{c}$  to match SciPy’s output  $\hat{\mathbf{c}}$ . Time your function from Problem 5, this function, and SciPy’s function on an array with 8192 entries.

(Hint: Use `%time` in Jupyter Notebook to time a single line of code.)

## Visualizing the DFT

The graph of the DFT of a sound wave is useful in a variety of applications. While the graph of the sound in the time domain gives information about the amplitude (volume) of a sound wave at a given time, the graph of the DFT shows which frequencies (pitches) are present in the sound wave. Plotting a sound’s DFT is referred to as plotting in the *frequency domain*.

As a simple example, the single-tone notes generated by the function in Problem 3 contain only one frequency. For instance, Figure 12.2a graphs the DFT of an A tone. However, this plot shows two frequency spikes, despite there being only one frequency present in the actual sound. This is due to symmetries inherent to the DFT; for frequency detection, the second half of the plot can be ignored as in Figure 12.2b.



(a) The DFT of an A tone with symmetries.      (b) The DFT of an A tone without symmetries.

Figure 12.2: Plots of the DFT with and without symmetries. Notice that the  $x$ -axis of the symmetrical plot on the left goes up to 44100 (the sample rate of the sound wave) while the  $x$ -axis of the non-symmetric plot on the right goes up to only 22050 (half the sample rate). Also notice that the spikes occur at 440 Hz and 43660 Hz (which is  $44100 - 440$ ).

The DFT of a more complicated sound wave has many frequencies, each of which corresponds to a different tone present in the sound wave. The magnitude of the coefficients indicates a frequency's influence in the sound wave; a greater magnitude means that the frequency is more influential.

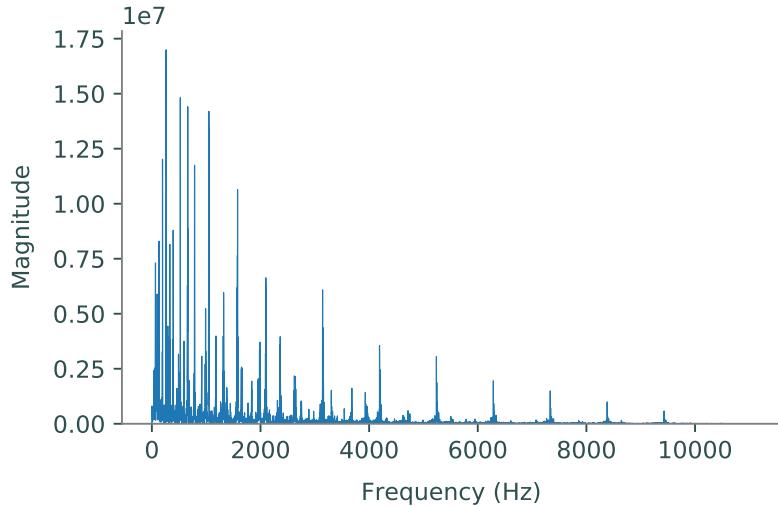


Figure 12.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency present in the sound wave. Since the sample rate of `tada.wav` is 22050 Hz, the plot of its DFT without symmetries only goes up to 11025 Hz, half of its sample rate.

## Plotting Frequencies

Since the DFT represents the frequency domain, the  $x$ -axis of a plot of the DFT should be in terms of Hertz, which has units  $1/s$ . In other words, to plot the magnitudes of the Fourier coefficients against the correct frequencies, we must convert the frequency index  $k$  of each  $c_k$  to Hertz. This can be done by multiplying the index by the sample rate and dividing by the number of samples:

$$\frac{k}{\text{num samples}} \times \frac{\text{num samples}}{\text{second}} = \frac{k}{\text{second}}. \quad (12.7)$$

In other words,  $kr/n = v$ , where  $r$  is the sample rate,  $n$  is the number of samples, and  $v$  is the resulting frequency.

**Problem 7.** Modify your `SoundWave` plotting method from Problem 1 so that it accepts a boolean defaulting to `False`. If the boolean is `True`, take the DFT of the stored samples and plot—in a new subplot—the frequencies present on the  $x$ -axis and the magnitudes of those frequencies (use `np.abs()` to compute the magnitude) on the  $y$ -axis. Only display the first half of the plot (as in Figures 12.2b and 12.2b), and use (12.7) to adjust the  $x$ -axis so that it correctly shows the frequencies in Hertz. Use SciPy to calculate the DFT.

Display the DFT plots of the A tone and the A minor chord from Problem 4. Compare your results to Figures 12.2a and 12.4.

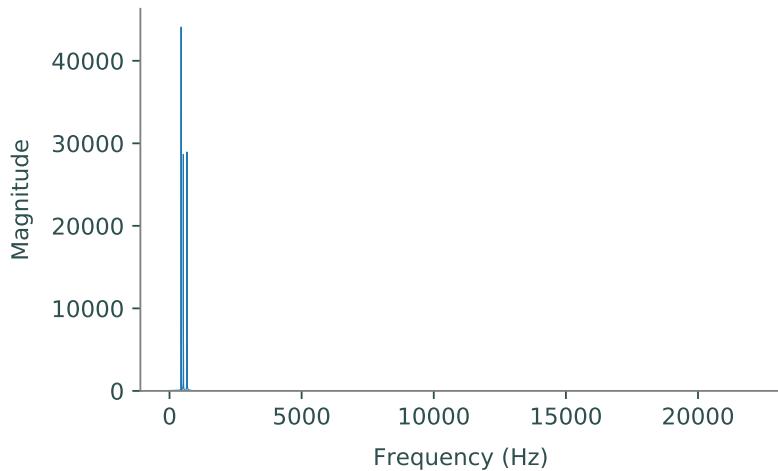


Figure 12.4: The DFT of the A minor chord.

If the frequencies present in a sound are already known before plotting its DFT, the plot may be interesting, but little new information is actually revealed. Thus, the main applications of the DFT involve sounds in which the frequencies present are unknown. One application in particular is sound filtering, which will be explored in greater detail in a subsequent lab. The first step in filtering a sound is determining the frequencies present in that sound by taking its DFT.

Consider the DFT of the A minor chord in Figure 12.4. This graph shows that there are three main frequencies present in the sound. To determine what those frequencies are, find which indices of the array of DFT coefficients have the three largest values, then scale these indices the same way as in (12.7) to translate the indices to frequencies in Hertz.

**Problem 8.** The file `mystery_chord.wav` contains an unknown chord. Use the DFT and the frequency table in Problem 3 to determine the individual notes that are present in the sound. (Hint: `np.argsort()` may be useful.)

# 13

## Introduction to Wavelets

**Lab Objective:** *Wavelets are used to sparsely represent information. This makes them useful in a variety of applications. We explore both the one- and two-dimensional discrete wavelet transforms using various types of wavelets. We then use a Python package called PyWavelets for further wavelet analysis including image cleaning and image compression.*

### Wavelet Functions

*Wavelets families* are sets of orthogonal functions (wavelets) designed to decompose nonperiodic, piecewise continuous functions. These families have four types of wavelets: mother, daughter, father, and son functions. Father and son wavelets contain information related to the general movement of the function, while mother and daughter wavelets contain information related to the details of the function. The father and mother wavelets are the basis of a family of wavelets. Son and daughter wavelets are just scaled translates of the father and mother wavelets, respectively.

### Haar Wavelets

The *Haar Wavelet* family is one of the most widely used wavelet families in wavelet analysis. This set includes the father, mother, son, and daughter wavelets defined below. The Haar father (scaling) function is given by

$$\varphi(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

The Haar son wavelets are scaled and translated versions of the father wavelet:

$$\varphi_{jk}(x) = \varphi(2^j x - k) = \begin{cases} 1 & \text{if } \frac{k}{2^j} \leq x < \frac{k+1}{2^j} \\ 0 & \text{otherwise.} \end{cases}$$

The Haar mother wavelet function is defined as

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

The Haar daughter wavelets are scaled and translated versions of the mother wavelet

$$\psi_{jk} = \psi(2^j x - k)$$

## Wavelet Decompositions

Information (such as a mathematical function or signal) can be stored and analyzed by considering its *wavelet decomposition*. A *wavelet decomposition* is a linear combination of wavelets. For example, a mathematical function  $f$  can be approximated as a combination of Haar son and daughter wavelets as follows:

$$f(x) = \sum_{k=-\infty}^{\infty} a_k \varphi_{m,k}(x) + \sum_{k=-\infty}^{\infty} b_{m,k} \psi_{m,k}(x) + \cdots + \sum_{k=-\infty}^{\infty} b_{n,k} \psi_{n,k}(x)$$

where  $m < n$ , and all but a finite number of the  $a_k$  and  $b_{j,k}$  terms are nonzero. The  $a_k$  terms are often referred to as *approximation coefficients* while the  $b_{j,k}$  terms are known as *detail coefficients*. The approximation coefficients typically capture the broader, more general features of a signal while the detail coefficients capture smaller details and noise.

A wavelet decomposition can be done with any family of wavelet functions. Depending on the properties of the wavelet and the function (or signal)  $f$ ,  $f$  can be approximated to an arbitrary level of accuracy. Each arbitrary wavelet family has a mother wavelet  $\psi$  and a father wavelet  $\varphi$  which are the basis of the family. A countably infinite set of wavelet functions (daughter and son wavelets) can be generated using dilations and shifts of the first two functions where  $m, k \in \mathbb{Z}$ :

$$\begin{aligned}\psi_{m,k}(x) &= \psi(2^m x - k) \\ \varphi_{m,k}(x) &= \varphi(2^m x - k).\end{aligned}$$

## The Discrete Wavelet Transform

The mapping from a function to a sequence of wavelet coefficients is called the *discrete wavelet transform*. The discrete wavelet transform is analogous to the discrete Fourier transform. Now, instead of using trigonometric functions, different families of basis functions are used.

In the case of finitely-sampled signals and images, there exists an efficient algorithm for computing the wavelet decomposition. Commonly used wavelets have associated high-pass and low-pass filters which are derived from the wavelet and scaling functions, respectively.

When the low-pass filter is convolved with the sampled signal, low frequency (also known as approximation) information is extracted. This is similar to turning up the bass on a speaker, which extracts the low frequencies of a sound wave. This filter highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details and extracts the approximation coefficients.

When the high-pass filter is convolved with the sampled signal, high frequency information (also known as detail) is extracted. This is similar to turning up the treble on a speaker, which extracts the high frequencies of a sound wave. This filter highlights the small changes found in the signal and extracts the detail coefficients.

The two primary operations of the algorithm are the discrete convolution and downsampling, denoted  $*$  and  $DS$ , respectively. First, a signal is convolved with both filters. The convolutions fold the signal back on itself so the resulting array is the same size but half the information is duplicated, *downsampling* is then required to eliminate the repeated information. In the context of this lab, a *filter bank* is the combined process of convolving with a filter, and then downsampling. The result will be an array of approximation coefficients  $A$  and an array of detail coefficients  $D$ . This process can be repeated on the new approximation to obtain another layer of approximation and detail coefficients. See Figure 13.1.

A common lowpass filter is the averaging filter. Given an array  $\mathbf{x}$ , the averaging filter produces an array  $\mathbf{y}$  where  $y_n$  is the average of  $x_n$  and  $x_{n-1}$ . In other words, the averaging filter convolves an array with the array  $L = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix}$ . This filter preserves the main idea of the data. The corresponding highpass filter is the distance filter. Given an array  $\mathbf{x}$ , the distance filter produces an array  $\mathbf{y}$  where  $y_n$  is the distance between  $x_n$  and  $x_{n-1}$  ( $|x_n - x_{n-1}|$ ). In other words, the difference filter convolves an array with the array  $H = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$ . This filter preserves the details of the data.

For the Haar Wavelet, we will use the lowpass and highpass filters mentioned. In order for this filters to have inverses, the filters must be normalized (for more on why this is, see Additional Materials). The resulting lowpass and highpass filters for the Haar Wavelets are the following:

$$L = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

$$H = \begin{bmatrix} -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

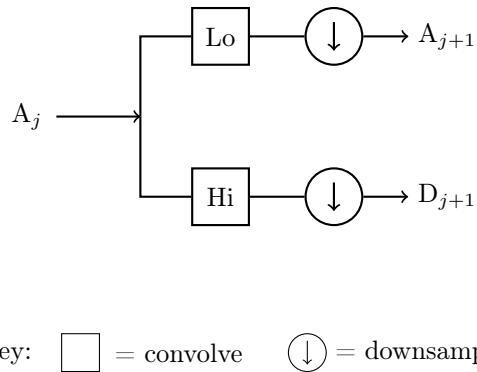


Figure 13.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

As noted earlier, the key mathematical operations of the discrete wavelet transform are convolution and downsampling. Given a filter and a signal, the convolution can be obtained using `scipy.signal.fftconvolve()`.

```
>>> from scipy.signal import fftconvolve
>>> # Initialize a filter.
>>> L = np.ones(2)/np.sqrt(2)
>>> # Initialize a signal X.
>>> X = np.sin(np.linspace(0,2*np.pi,16))
>>> # Convolve X with L.
>>> fftconvolve(X, L)
[ -1.84945741e-16   2.87606238e-01   8.13088984e-01   1.19798126e+00
  1.37573169e+00   1.31560561e+00   1.02799937e+00   5.62642704e-01
  7.87132986e-16  -5.62642704e-01  -1.02799937e+00  -1.31560561e+00
 -1.37573169e+00  -1.19798126e+00  -8.13088984e-01  -2.87606238e-01
 -1.84945741e-16]
```

The convolution operation alone gives redundant information, so it is downsampled to keep only what is needed. The array will be downsampled by a factor of 2, which means keeping only every other entry:

```
>>> # Downsample an array X.
>>> sampled = X[1::2] # Keeps odd entries
```

Both the approximation and detail coefficients are computed in this manner. The approximation uses the low-pass filter while the detail uses the high-pass filter. Implementation of a filter bank is found in Algorithm 1.

---

**Algorithm 1** The one-dimensional discrete wavelet transform.  $X$  is the signal to be transformed,  $L$  is the low-pass filter,  $H$  is the high-pass filter and  $n$  is the number of filter bank iterations.

---

```

1: procedure DWT( $X, L, H, n$ )
2:    $A_0 \leftarrow X$                                       $\triangleright$  Initialization.
3:   for  $i = 0 \dots n - 1$  do
4:      $D_{i+1} \leftarrow DS(A_i * H)$                     $\triangleright$  High-pass filter and downsample.
5:      $A_{i+1} \leftarrow DS(A_i * L)$                     $\triangleright$  Low-pass filter and downsample.
6:   return  $A_n, D_n, D_{n-1}, \dots, D_1$ .
```

---

**Problem 1.** Write a function that calculates the discrete wavelet transform using Algorithm 1. The function should return a list of one-dimensional NumPy arrays in the following form:  $[A_n, D_n, \dots, D_1]$ .

Test your function by calculating the Haar wavelet coefficients of a noisy sine signal with  $n = 4$ :

```
domain = np.linspace(0, 4*np.pi, 1024)
noise = np.random.randn(1024)*.1
noisysin = np.sin(domain) + noise
coeffs = dwt(noisysin, L, H, 4)
```

Plot the original signal with the approximation and detail coefficients and verify that they match the plots in Figure 13.2.

(Hint: Use array broadcasting).

Note: the plots in your jupyter notebook *do not* have to be labeled exactly like those in 13.2. As long as the signals are clearly visible that is enough.

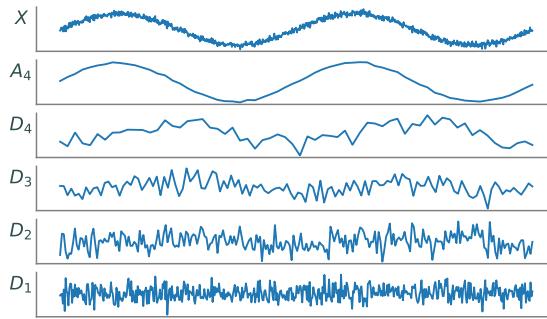


Figure 13.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

## Inverse Discrete Wavelet Transform

The process of the discrete wavelet transform is reversible. Using modified filters, a set of detail and approximation coefficients can be manipulated and combined to recreate a signal. The Haar wavelet filters for the inverse transformation are found by reversing the operations for each filter. The Haar inverse filters are given below:

$$L^{-1} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

$$H^{-1} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{bmatrix}$$

The first row refers to the inverse high-pass filter and the second row refers to the inverse low-pass filter.

Suppose the wavelet coefficients  $A_n$  and  $D_n$  have been computed.  $A_{n-1}$  can be recreated by tracing the schematic in Figure 13.1 backwards:  $A_n$  and  $D_n$  are first upsampled, and then are convolved with the inverse low-pass and high-pass filters, respectively. In the case of the Haar wavelet, *upsampling* involves doubling the length of an array by inserting a 0 at every other position. To complete the operation, the new arrays are convolved and added together to obtain  $A_{n-1}$ .

```
>>> # Upsample the coefficient arrays A and D.
>>> up_A = np.zeros(2*A.size)
>>> up_A[::2] = A
>>> up_D = np.zeros(2*D.size)
>>> up_D[::2] = D
>>> # Convolve and add, discarding the last entry.
>>> A = fftconvolve(up_A, L)[:-1] + fftconvolve(up_D, H)[:-1]
```

This process is continued with the newly obtained approximation coefficients and with the next detail coefficients until the original signal is recovered.

**Problem 2.** Write a function that performs the inverse wavelet transform. The function should accept three things as arguments: a list of arrays (of the same form as the output of Problem 1), a reverse low-pass filter, and a reverse high-pass filter. The function should return a single array, which represents the recovered signal.

Note that the input list of arrays has length  $n + 1$  (consisting of  $A_n$  together with  $D_n, D_{n-1}, \dots, D_1$ ). Your code should run once per  $D_i$  matrix so it should execute a total of  $n$  times.

To test your function, first perform the inverse transform on the noisy sine wave that you created in the first problem. Then, compare the original signal with the signal recovered by your inverse wavelet transform function using `np.allclose()`.

### ACHTUNG!

Although Algorithm 1 and the preceding discussion apply in the general case, the code implementations apply only to the Haar wavelet. Because of the nature of the discrete convolution, when convolving with longer filters, the signal to be transformed needs to undergo a different type of lengthening in order to avoid information loss during the convolution. As such, the functions written in Problems 1 and 2 will only work correctly with the Haar filters and would require modifications to be compatible with more wavelets.

## The Two-dimensional Wavelet Transform

The generalization of the wavelet transform to two dimensions is similar to one dimensional transforms. Again, the two primary operations used are convolution and downsampling. The main difference in the two-dimensional case is the number of convolutions and downsample per iteration. First, the convolution and downsampling are performed along the rows of an array. This results in two new arrays, as in the one dimensional case. Then, convolution and downsampling are performed along the columns of the two new arrays. This results in four final arrays that make up the new approximation and detail coefficients. See Figure 13.3.

When implemented as an iterative filter bank, each pass through the filter bank yields one set of approximation coefficients plus three sets of detail coefficients. More specifically, if the two-dimensional array  $X$  is the input to the filter bank, the arrays  $Approx$ ,  $H$ ,  $V$ , and  $D$  are obtained.  $Approx$  is a smoothed approximation of  $X$  (similar to  $A_n$  in the one-dimensional case), and the other three arrays contain detail coefficients that capture high-frequency oscillations in horizontal ( $H$ ), vertical ( $V$ ), and diagonal ( $D$ ) directions. The arrays  $A$ ,  $H$ ,  $V$ , and  $D$  are known as *subbands*. Any or all of the subbands can be fed into a filter bank to further decompose the signal into additional subbands. This decomposition can be represented by a partition of a rectangle, called a *subband pattern*. The subband pattern for one pass of the filter bank is shown in Figure 13.4, with an example of an image decomposition given in Figure 13.5.

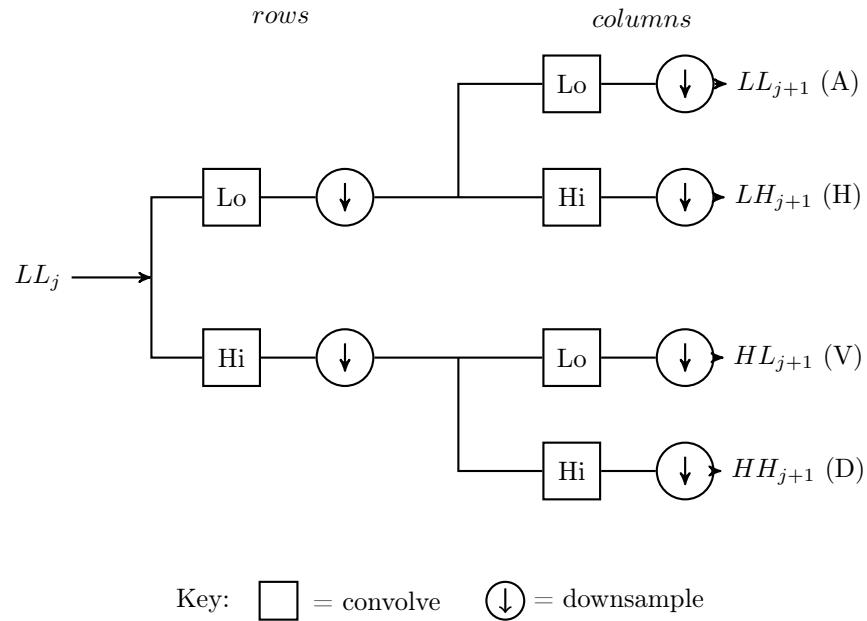


Figure 13.3: The two-dimensional discrete wavelet transform implemented as a filter bank.

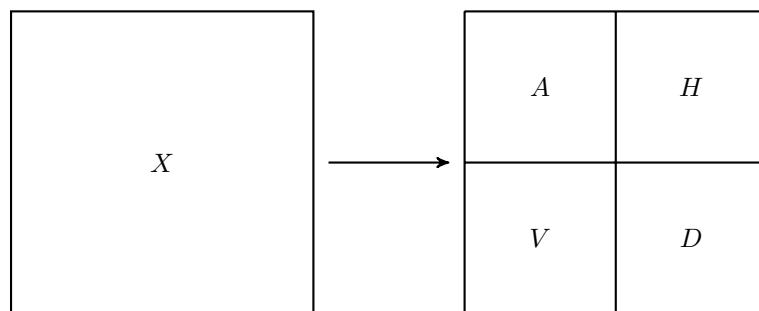


Figure 13.4: The subband pattern for one step in the 2-dimensional wavelet transform.

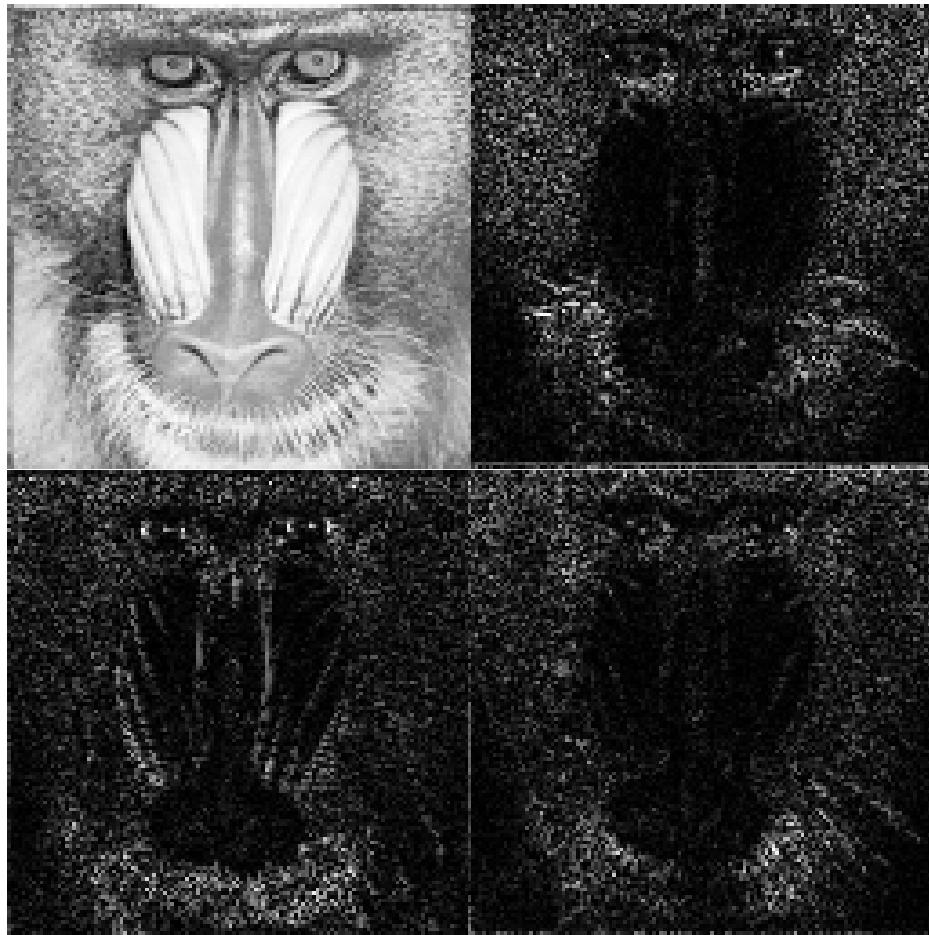


Figure 13.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband ( $A$ ) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.  
Original image source: <http://sipi.usc.edu/database/>.

The wavelet coefficients obtained from a two-dimensional wavelet transform are used to analyze and manipulate images at differing levels of resolution. Images are often sparsely represented by wavelets; that is, most of the image information is captured by a small subset of the wavelet coefficients. This is a key fact for wavelet-based image compression and will be discussed in further detail later in the lab.

## The PyWavelets Module

PyWavelets is a Python package designed for wavelet analysis. Although it has many other uses, in this lab it will primarily be used for image manipulation. PyWavelets can be installed using the following command:

```
$ pip install PyWavelets
```

PyWavelets provides a simple way to calculate the subbands resulting from one pass through the filter bank. The following code demonstrates how to find the approximation and detail subbands of an image.

```
>>> from imageio import imread
>>> import pywt                      # The PyWavelets package.
# The True parameter produces a grayscale image.
>>> mandrill = imread('mandrill1.png', cmap=True)
# Use the Daubechies 4 wavelet with periodic extension.
>>> lw = pywt.dwt2(mandrill, 'db4', mode='per')
```

The function `pywt.dwt2()` calculates the subbands resulting from one pass through the filter bank. The second positional argument specifies the type of wavelet to be used in the transform. The `mode` keyword argument sets the extension mode, which determines the type of padding used in the convolution operation. For the problems in this lab, always use `mode='per'`, which is the periodic extension. The function `dwt2()` returns a tuple. The first entry of the list is the  $A$ , or approximation, subband. The second entry of the list is a separate tuple containing the remaining subbands,  $H$ ,  $V$ , and  $D$  (in that order).

PyWavelets supports a number of different wavelets which are divided into different classes known as families. The supported families and their wavelet instances can be listed by executing the following code:

```
>>> # List the available wavelet families.
>>> print(pywt.families())
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey', 'gaus', 'mexh', 'morl', '↔
    cgau', 'shan', 'fbsp', 'cmor']
>>> # List the available wavelets in a given family.
>>> print(pywt.wavelist('coif'))
['coif1', 'coif2', 'coif3', 'coif4', 'coif5', 'coif6', 'coif7', 'coif8', 'coif9↔
    ', 'coif10', 'coif11', 'coif12', 'coif13', 'coif14', 'coif15', 'coif16', '↔
    coif17']
```

It's important to note that the names from the wavelist are what we use when we call `dwt2`. Sometimes the name of a family will also exist as a wavelet transform in the wave list, but not always. Different wavelets have different properties; the most suitable wavelet is dependent on the specific application. For example, the morlet wavelet is closely related to human hearing and vision. Note that not all of these families work with the function `pywt.dwt2()`, because they are continuous wavelets. Choosing which wavelet is used is partially based on the properties of a wavelet, but since many wavelets share desirable properties, the best wavelet for a particular application is often not known without some type of testing.

#### NOTE

The numerical value in a wavelets name refers to the filter length. This value is multiplied by the standard filter length of the given wavelet, resulting in the new filter length. For example, `coif1` has filter length 6 and `coif2` has filter length 12.

**Problem 3.** Explore the two-dimensional wavelet transform by completing the following:

1. Save a picture of a raccoon with the following code

```
>>> from scipy.misc import face
>>> raccoon = face(True)
```

2. Plot the subbands of raccoon as described above (using the Daubechies 4 wavelet with periodic extension). Compare this with the subbands of the mandrill image shown in Figure 13.5.
3. Compare the subband patterns of the haar, symlet, and coiflet wavelets of the raccoon picture by plotting the subbands after one pass through the filter bank. The haar subband should have more detail than the symlet subband, and the symlet subband should have more detail than the coiflet wavelet.

A few Hints: for plotting, find a function that will plot an image given an array. Also, when plotting the transformations that represent the horizontal, vertical, or diagonal components of the image take `np.abs()` of the array when you plot it. This will radicalize the array and make its detections easier to view. Finally, you will run into an error if you try to use just 'sym' as an argument. Consider why this could be and how you might find the proper argument to use.

The function `pywt.wavedec2()` is similar to `pywt.dwt2()`, but it also includes a keyword argument, `level`, which specifies the number of times to pass an image through the filter bank. It will return a list of subbands, the first of which is the final approximation subband, while the remaining elements are tuples which contain sets of detail subbands ( $H$ ,  $V$ , and  $D$ ). For example, if I were to call `pywt.wavedec2` with `level=4`, the output would be of the form [Approx, (H4,V4,D4),(H3,V3,D3),(H2,V2,D2),(H1,V1,D1)].

If `level` is not specified, the number of passes through the filter bank will be the maximum level where the decomposition is still useful. The function `pywt.waverec2()` accepts a list of subband patterns (like the output of `pywt.wavedec2()` or `pywt.dwt2()`), a name string denoting the wavelet, and a keyword argument `mode` for the extension mode. It returns a reconstructed image using the reverse filter bank. When using this function, be sure that the wavelet and mode match the deconstruction parameters. PyWavelets has many other useful functions including `dwt()`, `idwt()` and `idwt2()` which can be explored further in the documentation for PyWavelets, <https://pywavelets.readthedocs.io/en/latest/index.html>.

## Applications

### Noise Reduction

Noise in an image is defined as unwanted visual artifacts that obscure the true image. Images acquire noise from a variety of sources, including cameras, data transfer, and image processing algorithms. This section will focus on reducing a particular type of noise in images called *Gaussian white noise*.

Gaussian white noise causes every pixel in an image to be perturbed by a small amount. Many types of noise, including Gaussian white noise, are very high-frequency. Since many images are relatively sparse in high-frequency domains, noise in an image can be safely removed from the high frequency subbands while minimally distorting the true image. A basic, but effective, approach to reducing Gaussian white noise in an image is thresholding. Thresholding can be done in two ways, referred to as hard and soft thresholding.

Given a positive threshold value  $\tau$ , hard thresholding sets every detail coefficient whose magnitude is less than  $\tau$  to zero, while leaving the remaining coefficients untouched. Soft thresholding also zeros out all coefficients of magnitude less than  $\tau$ , but in addition maps the remaining positive coefficients  $\beta$  to  $\beta - \tau$  and the remaining negative coefficients  $\alpha$  to  $\alpha + \tau$ .

Once the coefficients have been thresholded, the inverse wavelet transform is used to recover the denoised image. The threshold value is generally a function of the variance of the noise, and in real situations, is not known. In fact, noise variance estimation in images is a research area in its own right, but that goes beyond the scope of this lab.

**Problem 4.** Write two functions that accept a list of wavelet coefficients in the usual form, as well as a threshold value. Each function returns the thresholded wavelet coefficients (also in the usual form). The first function should implement hard thresholding and the second should implement soft thresholding. While writing these two functions, remember that only the detail coefficients (meaning the elements of H,V, or D arrays) are thresholded, so the first entry of the input coefficient list (The A matrix) should remain unchanged.

To test your functions, perform hard and soft thresholding on `noisy_darkhair.png` and plot the resulting images together. When testing your function, use the Daubechies 4 wavelet and four sets of detail coefficients (`level=4` when using `wavedec2()`). For soft thresholding use  $\tau = 20$ , and for hard thresholding use  $\tau = 40$ .

Some notes:

1. Masking will be helpful however it's important to consider the order in which you change values since adjusting certain values by  $\tau$  initially can skew which conditions that value satisfies. Use your masks in a safe order.
2. In previous problems there was only one tuple of detail coefficients now there may be more. Make sure your code is robust enough to handle any number of tuples of detail coefficients.

## Image Compression

Transform methods based on Fourier and wavelet analysis play an important role in image compression; for example, the popular JPEG image compression standard is based on the discrete cosine transform. The JPEG2000 compression standard and the FBI Fingerprint Image database, along with other systems, take the wavelet approach.

The general framework for compression is as follows. First, the image to be compressed undergoes some form of preprocessing, depending on the particular application. Next, the discrete wavelet transform is used to calculate the wavelet coefficients, and these are then *quantized*, i.e. mapped to a set of discrete values (for example, rounded to the nearest integer). The quantized coefficients are then passed through an entropy encoder (such as Huffman Encoding), which reduces the number of bits required to store the coefficients. What remains is a compact stream of bits that can be saved or transmitted much more efficiently than the original image. The steps above are nearly all invertible (the only exception being quantization), allowing the original image to be almost perfectly reconstructed from the compressed bitstream. See Figure 13.6.

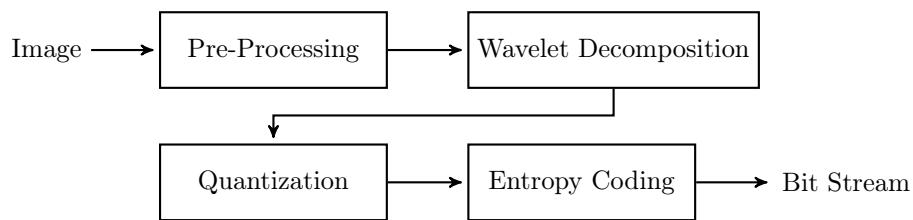


Figure 13.6: Wavelet Image Compression Schematic

## WSQ: The FBI Fingerprint Image Compression Algorithm

The Wavelet Scalar Quantization (WSQ) algorithm is among the first successful wavelet-based image compression algorithms. It solves the problem of storing millions of fingerprint scans efficiently while meeting the law enforcement requirements for high image quality. This algorithm is capable of achieving compression ratios in excess of 10-to-1 while retaining excellent image quality; see Figure 13.7. This section of the lab steps through a simplified version of this algorithm by writing a Python class that performs both the compression and decompression. Differences between this simplified algorithm and the complete algorithm are found in the Additional Material section at the end of this lab. Most of the methods of the class have already been implemented. The following problems will detail the methods you will need to implement yourself.

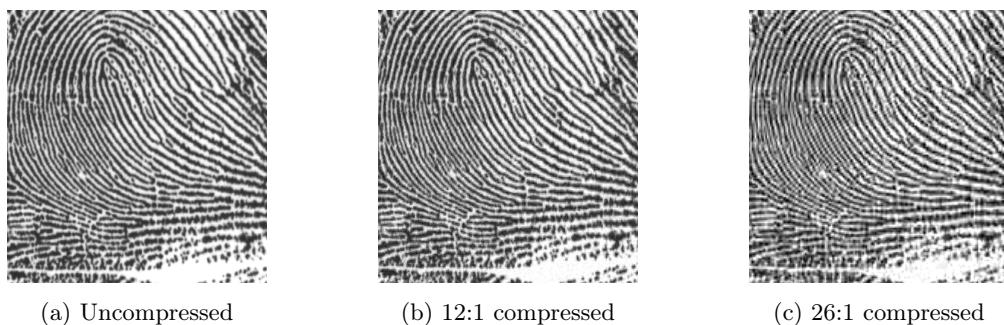


Figure 13.7: Fingerprint scan at different levels of compression. Original image source: <http://www.nist.gov/itl/iad/ig/wsqa.cfm>.

## WSQ: Preprocessing

Preprocessing in this algorithm ensures that roughly half of the new pixel values are negative, while the other half are positive, and all fall in the range  $[-128, 128]$ . The input to the algorithm is a matrix of nonnegative 8-bit integer values giving the grayscale pixel values for the fingerprint image. The image is processed by the following formula:

$$M' = \frac{M - m}{s},$$

where  $M$  is the original image matrix,  $M'$  is the processed image,  $m$  is the mean pixel value, and  $s = \max\{\max(M) - m, m - \min(M)\}/128$  (here  $\max(M)$  and  $\min(M)$  refer to the maximum and minimum pixel values in the matrix). We've provided the code for this part of the algorithm.

## WSQ: Calculating the Wavelet Coefficients

The next step in the compression algorithm is decomposing the image into subbands of wavelet coefficients. In this implementation of the WSQ algorithm, the image is decomposed into five sets of detail coefficients (`level=5`) and one approximation subband, as shown in Figure 13.8. Each of these subbands should be placed into a list in the same ordering as in Figure 13.8 (another way to consider this ordering is the approximation subband followed by each level of detail coefficients  $[A, H_5, V_5, D_5, H_4, V_4, \dots, D_1]$ ).

**Problem 5.** Implement the class method `decompose()`. This function should accept an image to decompose and should return a list of ordered subbands. Use the function `pywt.wavedec2()` with the '`coif1`' wavelet to obtain the subbands. These subbands should then be ordered in a single list as described above.

Implement the inverse of the decomposition by writing the class method `recreate()`. This function should accept a list of 16 subbands (ordered like the output of `decompose()`) and should return a reconstructed image. Use `pywt.waverec2()` to reconstruct an image from the subbands. Note that you will need to adjust the accepted list in order to adhere to the required input for `waverec2()`.

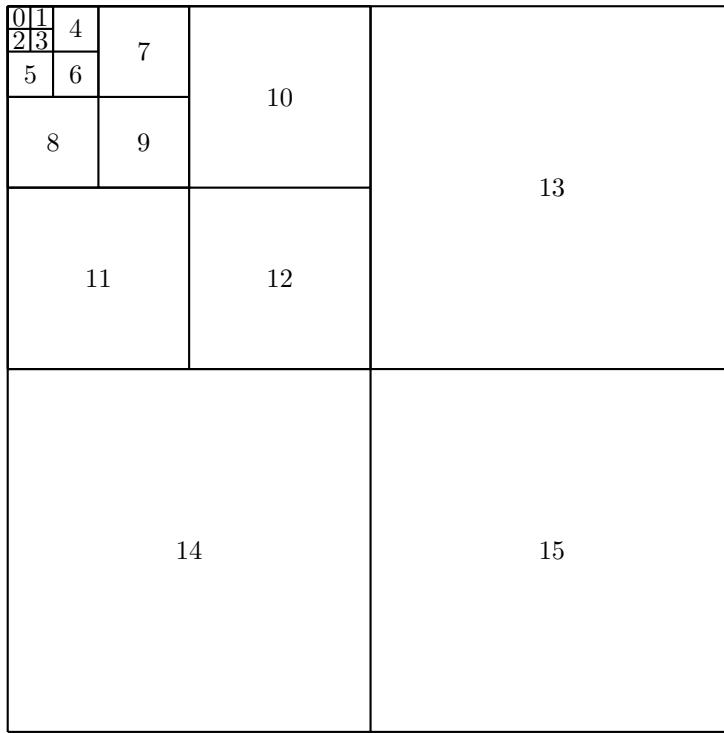


Figure 13.8: Subband Pattern for simplified WSQ algorithm.

### WSQ: Quantization

Quantization is the process of mapping each wavelet coefficient to an integer value and is the main source of compression in the algorithm. By mapping the wavelet coefficients to a relatively small set of integer values, the complexity of the data is reduced, which allows for efficient encoding of the information in a bit string. Further, a large portion of the wavelet coefficients will be mapped to 0 and discarded completely. The fact that fingerprint images tend to be very nearly sparse in the wavelet domain means that little information is lost during quantization. Care must be taken, however, to perform this quantization in a manner that achieves good compression without discarding so much information that the image cannot be accurately reconstructed.

Given a wavelet coefficient  $a$  in subband  $k$ , the corresponding quantized coefficient  $p$  is given by

$$p = \begin{cases} \left\lfloor \frac{a - Z_k/2}{Q_k} \right\rfloor + 1, & a > Z_k/2 \\ 0, & -Z_k/2 \leq a \leq Z_k/2 \\ \left\lceil \frac{a + Z_k/2}{Q_k} \right\rceil - 1, & a < -Z_k/2, \end{cases}$$

where  $Z_k$  and  $Q_k$  are dependent on the subband. They determine how much compression is achieved. If  $Q_k = 0$ , all coefficients are mapped to 0.

An array of detail coefficients (such as  $H$ ) can be quantized by running each value of the array through the piecewise function.

Selecting appropriate values for  $Z_k$  and  $Q_k$  is a tricky problem in itself, and relies on heuristics based on the statistical properties of the wavelet coefficients. The methods that calculate these values have already been initialized.

Quantization is not a perfectly invertible process. Once the wavelet coefficients have been quantized, some information is permanently lost. However, wavelet coefficients  $\hat{a}_k$  in subband  $k$  can be roughly reconstructed from the quantized coefficients  $p$  using

$$\hat{a}_k = \begin{cases} (p - C)Q_k + Z_k/2, & p > 0 \\ 0, & p = 0 \\ (p + C)Q_k - Z_k/2, & p < 0, \end{cases}$$

where  $C$  is a new dequantization parameter. This process is called *dequantization*. Again, if  $Q_k = 0$ ,  $\hat{a}_k = 0$  should be returned.

**Problem 6.** Implement the quantization step by writing the `quantize()` method of your class. This method should accept a NumPy array of coefficients and the quantization parameters  $Q_k$  and  $Z_k$ . The function should return a NumPy array of the quantized coefficients.

Also implement the `dequantize()` method of your class using the formula given above. This function should accept the same parameters as `quantize()` as well as a parameter  $C$  which defaults to .44. The function should return a NumPy array of dequantized coefficients.

(Hint: Masking and array slicing will help keep your code short and fast when implementing both of these methods. Remember the case for  $Q_k = 0$ . Test your functions by comparing the output of your functions to a hand calculation on a small matrix.)

## WSQ: The Rest

The remainder of the compression and decompression methods have already been implemented in the WSQ class. The following discussion explains the basics of what happens in those methods. Once all of the subbands have been quantized, they are divided into three groups. The first group contains the smallest ten subbands (positions zero through nine), while the next two groups contain the three subbands of next largest size (positions ten through twelve and thirteen through fifteen, respectively). All of the subbands of each group are then flattened and concatenated with the other subbands in the group. These three arrays of values are then mapped to Huffman indices. Since the wavelet coefficients for fingerprint images are typically very sparse, special indices are assigned to lists of sequential zeros of varying lengths. This allows large chunks of information to be stored as a single index, greatly aiding in compression. The Huffman indices are then assigned a bit string representation through a Huffman map.

Python does not natively include all of the tools necessary to work with bit strings, but the Python package `bitstring` does have these capabilities. Download `bitstring` using the following command:

```
$ pip install bitstring
```

You will not use `bitstring` functions in this lab, but the code provided in the lab does call functions from the `bitstring` module. So you'll need to import the package with the following line of code:

```
>>> import bitstring as bs
```

### WSQ: Calculating the Compression Ratio

The methods of compression and decompression are now fully implemented. The final task is to verify how much compression has taken place. The compression ratio is the ratio of the number of bits in the original image to the number of bits in the encoding. Assuming that each pixel of the input image is an 8-bit integer, the number of bits in the original image is just eight times the number of pixels (the number of pixels in the original source image is stored in the class attribute `_pixels`). The number of bits in the encoding can be calculated by adding up the lengths of each of the three bit strings stored in the class attribute `_bitstrings`.

**Problem 7.** Implement the method `get_ratio()` by calculating the ratio of compression. The function should not accept any parameters and should return the compression ratio.

Your compression algorithm is now complete! Test your class with the following code. The compression ratio should be approximately 18.

```
# Try out different values of r between .1 to .9.
r = .5
finger = imread('uncompressed_finger.png', cmap=True)
wsq = WSQ()
wsq.compress(finger, r)
print(wsq.get_ratio())
new_finger = wsq.decompress()
plt.subplot(211)
plt.imshow(finger, cmap=plt.cm.Greys_r)
plt.subplot(212)
plt.imshow(np.abs(new_finger), cmap=plt.cm.Greys_r)
plt.show()
```

## Additional Material

### Haar Wavelet Transform

The Haar Wavelet Transform is a general matrix transform used to convolve Haar Wavelets. It is found by combining the convolution matrices for a lowpass and highpass filter such that one is directly on top of the other. The lowpass filter is taking the average of every two elements in an array and the highpass filter is taking the difference of every two elements in an array. Redundant information given in the new matrix is then removed via downsampling. However, in order for the transform matrix to have the property  $A^T = A^{-1}$ , the columns of the matrix must be normalized. Thus, each column is normalized (and subsequently the filters) and the resulting matrix is the Haar Wavelet Transform.

For more on the Haar Wavelet Transform, see *Discrete Wavelet Transformations: An Elementary Approach with Applications* by Patrick J. Van Fleet.

### WSQ Algorithm

The official standard for the WSQ algorithm is slightly different from the version implemented in this lab. One of the largest differences is the subband pattern that is used in the official algorithm; this pattern is demonstrated in Figure 13.9. The pattern used may seem complicated and somewhat arbitrary, but it is used because of the relatively good empirical results when used in compression. This pattern can be obtained by performing a single pass of the 2-dimensional filter bank on the image then passing each of the resulting subbands through the filter bank resulting in 16 total subbands. This same process is then repeated with the  $A$ ,  $H$  and  $V$  subbands of the original approximation subband creating 46 additional subbands. Finally, the subband corresponding to the top left of Figure 13.9 should be passed through the 2-dimensional filter bank a single time.

As in the implementation given above, the subbands of the official algorithm are divided into three groups. The subbands 0 through 18 are grouped together, as are 19 through 51 and 52 through 63. The official algorithm also uses a wavelet specialized for image compression that is not included in the PyWavelets distribution. There are also some slight modifications made to the implementation of the discrete wavelet transform that do not drastically affect performance.

0	1	4	7	8	19	20	23	24	52	53						
2	3															
5	6	9	10	21	22	25	26									
11	12	15	16	27	28	31	32									
13	14	17	18	29	30	33	34			54	55					
35	36	39	40	51												
37	38	41	42													
43	44	47	48													
45	46	49	50													
56				57				60	61							
58				59				62	63							

Figure 13.9: True subband pattern for WSQ algorithm.

# 14

# Polynomial Interpolation

**Lab Objective:** *Learn and compare three methods of polynomial interpolation: standard Lagrange interpolation, Barycentric Lagrange interpolation and Chebyshev interpolation. Explore Runge's phenomenon and how the choice of interpolating points affect the results. Use polynomial interpolation to study air pollution by approximating graphs of particulates in air.*

## Polynomial Interpolation

Polynomial interpolation is the method of finding a polynomial that matches a function at specific points in its range. More precisely, if  $f(x)$  is a function on the interval  $[a, b]$  and  $p(x)$  is a polynomial then  $p(x)$  interpolates the function  $f(x)$  at the points  $x_0, x_1, \dots, x_n$  if  $p(x_j) = f(x_j)$  for all  $j = 0, 1, \dots, n$ . In this lab most of the discussion is focused on using interpolation as a means of approximating functions or data, however, polynomial interpolation is useful in a much wider array of applications.

Given a function  $f(x)$  and a set of unique points  $\{x_i\}_{i=0}^n$ , it can be shown that there exists a unique interpolating polynomial  $p(x)$ . That is, there is one and only one polynomial of degree  $n$  that interpolates  $f(x)$  through those points. This uniqueness property is why, for the remainder of this lab, an interpolating polynomial is referred to as *the* interpolating polynomial. One approach to finding the unique interpolating polynomial of degree  $n$  is Lagrange interpolation.

### Lagrange interpolation

Given a set  $\{x_i\}_{i=1}^n$  of  $n$  points to interpolate, a family of  $n$  basis functions with the following property is constructed:

$$L_j(x_i) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}.$$

The Lagrange form of this family of basis functions is

$$L_j(x) = \frac{\prod_{k=1, k \neq j}^n (x - x_k)}{\prod_{k=1, k \neq j}^n (x_j - x_k)} \quad (14.1)$$

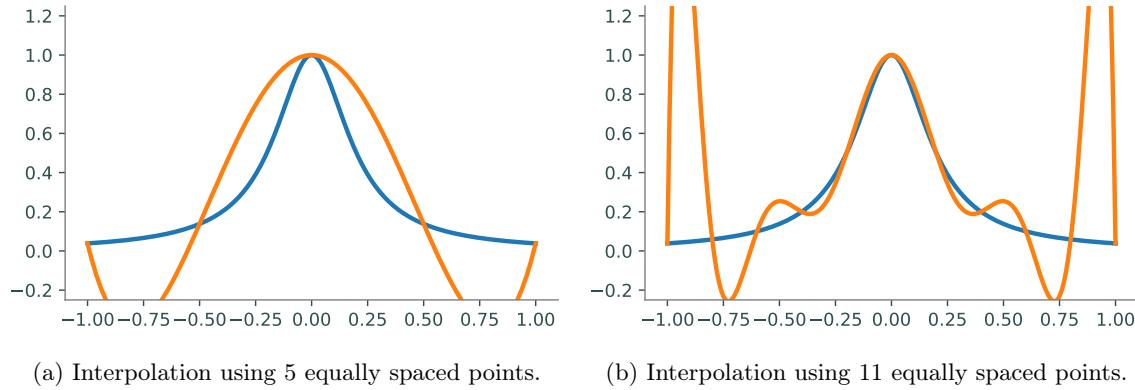


Figure 14.1: Interpolations of Runge's function  $f(x) = \frac{1}{1+25x^2}$  with equally spaced interpolating points.

Each of these Lagrange basis functions is a polynomial of degree  $n-1$  and has the necessary properties as given above.

**Problem 1.** Define a function `lagrange()` that will be used to construct and evaluate an interpolating polynomial on a domain of  $x$  values. The function should accept two NumPy arrays of length  $n$  which contain the  $x$  and  $y$  values of the interpolating points as well as a NumPy array of values of length  $m$  at which the interpolating polynomial will be evaluated.

Within `lagrange()`, write a subroutine that will evaluate each of the  $n$  Lagrange basis functions at every point in the domain. It may be helpful to follow these steps:

1. Compute the denominator of each  $L_j$  (as in Equation 14.1) .
2. Using the previous step, evaluate  $L_j$  at all points in the computational domain (this will give you  $m$  values for each  $L_j$ .)
3. Combine the results into an  $n \times m$  NumPy array, consisting of each of the  $n L_j$  evaluated at each of the  $m$  points in the domain.

You may find the functions `np.product()` and `np.delete()` to be useful while writing this method.

*Lagrange interpolation* is completed by combining the Lagrange basis functions with the  $y$ -values of the function to be interpolated  $y_i = f(x_i)$  in the following manner:

$$p(x) = \sum_{j=1}^n y_j L_j(x) \quad (14.2)$$

This will create the unique interpolating polynomial.

Since polynomials are typically represented in their expanded form with coefficients on each of the terms, it may seem like the best option when working with polynomials would be to use Sympy, or NumPy's `poly1d` class to compute the coefficients of the interpolating polynomial individually. This is rarely the best approach, however, since expanding out the large polynomials that are required can quickly lead to instability (especially when using large numbers of interpolating points). Instead, it is usually best just to leave the polynomials in unexpanded form (which is still a polynomial, just not a pretty-looking one), and compute values of the polynomial directly from this unexpanded form.

```
# Evaluate the polynomial (x-2)(x+1) at 10 points without expanding the ←
expression.
>>> pts = np.arange(10)
>>> (pts - 2) * (pts + 1)
array([ 2,  0,  0,  2,  6, 12, 20, 30, 42, 56])
```

In the given example, there would have been no instability if the expression had actually been expanded but in the case of a large polynomial, stability issues can dominate the computation. Although the coefficients of the interpolating polynomials will not be explicitly computed in this lab, polynomials are still being used, albeit in a different form.

**Problem 2.** Complete the implementation of `lagrange()`.

Evaluate the interpolating polynomial at each point in the domain by combining the  $y$  values of the interpolation points and the evaluated Lagrange basis functions from Problem 1 as in Equation 14.2. Return the final array of length  $m$  that consists of the interpolating polynomial evaluated at each point in the domain.

You can test your function by plotting Runge's function  $f(x) = \frac{1}{1+25x^2}$  and your interpolating polynomial on the same plot for different values of  $n$  equally spaced interpolating values then comparing your plot to the plots given in Figure 14.1.

The Lagrange form of polynomial interpolation is useful in some theoretical contexts and is easier to understand than other methods, however, it has some serious drawbacks that prevent it from being a useful method of interpolation. First, Lagrange interpolation is  $O(n^2)$  where other interpolation methods are  $O(n^2)$  (or faster) at startup but only  $O(n)$  at run-time, Second, Lagrange interpolation is an unstable algorithm which causes it to return inaccurate answers when larger numbers of interpolating points are used. Thus, while useful in some situations, Lagrange interpolation is not desirable in most instances.

## Barycentric Lagrange interpolation

Barycentric Lagrange interpolation is simple variant of Lagrange interpolation that performs much better than plain Lagrange interpolation. It is essentially just a rearrangement of the order of operations in Lagrange multiplication which results in vastly improved performance, both in speed and stability.

Barycentric Lagrange interpolation relies on the observation that each basis function  $L_j$  can be rewritten as

$$L_j(x) = \frac{w(x)}{(x - x_j)} w_j$$

where

$$w(x) = \prod_{j=1}^n (x - x_j)$$

and

$$w_j = \frac{1}{\prod_{k=1, k \neq j}^n (x_j - x_k)}.$$

The  $w_j$ 's are known as the *barycentric weights*.

Using the previous equations, the interpolating polynomial can be rewritten

$$p(x) = w(x) \sum_{j=1}^n \frac{w_j y_j}{x - x_j}$$

which is the *first barycentric form*. The computation of  $w(x)$  can be avoided by first noting that

$$1 = w(x) \sum_{j=1}^n \frac{w_j}{x - x_j}$$

which allows the interpolating polynomial to be rewritten as

$$p(x) = \frac{\sum_{j=1}^n \frac{w_j y_j}{x - x_j}}{\sum_{j=1}^n \frac{w_j}{x - x_j}}$$

This form of the Lagrange interpolant is known as the *second barycentric form* which is the form used in Barycentric Lagrange interpolation. So far, the changes made to Lagrange interpolation have resulted in an algorithm that is  $O(n)$  once the barycentric weights ( $w_j$ ) are known. The following adjustments will improve the algorithm so that it is numerically stable and later discussions will allow for the quick addition of new interpolating points after startup.

The second barycentric form makes it clear that any factors that are common to the  $w_k$  can be ignored (since they will show up in both the numerator and denominator). This allows for an important improvement to the formula that will prevent overflow error in the arithmetic. When computing the barycentric weights, each element of the product  $\prod_{k=1, k \neq j}^n (x_j - x_k)$  should be multiplied by  $C^{-1}$ , where  $4C$  is the width of the interval being interpolated ( $C$  is known as the *capacity* of the interval). In effect, this scales each barycentric weight by  $C^{1-n}$  which helps to prevent overflow during computation. Thus, the new barycentric weights are given by

$$w_j = \frac{1}{\prod_{k=1, k \neq j}^n [(x_j - x_k)/C]}.$$

Once again, this change is possible since the extra factor  $C^{1-n}$  is cancelled out in the final product. This process is summed up in the following code:

```
# Given a NumPy array xint of interpolating x-values, calculate the weights.
>>> n = len(xint)                                # Number of interpolating points.
>>> w = np.ones(n)                               # Array for storing barycentric weights.
# Calculate the capacity of the interval.
>>> C = (np.max(xint) - np.min(xint)) / 4
```

```
>>> shuffle = np.random.permutation(n-1)
>>> for j in range(n):
>>>     temp = (xint[j] - np.delete(xint, j)) / c
>>>     temp = temp[shuffle]           # Randomize order of product.
>>>     w[j] /= np.product(temp)
```

The order of `temp` was randomized so that the arithmetic does not overflow due to poor ordering (if standard ordering is used, overflow errors can be encountered since all of the points of similar magnitude are multiplied together at once). When these two fixes are combined, the Barycentric Algorithm becomes numerically stable.

**Problem 3.** Create a class that performs Barycentric Lagrange interpolation. The constructor of your class should accept two NumPy arrays which contain the  $x$  and  $y$  values of the interpolation points. Store these arrays as attributes. In the constructor, compute the corresponding barycentric weights and store the resulting array as a class attribute. Be sure that the relative ordering of the arrays remains unchanged.

Implement the `__call__()` method so that it accepts a NumPy array of values at which to evaluate the interpolating polynomial and returns an array of the evaluated points. Your class can be tested in the same way as the Lagrange function written in Problem 2

### ACHTUNG!

As currently explained and implemented, the Barycentric class from Problem 3 will fail when a point to be evaluated exactly matches one of the  $x$ -values of the interpolating points. This happens because a divide by zero error is encountered in the final step of the algorithm. The fix for this, although not required here, is quite easy: keep track of any problem points and replace the final computed value with the corresponding  $y$ -value (since this is a point that is exactly interpolated). If you do not implement this fix, just be sure not to pass in any points that exactly match your interpolating values.

Another advantage of the barycentric method is that it allows for the addition of new interpolating points in  $O(n)$  time. Given a set of existing barycentric weights  $\{w_j\}_{j=1}^n$  and a new interpolating point  $x_i$ , the new barycentric weight is given by

$$w_i = \frac{1}{\prod_{k=1}^n (x_i - x_k)}.$$

In addition to calculating the new barycentric weight, all existing weights should be updated as follows  $w_j = \frac{w_j}{x_j - x_i}$ .

**Problem 4.** Include a method in the class written in Problem 3 that allows for the addition of new interpolating points by updating the barycentric weights. Your function should accept two NumPy arrays which contain the  $x$  and  $y$  values of the new interpolation points. Update and store the old weights then extend the class attribute arrays that store the weights, and the  $x$  and  $y$  values of the interpolation points with the new data. When updating all class attributes, make sure to maintain the same relative order.

The implementation outlined here calls for the  $y$ -values of the interpolating points to be known during startup, however, these values are not needed until run-time. This allows the  $y$ -values to be changed without having to recompute the barycentric weights. This is an additional advantage of Barycentric Lagrange interpolation.

### Scipy's Barycentric Lagrange class

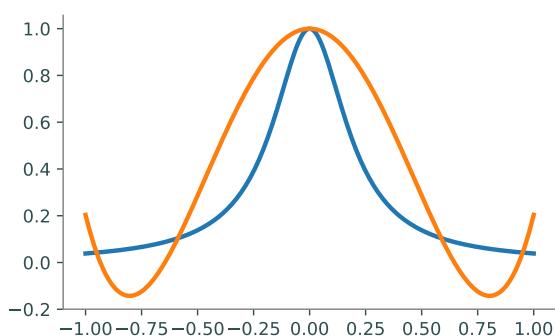
Scipy includes a Barycentric interpolator class. This class includes the same functionality as the class described in Problems 3 and 4 in addition to the ability to update the  $y$ -values of the interpolation points. The following code will produce a figure similar to Figure 14.1b.

```
>>> from scipy.interpolate import BarycentricInterpolator

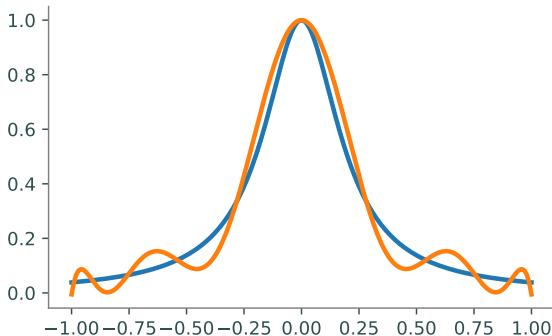
>>> f = lambda x: 1/(1+25 * x**2)    # Function to be interpolated.
# Obtain the Chebyshev extremal points on [-1,1].
>>> n = 11
>>> pts = np.linspace(-1, 1, n)
>>> domain = np.linspace(-1, 1, 200)

>>> poly = BarycentricInterpolator(pts[:-1])
>>> poly.add_xi(pts[-1])           # Oops, forgot one of the points.
>>> poly.set_yi(f(pts))          # Set the y values.

>>> plt.plot(domain, f(domain))
>>> plt.plot(domain, poly.eval(domain))
```



(a) Polynomial using 5 Chebyshev roots.



(b) Polynomial using 11 Chebyshev roots.

Figure 14.2: Example of overcoming Runge's phenomenon by using Chebyshev nodes for interpolating values. Plots made using Runge's function  $f(x) = \frac{1}{1+25x^2}$ . Compare with Figure 14.1

## Chebyshev Interpolation

### Chebyshev Nodes

As has been mentioned previously, the Barycentric version of Lagrange interpolation is a stable process that does not accumulate large errors, even with extreme inputs. However, polynomial interpolation itself is, in general, an ill-conditioned problem. Thus, even small changes in the interpolating values can give drastically different interpolating polynomials. In fact, poorly chosen interpolating points can result in a very bad approximation of a function. As more points are added, this approximation can worsen. This increase in error is called *Runge's phenomenon*.

The set of equally spaced points is an example of a set of points that may seem like a reasonable choice for interpolation but in reality produce very poor results. Figure 14.1 gives an example of this using Runge's function. As the number of interpolating points increases, the quality of the approximation deteriorates, especially near the endpoints.

Although polynomial interpolation has a great deal of potential error, a good set of interpolating points can result in fast convergence to the original function as the number of interpolating points is increased. One such set of points is the Chebyshev extremal points which are related to the Chebyshev polynomials (to be discussed shortly). The  $n + 1$  Chebyshev extremal points on the interval  $[a, b]$  are given by the formula  $y_j = \frac{1}{2}(a + b + (b - a) \cos(\frac{j\pi}{n}))$  for  $j = 0, 1, \dots, n$ . These points are shown in Figure 14.3. One important feature of these points is that they are clustered near the endpoints of the interval, this is key to preventing Runge's phenomenon.

**Problem 5.** Write a function that defines a domain  $\mathbf{x}$  of 400 equally spaced points on the interval  $[-1, 1]$ . For  $n = 2^2, 2^3, \dots, 2^8$ , repeat the following experiment.

1. Interpolate Runge's function  $f(x) = 1/(1+25x^2)$  with  $n$  equally spaced points over  $[-1, 1]$  using SciPy's `BarycentricInterpolator` class, resulting in an approximating function  $\tilde{f}$ . Compute the absolute error  $\|f(\mathbf{x}) - \tilde{f}(\mathbf{x})\|_\infty$  of the approximation using `la.norm()` with `ord=np.inf`.
2. Interpolate Runge's function with  $n + 1$  Chebyshev extremal points, also via SciPy, and compute the absolute error.

Plot the errors of each method against the number of interpolating points  $n$  in a log-log plot.

To verify that your figure make sense, try plotting the interpolating polynomials with the original function for a few of the larger values of  $n$ .

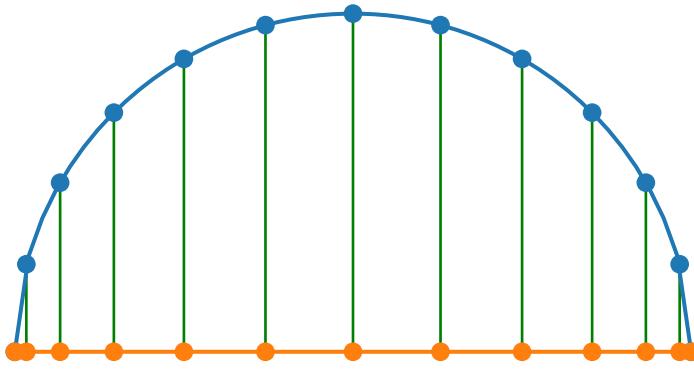


Figure 14.3: The Chebyshev extremal points. The  $n$  points where the Chebyshev polynomial of degree  $n$  reaches its local extrema. These points are also the projection onto the x-axis of  $n$  equally spaced points around the unit circle.

## Chebyshev Polynomials

The Chebyshev roots and Chebyshev extremal points are closely related to a set of polynomials known as the Chebyshev polynomials. The first two Chebyshev polynomials are defined as  $T_0(x) = 1$  and  $T_1(x) = x$ . The remaining polynomials are defined by the recursive algorithm  $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ . The Chebyshev polynomials form a complete basis for the polynomials in  $\mathbb{R}$  which means that for any polynomial  $p(x)$ , there exists a set of unique coefficients  $\{a_k\}_{k=0}^n$  such that

$$p(x) = \sum_{k=0}^n a_k T_k.$$

Finding the Chebyshev representation of an interpolating polynomial is a slow process (dominated by matrix multiplication or solving a linear system), but when the interpolating values are the Chebyshev extrema, there exists a fast algorithm for computing the Chebyshev coefficients of the interpolating polynomial. This algorithm is based on the Fast Fourier transform which has temporal complexity  $O(n \log n)$ . Given the  $n + 1$  Chebyshev extremal points  $y_j = \cos(\frac{j\pi}{n})$  for  $j = 0, 1, \dots, n$  and a function  $f$ , the unique  $n$ -degree interpolating polynomial  $p(x)$  is given by

$$p(x) = \sum_{k=0}^n a_k T_k$$

where

$$a_k = \gamma_k \Re [DFT(f(y_0), f(y_1), \dots, f(y_{2n-1}))]_k.$$

Note that although this formulation includes  $y_j$  for  $j > n$ , there are really only  $n + 1$  distinct values being used since  $y_{n-k} = y_{n+k}$ . Also,  $\Re$  denotes the real part of the Fourier transform and  $\gamma_k$  is defined as

$$\gamma_k = \begin{cases} 1 & k \in \{0, n\} \\ 2 & \text{otherwise.} \end{cases}$$

**Problem 6.** Write a function that accepts a function  $f$  and an integer  $n$ . Compute the  $n + 1$  Chebyshev coefficients for the degree  $n$  interpolating polynomial of  $f$  using the Fourier transform (`np.real()` and `np.fft.fft()` will be helpful). When using NumPy's `fft()` function, multiply every entry of the resulting array by the scaling factor  $\frac{1}{2n}$  to match the derivation given above.

Validate your function with `np.polynomial.chebyshev.poly2cheb()`. The results should be exact for polynomials.

```
# Define f(x) = -3 + 2x^2 - x^3 + x^4 by its (ascending) coefficients.
>>> f = lambda x: -3 + 2*x**2 - x**3 + x**4
>>> pcoeffs = [-3, 0, 2, -1, 1]
>>> ccoeffs = np.polynomial.chebyshev.poly2cheb(pcoeffs)

# The following callable objects are equivalent to f().
>>> fpoly = np.polynomial.Polynomial(pcoeffs)
>>> fcheb = np.polynomial.Chebyshev(ccoeffs)
```

## Lagrange vs. Chebyshev

As was previously stated, Barycentric Lagrange interpolation is  $O(n^2)$  at startup and  $O(n)$  at runtime while Chebyshev interpolation is  $O(n \log n)$ . This improved speed is one of the greatest advantages of Chebyshev interpolation. Chebyshev interpolation is also more accurate than Barycentric interpolation, even when using the same points. Despite these significant advantages in accuracy and temporal complexity, Barycentric Lagrange interpolation has one very important advantage over Chebyshev interpolation: Barycentric interpolation can be used on any set of interpolating points while Chebyshev is restricted to the Chebyshev nodes. In general, because of their better accuracy, the Chebyshev nodes are more desirable for interpolation, but there are situations when the Chebyshev nodes are not available or when specific points are needed in an interpolation. In these cases, Chebyshev interpolation is not possible and Barycentric Lagrange interpolation must be used.

## Utah Air Quality

The Utah Department of Environmental Quality has air quality stations throughout the state of Utah that measure the concentration of particles found in the air. One particulate of particular interest is  $PM_{2.5}$  which is a set of extremely fine particles known to cause tissue damage to the lungs. The file `airdata.npy` has the hourly concentration of  $PM_{2.5}$  in micrograms per cubic meter for a particular measuring station in Salt Lake County for the year 2016. The given data presents a fairly smooth function which can be reasonably approximated by an interpolating polynomial. Although Chebyshev interpolation would be preferable (because of its superior speed and accuracy), it is not possible in this case because the data is not continuous and the information at the Chebyshev nodes is not known. In order to get the best possible interpolation, it is still preferable to use points close to the Chebyshev extrema with Barycentric interpolation. The following code will take the  $n+1$  Chebyshev extrema and find the closest match in the non-continuous data found in the variable `data` then calculate the barycentric weights.

```
>>> fx = lambda a, b, n: .5*(a+b + (b-a) * np.cos(np.arange(n+1) * np.pi / n))
```

```
>>> a, b = 0, 366 - 1/24
>>> domain = np.linspace(0, b, 8784)
>>> points = fx(a, b, n)
>>> temp = np.abs(points - domain.reshape(8784, 1))
>>> temp2 = np.argmin(temp, axis=0)

>>> poly = barycentric(domain[temp2], data[temp2])
```

**Problem 7.** Write a function that interpolates the given data along the whole interval at the closest approximations to the  $n + 1$  Chebyshev extremal nodes. The function should accept  $n$ , perform the Barycentric interpolation then plot the original data and the approximating polynomial on the same domain on two separate subplots. Your interpolating polynomial should give a fairly good approximation starting at around 50 points. Note that beyond about 200 points, the given code will break down since it will attempt to return multiple of the same points causing a divide by 0 error. If you did not perform the fix suggested in the ACHTUNG box, make sure not to pass in any points that exactly match the interpolating values.

## Additional Material

The *Clenshaw Algorithm* is a fast algorithm commonly used to evaluate a polynomial given its representation in Chebyshev coefficients. This algorithm is based on the recursive relation between Chebyshev polynomials and is the algorithm used by NumPy's `polynomial.chebyshev` module.

---

**Algorithm 1** Accepts an array  $x$  of points at which to evaluate the polynomial and an array  $a = [a_0, a_1, \dots, a_{n-1}]$  of Chebyshev coefficients.

---

```

1: procedure CLENSHAWRECURSION( $x, a$ )
2:    $u_{n+1} \leftarrow 0$ 
3:    $u_n \leftarrow 0$ 
4:    $k \leftarrow n - 1$ 
5:   while  $k \geq 1$  do
6:      $u_k \leftarrow 2xu_{k+1} - u_{k+2} + a_k$ 
7:      $k \leftarrow k - 1$ 
8:   return  $a_0 + xu_1 - u_2$ 
```

---



# 15

## Gaussian Quadrature

**Lab Objective:** *Learn the basics of Gaussian quadrature and its application to numerical integration. Build a class to perform numerical integration using Legendre and Chebyshev polynomials. Compare the accuracy and speed of both types of Gaussian quadrature with the built-in Scipy package. Perform multivariate Gaussian quadrature.*

### Legendre and Chebyshev Gaussian Quadrature

It can be shown that for any class of orthogonal polynomials  $p \in \mathbb{R}[x; 2n + 1]$  with corresponding weight function  $w(x)$ , there exists a set of points  $\{x_i\}_{i=0}^n$  and weights  $\{w_i\}_{i=0}^n$  such that

$$\int_a^b p(x)w(x)dx = \sum_{i=0}^n p(x_i)w_i.$$

Since this relationship is exact, a good approximation for the integral

$$\int_a^b f(x)w(x)dx$$

can be expected as long as the function  $f(x)$  can be reasonably interpolated by a polynomial at the points  $x_i$  for  $i = 0, 1, \dots, n$ . In fact, it can be shown that if  $f(x)$  is  $2n + 1$  times differentiable, the error of the approximation will decrease as  $n$  increases.

Gaussian quadrature can be performed using any basis of orthonormal polynomials, but the most commonly used are the Legendre polynomials and the Chebyshev polynomials. Their weight functions are  $w_l(x) = 1$  and  $w_c(x) = \frac{1}{\sqrt{1-x^2}}$ , respectively, both defined on the open interval  $(-1, 1)$ .

**Problem 1.** Define a class for performing Gaussian quadrature. The constructor should accept an integer  $n$  denoting the number of points and weights to use (this will be explained later) and a label indicating which class of polynomials to use. If the label is not either "`legendre`" or "`chebyshev`", raise a `ValueError`; otherwise, store it as an attribute.

The weight function  $w(x)$  will show up later in the denominator of certain computations. Define the reciprocal function  $w(x)^{-1} = 1/w(x)$  as a `lambda` function and save it as an attribute.

## Calculating Points and Weights

All sets of orthogonal polynomials  $\{u_k\}_{k=0}^n$  satisfy the three-term recurrence relation

$$u_0 = 1, \quad u_1 = x - \alpha_1, \quad u_{k+1} = (x - \alpha_k)u_k - \beta_k u_{k-1}$$

for some coefficients  $\{\alpha_k\}_{k=1}^n$  and  $\{\beta_k\}_{k=1}^n$ . For the Legendre polynomials, they are given by

$$\alpha_k = 0, \quad \beta_k = \frac{k^2}{4k^2 - 1},$$

and for the Chebyshev polynomials, they are

$$\alpha_k = 0, \quad \beta_k = \begin{cases} \frac{1}{2} & \text{if } k = 1 \\ \frac{1}{4} & \text{otherwise.} \end{cases}$$

Given these values, the corresponding *Jacobi matrix* is defined as follows.

$$J = \begin{bmatrix} \alpha_1 & \sqrt{\beta_1} & 0 & \dots & 0 \\ \sqrt{\beta_1} & \alpha_2 & \sqrt{\beta_2} & \dots & 0 \\ 0 & \sqrt{\beta_2} & \alpha_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & & \sqrt{\beta_{n-1}} & \\ 0 & \dots & \sqrt{\beta_{n-1}} & & \alpha_n \end{bmatrix}$$

According to the *Golub-Welsch algorithm*,<sup>1</sup> the  $n$  eigenvalues of  $J$  are the points  $x_i$  to use in Gaussian quadrature, and the corresponding weights are given by  $w_i = \mu_w(\mathbb{R})v_{i,0}^2$  where  $v_{i,0}$  is the first entry of the  $i$ th eigenvector and  $\mu_w(\mathbb{R}) = \int_{-\infty}^{\infty} w(x)dx$  is the *measure* of the weight function. Since the weight functions for Legendre and Chebyshev polynomials have compact support on the interval  $(-1, 1)$ , their measures are given as follows.

$$\mu_{w_l}(\mathbb{R}) = \int_{-\infty}^{\infty} w_l(x)dx = \int_{-1}^1 1dx = 2 \quad \mu_{w_c}(\mathbb{R}) = \int_{-\infty}^{\infty} w_c(x)dx = \int_{-1}^1 \frac{1}{\sqrt{1-x^2}}dx = \pi$$

**Problem 2.** Write a method for your class from Problem 1 that accepts an integer  $n$ . Construct the  $n \times n$  Jacobi matrix  $J$  for the polynomial family indicated in the constructor. Use SciPy to compute the eigenvalues and eigenvectors of  $J$ , then compute the points  $\{x_i\}_{i=1}^n$  and weights  $\{w_i\}_{i=1}^n$  for the quadrature. Return both the array of points and the array weights.

Test your method by checking your points and weights against the following values using the Legendre polynomials with  $n = 5$ .

$x_i$	$-\frac{1}{3}\sqrt{5 + 2\sqrt{\frac{10}{7}}}$	$-\frac{1}{3}\sqrt{5 - 2\sqrt{\frac{10}{7}}}$	0	$\frac{1}{3}\sqrt{5 - 2\sqrt{\frac{10}{7}}}$	$\frac{1}{3}\sqrt{5 + 2\sqrt{\frac{10}{7}}}$
$w_i$	$\frac{322 - 13\sqrt{70}}{900}$	$\frac{322 + 13\sqrt{70}}{900}$	128	$\frac{322 + 13\sqrt{70}}{900}$	$\frac{322 - 13\sqrt{70}}{900}$

<sup>1</sup>See <http://gubner.ece.wisc.edu/gaussquad.pdf> for a complete treatment of the Golub-Welsch algorithm, including the computation of the recurrence relation coefficients for arbitrary orthogonal polynomials.

Finally, modify the constructor of your class so that it calls your new function and stores the resulting points and weights as attributes.

(Note: The order of the points and weights in the table may differ depending on whether you used `scipy.linalg.eig()` or `scipy.linalg.eigh()`. The order doesn't matter but it is important that each point corresponds to the correct weight.)

## UNIT TEST

In the file `test_gaussian_quadrature.py`, write unit tests for the `points_weights()` method in your `GaussianQuadrature` class. Some unit tests have been provided for Problem 1.

**Note:** The unit tests provided may have different attribute names than those you have written for your class. You may adjust the provided tests to match your code, or change your code to conform with the given tests.

## Integrating with Given Weights and Points

Now that the points and weights have been obtained, they can be used to approximate the integrals of different functions. For a given function  $f(x)$  with points  $x_i$  and weights  $w_i$ ,

$$\int_{-1}^1 f(x)w(x)dx \approx \sum_{i=1}^n f(x_i)w_i.$$

There are two problems with the preceding formula. First, the weight function is part of the integral being approximated, and second, the points obtained are only found on the interval  $(-1, 1)$  (in the case of the Legendre and Chebyshev polynomials). To solve the first problem, define a new function  $g(x) = f(x)/w(x)$  so that

$$\int_{-1}^1 f(x)dx = \int_{-1}^1 g(x)w(x)dx \approx \sum_{i=1}^n g(x_i)w_i. \quad (15.1)$$

The integral of  $f(x)$  on  $[-1, 1]$  can thus be approximated with the inner product  $\mathbf{w}^T g(\mathbf{x})$ , where  $g(\mathbf{x}) = [g(x_1), \dots, g(x_n)]^T$  and  $\mathbf{w} = [w_1, \dots, w_n]^T$ .

**Problem 3.** Write a method for your class that accepts a callable function  $f$ . Use (15.1) and the stored points and weights to approximate of the integral of  $f$  on the interval  $[-1, 1]$ .  
(Hint: Use  $w(x)^{-1}$  from Problem 1 to compute  $g(x)$  without division.)

Test your method with examples that are easy to compute by hand and by comparing your results to `scipy.integrate.quad()` (The answer given below is found by using Chebyshev, even though the default mode is Lagrange.)

```
>>> import numpy as np
>>> from scipy.integrate import quad

# Integrate f(x) = 1 / sqrt(1 - x**2) from -1 to 1.
>>> f = lambda x: 1 / np.sqrt(1 - x**2)
```

```
>>> quad(f, -1, 1)[0]
3.141592653589591
```

### NOTE

Since the points and weights for Gaussian quadrature do not depend on  $f$ , they only need to be computed once and can then be reused to approximate the integral of any function. The class structure in Problems 1–4 takes advantage of this fact, but `scipy.integrate.quad()` does not. If a larger  $n$  is needed for higher accuracy, however, the computations must be repeated to get a new set of points and weights.

## Shifting the Interval of Integration

Since the weight functions for the Legendre and Chebyshev polynomials have compact support on the interval  $(-1, 1)$ , all of the quadrature points are found on that interval as well. To integrate a function on an arbitrary interval  $[a, b]$  requires a change of variables. Let

$$u = \frac{2x - b - a}{b - a}$$

so that  $u = -1$  when  $x = a$  and  $u = 1$  when  $x = b$ . Then

$$x = \frac{b-a}{2}u + \frac{a+b}{2} \quad \text{and} \quad dx = \frac{b-a}{2}du,$$

so the transformed integral is given by

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 f\left(\frac{b-a}{2}u + \frac{a+b}{2}\right) du.$$

By defining a new function  $h(x)$  as

$$h(x) = f\left(\frac{(b-a)}{2}x + \frac{(a+b)}{2}\right),$$

the integral of  $f$  can be approximated by integrating  $h$  over  $[-1, 1]$  with (15.1). This results in the final quadrature formula

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 h(x)dx = \frac{b-a}{2} \int_{-1}^1 g(x)w(x)dx \approx \frac{b-a}{2} \sum_{i=1}^n g(x_i)w_i, \quad (15.2)$$

where now  $g(x) = h(x)/w(x)$ .

**Problem 4.** Write a method for your class that accepts a callable function  $f$  and bounds of integration  $a$  and  $b$ . Use (15.2) to approximate the integral of  $f$  from  $a$  to  $b$ .  
(Hint: Define  $h(x)$  and use your method from Problem 3.)

**Problem 5.** The *standard normal distribution* has the following probability density function.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

This function has no symbolic antiderivative, so it can only be integrated numerically. The following code gives an “exact” value of the integral of  $f(x)$  from  $-\infty$  to a specified value.

```
>>> from scipy.stats import norm

>>> norm.cdf(1)                                # Integrate f from -infinity to 1.
0.84134474606854293
>>> norm.cdf(1) - norm.cdf(-1)                 # Integrate f from -1 to 1.
0.68268949213708585
```

Write a function that uses `scipy.stats` to calculate the “exact” value

$$F = \int_{-3}^2 f(x) dx.$$

Then repeat the following experiment for  $n = 5, 10, 15, \dots, 50$ .

1. Use your class from Problems 1–4 with the Legendre polynomials to approximate  $F$  using  $n$  points and weights. Calculate and record the error of the approximation.
2. Use your class with the Chebyshev polynomials to approximate  $F$  using  $n$  points and weights. Calculate and record the error of the approximation.

Plot the errors against the number of points and weights  $n$ , using a log scale for the  $y$ -axis. Finally, plot a horizontal line showing the error of `scipy.integrate.quad()` (which doesn’t depend on  $n$ ).

## Multivariate Quadrature

The extension of Gaussian quadrature to higher dimensions is fairly straightforward. The same set of points  $\{z_i\}_{i=1}^n$  and weights  $\{w_i\}_{i=1}^n$  can be used in each direction, so the only difference from 1-D quadrature is how the function is shifted and scaled. To begin, let  $h : \mathbb{R}^2 \rightarrow \mathbb{R}$  and define  $g : \mathbb{R}^2 \rightarrow \mathbb{R}$  by  $g(x, y) = h(x, y)/(w(x)w(y))$  so that

$$\int_{-1}^1 \int_{-1}^1 h(x, y) dx dy = \int_{-1}^1 \int_{-1}^1 g(x, y) w(x) w(y) dx dy \approx \sum_{i=1}^n \sum_{j=1}^n w_i w_j g(z_i, z_j). \quad (15.3)$$

To integrate  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  over an arbitrary box  $[a_1, b_1] \times [a_2, b_2]$ , set

$$h(x, y) = f\left(\frac{b_1 - a_1}{2}x + \frac{a_1 + b_1}{2}, \frac{b_2 - a_2}{2}y + \frac{a_2 + b_2}{2}\right)$$

so that

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x) dx dy = \frac{(b_1 - a_1)(b_2 - a_2)}{4} \int_{-1}^1 \int_{-1}^1 h(x, y) dx dy. \quad (15.4)$$

Combining (15.3) and (15.4) gives the final 2-D Gaussian quadrature formula. Compare it to (15.2).

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x) dx dy \approx \frac{(b_1 - a_1)(b_2 - a_2)}{4} \sum_{i=1}^n \sum_{j=1}^n w_i w_j g(z_i, z_j) \quad (15.5)$$

**Problem 6.** Write a method for your class that accepts a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  (which actually accepts two separate arguments, not one array with two elements) and bounds of integration  $a_1$ ,  $a_2$ ,  $b_1$ , and  $b_2$ . Use (15.5) to compute the double integral

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x) dx dy.$$

Validate your method by comparing it `scipy.integrate.nquad()`. Note carefully that this function has slightly different syntax for the bounds of integration.

```
>>> from scipy.integrate import nquad

# Integrate f(x,y) = sin(x) + cos(y) over [-10,10] in x and [-1,1] in y.
>>> f = lambda x, y: np.sin(x) + np.cos(y)
>>> nquad(f, [[-10, 10], [-1, 1]])[0]
33.658839392315855
```

### NOTE

Although Gaussian quadrature can obtain reasonable approximations in lower dimensions, it quickly becomes intractable in higher dimensions due to the curse of dimensionality. In other words, the number of points and weights required to obtain a good approximation becomes so large that Gaussian quadrature become computationally infeasible. For this reason, high-dimensional integrals are often computed via *Monte Carlo methods*, numerical integration techniques based on random sampling. However, quadrature methods are generally significantly more accurate in lower dimensions than Monte Carlo methods.

# 16

## One-dimensional Optimization

**Lab Objective:** *Most mathematical optimization problems involve estimating the minimizer(s) of a scalar-valued function. Many algorithms for optimizing functions with a high-dimensional domain depend on routines for optimizing functions of a single variable. There are many techniques for optimization in one dimension, each with varying degrees of precision and speed. In this lab, we implement the golden section search method, Newton's method, and the secant method, then apply them to the backtracking problem.*

### Golden Section Search

A function  $f : [a, b] \rightarrow \mathbb{R}$  satisfies the *unimodal property* if it has exactly one local minimum and is monotonic on either side of the minimizer. In other words,  $f$  decreases from  $a$  to its minimizer  $x^*$ , then increases up to  $b$  (see Figure 16.1). The *golden section search* method optimizes a unimodal function  $f$  by iteratively defining smaller and smaller intervals containing the unique minimizer  $x^*$ . This approach is especially useful if the function's derivative does not exist, is unknown, or is very costly to compute.

By definition, the minimizer  $x^*$  of  $f$  must lie in the interval  $[a, b]$ . To shrink the interval around  $x^*$ , we test the following strategically chosen points.

$$\tilde{a} = b - \frac{b-a}{\varphi} \quad \tilde{b} = a + \frac{b-a}{\varphi}$$

Here  $\varphi = \frac{1+\sqrt{5}}{2}$  is the *golden ratio*. At each step of the search,  $[a, b]$  is refined to either  $[a, \tilde{b}]$  or  $[\tilde{a}, b]$ , called the *golden sections*, depending on the following criteria.

If  $f(\tilde{a}) < f(\tilde{b})$ , then since  $f$  is unimodal, it must be increasing in a neighborhood of  $\tilde{b}$ . The unimodal property also guarantees that  $f$  must be increasing on  $[\tilde{b}, b]$  as well, so  $x^* \in [a, \tilde{b}]$  and we set  $b = \tilde{b}$ . By similar reasoning, if  $f(\tilde{a}) > f(\tilde{b})$ , then  $x^* \in [\tilde{a}, b]$  and we set  $a = \tilde{a}$ . If, however,  $f(\tilde{a}) = f(\tilde{b})$ , then the unimodality of  $f$  does not guarantee anything about where the minimizer lies. Assuming either  $x^* \in [a, \tilde{b}]$  or  $x^* \in [\tilde{a}, b]$  allows the iteration to continue, but the method is no longer guaranteed to converge to the local minimum.

At each iteration, the length of the search interval is divided by  $\varphi$ . The method therefore converges linearly, which is somewhat slow. However, the idea is simple and each step is computationally inexpensive.

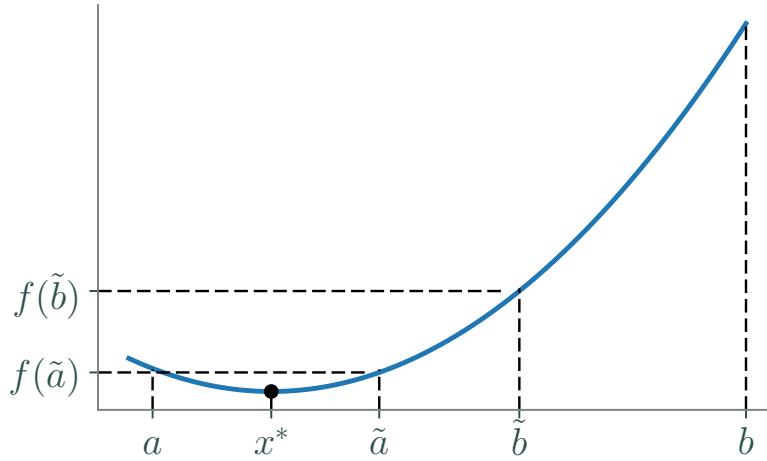


Figure 16.1: The unimodal  $f : [a, b] \rightarrow \mathbb{R}$  can be minimized with a golden section search. For the first iteration,  $f(\tilde{a}) < f(\tilde{b})$ , so  $x^* \in [a, \tilde{b}]$ . New values of  $\tilde{a}$  and  $\tilde{b}$  are then calculated from this new, smaller interval.

---

**Algorithm 1** The Golden Section Search

---

```

1: procedure GOLDEN_SECTION( $f, a, b, \text{tol}, \text{maxiter}$ )
2:    $x_0 \leftarrow (a + b)/2$             $\triangleright$  Set the initial minimizer approximation as the interval midpoint.
3:    $\varphi = (1 + \sqrt{5})/2$ 
4:   for  $i = 1, 2, \dots, \text{maxiter}$  do            $\triangleright$  Iterate only  $\text{maxiter}$  times at most.
5:      $c \leftarrow (b - a)/\varphi$ 
6:      $\tilde{a} \leftarrow b - c$ 
7:      $\tilde{b} \leftarrow a + c$ 
8:     if  $f(\tilde{a}) \leq f(\tilde{b})$  then            $\triangleright$  Get new boundaries for the search interval.
9:        $b \leftarrow \tilde{b}$ 
10:    else
11:       $a \leftarrow \tilde{a}$ 
12:     $x_1 \leftarrow (a + b)/2$             $\triangleright$  Set the minimizer approximation as the interval midpoint.
13:    if  $|x_0 - x_1| < \text{tol}$  then            $\triangleright$  Stop iterating if the approximation stops changing enough.
14:      break
15:     $x_0 \leftarrow x_1$ 
16:  return  $x_1$ 

```

---

**Problem 1.** Write a function that accepts a function  $f : \mathbb{R} \rightarrow \mathbb{R}$ , interval limits  $a$  and  $b$ , a stopping tolerance  $\text{tol}$ , and a maximum number of iterations  $\text{maxiter}$ . Use Algorithm 1 to implement the golden section search. Return the approximate minimizer  $x^*$ , whether or not the algorithm converged (`true` or `false`), and the number of iterations computed.

Test your function by minimizing  $f(x) = e^x - 4x$  on the interval  $[0, 3]$ , then plotting the function and the computed minimizer together. Also compare your results to SciPy's golden section search, `scipy.optimize.golden()`.

```
>>> from scipy import optimize as opt
>>> import numpy as np

>>> f = lambda x : np.exp(x) - 4*x
>>> opt.golden(f, brack=(0,3), tol=.001)
1.3862578679031485                                # ln(4) is the minimizer.
```

## Newton's Method

*Newton's method* is an important root-finding algorithm that can also be used for optimization. Given  $f : \mathbb{R} \rightarrow \mathbb{R}$  and a good initial guess  $x_0$ , the sequence  $(x_k)_{k=1}^{\infty}$  generated by the recursive rule

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

converges to a point  $\bar{x}$  satisfying  $f(\bar{x}) = 0$ . The first-order necessary conditions from elementary calculus state that if  $f$  is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of  $f'$  is a way to identify potential minima or maxima of  $f$ . Specifically, starting with an initial guess  $x_0$ , set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{16.1}$$

and iterate until  $|x_k - x_{k-1}|$  is satisfactorily small. Note that this procedure does not use the actual function  $f$  at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

Each step of (16.1) can be thought of approximating the objective function  $f$  by a quadratic function  $q$  and finding its unique extrema. That is, we first approximate  $f$  with its second-degree Taylor polynomial centered at  $x_k$ .

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

This quadratic function satisfies  $q(x_k) = f(x_k)$  and matches  $f$  fairly well close to  $x_k$ . Thus the optimizer of  $q$  is a reasonable guess for an optimizer of  $f$ . To compute that optimizer, solve  $q'(x) = 0$ .

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \implies x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (16.1) using  $x_{k+1}$  for  $x$ . See Figure 16.2.

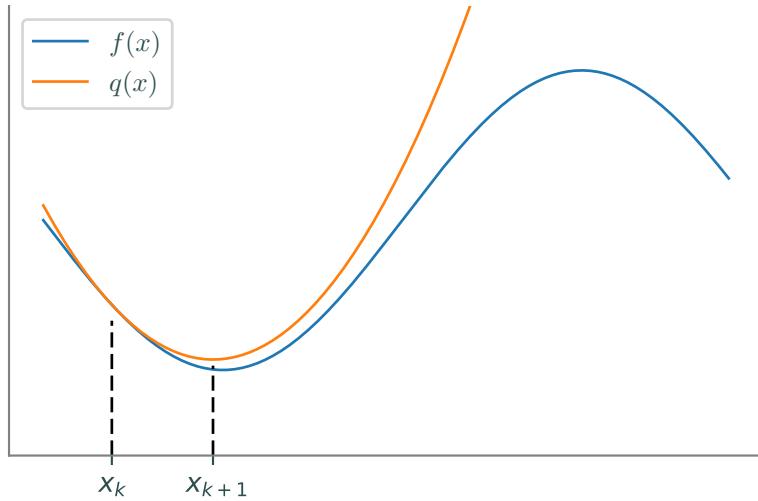


Figure 16.2: A quadratic approximation of  $f$  at  $x_k$ . The minimizer  $x_{k+1}$  of  $q$  is close to the minimizer of  $f$ .

Newton's method for optimization works well to locate minima when  $f''(x) > 0$  on the entire domain. However, it may fail to converge to a minimizer if  $f''(x) \leq 0$  for some portion of the domain. If  $f$  is not unimodal, the initial guess  $x_0$  must be sufficiently close to a local minimizer  $x^*$  in order to converge.

**Problem 2.** Let  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Write a function that accepts  $f'$ ,  $f''$ , a starting point  $x_0$ , a stopping tolerance  $\text{tol}$ , and a maximum number of iterations  $\text{maxiter}$ . Implement Newton's method using (16.1) to locate a local optimizer. Return the approximate optimizer, whether or not the algorithm converged, and the number of iterations computed.

Test your function by minimizing  $f(x) = x^2 + \sin(5x)$  with an initial guess of  $x_0 = 0$ . Compare your results to `scipy.optimize.newton()`, which implements the root-finding version of Newton's method.

```
>>> df = lambda x : 2*x + 5*np.cos(5*x)
>>> d2f = lambda x : 2 - 25*np.sin(5*x)
>>> opt.newton(df, x0=0, fprime=d2f, tol=1e-10, maxiter=500)
-1.4473142236328096
```

Note that other initial guesses can yield different minima for this function.

## The Secant Method

The second derivative of an objective function is not always known or may be prohibitively expensive to evaluate. The *secant method* solves this problem by numerically approximating the second derivative with a difference quotient.

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

Selecting  $x = x_k$  and  $h = x_{k-1} - x_k$  gives the following approximation.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \quad (16.2)$$

Inserting (16.2) into (16.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1}f'(x_k) - x_kf'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \quad (16.3)$$

Notice that this recurrence relation requires two previous points (both  $x_k$  and  $x_{k-1}$ ) to calculate the next estimate. This method converges superlinearly—slower than Newton’s method, but faster than the golden section search—with convergence criteria similar to Newton’s method.

**Problem 3.** Write a function that accepts a first derivative  $f'$ , starting points  $x_0$  and  $x_1$ , a stopping tolerance `tol`, and a maximum of iterations `maxiter`. Use (16.3) to implement the Secant method. Try to make as few computations as possible by only computing  $f'(x_k)$  once for each  $k$ . Return the minimizer approximation, whether or not the algorithm converged, and the number of iterations computed.

Test your code with the function  $f(x) = x^2 + \sin(x) + \sin(10x)$  and with initial guesses of  $x_0 = 0$  and  $x_1 = -1$ . Plot your answer with the graph of the function. Also compare your results to `scipy.optimize.newton()`; without providing the `fprime` argument, this function uses the secant method. However, it still only takes in one initial condition, so it may converge to a different local minimum than your function.

```
>>> df = lambda x: 2*x + np.cos(x) + 10*np.cos(10*x)
>>> opt.newton(df, x0=0, tol=1e-10, maxiter=500)
0.45308663951300454
```

## Descent Methods

Consider now a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ . *Descent methods*, also called *line search methods*, are optimization algorithms that create a convergent sequence  $(x_k)_{k=1}^\infty$  by the following rule.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (16.4)$$

Here  $\alpha_k \in \mathbb{R}$  is called the *step size* and  $\mathbf{p}_k \in \mathbb{R}^n$  is called the *search direction*. The choice of  $\mathbf{p}_k$  is usually what distinguishes an algorithm; in the one-dimensional case ( $n = 1$ ),  $p_k = f'(x_k)/f''(x_k)$  results in Newton’s method, and using the approximation in (16.2) results in the secant method.

To be effective, a descent method must also use a good step size  $\alpha_k$ . If  $\alpha_k$  is too large, the method may repeatedly overstep the minimum; if  $\alpha_k$  is too small, the method may converge extremely slowly. See Figure 16.3.

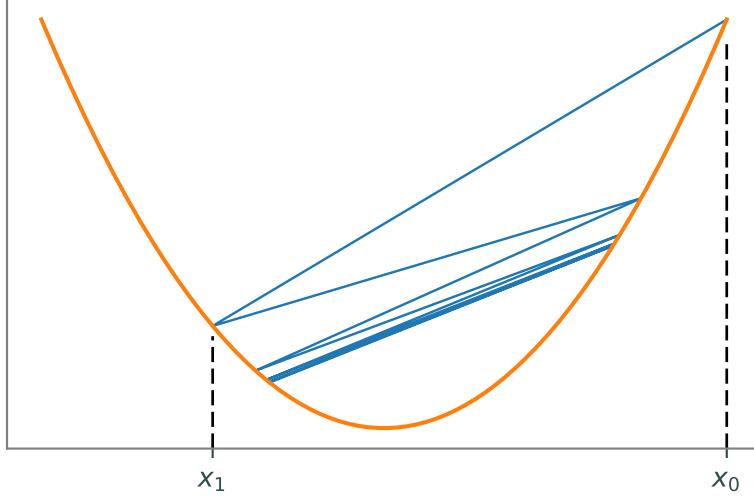


Figure 16.3: If the step size  $\alpha_k$  is too large, a descent method may repeatedly overstep the minimizer.

Given a search direction  $\mathbf{p}_k$ , the best step size  $\alpha_k$  minimizes the function  $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha\mathbf{p}_k)$ . Since  $f$  is scalar-valued,  $\phi_k : \mathbb{R} \rightarrow \mathbb{R}$ , so any of the optimization methods discussed previously can be used to minimize  $\phi_k$ . However, computing the best  $\alpha_k$  at every iteration is not always practical. Instead, some methods use a cheap routine to compute a step size that may not be optimal, but which is good enough. The most common approach is to find an  $\alpha_k$  that satisfies the *Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (16.5)$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (16.6)$$

where  $0 < c_1 < c_2 < 1$  (for the best results, choose  $c_1 \ll c_2$ ). The condition (16.5) is also called the *Armijo rule* and ensures that the step decreases  $f$ . However, this condition is not enough on its own. By Taylor's theorem,

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small  $\alpha_k$  will always satisfy (16.5) since  $Df(\mathbf{x}_k)^T \mathbf{p}_k < 0$  (as  $\mathbf{p}_k$  is a descent direction). The condition (16.6), called the *curvature condition*, ensures that the  $\alpha_k$  is large enough for the algorithm to make significant progress.

It is possible to find an  $\alpha_k$  that satisfies the Wolfe conditions, but that is far from the minimizer of  $\phi_k(\alpha)$ . The *strong Wolfe conditions* modify (16.6) to ensure that  $\alpha_k$  is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k| \leq c_2 |Df(\mathbf{x}_k)^T \mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (16.6):

$$f(\mathbf{x}_k) + (1 - c)\alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k,$$

where  $0 < c < 1$ . These conditions are very similar to the Wolfe conditions (the right inequality is (16.5)), but they do not require the calculation of the directional derivative  $Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k$ .

## Backtracking

A *backtracking line search* is a simple strategy for choosing an acceptable step size  $\alpha_k$ : start with an fairly large initial step size  $\alpha$ , then repeatedly scale it down by a factor  $\rho$  until the desired conditions are satisfied. The following algorithm only requires  $\alpha$  to satisfy (16.5). This is usually sufficient, but if it finds  $\alpha$ 's that are too small, the algorithm can be modified to satisfy (16.6) or one of its variants.

---

**Algorithm 2** Backtracking using the Armijo Rule

---

```

1: procedure BACKTRACKING( $f$ ,  $Df$ ,  $\mathbf{x}_k$ ,  $\mathbf{p}_k$ ,  $\alpha$ ,  $\rho$ ,  $c$ )
2:    $Dfp \leftarrow Df(\mathbf{x}_k)^T \mathbf{p}_k$                                  $\triangleright$  Compute these values only once.
3:    $fx \leftarrow f(\mathbf{x}_k)$ 
4:   while  $(f(\mathbf{x}_k + \alpha \mathbf{p}_k) > fx + c\alpha Dfp)$  do
5:      $\alpha \leftarrow \rho\alpha$ 
return  $\alpha$ 
```

---

**Problem 4.** Write a function that accepts a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , its derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , an approximate minimizer  $\mathbf{x}_k$ , a search direction  $\mathbf{p}_k$ , an initial step length  $\alpha$ , and parameters  $\rho$  and  $c$ . Implement the backtracking method of Algorithm 2. Return the computed step size.

The functions  $f$  and  $Df$  should both accept 1-D NumPy arrays of length  $n$ . For example, if  $f(x, y, z) = x^2 + y^2 + z^2$ , then  $f$  and  $Df$  could be defined as follows.

```
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> Df = lambda x: np.array([2*x[0], 2*x[1], 2*x[2]])
```

SciPy's `scipy.optimize.linesearch.scalar_search_armijo()` finds an acceptable step size using the Armijo rule. It may not give the exact answer as your implementation since it decreases  $\alpha$  differently, but the answers should be similar.

```
>>> from scipy.optimize import linesearch
>>> from jax import numpy as jnp
>>> from jax import grad

# Get a step size for  $f(x,y,z) = x^2 + y^2 + z^2$ .
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> x = jnp.array([150., .03, 40.])          # Current minimizer guesss.
>>> p = jnp.array([-5., -100., -4.5])        # Current search direction.
>>> phi = lambda alpha: f(x + alpha*p)         # Define phi(alpha).
>>> dphi = grad(phi)
>>> alpha, _ = linesearch.scalar_search_armijo(phi, phi(0.), dphi(0.))
```



# 17

# Regular Expressions

**Lab Objective:** Cleaning and formatting data are fundamental problems in data science. Regular expressions are an important tool for working with text carefully and efficiently, and are useful for both gathering and cleaning data. This lab introduces regular expression syntax and common practices, including an application to a data cleaning problem. This link may be helpful as a reference: <https://docs.python.org/3/library/re.html>

A *regular expression* or *regex* is a string of characters that follows a certain syntax to specify a pattern, like generalized shorthand for strings. Strings that follow the pattern are said to *match* the expression (and vice versa). A single regular expression can match a large set of strings, such as the set of all valid email addresses.

## ACHTUNG!

There are some universal standards for regular expression syntax, but the exact syntax varies slightly depending on the program or language. However, the syntax presented in this lab (for Python) is sufficiently similar to any other regex system. Consider learning to use regular expressions in Vim or your favorite text editor, keeping in mind that there will be slight syntactic differences from what is presented here.

## Regular Expression Syntax in Python

The `re` module implements regular expressions in Python. The function `re.compile()` takes in a regular expression string and returns a corresponding *pattern* object, which has methods for determining if and how other strings match the pattern. You can think of the `re.compile` object as a box with a certain shape cut out of the bottom. When a lot of differently shaped objects are put into the box and shaken around only the objects with the same exact shape as the one cut out of the box will fall out. One method that `re.compile()` uses is the `search()` method, which returns `None` for a string that doesn't match, and a *match* object for a string that does match.

The `match()` method is different than the `match` object mentioned previously. The `match` method only matches strings that satisfy the pattern **at the beginning** of the string. To answer the question “does any part of my target string match this regular expression?” always use the `search()` method.

```
>>> import re
>>> pattern = re.compile("cat")      # Make a pattern object for finding 'cat'.
>>> bool(pattern.search("cat"))     # 'cat' matches 'cat', of course.
True
>>> bool(pattern.match("catfish"))   # 'catfish' starts with 'cat'.
True
>>> bool(pattern.match("fishcat"))   # 'fishcat' doesn't start with 'cat'.
False
>>> bool(pattern.search("fishcat"))  # but it does contain 'cat'.
True
>>> bool(pattern.search("hat"))     # 'hat' does not contain 'cat'.
False
```

Most of the functions in the `re` module are shortcuts for compiling a pattern object and calling one of its methods. Using `re.compile()` is good practice because the resulting object (analogously, the box you made) is reusable, while each call to `re.search()` compiles a new (but redundant) pattern object. For example, the following lines of code are equivalent.

```
>>> bool(re.compile("cat").search("catfish"))
True
>>> bool(re.search("cat", "catfish"))
True
```

Assigning `re.compile("cat").search("catfish")` or `re.search("cat", "catfish")` without the `bool()` around them to variables will create match objects.

**Problem 1.** Write a function that compiles and returns a regular expression pattern object with the pattern string "`python`".

## Literal Characters and Metacharacters

The following string characters (separated by spaces) are *metacharacters* in Python's regular expressions, meaning they have special significance in a pattern string:

. ^ \$ \* + ? { } [ ] \ | ( ).

A regular expression that matches strings with one or more metacharacters requires two things.

1. Use *raw strings* instead of regular Python strings by prefacing the string with an `r`, such as `r"cat"`. The resulting string interprets backslashes as actual backslash characters, rather than the start of an escape sequence like `\n` or `\t`.
2. Preface any metacharacters with a backslash to indicate a literal character. For example, to match the string "\$3.99? Thanks.", use `r"\$3\.99\? Thanks\."`.

Without raw strings, every backslash has to be written as a double backslash, which makes many regular expression patterns hard to read (`"\\\$3\\.99\\? Thanks\\."`).

**Problem 2.** Write a function that compiles and returns a regular expression pattern object that matches the string "`~{@}{?}{%}{.}{*}{_}{&}`\$".

**Hint:** There are online sites like <https://regex101.com/> that can help check answers. Consider building regex expressions one character at a time at this website.

The regular expressions of Problems 1 and 2 only match strings that are or include the exact pattern. The metacharacters allow regular expressions to have much more flexibility and control so that a single pattern can match a wide variety of strings, or a very specific set of strings. The *line anchor* metacharacters `^` and `$` are used to match the **start** and the **end** of a line of text, respectively. This shrinks the matching set, even when using the `search()` method instead of the `match()` method. For example, the only single-line string that the expression '`~x$`' matches is '`x`', whereas the expression '`x`' can match any string with an '`x`' in it.

The *pipe* character `|` is a logical OR in a regular expression: `A|B` matches A or B. The parentheses `()` create a *group* in a regular expression. A group establishes an order of operations in an expression. For example, in the regex "`^one|two fish$`", precedence is given to the invisible string concatenation between "`two`" and "`fish`", while "`^(one|two) fish$`" gives precedence to the '`|`' metacharacter. Notice that the *pipe* is inside the *group*.

```
>>> fish = re.compile(r"^(one|two) fish$")
>>> for test in ["one fish", "two fish", "red fish", "one two fish"]:
...     print(test + ':', bool(fish.search(test)))
...
one fish: True
two fish: True
red fish: False
one two fish: False
```

**Problem 3.** Write a function that compiles and returns a regular expression pattern object that matches the following strings, and no other strings, even with `re.search()`.

`"Book store"`    `"Mattress store"`    `"Grocery store"`  
`"Book supplier"`    `"Mattress supplier"`    `"Grocery supplier"`

**Hint:** The naive way to do this is create a very long chain of `or` operators with the exact phrases as options. Instead, think about dividing it into two groups of `or` operators where the first group picks the first word and the second group picks the second word.

There is a file called `test_regular_expressions.py` that contains some prewritten unit tests to help you test your function for this problem.

## Character Classes

The hard bracket metacharacters `[` and `]` are used to create *character classes*, a part of a regular expression that can match a variety of characters. For example, the pattern `[abc]` matches any of the characters `a`, `b`, or `c`. This is different than a group delimited by parentheses: a group can match multiple characters, while a character class matches only one character. For instance, `[abc]` does not match `ab` or `abc`, and `(abc)` matches `abc` but not `ab` or even `a`.

Within character classes, there are two additional metacharacters. When `^` appears **as the first character** in a character class, right after the opening bracket `[`, the character class matches anything **not** specified instead. In other words, `^` is the set complement operation on the character class. Additionally, the dash `-` specifies a range of values. For instance, `[0-9]` matches any digit, and `[a-z]` matches any lowercase letter. Thus `[^0-9]` matches any character **except** for a digit, and `[^a-z]` matches any character **except** for lowercase letters. Keep in mind that the dash `-`, when at the beginning or end of the character class, will match the literal `'-'`. Note that `[0-27-9]` acts like `[(0-2)|(7-9)]`.

```
>>> p1, p2 = re.compile(r"^[a-z][^0-7]$"), re.compile(r"^[^abcA-C][0-27-9]$")
>>> for test in ["d8", "aa", "E9", "EE", "d88"]:
...     print(test + ':', bool(p1.search(test)), bool(p2.search(test)))
...
d8: True True
aa: True False # a is not in [^abcA-C] or [0-27-9].
E9: False True # E is not in [a-z].
EE: False False # E is not in [a-z] or [0-27-9].
d88: False False # Too many characters.
```

There are also a variety of shortcuts that represent common character classes, listed in Table 17.1. Familiarity with these shortcuts makes some regular expressions significantly more readable.

Character	Description
<code>\b</code>	Matches the empty string, but only at the start or end of a word.
<code>\s</code>	Matches any whitespace character; equivalent to <code>[ \t\n\r\f\v]</code> .
<code>\S</code>	Matches any non-whitespace character; equivalent to <code>[^\s]</code> .
<code>\d</code>	Matches any decimal digit; equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches any non-digit character; equivalent to <code>[^\d]</code> .
<code>\w</code>	Matches any alphanumeric character; equivalent to <code>[a-zA-Z0-9_]</code> .
<code>\W</code>	Matches any non-alphanumeric character; equivalent to <code>[^\w]</code> .

Table 17.1: Character class shortcuts.

Any of the character class shortcuts can be used within other custom character classes. For example, `[_A-Z\s]` matches an underscore, capital letter, or whitespace character.

Finally, a period `.` matches **any** character except for a line break. This is a very powerful metacharacter; be careful to only use it when part of the regular expression really should match **any** character.

```
# Match any three-character string with a digit in the middle.
>>> pattern = re.compile(r"^.d.$")
>>> for test in ["a0b", "888", "n2%", "abc", "cat"]:
...     print(test + ':', bool(pattern.search(test)))
...
a0b: True
888: True
n2%: True
abc: False
cat: False
```

```
# Match two letters followed by a number and two non-newline characters.
>>> pattern = re.compile(r"^[a-zA-Z][a-zA-Z]\d..$")
>>> for test in ["tk421", "bb8!?", "JB007", "Boba?"]:
...     print(test + ':', bool(pattern.search(test)))
...
tk421: True
bb8!?: True
JB007: True
Boba?: False
```

The following table is a useful recap of some common regular expression metacharacters.

Character	Description
.	Matches any character except a newline.
^	Matches the start of the string.
\$	Matches the end of the string or just before the newline at the end of the string.
	A B creates an regular expression that will match either A or B.
[...]	Indicates a set of characters. A ^ as the first character indicates a complementing set.
(...)	Matches the regular expression inside the parentheses.
	The contents can be retrieved or matched later in the string.

Table 17.2: Standard regular expression metacharacters in Python.

## Repetition

The remaining metacharacters are for matching a specified number of characters. This allows a single regular expression to match strings of varying lengths.

Character	Description
*	Matches 0 or more repetitions of the preceding regular expression.
+	Matches 1 or more repetitions of the preceding regular expression.
?	Matches 0 or 1 of the preceding regular expression.
{m,n}	Matches from m to n repetitions of the preceding regular expression.
*?, +?, ??, {m,n}?	Non-greedy versions of the previous four special characters.

Table 17.3: Repetition metacharacters for regular expressions in Python.

Each of the repetition operators acts on the expression immediately preceding it. This could be a single character, a group, or a character class. For instance, `(abc)+` matches `abc`, `abcaabc`, `abcabcabc`, and so on, but not `aba` or `cba`. On the other hand, `[abc]*` matches any sequence of `a`, `b`, and `c`, including `abcabc` and `aabbcc`.

The curly braces `{}` specify a custom number of repetitions allowed. `{,n}` matches **up to n** instances, `{m,}` matches **at least m** instances, `{k}` matches **exactly k** instances, and `{m,n}` matches from `m` to `n` instances. Thus the `?` operator is equivalent to `{,1}` and `+` is equivalent to `{1,}`.

```
# Match exactly 3 'a' characters.
>>> pattern = re.compile(r"^\{a\}\{3\}$")
```

```
>>> for test in ["aa", "aaa", "aaaa", "aba"]:
...     print(test + ':', bool(pattern.search(test)))
...
aa: False                                # Too few.
aaa: True
aaaa: False                               # Too many.
aba: False
```

Be aware that line anchors are especially important when using repetition operators. Consider the following (bad) example and compare it to the previous example.

```
# Match exactly 3 'a' characters, hopefully.
>>> pattern = re.compile(r"a{3}")
>>> for test in ["aaa", "aaaa", "aaaaaa", "aaaab"]:
...     print(test + ':', bool(pattern.search(test)))
...
aaa: True
aaaa: True                                # Should be too many!
aaaaaa: True                               # Should be too many!
aaaab: True                                # Too many, and even with the 'b'?
```

The unexpected matches occur because `"aaa"` is at the beginning of each of the test strings. With the line anchors `^` and `$`, the search truly only matches the exact string `"aaa"`.

**Problem 4.** A *valid Python identifier* (a valid variable name) is any string starting with an alphabetic character or an underscore, followed by any (possibly empty) sequence of alphanumeric characters and underscores.

A *valid python parameter definition* is defined as the concatenation of the following strings:

- any valid python identifier
- any number of spaces
- (optional) an equals sign followed by any number of spaces and ending with one of the following three things: any real number, a single quote followed by any number of non-single-quote characters followed by a single quote (ex: `'example'`), or any valid python identifier

Define a function that compiles and returns a regular expression pattern object that matches any valid Python parameter definition.

(Hint: Use the `\w` character class shortcut to keep your regular expression clean.)

To help in debugging, the following examples may be useful. These test cases are a good start, but are not exhaustive. The first table should match valid Python identifiers. The second should match a valid python parameter definition, as defined in this problem. Note that some strings which would be valid in python will not be for this problem.

Matches:	<code>"Mouse"</code>	<code>"_num = 2.3"</code>	<code>"arg_ = 'hey'"</code>	<code>"__x__"</code>	<code>"var24"</code>
Non-matches:	<code>"3rats"</code>	<code>"_num = 2.3.2"</code>	<code>"arg_ = 'one'two"</code>	<code>"sq(x)"</code>	<code>" x"</code>

Matches:	<code>"max=total" "string= " "num_guesses"</code>
Non-matches:	<code>"max=2total" "is_4=(value==4)" "pattern = r'^one two fish\$'"</code>

**Hint:** It may seem more efficient to keep the equals sign part of the expression outside of your `or` group (parentheses with pipelines) but that is a really tricky way to do it. It is easier to include the equals sign and space in each case individually. For example, `(=\s*...|=\s*...|...)` instead of `(=\s*(...|...|...))`.

**Note:** The equals sign is not a special character, but is matching the character '=' exactly. The example above matches an equal sign followed by any number of whitespace.

## UNIT TEST

There is a file called `test_regular_expressions.py` that contains prewritten unit tests for Problem 3. There is a place for you to add your own unit tests for your function in Problem 4 which will be graded.

## Manipulating Text with Regular Expressions

So far we have been solely concerned with whether or not a regular expression and a string match, but the power of regular expressions comes with what can be done with a match. In addition to the `search()` method, regular expression pattern objects have the following useful methods.

Method	Description
<code>match()</code>	Match a regular expression pattern to the beginning of a string.
<code>fullmatch()</code>	Match a regular expression pattern to all of a string.
<code>search()</code>	Search a string for the presence of a pattern.
<code>sub()</code>	Substitute occurrences of a pattern found in a string.
<code>subn()</code>	Same as <code>sub</code> , but also return the number of substitutions made.
<code>split()</code>	Split a string by the occurrences of a pattern.
<code>findall()</code>	Find all occurrences of a pattern in a string.
<code>finditer()</code>	Return an iterator yielding a match object for each match.

Table 17.4: Methods of regular expression pattern objects.

Some substitutions require remembering part of the text that the regular expression matches. Groups are useful here: each group in the regular expression can be represented in the substitution string by `\n`, where  $n$  is an integer (starting at 1) specifying which group to use.

```
# Find words that start with 'cat', remembering what comes after the 'cat'.
>>> pig_latin = re.compile(r"\bcat(\w*)")
>>> target = "Let's catch some catfish for the cat"

>>> pig_latin.sub(r"at\1clay", target) # \1 = (\w*) from the expression.
"Let's atchclay some atfishclay for the atclay"
```

The repetition operators ?, +, \*, and {m,n} are *greedy*, meaning that they match the largest string possible. On the other hand, the operators ??, +?, \*?, and {m,n}? are *non-greedy*, meaning they match the smallest strings possible. This is very often the desired behavior for a regular expression.

```
>>> target = "<abc> <def> <ghi>"  
  
# Match angle brackets and anything in between.  
>>> greedy = re.compile(r"^.+>$") # Greedy *  
>>> greedy.findall(target)  
['<abc> <def> <ghi>'] # The entire string matched!  
  
# Try again, using the non-greedy version.  
>>> nongreedy = re.compile(r"<.*?>") # Non-greedy *?  
>>> nongreedy.findall(target)  
['<abc>', '<def>', '<ghi>'] # Each <> set is an individual match.
```

Finally, there are a few customizations that make searching larger texts manageable. Each of these *flags* can be used as keyword arguments to `re.compile()`.

Flag	Description
<code>re.DOTALL</code>	. matches any character at all, including the newline.
<code>re.IGNORECASE</code>	Perform case-insensitive matching.
<code>re.MULTILINE</code>	^ matches the beginning of lines (after a newline) as well as the string; \$ matches the end of lines (before a newline) as well as the end of the string.

Table 17.5: Regular expression flags.

A benefit of using '^' and '\$' is that they allow you to search across multiple lines. For example, how would we match "World" in the string "Hello\nWorld"? Using `re.MULTILINE` in the `re.search` function will allow us to match at the beginning of each new line, instead of just the beginning of the string. The following shows how to implement multiline searching:

```
>>> pattern1 = re.compile("^\W")  
>>> pattern2 = re.compile("^\W", re.MULTILINE)  
>>> bool(pattern1.search("Hello\nWorld"))  
False  
>>> bool(pattern2.search("Hello\nWorld"))  
True
```

**Problem 5.** A Python *block* is composed of several lines of code with the same indentation level. Blocks are delimited by key words and expressions, followed by a colon. Possible key words are `if`, `elif`, `else`, `for`, `while`, `try`, `except`, `finally`, `with`, `def`, and `class`. Some of these keywords require an expression to precede the colon (`if`, `elif`, `for`, etc.). Some require no expressions to precede the colon (`else`, `finally`), and `except` may or may not have an expression before the colon.

Write a function that accepts a string of Python code and uses regular expressions to place colons in the appropriate spots. Assume that every colon is missing in the input string and that any keyword that should have an expression after it does. (Note that this will simplify your regex expression since you won't have to design it to handle cases where some colons are present or to detect the key words that need expressions versus ones that don't.) Return the string of code with colons in the correct places.

```
"""
k, i, p = 999, 1, 0
while k > i
    i *= 2
    p += 1
    if k != 999
        print("k should not have changed")
    else
        pass
print(p)
"""

# The string given above should become this string.
"""
k, i, p = 999, 1, 0
while k > i:
    i *= 2
    p += 1
    if k != 999:
        print("k should not have changed")
    else:
        pass
print(p)
"""
```

## Extracting Text with Regular Expressions

Regular expressions are useful for locating and extracting information that matches a certain format. The method `pattern.findall(string)` returns a list containing all non-overlapping matches of `pattern` found in `string`. The method scans the string from left to right and returns the matches in that order. If two matches overlap, the match that begins first is returned.

When at least one group, indicated by `()`, is present in the pattern, then only information contained in a group is returned. Each match is returned as a tuple containing the part of the string that matches each group in the pattern.

```
>>> pattern = re.compile("\w* fish")

# Without any groups, the entirety of each match is returned.
>>> pattern.findall("red fish, blue fish, one fish, two fish")
['red fish', 'blue fish', 'one fish', 'two fish']
```

```
# When a group is present, only information contained in a group is returned.
>>> pattern2 = re.compile("(\\w*) (fish|dish)")
>>> pattern2.findall("red dish, blue dish, one fish, two fish")
[('red', 'dish'), ('blue', 'dish'), ('one', 'fish'), ('two', 'fish')]
```

If you wish to extract the characters that match some groups, but not others, you can choose to exclude a group from being returned using the syntax `(?:)`:

```
>>> pattern = re.compile("(\\w*)(?:fish|dish)")
>>> pattern.findall("red dish, blue dish, one fish, two fish")
['red', 'blue', 'one', 'two']
```

**Problem 6.** The file `fake_contacts.txt` contains poorly formatted contact data for 2000 fictitious individuals. Each line of the file contains data for one person, including their name and possibly their birthday, email address, and/or phone number. The formatting of the data is not consistent, and much of it is missing. Each contact name includes a first and last name. Some names have middle initials, in the form `Jane C. Doe`. Each birthday lists the month, then the day, and then the year, though the format varies from `1/1/11`, `1/01/2011`, etc. If century is not specified for birth year, as in `1/01/XX`, birth year is assumed to be `20XX`. Remember, not all information is listed for each contact.

Use regular expressions to extract the necessary data and format it uniformly, writing birthdays as `mm/dd/yyyy` and phone numbers as `(xxx)xxx-xxxx`. Return a dictionary where the key is the name of an individual and the value is another dictionary containing their information. Each of these inner dictionaries should have the keys `"birthday"`, `"email"`, and `"phone"`. In the case of missing data, map the key to `None`.

The first two entries of the completed dictionary are given below.

```
{
    "John Doe": {
        "birthday": "01/01/2099",
        "email": "john_doe90@hopefullynotarealaddress.com",
        "phone": "(123)456-7890"
    },
    "Jane Smith": {
        "birthday": None,
        "email": None,
        "phone": "(222)111-3333"
    },
    # ...
}
```

**Hint:** Think about creating a separate re.compile() object to ‘catch’ each piece of identifying information. Extract and clean each piecece individually and build your dictionary incrementally. If the piece of information exists for a specific person, reformat and assign it to that pesron’s dictionary. If the information doesn’t exist assign it to be [None](#).

## Additional Material

### Regular Expressions in the Unix Shell

As we have seen,, regular expressions are very useful when we want to match patterns. Regular expressions can be used when matching patterns in the Unix Shell. Though there are many Unix commands that take advantage of regular expressions, we will focus on `grep` and `awk`.

#### Regular Expressions and grep

Recall from Lab 1 that `grep` is used to match patterns in files or output. It turns out we can use regular expressions to define the pattern we wish to match.

In general, we use the following syntax:

```
$ grep 'regexp' filename
```

We can also use regular expressions when piping output to `grep`.

```
# List details of directories within current directory.  
$ ls -l | grep ^d
```

#### Regular Expressions and awk

By incorporating regular expressions, the `awk` command becomes much more robust. Before GUI spreadsheet programs like Microsoft Excel, `awk` was commonly used to visualize and query data from a file.

Including `if` statements inside `awk` commands gives us the ability to perform actions on lines that match a given pattern. The following example prints the filenames of all files that are owned by `freddy`.

```
$ ls -l | awk ' {if ($3 ~ /freddy/) print $9} '
```

Because there is a lot going on in this command, we will break it down piece-by-piece. The output of `ls -l` is getting piped to `awk`. Then we have an `if` statement. The syntax here means if the condition inside the parenthesis holds, print field 9 (the field with the filename). The condition is where we use regular expressions. The `~` checks to see if the contents of field 3 (the field with the username) matches the regular expression found inside the forward slashes. To clarify, `freddy` is the regular expression in this example and the expression must be surrounded by forward slashes. Consider a similar example. In this example, we will list the names of the directories inside the current directory. (This replicates the behavior of the Unix command `ls -d */`)

```
$ ls -l | awk ' {if ($1 ~ /^d/) print $9} '
```

Notice in this example, we printed the names of the directories, whereas in one of the example using `grep`, we printed all the details of the directories as well.

**ACHTUNG!**

Some of the definitions for character classes we used earlier in this lab will not work in the Unix Shell. For example, \w and \d are not defined. Instead of \w, use [:alnum:]]. Instead of \d, use [:digit:]]. For a complete list of similar character classes, search the internet for *POSIX Character Classes* or *Bracket Character Classes*.



# 18

## Gradient Descent Methods

**Lab Objective:** *Iterative optimization methods choose a search direction and a step size at each iteration. One simple choice for the search direction is the negative gradient, resulting in the method of steepest descent. While theoretically foundational, in practice this method is often slow to converge. An alternative method, the conjugate gradient algorithm, uses a similar idea that results in much faster convergence in some situations. In this lab we implement a method of steepest descent and two conjugate gradient methods, then apply them to regression problems.*

### The Method of Steepest Descent

Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  with first derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ . The following iterative technique is a common template for methods that aim to compute a local minimizer  $\mathbf{x}^*$  of  $f$ .

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (18.1)$$

Here  $\mathbf{x}_k$  is the  $k$ th approximation to  $\mathbf{x}^*$ ,  $\alpha_k$  is the *step size*, and  $\mathbf{p}_k$  is the *search direction*. Newton's method and its relatives follow this pattern, but they require the calculation (or approximation) of the inverse Hessian matrix  $Df^2(\mathbf{x}_k)^{-1}$  at each step. The following idea is a simpler and less computationally intensive approach than Newton and quasi-Newton methods.

The derivative  $Df(\mathbf{x})^\top$  (often called the *gradient* of  $f$  at  $\mathbf{x}$ , sometimes notated  $\nabla f(\mathbf{x})$ ) is a vector that points in the direction of greatest **increase** of  $f$  at  $\mathbf{x}$ . It follows that the negative derivative  $-Df(\mathbf{x})^\top$  points in the direction of steepest **decrease** at  $\mathbf{x}$ . The *method of steepest descent* chooses the search direction  $\mathbf{p}_k = -Df(\mathbf{x}_k)^\top$  at each step of (18.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top \quad (18.2)$$

Setting  $\alpha_k = 1$  for each  $k$  is often sufficient for Newton and quasi-Newton methods. However, a constant choice for the step size in (18.2) can result in oscillating approximations or even cause the sequence  $(\mathbf{x}_k)_{k=1}^\infty$  to travel away from the minimizer  $\mathbf{x}^*$ . To avoid this problem, the step size  $\alpha_k$  can be chosen in a few ways.

- Start with  $\alpha_k = 1$ , then set  $\alpha_k = \frac{\alpha_k}{2}$  until  $f(\mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top) < f(\mathbf{x}_k)$ , terminating the iteration if  $\alpha_k$  gets too small. This guarantees that the method actually descends at each step and that  $\alpha_k$  satisfies the Armijo rule, without endangering convergence.

- At each step, solve the following one-dimensional optimization problem.

$$\alpha_k = \operatorname{argmin}_{\alpha} f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^T)$$

Using this choice is called *exact steepest descent*. This option is more expensive per iteration than the above strategy, but it results in fewer iterations before convergence.

**Problem 1.** Write a function that accepts an objective function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , its derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , a convergence tolerance `tol` defaulting to  $1e^{-5}$ , and a maximum number of iterations `maxiter` defaulting to 100. Implement the exact method of steepest descent, using a one-dimensional optimization method to choose the step size (use `opt.minimize_scalar()` or your own 1-D minimizer). Iterate until  $\|Df(\mathbf{x}_k)\|_\infty < \text{tol}$  or  $k > \text{maxiter}$ . Return the approximate minimizer  $\mathbf{x}^*$ , whether or not the algorithm converged (`True` or `False`), and the number of iterations computed.

Test your function on  $f(x, y, z) = x^4 + y^4 + z^4$  (easy) and the Rosenbrock function (hard). It should take many iterations to minimize the Rosenbrock function, but it should converge eventually with a large enough choice of `maxiter`.

## The Conjugate Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems. Depending on the nature of the objective function, the sequence of points can zig-zag back and forth or get stuck on flat areas without making significant progress toward the true minimizer.

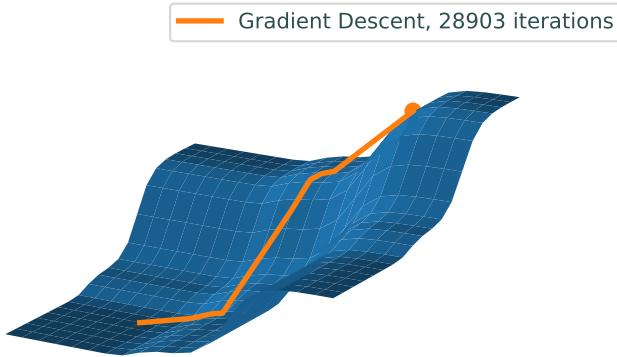


Figure 18.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

Unlike the method of steepest descent, the *conjugate gradient algorithm* chooses a search direction that is guaranteed to be a descent direction, though not the direction of greatest descent. These directions are using a generalized form of orthogonality called *conjugacy*.

Let  $Q$  be a square, positive definite matrix. A set of vectors  $\{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_m\}$  is called  *$Q$ -conjugate* if each distinct pair of vectors  $\mathbf{x}_i, \mathbf{x}_j$  satisfy  $\mathbf{x}_i^\top Q \mathbf{x}_j = 0$ . A  $Q$ -conjugate set of vectors is linearly independent and can form a basis that diagonalizes the matrix  $Q$ . This guarantees that an iterative method to solve  $Q\mathbf{x} = \mathbf{b}$  only require as many steps as there are basis vectors.

Solve a positive definite system  $Q\mathbf{x} = \mathbf{b}$  is valuable in and of itself for certain problems, but it is also equivalent to minimizing certain functions. Specifically, consider the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top Q\mathbf{x} - \mathbf{b}^\top \mathbf{x} + c.$$

Because  $Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b}$ , minimizing  $f$  is the same as solving the equation

$$\mathbf{0} = Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b} \Rightarrow Q\mathbf{x} = \mathbf{b},$$

which is the original linear system. Note that the constant  $c$  does not affect the minimizer, since if  $\mathbf{x}^*$  minimizes  $f(\mathbf{x})$  it also minimizes  $f(\mathbf{x}) + c$ .

Using the conjugate directions guarantees an iterative method to converge on the minimizer because each iteration minimizes the objective function over a subspace of dimension equal to the iteration number. Thus, after  $n$  steps, where  $n$  is the number of conjugate basis vectors, the algorithm has found a minimizer over the entire space. In certain situations, this has a great advantage over gradient descent, which can bounce back and forth. This comparison is illustrated in Figure 18.2. Additionally, because the method utilizes a basis of conjugate vectors, the previous search direction can be used to find a conjugate projection onto the next subspace, saving computational time.

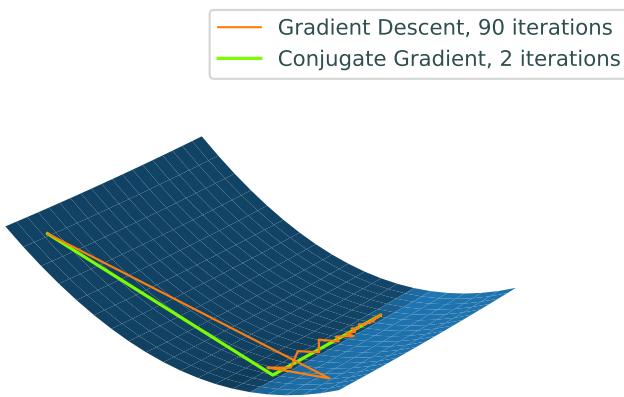


Figure 18.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

**Algorithm 1**


---

```

1: procedure CONJUGATE GRADIENT( $\mathbf{x}_0, Q, \mathbf{b}, \text{tol}$ )
2:    $\mathbf{r}_0 \leftarrow Q\mathbf{x}_0 - \mathbf{b}$ 
3:    $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$ 
4:    $k \leftarrow 0$ 
5:   while  $\|\mathbf{r}_k\| \geq \text{tol}, k < n$  do
6:      $\alpha_k \leftarrow \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{d}_k^\top Q \mathbf{d}_k$ 
7:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ 
8:      $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k Q \mathbf{d}_k$ 
9:      $\beta_{k+1} \leftarrow \mathbf{r}_{k+1}^\top \mathbf{r}_{k+1} / \mathbf{r}_k^\top \mathbf{r}_k$ 
10:     $\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$ 
11:     $k \leftarrow k + 1$ .
return  $\mathbf{x}_{k+1}$ 

```

---

The points  $\mathbf{x}_k$  are the successive approximations to the minimizer, the vectors  $\mathbf{d}_k$  are the conjugate descent directions, and the vectors  $\mathbf{r}_k$  (which actually correspond to the steepest descent directions) are used in determining the conjugate directions. The constants  $\alpha_k$  and  $\beta_k$  are used, respectively, in the line search, and in ensuring the  $Q$ -conjugacy of the descent directions.

**Problem 2.** Write a function that accepts an  $n \times n$  positive definite matrix  $Q$ , a vector  $\mathbf{b} \in \mathbb{R}^n$ , an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , and a stopping tolerance. Use Algorithm 1 to solve the system  $Q\mathbf{x} = \mathbf{b}$ . Continue the algorithm until  $\|\mathbf{r}_k\|$  is less than the tolerance, iterating no more than  $n$  times. Return the solution  $\mathbf{x}$ , whether or not the algorithm converged in  $n$  iterations or less, and the number of iterations computed. Test your function on the simple system

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix},$$

which has solution  $\mathbf{x}^* = [\frac{1}{2}, 2]^\top$ . This is equivalent to minimizing the quadratic function  $f(x, y) = x^2 + 2y^2 - x - 8y$ ; check that your function from Problem 1 gets the same solution.

More generally, you can generate a random positive definite matrix  $Q$  for testing by setting  $Q = A^\top A$  for any  $A$  of full rank. Note, for values of  $n \leq 5$  this method is not stable enough to always converge in exactly  $n$  iterations. Try using the code given below to test your function for values of  $n < 5$ .

There is a file called `test_gradient_methods.py` that contains some prewritten unit tests that you can use to test your function.

```

>>> import numpy as np
>>> from scipy import linalg as la

# Generate Q, b, and the initial guess x0.
>>> n = 4
>>> A = np.random.random((n,n))
>>> Q = A.T @ A
>>> b, x0 = np.random.random((2,n))

>>> x = la.solve(Q, b)      # Use your function here.

```

```
>>> np.allclose(Q @ x, b)
True
```

## Non-linear Conjugate Gradient

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions.

Though the non-linear version does not have guaranteed convergence as the linear formulation does, it can still converge in less iterations than the method of steepest descent. Modifying the algorithm for more general functions requires new formulas for  $\alpha_k$ ,  $\mathbf{r}_k$ , and  $\beta_k$ .

- The scalar  $\alpha_k$  is simply the result of performing a line-search in the given direction  $\mathbf{d}_k$  and is thus defined  $\alpha_k = \operatorname{argmin}_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ .
- The vector  $\mathbf{r}_k$  in the original algorithm was really just the gradient of the objective function, so now define  $\mathbf{r}_k = Df(\mathbf{x}_k)^\top$ .
- The constants  $\beta_k$  can be defined in various ways, and the most correct choice depends on the nature of the objective function. A well-known formula, attributed to Fletcher and Reeves, is  $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^\top / Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^\top$ .

---

### Algorithm 2

---

```

1: procedure NON-LINEAR CONJUGATE GRADIENT( $f$ ,  $Df$ ,  $\mathbf{x}_0$ ,  $\text{tol}$ ,  $\text{maxiter}$ )
2:    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^\top$ 
3:    $\mathbf{d}_0 \leftarrow \mathbf{r}_0$ 
4:    $\alpha_0 \leftarrow \operatorname{argmin}_{\alpha} f(\mathbf{x}_0 + \alpha \mathbf{d}_0)$ 
5:    $\mathbf{x}_1 \leftarrow \mathbf{x}_0 + \alpha_0 \mathbf{d}_0$ 
6:    $k \leftarrow 1$ 
7:   while  $\|\mathbf{r}_k\| \geq \text{tol}$ ,  $k < \text{maxiter}$  do
8:      $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^\top$ 
9:      $\beta_k = \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{r}_{k-1}^\top \mathbf{r}_{k-1}$ 
10:     $\mathbf{d}_k \leftarrow \mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$ .
11:     $\alpha_k \leftarrow \operatorname{argmin}_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{d}_k).$ 
12:     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k.$ 
13:     $k \leftarrow k + 1.$ 
```

---

**Problem 3.** Write a function that accepts a convex objective function  $f$ , its derivative  $Df$ , an initial guess  $\mathbf{x}_0$ , a convergence tolerance defaulting to  $1e^{-5}$ , and a maximum number of iterations defaulting to 100. Use Algorithm 2 to compute the minimizer  $\mathbf{x}^*$  of  $f$ . Return the approximate minimizer, whether or not the algorithm converged, and the number of iterations computed.

Compare your function to SciPy's `opt.fmin_cg()`.

```
>>> opt.fmin_cg(opt.rosen, np.array([10, 10]), fprime=opt.rosen_der)
```

```
Optimization terminated successfully.
    Current function value: 0.000000
    Iterations: 44
    Function evaluations: 102 # Much faster than steepest descent!
    Gradient evaluations: 102
array([ 1.00000007,  1.00000015])
```

## UNIT TEST

There is a file called `test_gradient_methods.py` that contains some prewritten unit tests for Problem 2. There is a place for you to add your own unit tests to test your function from Problem 3 which will be graded.

## Regression Problems

A major use of the conjugate gradient method is solving linear least squares problems. Recall that a least squares problem can be formulated as an optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2,$$

where  $A$  is an  $m \times n$  matrix with full column rank,  $\mathbf{x} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^m$ . The solution can be calculated analytically, and is given by

$$\mathbf{x}^* = (A^T A)^{-1} A^T \mathbf{b}.$$

In other words, the minimizer solves the linear system

$$A^T A \mathbf{x} = A^T \mathbf{b}. \quad (18.3)$$

Since  $A$  has full column rank, it is invertible,  $A^T A$  is positive definite, and for any non-zero vector  $\mathbf{z}$ ,  $\mathbf{z}^T A^T A \mathbf{z} = \|A\mathbf{z}\|^2 > 0$ . As  $A^T A$  is positive definite, conjugate gradient can be used to solve Equation 18.3.

Linear least squares is the mathematical underpinning of *linear regression*. Linear regression involves a set of real-valued data points  $\{y_1, \dots, y_m\}$ , where each  $y_i$  is paired with a corresponding set of predictor variables  $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$  with  $n < m$ . The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \dots + \beta_n x_{i,n} + \varepsilon_i$$

for  $i = 1, 2, \dots, m$ . The real numbers  $\beta_0, \dots, \beta_n$  are known as the parameters of the model, and the  $\varepsilon_i$  are independent, normally-distributed error terms. The goal of linear regression is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares. Define

$$\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}.$$

The solution  $\mathbf{x}^* = [\beta_0^*, \beta_1^*, \dots, \beta_n^*]^\top$  to the system  $A^\top A \mathbf{x} = A^\top \mathbf{b}$  gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data.

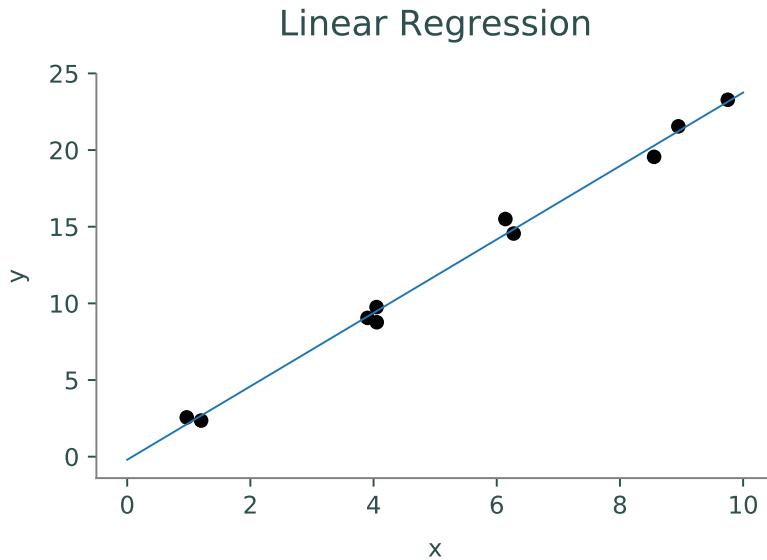


Figure 18.3: Solving the linear regression problem results in a best-fit hyperplane.

**Problem 4.** Using your function from Problem 2, solve the linear regression problem specified by the data contained in the file<sup>a</sup> `linregression.txt`. This is a whitespace-delimited text file formatted so that the  $i$ -th row consists of  $y_i, x_{i,1}, \dots, x_{i,n}$ . Use `np.loadtxt()` to load in the data and return the solution to the normal equations.

<sup>a</sup>Source: Statistical Reference Datasets website at <http://www.itl.nist.gov/div898/strd/lls/data/LINKS/v-Longley.shtml>.

## Logistic Regression

*Logistic regression* is another important technique in statistical analysis and machine learning that builds off of the concepts of linear regression. As in linear regression, there is a set of predictor variables  $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}_{i=1}^m$  with corresponding outcome variables  $\{y_i\}_{i=1}^m$ . In logistic regression, the outcome variables  $y_i$  are binary and can be modeled by a *sigmoidal* relationship. The value of the predicted  $y_i$  can be thought of as the probability that  $y_i = 1$ . In mathematical terms,

$$\mathbb{P}(y_i = 1 | x_{i,1}, \dots, x_{i,n}) = p_i,$$

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers  $\beta_0, \beta_1, \dots, \beta_n$ . Note that  $p_i \in (0, 1)$  regardless of the values of the predictor variables and parameters.

The probability of observing the outcome variables  $y_i$  under this model, assuming they are independent, is given by the *likelihood function*  $\mathcal{L} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$

$$\mathcal{L}(\beta_0, \dots, \beta_n) = \prod_{i=1}^m p_i^{y_i} (1 - p_i)^{1-y_i}.$$

The goal of logistic regression is to find the parameters  $\beta_0, \dots, \beta_k$  that maximize this likelihood function. Thus, the problem can be written as:

$$\max_{(\beta_0, \dots, \beta_n)} \mathcal{L}(\beta_0, \dots, \beta_n).$$

Maximizing this function is often a numerically unstable calculation. Thus, to make the objective function more suitable, the logarithm of the objective function may be maximized because the logarithmic function is strictly monotone increasing. Taking the log and turning the problem into a minimization problem, the final problem is formulated as:

$$\min_{(\beta_0, \dots, \beta_n)} -\log \mathcal{L}(\beta_0, \dots, \beta_n).$$

A few lines of calculation reveal that this objective function can also be rewritten as

$$\begin{aligned} -\log \mathcal{L}(\beta_0, \dots, \beta_n) &= \sum_{i=1}^m \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))) + \\ &\quad \sum_{i=1}^m (1 - y_i)(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}). \end{aligned}$$

The values for the parameters  $\{\beta_i\}_{i=1}^n$  that we obtain are known as the *maximum likelihood estimate* (MLE). To find the MLE, conjugate gradient can be used to minimize the objective function.

For a one-dimensional binary logistic regression problem, we have predictor data  $\{x_i\}_{i=1}^m$  with labels  $\{y_i\}_{i=1}^m$  where each  $y_i \in \{0, 1\}$ . The negative log likelihood then becomes the following.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^m \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \quad (18.4)$$

**Problem 5.** Write a class for doing binary logistic regression in one dimension that implement the following methods.

1. `fit()`: accept an array  $\mathbf{x} \in \mathbb{R}^n$  of data, an array  $\mathbf{y} \in \mathbb{R}^n$  of labels (0s and 1s), and an initial guess  $\boldsymbol{\beta}_0 \in \mathbb{R}^2$ . Define the negative log likelihood function as given in (18.4), then minimize it (with respect to  $\boldsymbol{\beta}$ ) with your function from Problem 3 or `opt.fmin_cg()`. Store the resulting parameters  $\beta_0$  and  $\beta_1$  as attributes.
2. `predict()`: accept a float  $x \in \mathbb{R}$  and calculate

$$\sigma(x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x))},$$

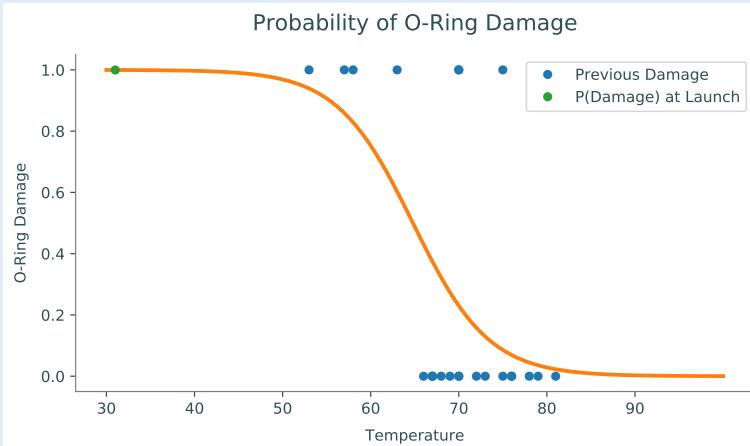
where  $\beta_0$  and  $\beta_1$  are the optimal values calculated in `fit()`. The value  $\sigma(x)$  is the probability that the observation  $x$  should be assigned the label  $y = 1$ .

This class does not need an explicit constructor. You may assume that `predict()` will be called after `fit()`.

**Problem 6.** On January 28, 1986, less than two minutes into the Challenger space shuttle's 10th mission, there was a large explosion that originated from the spacecraft, killing all seven crew members and destroying the shuttle. The investigation that followed concluded that the malfunction was caused by damage to O-rings that are used as seals for parts of the rocket engines. There were 24 space shuttle missions before this disaster, some of which had noted some O-ring damage. Given the data, could this disaster have been predicted?

The file `challenger.npy` contains data for 23 missions (during one of the 24 missions, the engine was lost at sea). The first column ( $\mathbf{x}$ ) contains the ambient temperature, in Fahrenheit, of the shuttle launch. The second column ( $\mathbf{y}$ ) contains a binary indicator of the presence of O-ring damage (1 if O-ring damage was present, 0 otherwise).

Instantiate your class from Problem 5 and fit it to the data, using an initial guess of  $\beta_0 = [20, -1]^\top$ . Plot the resulting curve  $\sigma(x)$  for  $x \in [30, 100]$ , along with the raw data. Return the predicted probability (according to this model) of O-ring damage on the day the shuttle was launched, given that it was 31°F.





# 19

## The Simplex Method

**Lab Objective:** *The Simplex Method is a straightforward algorithm for finding optimal solutions to optimization problems with linear constraints and cost functions. Because of its simplicity and applicability, this algorithm has been named one of the most important algorithms invented within the last 100 years. In this lab we implement a standard Simplex solver for the primal problem.*

### Standard Form

The Simplex Algorithm accepts a linear constrained optimization problem, also called a *linear program*, in the form given below:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

Note that any linear program can be converted to standard form, so there is no loss of generality in restricting our attention to this particular formulation.

Such an optimization problem defines a region in space called the *feasible region*, the set of points satisfying the constraints. Because the constraints are all linear, the feasible region forms a geometric object called a *polytope*, having flat faces and edges (see Figure 19.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.

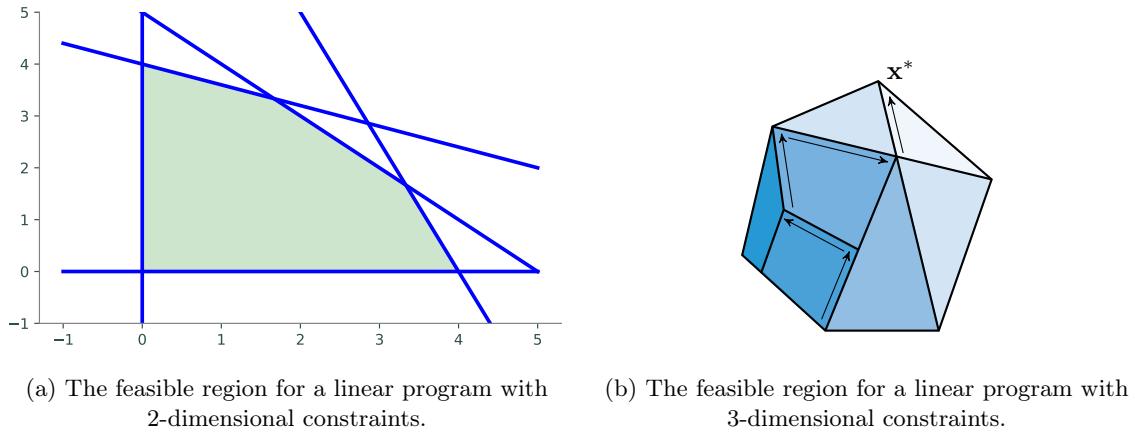


Figure 19.1: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

Implementing the Simplex Algorithm is straightforward, provided one carefully follows the procedure. We will break the algorithm into several small steps, and write a function to perform each one. To become familiar with the execution of the Simplex algorithm, it is helpful to work several examples by hand.

## The Simplex Solver

Our program will be more lengthy than many other lab exercises and will consist of a collection of functions working together to produce a final result. It is important to clearly define the task of each function and how all the functions will work together. If this program is written haphazardly, it will be much longer and more difficult to read than it needs to be. We will walk you through the steps of implementing the Simplex Algorithm as a Python class.

For demonstration purposes, we will use the following linear program.

$$\begin{aligned}
 &\text{minimize} && -3x_0 - 2x_1 \\
 &\text{subject to} && x_0 - x_1 \leq 2 \\
 & && 3x_0 + x_1 \leq 5 \\
 & && 4x_0 + 3x_1 \leq 7 \\
 & && x_0, x_1 \geq 0.
 \end{aligned}$$

## Accepting a Linear Program

Our first task is to determine if we can even use the Simplex algorithm. Assuming that the problem is presented to us in standard form, we need to check that the feasible region includes the origin. For now, we only check for feasibility at the origin. A more robust solver sets up the auxiliary problem and solves it to find a starting point if the origin is infeasible.

**Problem 1.** Write a class that accepts the arrays  $\mathbf{c}$ ,  $A$ , and  $\mathbf{b}$  of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin. That is, check that  $A\mathbf{x} \leq \mathbf{b}$  when  $\mathbf{x} = \mathbf{0}$ . Raise a `ValueError` if the problem is not feasible at the origin.

## Adding Slack Variables

The next step is to convert the inequality constraints  $A\mathbf{x} \leq \mathbf{b}$  into equality constraints by introducing a slack variable for each constraint equation. If the constraint matrix  $A$  is an  $m \times n$  matrix, then there are  $m$  slack variables, one for each row of  $A$ . Grouping all of the slack variables into a vector  $\mathbf{w}$  of length  $m$ , the constraints now take the form  $A\mathbf{x} + \mathbf{w} = \mathbf{b}$ . In our example, we have

$$\mathbf{w} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

When adding slack variables, it is useful to represent all of your variables, both the original primal variables and the additional slack variables, in a convenient manner. One effective way is to refer to a variable by its subscript. For example, we can use the integers 0 through  $n - 1$  to refer to the original (non-slack) variables  $x_0$  through  $x_{n-1}$ , and we can use the integers  $n$  through  $n + m - 1$  to track the slack variables (where the slack variable corresponding to the  $i$ th row of the constraint matrix is represented by the index  $n + i - 1$ ).

We also need some way to track which variables are *independent* (non-zero) and which variables are *dependent* (those that have value 0). This can be done using the objective function. At anytime during the optimization process, the non-zero variables in the objective function are *independent* and all other variables are *dependent*.

## Creating a Dictionary

After we have determined that our program is feasible, we need to create the *dictionary* (sometimes called the *tableau*), a matrix to track the state of the algorithm.

There are many different ways to build your dictionary. We will do this by setting the corresponding dependent variable equations to 0. For example, if  $x_5$  were a dependent variable we would expect to see a -1 in the column that represents  $x_5$ . Define

$$\bar{A} = [ A \quad I_m ],$$

where  $I_m$  is the  $m \times m$  identity matrix we will use to represent our slack variables, and define

$$\bar{\mathbf{c}} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}.$$

That is,  $\bar{\mathbf{c}} \in \mathbb{R}^{n+m}$  such that the first  $n$  entries are  $\mathbf{c}$  and the final  $m$  entries are zeros. Then the initial dictionary has the form

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^\top \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{19.1}$$

The columns of the dictionary correspond to each of the variables (both primal and slack), and the rows of the dictionary correspond to the dependent variables.

For our example the initial dictionary is

$$D = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}.$$

The advantage of using this kind of dictionary is that it is easy to check the progress of your algorithm by hand.

**Problem 2.** Add a method to your Simplex solver that takes in arrays  $c$ ,  $A$ , and  $b$  to create the initial dictionary as a NumPy array. Make sure to initialize the dictionary in `__init__` by calling the method you just created and name the parameter `self.dictionary` (failure to do this will result in no points received for the problem).

## Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping dependent and independent variables, and transforming the dictionary appropriately. This has the effect of moving from one vertex of the feasible polytope to another vertex in a way that increases the value of the objective function. Depending on how you store your variables, you may need to modify a few different parts of your solver to reflect this swapping.

When initiating a pivot, you need to determine which variables will be swapped. In the dictionary representation, you first find a specific element on which to pivot, and the row and column that contain the pivot element correspond to the variables that need to be swapped. Row operations are then performed on the dictionary so that the pivot column becomes a negative elementary vector.

Let's break it down, starting with the pivot selection. We need to use some care when choosing the pivot element. To find the pivot column, search from left to right along the top row of the dictionary (ignoring the first column), and stop once you encounter the first negative value. The index corresponding to this column will be designated the *entering index*, since after the full pivot operation, it will enter the basis and become a dependent variable.

Using our initial dictionary  $D$  in the example, we stop at the second column:

$$D = \left[ \begin{array}{c|ccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right]$$

We now know that our pivot element will be found in the second column. The entering index is thus 1.

Next, we select the pivot element from among the negative entries in the pivot column (ignoring the entry in the first row). *If all entries in the pivot column are non-negative, the problem is unbounded and has no solution.* In this case, the algorithm should terminate. Otherwise, assuming our pivot column is the  $j$ th column of the dictionary and that the negative entries of this column are  $D_{i_1,j}, D_{i_2,j}, \dots, D_{i_k,j}$ , we calculate the ratios

$$\frac{-D_{i_1,0}}{D_{i_1,j}}, \frac{-D_{i_2,0}}{D_{i_2,j}}, \dots, \frac{-D_{i_k,0}}{D_{i_k,j}},$$

and we choose our pivot element to be one that minimizes this ratio. If multiple entries minimize the ratio, then we utilize *Bland's Rule*, which instructs us to choose the entry in the row corresponding to the smallest index (obeying this rule is important, as it prevents the possibility of the algorithm cycling back on itself infinitely). The index corresponding to the pivot row is designated as the *leaving index*, since after the full pivot operation, it will leave the basis and become an independent variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are negative, and hence they are all potential choices for the pivot element. We then calculate the ratios, and obtain

$$\frac{-2}{-1} = 2, \quad \frac{-5}{-3} = 1.66..., \quad \frac{-7}{-4} = 1.75.$$

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column (index 1), third row (index 2) is our designated pivot element.

$$D = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & \boxed{-3} & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}$$

**Problem 3.** Write a method that will determine the pivot row and pivot column according to Bland's Rule.

**Definition 19.1 (Bland's Rule).** Choose the independent variable with the smallest index that has a negative coefficient in the objective function as the leaving variable. Choose the dependent variable with the smallest index among all the binding dependent variables.

Bland's Rule is important in avoiding cycles when performing pivots. This rule guarantees that a feasible Simplex problem will terminate in a finite number of pivots. *Hint:* Avoid dividing by zero.

Finally, we perform row operations on our dictionary in the following way: divide the pivot row by the negative value of the pivot entry. Then use the pivot row to zero out all entries in the pivot column above and below the pivot entry. In our example, our pivot is -3. So, we must first divide the pivot row by 3, and then zero out the two entries above the pivot element and the single entry below it:

$$\begin{array}{ccccccc} \left[ \begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] & \xrightarrow{\quad} & \left[ \begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] & \xrightarrow{\quad} \\ \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] & \xrightarrow{\quad} & \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & 4/3 & -1 & 1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] & \xrightarrow{\quad} \\ \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & 4/3 & -1 & 1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 1/3 & 0 & -5/3 & 0 & 4/3 & -1 \end{array} \right]. \end{array}$$

The result of these row operations is our updated dictionary, and the pivot operation is complete.

**Problem 4.** Add a method to your solver that checks for unboundedness and performs a single pivot operation from start to completion. If the problem is unbounded, raise a `ValueError`.

## Termination and Reading the Dictionary

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial dictionary. After carrying out these initial steps, it then performs the pivoting operation iteratively until the optimal point is found. But how do we determine when the optimal point is found? The answer is to look at the top row of the dictionary, which represents the objective function. More specifically, before each pivoting operation, check whether all of the entries in the top row of the dictionary (ignoring the entry in the first column) are nonnegative. If this is the case, then we have found an optimal solution, and so we terminate the algorithm.

The final step is to report the solution. The ending state of the dictionary and index list tells us everything we need to know. The minimal value attained by the objective function is found in the upper leftmost entry of the dictionary. Variables with nonzero entries in the objective function, or first row of our dictionary array, are independent variables. Variables with an entry of 0 in the objective function are dependent variables, and their values are given by the first column of the dictionary. Specifically, independent variables are given the value of 0 while the dependent variable whose index is located at the  $i$ th entry of the index list has the value  $T_{i+1,0}$ .

In our example, suppose that our algorithm terminates with the dictionary and index list in the following state:

$$D = \begin{bmatrix} -5.2 & 0 & 0 & 0 & 0.2 & 0.6 \\ 0.6 & 0 & 0 & -1 & 1.4 & -0.8 \\ 1.6 & -1 & 0 & 0 & -0.6 & 0.2 \\ 0.2 & 0 & -1 & 0 & 0.8 & -0.6 \end{bmatrix}$$

Then the minimal value of the objective function is  $-5.2$ . The independent variables have indices 4, 5 and have the value 0. The dependent variables have indices 3, 1, and 2, and have values .6, 1.6, and .2, respectively. In the notation of the original problem statement, the solution is given by

$$\begin{aligned}x_0 &= 1.6 \\x_1 &= .2.\end{aligned}$$

**Problem 5.** Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the minimal value, the dependent variables, and the independent variables. The dependent and independent variables should be represented as two dictionaries that map the index of the variable to its corresponding value.

For our example, we would return the tuple

$(-5.2, \{0: 1.6, 1: .2, 2: .6\}, \{3: 0, 4: 0\})$ .

## UNIT TEST

There is a file called `test_simplex.py` that contains the following block of code as a unit test. There is a place for you to write your own unit tests for Problem 5, the simplex solver, which will be graded.

At this point, you should have a Simplex solver that is ready to use. The following code demonstrates how your solver is expected to behave:

```
>>> import SimplexSolver

# Initialize objective function and constraints.
>>> c = np.array([-3., -2.])
>>> b = np.array([2., 5, 7])
>>> A = np.array([[1., -1], [3, 1], [4, 3]])

# Instantiate the simplex solver, then solve the problem.
>>> solver = SimplexSolver(c, A, b)
>>> sol = solver.solve()
>>> print(sol)
(-5.2,
 {0: 1.6, 1: 0.2, 2: 0.6},
 {3: 0, 4: 0})
```

If the linear program were infeasible at the origin or unbounded, we would expect the solver to alert the user by raising an error.

Note that this simplex solver is *not* fully operational. It can't handle the case of infeasibility at the origin. This can be fixed by adding methods to your class that solve the *auxiliary problem*, that of finding an initial feasible dictionary when the problem is not feasible at the origin. Solving the auxiliary problem involves pivoting operations identical to those you have already implemented, so adding this functionality is not overly difficult.

## The Product Mix Problem

We now use our Simplex implementation to solve the *product mix problem*, which in its dependent form can be expressed as a simple linear program. Suppose that a manufacturer makes  $n$  products using  $m$  different resources (labor, raw materials, machine time available, etc). The  $i$ th product is sold at a unit price  $p_i$ , and there are at most  $m_j$  units of the  $j$ th resource available. Additionally, each unit of the  $i$ th product requires  $a_{j,i}$  units of resource  $j$ . Given that the demand for product  $i$  is  $d_i$  units per a certain time period, how do we choose the optimal amount of each product to manufacture in that time period so as to maximize revenue, while not exceeding the available resources?

Let  $x_1, x_2, \dots, x_n$  denote the amount of each product to be manufactured. The sale of product  $i$  brings revenue in the amount of  $p_i x_i$ . Therefore our objective function, the profit, is given by

$$\sum_{i=1}^n p_i x_i.$$

Additionally, the manufacture of product  $i$  requires  $a_{j,i} x_i$  units of resource  $j$ . Thus we have the resource constraints

$$\sum_{i=1}^n a_{j,i} x_i \leq m_j \text{ for } j = 1, 2, \dots, m.$$

Finally, we have the demand constraints which tell us not to exceed the demand for the products:

$$x_i \leq d_i \text{ for } i = 1, 2, \dots, n$$

The variables  $x_i$  are constrained to be nonnegative, of course. We therefore have a linear program in the appropriate form that is feasible at the origin. It is a simple task to solve the problem using our Simplex solver.

**Problem 6.** Solve the product mix problem for the data contained in the file `productMix.npz`. In this problem, there are 4 products and 3 resources. The archive file, which you can load using the function `np.load`, contains a dictionary of arrays. The array with key '`A`' gives the resource coefficients  $a_{i,j}$  (i.e. the  $(i, j)$ -th entry of the array give  $a_{i,j}$ ). The array with key '`p`' gives the unit prices  $p_i$ . The array with key '`m`' gives the available resource units  $m_j$ . The array with key '`d`' gives the demand constraints  $d_i$ .

Return a 1-d numpy array of the number of units that should be produced for each product. (For `productMix.npz`, the function should return an array of length four). *Hint:* Because this is a maximization problem and your solver works with minimizations, you will need to change the sign of the array `c`.

## Beyond Simplex

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [Nas00]. However, like any other algorithm, Simplex has its drawbacks.

In 1972, Victor Klee and George Minty published a paper with several examples of worst-case polytopes for the Simplex algorithm [KM72]. In their paper, they give several examples of polytopes that the Simplex algorithm struggles to solve.

Consider the following linear program from Klee and Minty.

$$\begin{array}{lllll}
 \max & 2^{n-1}x_1 & +2^{n-2}x_2 & +\cdots & +2x_{n-1} & +x_n \\
 \text{subject to } & x_1 & & & & \leq 5 \\
 & 4x_1 & & & & \leq 25 \\
 & 8x_1 & +4x_2 & & & \leq 125 \\
 & \vdots & & & & \vdots \\
 & 2^n x_1 & +2^{n-1}x_2 & +\cdots & +4x_{n-1} & +x_n \leq 5
 \end{array}$$

Klee and Minty show that for this example, the worst case scenario has exponential time complexity. With only  $n$  constraints and  $n$  variables, the simplex algorithm goes through  $2^n$  iterations. This is because there are  $2^n$  extreme points, and when starting at the point  $x = 0$ , the simplex algorithm goes through all of the extreme points before reaching the optimal point  $(0, 0, \dots, 0, 5^n)$ . Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.



# 20 Gymnasium

**Lab Objective:** *Gymnasium is a module designed to learn and apply reinforcement learning. The purpose of this lab is to learn the variety of functionalities available in Gymnasium and to implement them in various environments.*

Gymnasium is a module used to perform reinforcement learning. It contains a collection of environments where reinforcement learning can be used to accomplish various tasks. These environments include performing computer functions such as copy and paste, playing Atari video games, and controlling robots. To install Gymnasium, simply run the following code:

```
>>> pip install gymnasium
>>> # You may also need to install these dependencies
>>> pip install gymnasium[all]
>>> pip install gymnasium[classic-control]
```

## Environments

Each environment in Gymnasium can be thought of as a different scenario where reinforcement learning can be applied. A catalog of environments can be found using the following code.

```
>>> from gymnasium import envs
>>> print(envs.registry.values())
dict_values([EnvSpec(id='CartPole-v0', entry_point='gymnasium.envs.classic_control.cartpole:CartPoleEnv', reward_threshold=195.0, nondeterministic=False, max_episode_steps=200, order_enforce=True, autoreset=False, disable_env_checker=False, apply_api_compatibility=False, kwargs={}, namespace=None, name='CartPole', version=0, additional_wrappers=(), vector_entry_point='gymnasium.envs.classic_control.cartpole:CartPoleVectorEnv'), ...])
```

To learn more about Gymnasium and its environments, visit [gymnasium.farama.org](https://gymnasium.farama.org).

We will demonstrate how to work with Gymnasium environments by walking through the environment "[Blackjack-v1](#)". The game Blackjack<sup>1</sup> is a card game where the player receives two cards from a face card deck. The goal of the player is to get cards whose sum is as close to 21 as possible without exceeding 21. In this version of Blackjack, an ace is considered 1 or 11 and any face card is considered 10. On each turn, the player may choose to take another card or stop drawing cards. If their card sum does not exceed 21, they may take another card, but if it does, they lose. After the player stops drawing cards, the computer may play the same game. If the computer gets closer to 21 than the player (without exceeding 21), the player loses.

To begin working in an environment, the environment must be initialized and reset. Resetting the environment puts everything in the correct starting position and is necessary to begin using the environment. For example, in "[Blackjack-v1](#)", restarting the environment deals out a new game of Blackjack. Once the environment is complete, it should then be closed. Closing the environment tells the computer to stop running the environment (otherwise it will continue to run in the background).

```
>>> import gymnasium as gym
>>> env = gym.make('Blackjack-v1') # Initialize Blackjack-v1 environment
>>> env.reset() # Reset the environment
((16, 6, 1), {})

>>> env.close() # Close the environment
```

## Action Space

Once the environment is initialized and reset, the player can perform actions from the action space. To perform an action, use the function `step()`, which accepts the action as a parameter and returns an observation (more on those later). Environments may have discrete or continuous action spaces, but the environments presented in this lab all have discrete action spaces. When the action space is discrete, actions are defined as integers 0 through  $n$ , where  $n$  is the number of actions. The action space in "[Blackjack-v1](#)" has 2 actions, represented by 0 and 1: 0 indicates that the player will stop drawing cards, and 1 indicates that the player will draw another card.

```
>>> env = gym.make('Blackjack-v1')
>>> env.reset()
((12,9,0),{})
>>> env.action_space # Determine the number of actions available
Discrete(2)
# Select a random action and take a step using that action
>>> random_action = env.action_space.sample()
>>> random_action
1
# In this case, the random action was to draw another card
>>> env.step(random_action)
((16,9,0), 0.0, False, {})
```

---

<sup>1</sup>For more on how to play Blackjack, see <https://en.wikipedia.org/wiki/Blackjack>.

## Observation Space

The observation space of an environment contains all possible observations given an action. For example, in "`Blackjack-v1`", an observation is a tuple containing the total sum of the player's hand, the first card of the computer's hand, and a boolean indicating whether the player has an ace. The observation from each action can be found in the tuple returned by `step()`, which tells us the following information:

1. `observation`: The current state of the environment.
2. `reward`: The reward given from the observation. In most environments, maximizing the total reward increases performance. For example, the reward in '`Blackjack-v1`' is 1 if the player wins, -1 if the player loses, and 0 if there is a draw.
3. `done`: A boolean indicating whether the observation terminates the environment.
4. `truncated`: A boolean indicating whether the episode truncates for an abnormal reason.
5. `info`: Various information that may be helpful when debugging.

Consider the code below.

```
>>> env = gym.make('Blackjack-v1')
>>> env.reset()
((12,1,0),{})

>>> random_action = env.action_space.sample() # Make a random guess
>>> env.step(random_action)
((18,1,0), 0.0, False, False {})
```

This tuple can be interpreted as follows:

1. The sum of the player's hand is 18, the computer's first card is 1, and the player has no ace.
2. The reward is currently 0.0 (the game is not over yet).
3. The environment is not terminated.
4. The episode was not truncated
5. Information that may help debugging (which is currently empty).

In practice, this information is usually accessed by setting variables equal to `step()` as in

```
>>> obs, reward, done, trunc, info = env.step(random_action)
```

**Problem 1.** Write a function `random_blackjack()` that accepts an integer  $n$ . Initialize "`Blackjack-v1`"  $n$  times and each time take random actions until the game is terminated. Return the percentage of games the player wins. Use your function to print the win percentage after 50,000 games.

## Understanding Environments

Because each action and observation space is made up of numbers, good documentation is imperative to understanding any given environment. Fortunately, most environments in Gymnasium are very well documented, and most documentation follows the same pattern. There is a docstring which includes a description of the environment, a detailed action space, a detailed observation space, and explanation of rewards. It is always helpful to refer to this documentation when working in a Gymnasium environment.

In addition to documentation, certain environments can be understood better through visualization. For example, the environment "**Acrobot-v1**" displays a double pendulum. Rendering the environment allows the user to see the movement of the double pendulum as forces are applied to it. The best way to render an environment in Gymnasium is by running the following code through a python (.py) script, using the argument `render_mode='human'`.

```
>>> import gymnasium as gym

>>> env = gym.make('Acrobot-v1', render_mode='human')
>>> # env.reset() returns the observation space and corresponding info
>>> observation, info = env.reset()

>>> done = False
>>> while not done: # Until the environment terminates...
>>>     # Take random step
>>>     random_action = env.action_space.sample()
>>>     obs, reward, done, trunc, info = env.step(random_action)

>>> env.close()
```

However, this lab uses a Jupyter (.ipynb) file, and Gymnasium environments do not render well in Jupyter files. The visualization technique shown below is a simple workaround that uses the argument `render_mode='rgb_array'`, but unfortunately it renders slowly.

```
>>> from IPython import display
>>> from matplotlib import pyplot as plt

>>> env = gym.make('Acrobot-v1', render_mode='rgb_array')
>>> observation, info = env.reset()

>>> # Initialize visualization
>>> img = plt.imshow(env.render())

>>> done = False
>>> while not done:
>>>     # Take random step
>>>     random_action = env.action_space.sample()
>>>     obs, reward, done, trunc, info = env.step(random_action)

>>>     # Update visualization
>>>     img.set_data(env.render())
```

```
>>> display.display(plt.gcf())
>>> display.clear_output(wait=True)

>>> env.close()
```



Figure 20.1: Rendering of "Acrobot-v1"

## Solving An Environment

One way to solve an environment is to use information from the current observation to choose our next action. For example, consider "[Blackjack-v1](#)". Each observation tells us the player's current card sum. Based on the current card sum, we can decide whether we want to draw another card or stop drawing cards. To take the decided action, simply input the integer representing the action into the function `step()`.

**Problem 2.** Write a function `blackjack()` which runs a naïve algorithm to win blackjack. The function should receive an integer  $n$  as input. If the player's hand is less than or equal to  $n$ , the player should draw another card. If the player's hand is more than  $n$ , they should stop playing. Within the function, run the algorithm 10,000 times and return the percentage of games the player wins.

For  $n = 1, 2, \dots, 21$ , plot the average win rate returned by your function. Identify which value(s) of  $n$  wins most often.

Hint: Remember what the actions are in the action space of "[Blackjack-v1](#)".

**Problem 3.** The environment "[CartPole-v1](#)" presents a cart with a vertical pole. The goal of the environment is to keep the pole vertical as long as possible. The cart moves to the left with action 0, and it moves to the right with action 1. The observation space of this environment is a 4-dimensional array containing: the cart position, the cart velocity, the pole angle, and the pole angular velocity, respectively. More information about this environment can be found at [gymnasium.farama.org/environments/classic\\_control/cart\\_pole/](#).

Write a function `cartpole()` which initializes the "[CartPole-v1](#)" environment and keeps the pole vertical as long as possible based on the angular velocity of the tip of the pole. Return the number of steps it takes before it terminates (about 200 on average).

Run the game a single time and render the environment at each step. Then run your function 100 times without rendering, and print the average number of steps before it terminates.

**Problem 4.** The environment "[MountainCar-v0](#)" shows a car in a valley. The goal of the environment is to get the car to the top of the right mountain. The car can be driven forward (toward the goal) with action 2, can be driven backward with action 0, and will be put in neutral with action 1. Note that the car cannot immediately get up the hill because of gravity, so in order to move the car to goal, momentum will need to be gained by going back and forth between both sides of the valley. The observation space of this environment is a 2-dimensional array containing the  $x$  position and the velocity of the car, respectively. More information about this environment can be found at [gymnasium.farama.org/environments/classic\\_control/mountain\\_car/](#).

Using the given position and velocity of the car, write a function `car()` that solves the "[MountainCar-v0](#)" environment. Return the number of steps it takes before it terminates, which should be less than 180.

Run the game a single time and render the environment at each step. Then run your function 100 times without rendering, and print the average number of steps before it terminates.

## Q-Learning

While naïve methods like the ones above can be useful, reinforcement learning is a much better approach for using Gymnasium. Reinforcement learning is a subfield of machine learning where a problem is attempted over and over again. Each time a method is used to solve the problem, the method adapts based on the information gained from the previous attempt. Information can be gained from the sequence of observations and the total reward earned.

One simple reinforcement method is called *Q-learning*. While the theory behind Q-learning will not be explained in detail, the main idea is that the next action is not only based on the reward of the current action, but also of the next action. Q-learning creates a Q-table, which is an  $n \times m$  dimensional array, where  $n$  is the number of observations and  $m$  is the number of actions. For each state, the optimal action is the action that maximizes the value in the Q-table. In other words, if I am at observation  $i$ , the best action is the argmax of row  $i$  in the Q-table.

Q-learning requires 3 hyperparameters:

1. `alpha`: the learning rate. This determines whether to accept new values into the q-table.
2. `gamma`: the discount factor. The discount factor determines how important the reward of the current action is compared to the following action.

3. **epsilon**: the maximum value. This is the max reward that can be earned from a future action (not the current).

These hyperparameters can be changed to create different Q-tables. The following function will generate the optimal Q-table for a given environment.

```
def find_qvalues(env, alpha=.1, gamma=.6, epsilon=.1):
    """ Use the Q-learning algorithm to find qvalues.

    Parameters:
        env (str): environment name
        alpha (float): learning rate
        gamma (float): discount factor
        epsilon (float): maximum value

    Returns:
        q_table (ndarray nxm)
    """

    env = gym.make(env) # Make environment
    # Make Q-table
    q_table = np.zeros((env.observation_space.n, env.action_space.n))

    # Train
    for i in range(1,100001):
        state, info = env.reset() # Reset state

        epochs, penalties, reward, = 0,0,0
        done = False
        while not done:
            # Accept based on epsilon
            if random.uniform(0,1) < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state])

            # Take action
            next_state, reward, done, truncated, info = env.step(action)

            # Calculate new qvalue
            old_value = q_table[state,action]
            next_max = np.max(q_table[next_state])

            new_value = (1-alpha) * old_value + alpha * (reward + gamma * ↵
                next_max)
            q_table[state, action] = new_value

            # Check if penalty is made
            if reward == -10:
                penalties += 1

            # Get next observation
            state = next_state
```

```

    epochs += 1

    # Print episode number
    if i % 100 == 0:
        display.clear_output(wait=True)
        print(f"Episode: {i}")

    env.close()
    print("Training finished.")
    return q_table

```

**Problem 5.** The environment "[Taxi-v3](#)" depicts a taxi on a city grid, as shown in Figure 20.2. The goal of this environment is to pick up a passenger in a taxi and drop them off at their destination as fast as possible. You will have to look at the environment specifications at [gymnasium.farama.org/environments/toy\\_text/taxi/](https://gymnasium.farama.org/environments/toy_text/taxi/) to understand it better.

Initialize the environment, then randomly act until the environment is done and print the total reward. Since the taxi is acting randomly, it often takes so long for the environment to render that Jupyter will crash, so do NOT attempt to render this environment.

Next, use `find_qvalues()` to calculate the optimal Q-table of the environment using `alpha=.1`, `gamma=.6`, and `epsilon=.1`. Then, render "[Taxi-v3](#)", use the Q-table to move through it (as described above), and print the total reward.

Hint: `q_table[observation,:]` might be helpful.

Finally, write a function `taxis()` which initializes the "[Taxi-v3](#)" environment (without rendering). Run both scenarios described above 1000 times each, and return both the average random total reward and the average Q-learning total reward.

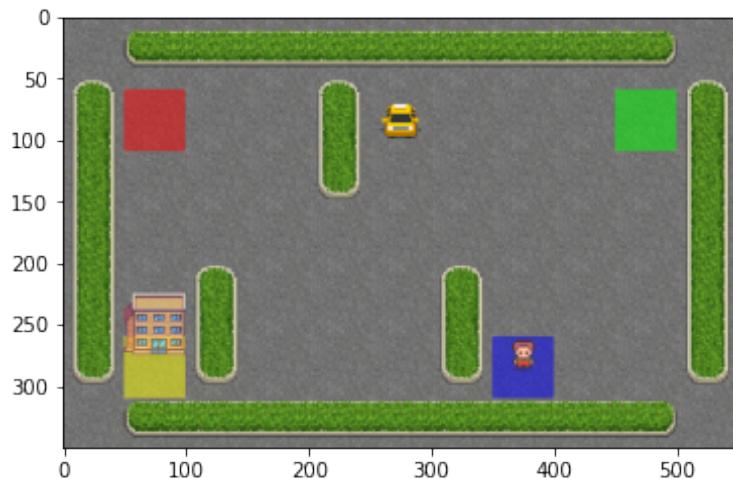


Figure 20.2: Example starting position of "[Taxi-v3](#)"

# 21

## CVXPY

**Lab Objective:** *CVXPY is a package of Python functions and classes designed for the purpose of convex optimization. In this lab we use these tools for linear and quadratic programming. We will solve various optimization problems using CVXPY and optimize eating healthily on a budget.*

### Linear Programs

A *linear program* is a linear constrained optimization problem. Such a problem can be stated in several different forms, one of which is

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && G\mathbf{x} \preceq \mathbf{h} \\ & && A\mathbf{x} = \mathbf{b}. \end{aligned}$$

The symbol  $\preceq$  denotes that the components of  $G\mathbf{x}$  are less than the components of  $\mathbf{h}$ . In other words, if  $\mathbf{x} \preceq \mathbf{y}$ , then  $x_i < y_i$  for all  $x_i \in \mathbf{x}$  and  $y_i \in \mathbf{y}$ . CVXPY accepts  $\leq$ ,  $\geq$ , and  $=$  in its constraints as long as the equations satisfy convexity requirements described later in this chapter, so we can reformulate this problem in yet another form:

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && G\mathbf{x} \preceq \mathbf{h} \\ & && P\mathbf{x} \succeq \mathbf{q} \\ & && A\mathbf{x} = \mathbf{b}. \end{aligned}$$

CVXPY accepts NumPy arrays and SciPy sparse matrices for the constraints, but the variable  $\mathbf{x}$  must be a CVXPY `Variable`.

Consider the following example:

$$\begin{aligned} & \text{minimize} && -4x_1 - 5x_2 \\ & \text{subject to} && x_1 + 2x_2 \leq 3 \\ & && 2x_1 + x_2 = 3 \\ & && x_1, x_2 \geq 0 \end{aligned}$$

We can solve this problem using the following code. Note that `cvxpy.Problem()` accepts the constraints as a single list and that `>=` represents both standard and elementwise greater than or equal to. The symbols `<=` and `==` are similarly versatile.

```
>>> import cvxpy as cp
>>> import numpy as np

# First we'll initialize the objective
# We can declare x with its size and sign
# nonneg = True is equivalent to the constraint P@ x >= 0 listed below
# Both are included for demonstration but are redundant
>>> x = cp.Variable(2, nonneg = True)
>>> c = np.array([-4, -5])
>>> objective = cp.Minimize(c.T @ x)

# Then we'll write the constraints
>>> A = np.array([2, 1])
>>> G = np.array([1, 2])
>>> P = np.eye(2)
>>> constraints = [A @ x == 3, G @ x <= 3, P @ x >= 0] #This must be a list

# Assemble the problem and then solve it
>>> problem = cp.Problem(objective, constraints)
>>> print(problem.solve())
-8.99999999850528
>>> print(x.value)
array([1., 1.])
```

### ACHTUNG!

If you are having trouble with `pip install cvxpy` check the following:

- CVXPY requires a C++ compiler, most MacOs ad Linux Systems have them built in. If you are running Windows, make sure that you have the "C++ builder tools" from the Visual Studio Build Tools installed.
- CVXPY requires specific versions of packages in order to run, check that you have the right version of your packages. The most common is NumPy is not up to date.

**Problem 1.** Solve the following convex optimization problem:

$$\begin{array}{ll}\text{minimize} & 2x_1 + x_2 + 3x_3 \\ \text{subject to} & x_1 + 2x_2 \leq 3 \\ & x_2 - 4x_3 \leq 1 \\ & 2x_1 + 10x_2 + 3x_3 \geq 12 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0\end{array}$$

Return the minimizer  $\mathbf{x}$  and the primal objective value.

## $l_1$ Norm

The  $l_1$  norm is defined

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|.$$

An  $l_1$  minimization problem is minimizing a vector's  $l_1$  norm, while fitting certain constraints. It can be written in the following form:

$$\begin{array}{ll}\text{minimize} & \|\mathbf{x}\|_1 \\ \text{subject to} & A\mathbf{x} = \mathbf{b}.\end{array}$$

CVXPY includes the  $l_1$  norm and many other useful functions. To specify a norm in CVXPY, use the syntax `cp.norm(x, a)` where  $a$  represents your choice of norm (1 in this case).

**Problem 2.** Write a function called `l1Min()` that accepts a matrix  $A$  and vector  $\mathbf{b}$  as NumPy arrays and solves the  $l_1$  minimization problem. Return the minimizer  $\mathbf{x}$  and the primal objective value.

To test your function consider the matrix  $A$  and vector  $\mathbf{b}$  below.

$$A = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & -2 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

The linear system  $A\mathbf{x} = \mathbf{b}$  has infinitely many solutions. Use `l1Min()` to verify that the solution which minimizes  $\|\mathbf{x}\|_1$  is approximately  $\mathbf{x} = [0., 2.571, 1.857, 0.]^T$  and the minimum objective value is approximately 4.429. There's a file called `test_cvxpy_intro.py` that contains this example as a prewritten unit test that you can use to test your function for this problem.

## The Transportation Problem

Consider the following transportation problem: A piano company needs to transport thirteen pianos from their three supply centers (denoted by 1, 2, 3) to two demand centers (4, 5). Transporting a piano from a supply center to a demand center incurs a cost, listed in Table 21.3. The company wants to minimize shipping costs for the pianos while meeting the demand.

Supply Center	Number of pianos available
1	7
2	2
3	4

Table 21.1: Number of pianos available at each supply center

Demand Center	Number of pianos needed
4	5
5	8

Table 21.2: Number of pianos needed at each demand center

Supply Center	Demand Center	Cost of transportation	Number of pianos
1	4	4	$p_1$
1	5	7	$p_2$
2	4	6	$p_3$
2	5	8	$p_4$
3	4	8	$p_5$
3	5	9	$p_6$

Table 21.3: Cost of transporting one piano from a supply center to a demand center

A system of constraints can be defined using the variables  $p_1, p_2, p_3, p_4, p_5$ , and  $p_6$ . First, there cannot be a negative number of pianos transported along any route. Next, use tables 21.1 and 21.2 and the variables  $p_1 \dots p_6$  to define a supply or demand constraint for each location. You may want to format this as a matrix. Finally, the objective function is the number of pianos shipped along each route multiplied by the respective costs (Table 21.3).

#### NOTE

Since our answers must be integers, in general this problem turns out to be an NP-hard problem. There is a whole field devoted to dealing with integer constraints, called *integer linear programming*, which is beyond the scope of this lab. Fortunately, we can treat this particular problem as a standard linear program and still obtain integer solutions.

**Problem 3.** Solve the piano transportation problem. Return the minimizer  $\mathbf{x}$  and the primal objective value.

## Quadratic Programming

Quadratic programming is similar to linear programming, but the objective function is quadratic rather than linear. The constraints, if there are any, are still of the same form. Thus,  $G$ ,  $\mathbf{h}$ ,  $A$ , and  $\mathbf{b}$  are optional. The formulation that we will use is

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{r}^T \mathbf{x} \\ \text{subject to} \quad & G \mathbf{x} \leq \mathbf{h} \\ & A \mathbf{x} = \mathbf{b}, \end{aligned}$$

where  $Q$  is a positive semidefinite symmetric matrix.

As an example, consider the quadratic function

$$f(x_1, x_2) = 2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2.$$

There are no constraints, so we only need to initialize the matrix  $Q$  and the vector  $\mathbf{r}$ . To find these, we first rewrite our function to match the formulation given above. If we let

$$Q = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \quad \mathbf{r} = \begin{bmatrix} d \\ e \end{bmatrix}, \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

then

$$\begin{aligned} \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{r}^T \mathbf{x} &= \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} d \\ e \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \frac{1}{2} ax_1^2 + bx_1x_2 + \frac{1}{2} cx_2^2 + dx_1 + ex_2 \end{aligned}$$

Thus, we see that the proper values to initialize our matrix  $Q$  and vector  $\mathbf{r}$  are:

$$\begin{aligned} a &= 4 & d &= 1 \\ b &= 2 & e &= -1 \\ c &= 2 & & \end{aligned}$$

Now that we have the matrix  $Q$  and vector  $\mathbf{r}$ , we are ready to use the CVXPY function for quadratic programming, `cp.quad_form()`.

```
>>> Q = np.array([[4, 2], [2, 2]])
>>> r = np.array([1, -1])
>>> x = cp.Variable(2)
>>> prob = cp.Problem(cp.Minimize(.5 * cp.quad_form(x, Q) + r.T @ x))
>>> print(prob.solve())
[-1. 1.5]
>>> print(x.value)
-1.25
```

**Problem 4.** Find the minimizer and minimum of

$$g(x_1, x_2, x_3) = \frac{3}{2}x_1^2 + 2x_1x_2 + x_1x_3 + 2x_2^2 + 2x_2x_3 + \frac{3}{2}x_3^2 + 3x_1 + x_3$$

(Hint: Write the function  $g$  to match the formulation given above before coding.)

So far we have only dealt with affine constraints. When working with non-affine constraints, be aware that CVXPY comes with some Disciplined Convex Programming (DCP) rules. A minimization problem requires a convex objective function; similarly, a maximization problem requires a concave objective function. Equality constraints ( $==$ ) must be affine. Less-than constraints ( $<=$ ) must have the left side convex and the right side concave. Greater-than constraints ( $>=$ ) must have the left side concave and the right side convex. This webpage provides a list of which of CVXPY's functions are concave or convex.

[https://www\\_cvxpy.org/tutorial/functions/index.html](https://www_cvxpy.org/tutorial/functions/index.html)

**Problem 5.** Write a function that accepts a matrix  $A$  and vector  $\mathbf{b}$  and solves the following problem.

$$\begin{aligned} &\text{minimize} && \|A\mathbf{x} - \mathbf{b}\|_2 \\ &\text{subject to} && \|\mathbf{x}\|_1 = 1 \\ & && \mathbf{x} \succeq 0 \end{aligned}$$

To test your function, use the matrix  $A$  and vector  $\mathbf{b}$  from Problem 2. The minimizer is approximately  $\mathbf{x} = [0, 1, 0, 0]$  with objective value 5.099. Hint: `norm()` is a convex function, so you will have to think of a different way to take the 1-norm.

## UNIT TEST

There is a file called `test_cvxpy_intro.py` that contains prewritten unit tests for Problem 2. There is a place for you to add your own unit tests to test your function for Problem 5, which will be graded.

## Eating on a Budget

In 2009, the inmates of Morgan County jail convinced Judge Clemon of the Federal District Court in Birmingham to put Sheriff Barlett in jail for malnutrition. Under Alabama law, in order to encourage less spending, "the chief lawman could go light on prisoners' meals and pocket the leftover change."<sup>1</sup>. Sheriffs had to ensure a minimum amount of nutrition for inmates, but minimizing costs meant more money for the sheriffs themselves. Judge Clemon jailed Sheriff Barlett until a plan was made to use all allotted funds, \$1.75 per inmate, to feed prisoners more nutritious meals. While this case made national news, the controversy of feeding prisoners in Alabama continues as of 2019<sup>2</sup>.

<sup>1</sup>Nossiter, Adam, 8 Jan 2009, "As His Inmates Grew Thinner, a Sheriff's Wallet Grew Fatter", *New York Times*, <https://www.nytimes.com/2009/01/09/us/09sheriff.html>

<sup>2</sup>Sheets, Connor, 31 January 2019, "Alabama sheriffs urge lawmakers to get them out of the jail food business", <https://www.al.com/news/2019/01/alabama-sheriffs-urge-lawmakers-to-get-them-out-of-the-jail-food-business.html>

The problem of minimizing cost while reaching healthy nutritional requirements can be approached as a convex optimization problem. Rather than viewing this problem from the sheriff's perspective, we view it from the perspective of a college student trying to minimize food cost in order to pay for higher education, all while meeting standard nutritional guidelines.

The file `food.npy` contains a dataset with nutritional facts for 18 foods that have been eaten frequently by college students working on this text. A subset of this dataset can be found in Table 21.4, where the "Food" column contains the list of all 18 foods.

The columns of the full dataset are:

- Column 1:  $p$ , price (dollars)
- Column 2:  $s$ , servings per container
- Column 3:  $c$ , calories per serving
- Column 4:  $f$ , fat per serving (grams)
- Column 5:  $\hat{s}$ , sugar per serving (grams)
- Column 6:  $\hat{c}$ , calcium per serving (milligrams)
- Column 7:  $\hat{f}$ , fiber per serving (grams)
- Column 8:  $\hat{p}$ , protein per serving (grams)

Food	Price $p$ dollars	Servings $s$	Calories $c$	Fat $f$ g	Sugar $\hat{s}$ g	Calcium $\hat{c}$ mg	Fiber $\hat{f}$ g	Protein $\hat{p}$ g
Ramen	6.88	48	190	7	0	0	0	5
Potatoes	0.48	1	290	0.4	3.2	53.8	6.9	7.9
Milk	1.79	16	130	5	12	250	0	8
Eggs	1.32	12	70	5	0	28	0	6
Pasta	3.88	8	200	1	2	0	2	7
Frozen Pizza	2.78	5	350	11	5	150	2	14
Potato Chips	2.12	14	160	11	1	0	1	1
Frozen Broccoli	0.98	4	25	0	1	25	2	1
Carrots	0.98	2	52.5	0.3	6.1	42.2	3.6	1.2
Bananas	0.24	1	105	0.4	14.4	5.9	3.1	1.3
Tortillas	3.48	18	140	4	0	0	0	3
Cheese	1.88	8	110	8	0	191	0	6
Yogurt	3.47	5	90	0	7	190	0	17
Bread	1.28	6	120	2	2	60	0.01	4
Chicken	9.76	20	110	3	0	0	0	20
Rice	8.43	40	205	0.4	0.1	15.8	0.6	4.2
Pasta Sauce	3.57	15	60	1.5	7	20	2	2
Lettuce	1.78	6	8	0.1	0.6	15.5	1	0.6

Table 21.4: Subset of table containing food data

According to the FDA<sup>1</sup> and US Department of Health, someone on a 2000 calorie diet should have no more than 2000 calories, no more than 65 grams of fat, no more than 50 grams of sugar<sup>2</sup>, at least 1000 milligrams of calcium<sup>1</sup>, at least 25 grams of fiber, and at least 46 grams of protein<sup>2</sup> per day. We can rewrite this as a convex optimization problem below.

$$\begin{aligned}
 & \text{minimize} && \sum_{i=1}^{18} p_i x_i, \\
 & \text{subject to} && \sum_{i=1}^{18} c_i x_i \leq 2000, \\
 & && \sum_{i=1}^{18} f_i x_i \leq 65, \\
 & && \sum_{i=1}^{18} \hat{s}_i x_i \leq 50, \\
 & && \sum_{i=1}^{18} \hat{c}_i x_i \geq 1000, \\
 & && \sum_{i=1}^{18} \hat{f}_i x_i \geq 25, \\
 & && \sum_{i=1}^{18} \hat{p}_i x_i \geq 46, \\
 & && x_i \geq 0.
 \end{aligned}$$

**Problem 6.** Read in the file `food.npy`. Use CVXPY to identify how much of each food item a college student should eat to minimize cost spent each day given these simplified nutrition requirements. Return the minimizing vector and the total amount of money spent.

According to this problem, what is the food you should eat most each day? What are the three foods you should eat most each week?

(Hint: Each nutritional value must be multiplied by the number of servings to get the nutrition value of the whole product).

You can learn more about CVXPY at <https://www=cvxpy.org/index.html>.

<sup>1</sup>url`https://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/pdv.html`

<sup>2</sup>`https://www.today.com/health/4-rules-added-sugars-how-calculate-your-daily-limit-t34731`

<sup>1</sup>26 Sept 2018, `https://ods.od.nih.gov/factsheets/Calcium-HealthProfessional/`

<sup>2</sup>`https://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/protein.html`

# 22

## Non-negative Matrix Factorization

**Lab Objective:** *Understand and implement the non-negative matrix factorization generator for recommendation systems with CVXPY.*

### Introduction

Collaborative filtering is the process of filtering data for patterns using collaboration techniques. More specifically, it refers to making prediction about a user's interests based on other users' interests. These predictions can be used to recommend items and are why collaborative filtering is one of the common methods of creating a recommendation system.

Recommendation systems look at the similarity between users to predict what item a user is most likely to enjoy. Common recommendation systems include Netflix's "Movies you Might Enjoy" list, Spotify's "Discover Weekly" playlist, and Amazon's "Products You Might Like" suggestions.

### Non-negative Matrix Factorization

Non-negative matrix factorization is one algorithm used in collaborative filtering. It can be applied to many other cases, including image processing, text mining, clustering, and community detection. The objective of non-negative matrix factorization is to take a non-negative matrix  $V$  and factor it into the product of two non-negative matrices.

For  $V \in \mathbb{R}^{m \times n}, 0 \preceq V$ ,

$$\begin{array}{ll}\text{minimize} & \|V - WH\| \\ \text{subject to} & 0 \preceq W, 0 \preceq H \\ \text{where} & W \in \mathbb{R}^{m \times k}, H \in \mathbb{R}^{k \times n}\end{array}$$

$k$  is the rank of the decomposition and can either be specified or found using the Root Mean Squared Error (the square root of the MSE), SVD, Non-negative Least Squares, or cross-validation techniques.

For this lab, we will use the Frobenius norm, given by

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}.$$

It is equivalent to the square root of the sum of the diagonal of  $A^H A$ .

**Problem 1.** Create the `NMFRecommender` class, which will be used to implement the NMF algorithm. Initialize the class with the following parameters: `random_state` defaulting to 15, `tol` defaulting to  $1e - 3$ , `maxiter` defaulting to 200, and `rank` defaulting to 3.

Add a method called `initialize_matrices` that takes in  $m$  and  $n$ , the dimensions of  $V$ . Set the random seed so that initializing the matrices can be replicated:

```
np.random.seed(self.random_state)
```

Initialize  $W$  and  $H$  using randomly generated numbers between 0 and 1, where  $W \in \mathbb{R}^{m \times k}$  and  $H \in \mathbb{R}^{k \times n}$ , where `k=rank`. Store  $W$  and  $H$  as attributes and return them.

## CVXPY

In order to compute the NMF of a matrix we will use the convex optimization package `CVXPY`. `CVXPY` is convex optimization package that takes a problem, converts it into standard form, calls a solver, and processes the result. Here is a basic example of how to use `CVXPY`

```
import cvxpy as cp

# Create two scalar optimization variables.
x = cp.Variable()
y = cp.Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve() # Returns the optimal value.

# print the status of the solution
print(prob.status)
```

The variables `x` and `y` are updated as the problem is solved. Constraints are not required.

Vector valued variables can be created by including the dimension of the variable as an argument like so: `x = cp.Variable(10)`. Similarly matrix valued variables can be created: `x = cp.Variable((5,5))`.

When initialized, variables have values of `None`. A value can be assigned to a variable before a problem is solved. The value can also be extracted after the optimal solution to a problem is found.

```
import numpy as np

random_matrix = np.random.random((5,5))
x = cp.Variable((5,5))
x.value = random_matrix
...
solve a problem using the variable x
...
solution_matrix = x.value
```

**Problem 2.** Finish the NMF class by adding a method `fit` that uses CVXPY to find an optimal  $W$  and  $H$ . It should accept  $V$  as a numpy array.

Constructing the problem so that it is known to be convex is important. Unfortunately, optimizing over 2 matrices being multiplied together breaks the rules of disciplined convex programming. Because of this, you must solve for  $W$  and  $H$  alternately. First solve for an optimal  $W$  given an  $H$ , then use that  $W$  to solve for an optimal  $H$  and so on. Continue until the difference between  $W$  and  $H$  and their previous values both have a Frobenius norm less than `tol`.

Finally, add a method called `reconstruct` that reconstructs and returns  $V$  by multiplying  $W$  and  $H$ .

HINT: You can build non-negativity into a CVXPY variable with `x = cp.Variable(n, nonneg=True)`. You can check if the solution is optimal by checking the status of your problem object with `prob.status`.

## Using NMF for Recommendations

Consider the following marketing problem where we have a list of five grocery store customers and their purchases. We want to create personalized food recommendations for their next visit. We start by creating a matrix representing each person and the number of items they purchased in different grocery categories. So from the matrix, we can see that John bought two fruits and one sweet.

$$V = \begin{pmatrix} & John & Alice & Mary & Greg & Peter & Jennifer \\ & 0 & 1 & 0 & 1 & 2 & 2 \\ & 2 & 3 & 1 & 1 & 2 & 2 \\ & 1 & 1 & 1 & 0 & 1 & 1 \\ & 0 & 2 & 3 & 4 & 1 & 1 \\ & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{array}{l} \\ Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

After performing NMF on  $V$ , we'll get the following  $W$  and  $H$ .

$$W = \begin{pmatrix} Component1 & Component2 & Component3 \\ 2.19 & 0. & 0.03 \\ 1.53 & 3.13 & 0.11 \\ 0.61 & 1.58 & 0. \\ 0.01 & 0. & 1.88 \\ 0.47 & 0. & 0. \end{pmatrix} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

$$H = \begin{pmatrix} John & Alice & Mary & Greg & Peter & Jennifer \\ 0. & 0.43 & 0. & 0.42 & 0.96 & 0.86 \\ 0.64 & 0.66 & 0.34 & 0. & 0.18 & 0.22 \\ 0. & 1.06 & 1.58 & 2.12 & 0.52 & 0.53 \end{pmatrix} \begin{array}{l} Component1 \\ Component2 \\ Component3 \end{array}$$

$W$  represents how much each grocery feature contributes to each component; a higher weight means it's more important to that component. For example, component 1 is heavily determined by vegetables followed by fruit, then sweets, coffee and finally bread. Component 2 is represented almost entirely by fruits, while component 3 is based on fruits and bread, with a small amount of vegetables.  $H$  is similar, except instead of showing how much each grocery category affects the component, it shows a much each person belongs to the component, again with a higher weight indicating that the person belongs more in that component. We can see the John belongs in component 2, while Jennifer mostly belongs in component 1.

To get our recommendations, we reconstruct  $V$  by multiplying  $W$  and  $H$ .

$$WH = \begin{pmatrix} John & Alice & Mary & Greg & Peter & Jennifer \\ 0. & 0.9735 & 0.0474 & 0.9834 & 2.118 & 1.8993 \\ 2.0032 & 2.8403 & 1.238 & 0.8758 & 2.0894 & 2.0627 \\ 1.0112 & 1.3051 & 0.5372 & 0.2562 & 0.87 & 0.8722 \\ 0. & 1.9971 & 2.9704 & 3.9898 & 0.9872 & 1.005 \\ 0. & 0.2021 & 0. & 0.1974 & 0.4512 & 0.4042 \end{pmatrix} \begin{array}{l} Vegetables \\ Fruits \\ Sweets \\ Bread \\ Coffee \end{array}$$

Most of the zeros from the original  $V$  have been filled in. This is the **collaborative filtering** portion of the algorithm. By sorting each column by weight, we can predict which items are more attractive to the customers. For instance, Mary has the highest weight for bread at 2.9704, followed by fruit at 1.238 and then sweets at 0.5372. So we would recommend bread to Mary.

Another way to interpret  $WH$  is to look at a feature and determine who is most likely to buy that item. So if we were having a sale on sweets but only had funds to let three people know, using the reconstructed matrix, we would want to target Alice, John, and Jennifer in that order. This gives us more information than  $V$  alone, which says that everyone except Greg bought one sweet.

**Problem 3.** Use the `NMFRecommender` class to run NMF on  $V$ , defined above, with 2 components. Return  $W$ ,  $H$ , and the number of people who have higher weights in component 2 than in component 1.

## Sklearn NMF

We also can compute the NMF using SkLearn. SkLearn uses a similar process as our class above. Here rank is represented by the parameter `n_components`.

```
from sklearn.decomposition import NMF

model = NMF(n_components=2, init='random', random_state=0)
W = model.fit_transform(V)
H = model.components_
```

SkLearn's NMF has other parameters that can also be included and adjusted. The `l1_ratio` is a value between 0 and 1 that gives a ratio of how much to weigh the l1 and l2 norms. When `l1_ratio = 1`, the l1 norm is used. When `l1_ratio = 0`, the l2 norm (Frobenius norm) is used. Any value between 0 and 1 is a weighted combination of the l1 and l2 norms. This ratio helps to prevent the model from over-fitting. The `alpha_W` and `alpha_H` parameters are floats that represent regularization constants; the default of 0 means that  $W$  and  $H$  are not regularized. If `alpha_H` is not specified, it will take on the value of `alpha_W` by default.

## Endmember Detection using NMF

NMF can be used for analysis and identification of images. If each image corresponds to a  $j \times k$  array of pixel intensities, then the data matrix is constructed by flattening the image into a vector of length  $jk$  and using these vectors as the columns of  $V$ , so  $V$  has dimensions  $jk \times n$ , where  $n$  is the number of images to analyze. Typically the materials in an image are referred to as the endmembers, which can be thought of as basis images which can be combined to reconstruct any image in the dataset. In the NMF decomposition  $H[k, j]$  represents the abundance of the  $k$ th endmember or basis face in the  $j$ th image.  $W[:, k]$  represents the spectral signature of the  $k$ th endmember. Then  $V[:, j] \cong WH[:, j]$ .

## Example

This example exhibits how to use SkLearn's NMF class. The example begins with augmenting and reformatting the images; this will be done for you in the problems following the example. Pay special attention to last two blocks of code which demonstrate using NMF with properly formatted images and plotting the results.

```
from sklearn.datasets import load_sample_images
import numpy as np

# function to convert colored image to gray scale image
def gray_convert(rgb):
    r, g, b = rgb[:, :, 0], rgb[:, :, 1], rgb[:, :, 2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray

# load in sample images
dataset = load_sample_images()
# grab the first image
```

```

image = dataset.images[0]
# convert image to gray scale
image = gray_convert(image)

# get augmentations for additional images
flipLR = np.fliplr(image)
flipUP = np.flipud(image)

# create matrix V
images = [np.ravel(image),np.ravel(flipLR),np.ravel(flipUP)]
images = np.transpose(images)

# decompose using NMF
model = NMF(n_components = 5,max_iter = 1000)
W = model.fit_transform(images)
H = model.components_

# plot basis images
plt.subplots_adjust(wspace = .02,hspace = .05)
plt.figure(figsize=(20, 8))
for i in range(W.shape[1]):
    plt.subplot(1,5,i+1)
    plt.xticks([], [])
    plt.yticks([],[])
    plt.imshow(np.reshape(W[:,i],(427,640)),cmap = 'gray')

```

The following three flattened images now make up the columns of  $V$ .



Figure 22.1: Original Images

Using a rank 5 reconstruction we see that the features used to reconstruct each image deal with the orientation of the building as well as the positive and negative space in each building.



Figure 22.2: Basis images for a rank 5 deconstruction. Each basis image comes from a column of  $W$ .

For the next two problems we will be using a dataset of facial images that we can load in with the code below. Notice that `get_faces` formats the faces and returns the  $V$  matrix for the face images.

```

def get_faces(path="./faces94"):
    """Traverse the specified directory to obtain one image per subdirectory.
       Flatten and convert each image to grayscale.

    Parameters:
        path (str): The directory containing the dataset of images.

    Returns:
        ((mn,k) ndarray) An array containing one column vector per
        subdirectory. k is the number of people, and each original
        image is mxn.
    """

    # Traverse the directory and get one image per subdirectory.
    faces = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":      # Only get jpg images.
                # Load the image, convert it to grayscale,
                # and flatten it into a vector.
                faces.append(np.ravel(imread(dirpath+"/"+fname, as_gray=True)))
                break
    # Put all the face vectors column-wise into a matrix.
    return np.transpose(faces)

def show(image, m=200, n=180, plt_show=False):
    """Plot the flattened grayscale 'image' of width 'w' and height 'h'.

    Parameters:
        image ((mn,) ndarray): A flattened image.
        m (int): The original number of rows in the image.
        n (int): The original number of columns in the image.
        plt_show (bool): if True, call plt.show() at the end
    """

    #scale image
    image = image / 255
    #reshape image

```

```

image = np.reshape(image,(m,n))
#show image
plt.imshow(image,cmap = "gray")

if plt_show:
    plt.show()

```

Similar to the example, we have basis faces that are used to reconstruct the images in the original dataset. A sample of basis faces for a rank 75 reconstruction of the faces dataset seem to correspond to the following endmembers: forehead, glasses, hair.

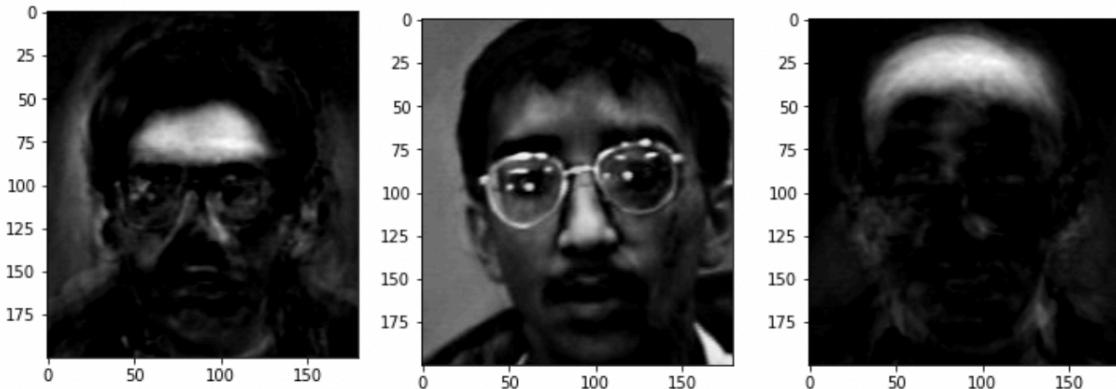


Figure 22.3: A sample of basis faces. Each basis face comes from a column of  $W$ .

**Problem 4.** Load in the facial dataset. SkLearn has the option to add regularization terms, `alpha_W` and `l1_ratio`, to the objective function  $\|V - WH\|$ . Reconstruct the third face in the dataset using SkLearn's NMF. Perform a grid search over the following: `n_components = [75]`, `alpha_W = [0, .2, .5]`, and `l1_ratio = [0,  $10^{-5}$ , 1]`.

A grid search is a way to hone in on the best parameters by looping through a "grid" of values. It involves running the process for each combination of parameters in the grid. For example, a grid search will run NMF with `n_components = 75`, `alpha_W = 0`, and `l1_ratio = 0`, then again with `n_components = 75`, `alpha_W = 0`, and `l1_ratio =  $10^{-5}$` , and so on, with all combinations of values. Note this will take a while to run (approximately 10 minutes) since it will run and fit the NMF model 9 total times. In the NMF function, make sure to set `init = "random"` or else the function won't converge correctly.

Determine which set of parameters best reconstructs the face. These are the parameters that most closely approximate the third face with  $W$  and  $H$ . (Hint: For each set of parameters, find the norm of `faces[:,2] - W @ H[:,2]` and see which is the smallest. Due to the randomness of the algorithm, answers may vary. Additionally, notice how we only look at the third face to avoid large matrix multiplication, which is computationally expensive.)

Plot all reconstructions of the third face and put the parameters in the title; use subplots.

**Problem 5.** Run NMF on the facial dataset again, using the best parameters from the problem above. Next, for the second and twelfth faces in the dataset, find the 10 basis faces with the largest coefficients. Plot these basis faces along with the original image using subplots. In a markdown block write a sentence or two about differences you notice in the features of the basis faces (look closely).



# 23 Interior Point 1: Linear Programs

**Lab Objective:** *For decades after its invention, the Simplex algorithm was the only competitive method for linear programming. The past 30 years, however, have seen the discovery and widespread adoption of a new family of algorithms that rival—and in some cases outperform—the Simplex algorithm, collectively called Interior Point methods. One of the major shortcomings of the Simplex algorithm is that the number of steps required to solve the problem can grow exponentially with the size of the linear system. Thus, for certain large linear programs, the Simplex algorithm is simply not viable. Interior Point methods offer an alternative approach and enjoy much better theoretical convergence properties. In this lab we implement an Interior Point method for linear programs, and in the next lab we will turn to the problem of solving quadratic programs.*

## Introduction

Recall that a linear program is a constrained optimization problem with a linear objective function and linear constraints. The linear constraints define a set of allowable points called the *feasible region*, the boundary of which forms a geometric object known as a *polytope*. The theory of convex optimization ensures that the optimal point for the objective function can be found among the vertices of the feasible polytope. The Simplex Method tests a sequence of such vertices until it finds the optimal point. Provided the linear program is neither unbounded nor infeasible, the algorithm is certain to produce the correct answer after a finite number of steps, but it does not guarantee an efficient path along the polytope toward the minimizer. Interior point methods do away with the feasible polytope and instead generate a sequence of points that cut through the interior (or exterior) of the feasible region and converge iteratively to the optimal point. Although it is computationally more expensive to compute such interior points, each step results in significant progress toward the minimizer. See Figure 23.1 for an example of a path using an Interior Point algorithm. In general, the Simplex Method requires many more iterations (though each iteration is less expensive computationally).

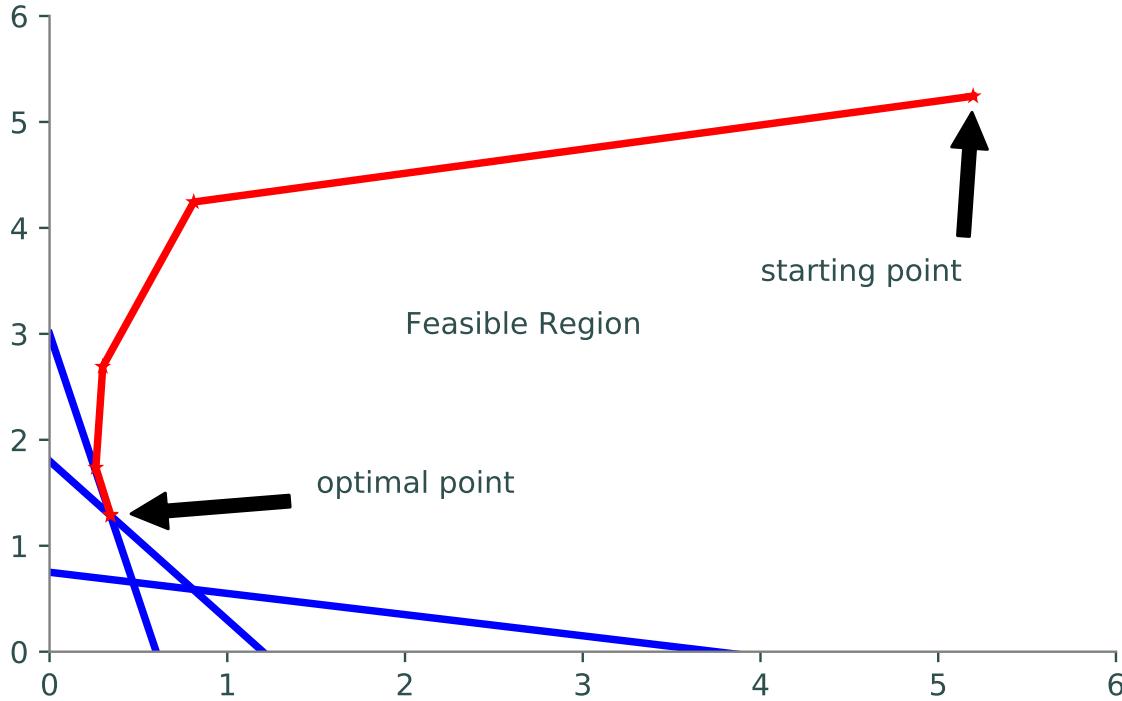


Figure 23.1: A path traced by an Interior Point algorithm.

## Primal-Dual Interior Point Methods

Some of the most popular and successful types of Interior Point methods are known as Primal-Dual Interior Point methods. Consider the following linear program:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{array}$$

Here,  $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ ,  $\mathbf{b} \in \mathbb{R}^m$ , and  $A \in \mathbb{R}^{m \times n}$  with full row rank. This is the *primal* problem, and its *dual* takes the form:

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T \boldsymbol{\lambda} \\ \text{subject to} & A^T \boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c} \\ & \boldsymbol{\mu}, \boldsymbol{\lambda} \geq \mathbf{0}, \end{array}$$

where  $\boldsymbol{\lambda} \in \mathbb{R}^m$  and  $\boldsymbol{\mu} \in \mathbb{R}^n$ .

## KKT Conditions

The theory of convex optimization gives us necessary and sufficient conditions for the solutions to the primal and dual problems via the Karush-Kuhn-Tucker (KKT) conditions. The Lagrangian for the primal problem is as follows:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{c}^T \mathbf{x} + \boldsymbol{\lambda}^T (\mathbf{b} - A\mathbf{x}) - \boldsymbol{\mu}^T \mathbf{x}$$

The KKT conditions are

$$\begin{aligned} A^T \boldsymbol{\lambda} + \boldsymbol{\mu} &= \mathbf{c} \\ A\mathbf{x} &= \mathbf{b} \\ x_i \mu_i &= 0, \quad i = 1, 2, \dots, n, \\ \mathbf{x}, \boldsymbol{\mu} &\succeq 0. \end{aligned}$$

It is convenient to write these conditions in a more compact manner, by defining an almost-linear function  $F$  and setting it equal to zero:

$$F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := \begin{bmatrix} A^T \boldsymbol{\lambda} + \boldsymbol{\mu} - \mathbf{c} \\ A\mathbf{x} - \mathbf{b} \\ M\mathbf{x} \end{bmatrix} = \mathbf{0},$$

$(\mathbf{x}, \boldsymbol{\mu} \succeq 0),$

where  $M = \text{diag}(\mu_1, \mu_2, \dots, \mu_n)$ . Note that the first row of  $F$  is the KKT condition for dual feasibility, the second row of  $F$  is the KKT condition for the primal problem, and the last row of  $F$  accounts for complementary slackness.

**Problem 1.** Define a function `interiorPoint()` that will be used to solve the complete interior point problem. This function should accept  $A$ ,  $\mathbf{b}$ , and  $\mathbf{c}$  as parameters, along with the keyword arguments `niter=20` and `tol=1e-16`. The keyword arguments will be used in a later problem.

In the next few problems, you will be writing functions within this function to solve the interior point problem one step at a time.

For this problem, within the `interiorPoint()` function, write a function for the vector-valued function  $F$  described above. This function should accept  $\mathbf{x}$ ,  $\boldsymbol{\lambda}$ , and  $\boldsymbol{\mu}$  as parameters and return a 1-dimensional NumPy array with  $2n + m$  entries.

## Search Direction

A Primal-Dual Interior Point method is a line search method that starts with an initial guess  $(\mathbf{x}_0^T, \boldsymbol{\lambda}_0^T, \boldsymbol{\mu}_0^T)$  and produces a sequence of points that converge to  $(\mathbf{x}^{*\top}, \boldsymbol{\lambda}^{*\top}, \boldsymbol{\mu}^{*\top})$ , the solution to the KKT equations and hence the solution to the original linear program. The constraints on the problem make finding a search direction and step length a little more complicated than for the unconstrained line search we have studied previously.

In the spirit of Newton's Method, we can form a linear approximation of the system  $F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}$  centered around our current point  $(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ , and calculate the direction  $(\Delta\mathbf{x}^T, \Delta\boldsymbol{\lambda}^T, \Delta\boldsymbol{\mu}^T)$  in which to step to set the linear approximation equal to  $\mathbf{0}$ . This equates to solving the linear system:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \\ \Delta\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \quad (23.1)$$

Here  $DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$  denotes the total derivative matrix of  $F$ . We can calculate this matrix block-wise by obtaining the partial derivatives of each block entry of  $F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$  with respect to  $\mathbf{x}$ ,  $\boldsymbol{\lambda}$ , and  $\boldsymbol{\mu}$ , respectively. We thus obtain:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ M & 0 & X \end{bmatrix}$$

where  $X = \text{diag}(x_1, x_2, \dots, x_n)$ .

Unfortunately, solving Equation 23.1 often leads to a search direction that is too greedy. Even small steps in this direction may lead the iteration out of the feasible region by violating one of the constraints. To remedy this, we define the *duality measure*  $\nu^1$  of the problem:

$$\nu = \frac{\mathbf{x}^T \boldsymbol{\mu}}{n}$$

The idea is to use Newton's method to identify a direction that strictly decreases  $\nu$ . Thus instead of solving Equation 23.1, we solve:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \\ \Delta \boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma \nu \mathbf{e} \end{bmatrix} \quad (23.2)$$

where  $\mathbf{e} = (1, 1, \dots, 1)^T$  and  $\sigma \in [0, 1]$  is called the *centering parameter*. The closer  $\sigma$  is to 0, the more similar the resulting direction will be to the plain Newton direction. The closer  $\sigma$  is to 1, the more the direction points inward to the interior of the feasible region.

**Problem 2.** Within `interiorPoint()`, write a subroutine to compute the search direction  $(\Delta \mathbf{x}^T, \Delta \boldsymbol{\lambda}^T, \Delta \boldsymbol{\mu}^T)$  by solving Equation 23.2. Use  $\sigma = \frac{1}{10}$  for the centering parameter.

Note that only the last block row of  $DF$  will need to be changed at each iteration (since  $M$  and  $X$  depend on  $\boldsymbol{\mu}$  and  $\mathbf{x}$ , respectively). Use the functions `lu_factor()` and `lu_solve()` from the `scipy.linalg` module to solving the system of equations efficiently.

## Step Length

Now that we have our search direction, it remains to choose our step length. We wish to step nearly as far as possible without violating the problem's constraints, thus remaining in the interior of the feasible region. First, we calculate the maximum allowable step lengths for  $\mathbf{x}$  and  $\boldsymbol{\mu}$ , respectively:

$$\alpha_{\max} = \min\{-\mu_i / \Delta \mu_i \mid \Delta \mu_i < 0\}$$

$$\delta_{\max} = \min\{-x_i / \Delta x_i \mid \Delta x_i < 0\}$$

If all values of  $\Delta \boldsymbol{\mu}$  are nonnegative, let  $\alpha_{\max} = 1$ . Likewise, if all values of  $\Delta \mathbf{x}$  are nonnegative, let  $\delta_{\max} = 1$ . Next, we back off from these maximum step lengths slightly:

$$\alpha = \min(1, 0.95 \alpha_{\max})$$

$$\delta = \min(1, 0.95 \delta_{\max}).$$

---

<sup>1</sup> $\nu$  is the Greek letter for  $n$ , pronounced “nu.”

These are our final step lengths. Thus, the next point in the iteration is given by:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + \delta \Delta \mathbf{x}_k \\ (\boldsymbol{\lambda}_{k+1}, \boldsymbol{\mu}_{k+1}) &= (\boldsymbol{\lambda}_k, \boldsymbol{\mu}_k) + \alpha(\Delta \boldsymbol{\lambda}_k, \Delta \boldsymbol{\mu}_k).\end{aligned}$$

**Problem 3.** Within `interiorPoint()`, write a subroutine to compute the step size after the search direction has been computed. Avoid using loops when computing  $\alpha_{\max}$  and  $\delta_{\max}$  (use masking and NumPy functions instead).

## Initial Point

Finally, the choice of initial point  $(\mathbf{x}_0, \boldsymbol{\lambda}_0, \boldsymbol{\mu}_0)$  is an important, nontrivial one. A naively or randomly chosen initial point may cause the algorithm to fail to converge. The following function will calculate an appropriate initial point.

```
def starting_point(A, b, c):
    """Calculate an initial guess to the solution of the linear program
    min c\trp x, Ax = b, x>=0.
    Reference: Nocedal and Wright, p. 410.
    """

    # Calculate x, lam, mu of minimal norm satisfying both
    # the primal and dual constraints.
    B = la.inv(A @ A.T)
    x = A.T @ B @ b
    lam = B @ A @ c
    mu = c - (A.T @ lam)

    # Perturb x and s so they are nonnegative.
    dx = max((-3./2)*x.min(), 0)
    dmu = max((-3./2)*mu.min(), 0)
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    # Perturb x and mu so they are not too small and not too dissimilar.
    dx = .5*(x*mu).sum()/mu.sum()
    dmu = .5*(x*mu).sum()/x.sum()
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    return x, lam, mu
```

**Problem 4.** Complete the implementation of `interiorPoint()`.

Use the function `starting_point()` provided above to select an initial point, then run the iteration `niter` times, or until the duality measure is less than `tol`. Return the optimal point  $\mathbf{x}^*$  and the optimal value  $\mathbf{c}^\top \mathbf{x}^*$ .

The duality measure  $\nu$  tells us in some sense how close our current point is to the minimizer. The closer  $\nu$  is to 0, the closer we are to the optimal point. Thus, by printing the value of  $\nu$  at each iteration, you can track how your algorithm is progressing and detect when you have converged.

To test your implementation, use the following code to generate a random linear program, along with the optimal solution.

```
def randomLP(j, k):
    """Generate a linear program min c\trp x s.t. Ax = b, x>=0.
    First generate m feasible constraints, then add
    slack variables to convert it into the above form.
    Inputs:
        j (int >= k): number of desired constraints.
        k (int): dimension of space in which to optimize.
    Outputs:
        A ((j, j+k) ndarray): Constraint matrix.
        b ((j,) ndarray): Constraint vector.
        c ((j+k,), ndarray): Objective function with j trailing 0s.
        x ((k,) ndarray): The first 'k' terms of the solution to the LP.
    """
    A = np.random.random((j,k))*20 - 10
    A[A[:, -1] < 0] *= -1
    x = np.random.random(k)*10
    b = np.zeros(j)
    b[:k] = A[:k, :] @ x
    b[k:] = A[k:, :] @ x + np.random.random(j-k)*10
    c = np.zeros(j+k)
    c[:k] = A[:k, :].sum(axis=0)/k
    A = np.hstack((A, np.eye(j)))
    return A, b, -c, x
```

```
>>> j, k = 7, 5
>>> A, b, c, x = randomLP(j, k)
>>> point, value = interiorPoint(A, b, c)
>>> np.allclose(x, point[:k])
True
```

## UNIT TEST

There is a file called `test_interior_point_linear.py` that contains a place for you to write unit tests to test your function from Problems 1-4. Use the `randomLP()` function to create test cases to use in your function. The tests you write will be graded.

## Least Absolute Deviations (LAD)

We now return to the familiar problem of fitting a line (or hyperplane) to a set of data. We have previously approached this problem by minimizing the sum of the squares of the errors between the data points and the line, an approach known as *least squares*. The least squares solution can be obtained analytically when fitting a linear function, or through a number of optimization methods (such as Conjugate Gradient) when fitting a nonlinear function.

The method of *least absolute deviations* (LAD) also seeks to find a best fit line to a set of data, but the error between the data and the line is measured differently. In particular, suppose we have a set of data points  $(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \dots, (y_m, \mathbf{x}_m)$ , where  $y_i \in \mathbb{R}$ ,  $\mathbf{x}_i \in \mathbb{R}^n$  for  $i = 1, 2, \dots, m$ . Here, the  $\mathbf{x}_i$  vectors are the *explanatory variables* and the  $y_i$  values are the *response variables*, and we assume the following linear model:

$$y_i = \boldsymbol{\beta}^\top \mathbf{x}_i + b, \quad i = 1, 2, \dots, m,$$

where  $\boldsymbol{\beta} \in \mathbb{R}^n$  and  $b \in \mathbb{R}$ . The error between the data and the proposed linear model is given by

$$\sum_{i=1}^n |\boldsymbol{\beta}^\top \mathbf{x}_i + b - y_i|,$$

and we seek to choose the parameters  $\boldsymbol{\beta}, b$  so as to minimize this error.

### Advantages of LAD

The most prominent difference between this approach and least squares is how they respond to outliers in the data. Least absolute deviations is robust in the presence of outliers, meaning that one (or a few) errant data points won't severely affect the fitted line. Indeed, in most cases, the best fit line is guaranteed to pass through at least two of the data points. This is a desirable property when the outliers may be ignored (perhaps because they are due to measurement error or corrupted data). Least squares, on the other hand, is much more sensitive to outliers, and so is the better choice when outliers cannot be dismissed. See Figure 23.2.

While least absolute deviations is robust with respect to outliers, small horizontal perturbations of the data points can lead to very different fitted lines. Hence, the least absolute deviations solution is less stable than the least squares solution. In some cases there are even infinitely many lines that minimize the least absolute deviations error term. However, one can expect a unique solution in most cases.

The least absolute deviations solution arises naturally when we assume that the residual terms  $\boldsymbol{\beta}^\top \mathbf{x}_i + b - y_i$  have a particular statistical distribution (the Laplace distribution). Ultimately, however, the choice between least absolute deviations and least squares depends on the nature of the data at hand, as well as your own good judgment.

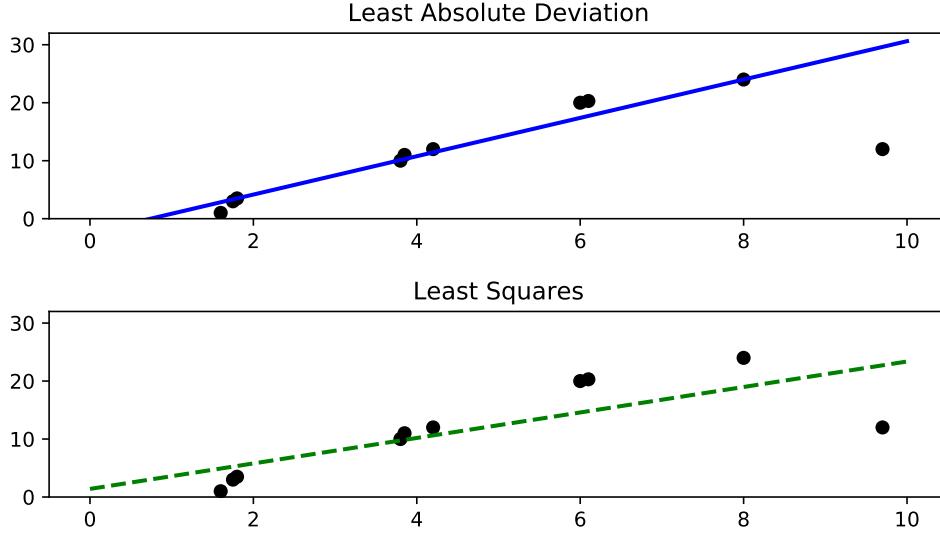


Figure 23.2: Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

## LAD as a Linear Program

We can formulate the least absolute deviations problem as a linear program, and then solve it using our interior point method. For  $i = 1, 2, \dots, m$  we introduce the artificial variable  $u_i$  to take the place of the error term  $|\beta^T \mathbf{x}_i + b - y_i|$ , and we require this variable to satisfy  $u_i \geq |\beta^T \mathbf{x}_i + b - y_i|$ . This constraint is not yet linear, but we can split it into an equivalent set of two linear constraints:

$$\begin{aligned} u_i &\geq \beta^T \mathbf{x}_i + b - y_i, \\ u_i &\geq y_i - \beta^T \mathbf{x}_i - b. \end{aligned}$$

The  $u_i$  are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^m u_i \\ \text{subject to} \quad & u_i \geq \beta^T \mathbf{x}_i + b - y_i, \\ & u_i \geq y_i - \beta^T \mathbf{x}_i - b. \end{aligned}$$

Now for each inequality constraint, we bring all variables ( $u_i, \beta, b$ ) to the left hand side and introduce a nonnegative slack variable to transform the constraint into an equality:

$$\begin{aligned} u_i - \beta^T \mathbf{x}_i - b - s_{2i-1} &= -y_i, \\ u_i + \beta^T \mathbf{x}_i + b - s_{2i} &= y_i, \\ s_{2i-1}, s_{2i} &\geq 0. \end{aligned}$$

Notice that the variables  $\beta, b$  are not assumed to be nonnegative, but in our interior point method, all variables are assumed to be nonnegative. We can fix this situation by writing these variables as the difference of nonnegative variables:

$$\begin{aligned}\beta &= \beta_1 - \beta_2, \\ b &= b_1 - b_2, \\ \beta_1, \beta_2 &\succeq \mathbf{0}; b_1, b_2 \geq 0.\end{aligned}$$

Substituting these values into our constraints, we have the following system of constraints:

$$\begin{aligned}u_i - \beta_1^T \mathbf{x}_i + \beta_2^T \mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i, \\ u_i + \beta_1^T \mathbf{x}_i - \beta_2^T \mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i, \\ \beta_1, \beta_2 &\succeq \mathbf{0}; u_i, b_1, b_2, s_{2i-1}, s_{2i} \geq 0.\end{aligned}$$

Writing  $\mathbf{y} = (-y_1, y_1, -y_2, y_2, \dots, -y_m, y_m)^\top$  and  $\beta_i = (\beta_{i,1}, \dots, \beta_{i,n})^\top$  for  $i = \{1, 2\}$ , we can aggregate all of our variables into one vector as follows:

$$\mathbf{v} = (u_1, \dots, u_m, \beta_{1,1}, \dots, \beta_{1,n}, \beta_{2,1}, \dots, \beta_{2,n}, b_1, b_2, s_1, \dots, s_{2m})^\top.$$

Defining  $\mathbf{c} = (1, 1, \dots, 1, 0, \dots, 0)^\top$  (where only the first  $m$  entries are equal to 1), we can write our objective function as

$$\sum_{i=1}^m u_i = \mathbf{c}^\top \mathbf{v}.$$

Hence, the final form of our linear program is:

$$\begin{aligned}&\text{minimize} && \mathbf{c}^\top \mathbf{v} \\ &\text{subject to} && A\mathbf{v} = \mathbf{y}, \\ & && \mathbf{v} \succeq \mathbf{0},\end{aligned}$$

where  $A$  is a matrix containing the coefficients of the constraints. Our constraints are now equalities, and the variables are all nonnegative, so we are ready to use our interior point method to obtain the solution.

## LAD Example

Consider the following example. We start with an array `data`, each row of which consists of the values  $y_i, x_{i,1}, \dots, x_{i,n}$ , where  $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^\top$ . We will have  $3m + 2(n+1)$  variables in our linear program. Below, we initialize the vectors  $\mathbf{c}$  and  $\mathbf{y}$ .

```
>>> m = data.shape[0]
>>> n = data.shape[1] - 1
>>> c = np.zeros(3*m + 2*(n + 1))
>>> c[:m] = 1
>>> y = np.empty(2*m)
>>> y[::2] = -data[:, 0]
>>> y[1::2] = data[:, 0]
>>> x = data[:, 1:]
```

The hardest part is initializing the constraint matrix correctly. It has  $2m$  rows and  $3m + 2(n + 1)$  columns. Try writing out the constraint matrix by hand for small  $m, n$ , and make sure you understand why the code below is correct.

```
>>> A = np.ones((2*m, 3*m + 2*(n + 1)))
>>> A[::2, :m] = np.eye(m)
>>> A[1::2, :m] = np.eye(m)
>>> A[::2, m:m+n] = -x
>>> A[1::2, m:m+n] = x
>>> A[::2, m+n:m+2*n] = x
>>> A[1::2, m+n:m+2*n] = -x
>>> A[::2, m+2*n] = -1
>>> A[1::2, m+2*n+1] = -1
>>> A[:, m+2*n+2:] = -np.eye(2*m, 2*m)
```

Now we can calculate the solution by calling our interior point function.

```
>>> sol = interiorPoint(A, y, c, niter=10)[0]
```

However, the variable `sol` holds the value for the vector

$$\mathbf{v} = (u_1, \dots, u_m, \beta_{1,1}, \dots, \beta_{1,n}, \beta_{2,1}, \dots, \beta_{2,n}, b_1, b_2, s_1, \dots, s_{2m+1})^T.$$

We extract values of  $\beta = \beta_1 - \beta_2$  and  $b = b_1 - b_2$  with the following code:

```
>>> beta = sol[m:m+n] - sol[m+n:m+2*n]
>>> b = sol[m+2*n] - sol[m+2*n+1]
```

**Problem 5.** The file `simdata.txt` contains two columns of data. The first gives the values of the response variables ( $y_i$ ), and the second column gives the values of the explanatory variables ( $x_i$ ). Find the least absolute deviations line for this data set, and plot it together with the data. Plot the least squares solution as well to compare the results.

```
>>> from scipy.stats import linregress
>>> slope, intercept = linregress(data[:,1], data[:,0])[:2]
>>> domain = np.linspace(0,10,200)
>>> plt.plot(domain, domain*slope + intercept)
```

# 24 Dynamic Programming

**Lab Objective:** Sequential decision making problems are a class of problems in which the current choice depends on future choices. They are a subset of Markov decision processes, an important class of problems with applications in business, robotics, and economics. Dynamic programming is a method of solving these problems that optimizes the solution by breaking the problem down into steps and optimizing the decision at each time period. In this lab we use dynamic programming to solve two classic dynamic optimization problems.

## The Marriage Problem

Many dynamic optimization problems can be classified as *optimal stopping* problems, where the goal is to determine at what time to take an action to maximize the expected reward. For example, how many people should you date before you get married? Or when hiring a secretary, how many people should you interview before hiring the current interviewee? These problems try to determine at which person  $t$  to stop in order to maximize the chance of getting the best candidate.

For instance, let  $N$  be the number of people you could date. After dating each person, you can either marry them or move on; you can't resume a relationship once it ends. In addition, you can rank your current relationship to all of the previous options, but not to future ones. The goal is to find the policy that maximizes the probability of choosing the best marriage partner. This policy may not always choose the best candidate, but it should get an almost-best candidate most of the time.

Let  $V(t - 1)$  be the probability that the best marriage partner is person  $t$ , assuming we didn't choose the first  $t - 1$  candidates while using an optimal policy. In other words, after dating  $t - 1$  people, you want to know the probability that the  $t^{th}$  person is the one you should marry. Note that the probability that the  $t^{th}$  person is the best candidate is  $\frac{1}{t}$  and the probability that they aren't is  $\frac{t-1}{t}$ . If the  $t^{th}$  person is not the best out of the first  $t$ , then the probability they are the best overall is 0 and the probability they are not is  $V(t)$ . If the  $t^{th}$  person is the best out of the first  $t$ , then the probability they are the best overall is  $\frac{t}{N}$  and the probability they are not is  $V(t)$ .

By Bellman's optimality equations,

$$V(t - 1) = \frac{t - 1}{t} \max \{0, V(t)\} + \frac{1}{t} \max \left\{ \frac{t}{N}, V(t) \right\} = \max \left\{ \frac{t - 1}{t} V(t) + \frac{1}{N}, V(t) \right\}. \quad (24.1)$$

Notice that (24.1) implies that  $V(t-1) \geq V(t)$  for all  $t \leq N$ . Hence, the probability of selecting the best match  $V(t)$  is non-increasing. Conversely,  $P(t \text{ is best overall} | t \text{ is best out of the first } t) = \frac{t}{N}$  is strictly increasing. Therefore, there is some  $t_0$ , called the *optimal stopping point*, such that  $V(t) \leq \frac{t}{N}$  for all  $t \geq t_0$ . After  $t_0$  relationships, we choose the next partner who is better than all of the previous ones. We can write (24.1) as

$$V(t-1) = \begin{cases} V(t_0) & t < t_0, \\ \frac{t-1}{t} V(t) + \frac{1}{N} & t \geq t_0. \end{cases}$$

The goal of an optimal stopping problem is to find  $t_0$ , which we can do by backwards induction. We start at the final candidate, who always has probability 0 of being the best overall if they are not the best so far, and work our way backwards, computing the expected value  $V(t)$ , for  $t = N, N-1, \dots, 1$ .

If  $N = 4$ , we have

$$\begin{aligned} V(4) &= 0, \\ V(3) &= \max \left\{ \frac{3}{4}V(4) + \frac{1}{4}, 0 \right\} = .25, \\ V(2) &= \max \left\{ \frac{2}{3}V(3) + \frac{1}{4}, .25 \right\} = .4166, \\ V(1) &= \max \left\{ \frac{1}{2}V(2) + \frac{1}{4}, .4166 \right\} = .4583. \end{aligned}$$

In this case, the maximum expected value is .4583 and the stopping point is  $t = 1$ . It is also useful to look at the percent of possible candidates you should pass over before selecting a candidate. This is called the optimal stopping percentage, which in this case is  $1/4 = .25$ . After passing over (and observing) the optimal percentage of candidates, the first candidate that is better than the candidates you observed should be selected<sup>1</sup>.

**Problem 1.** Write a function that accepts a number of candidates  $N$ . Calculate the expected values of choosing candidate  $t$  for  $t = 1, 2, \dots, N$ .

Return the highest expected value  $V(t_0)$  and the optimal stopping point  $t_0$ . (Hint: Python starts indices at 0, so you may need to adjust your indexing before returning the optimal stopping point.)

There is a file called `test_dynamic_programming.py` that contains a prewritten unit test for this problem. You can use it to make sure your function works for  $N = 4$ .

**Problem 2.** Write a function that takes in an integer  $M$  and runs your function from Problem 1 for each  $N = 3, 4, \dots, M$ . Graph the optimal stopping percentage of candidates ( $t_0/N$ ) to interview and the maximum probability  $V(t_0)$  against  $N$ . Return the optimal stopping percentage for  $M$ .

The optimal stopping percentage for  $M = 1000$  is .368.

---

<sup>1</sup>For examples and a more comprehensive explanation, see <https://www.americanscientist.org/article/knowing-when-to-stop>.

Both the stopping time and the probability of choosing the best person converge to  $\frac{1}{e} \approx .36788$ . Then to maximize the chance of having the best marriage, you should date at least  $\frac{N}{e}$  people before choosing the next best person. This famous problem is also known as the *secretary problem*, the *sultan's dowry problem*, and the *best choice problem*. For more information, see [https://en.wikipedia.org/wiki/Secretary\\_problem](https://en.wikipedia.org/wiki/Secretary_problem).

## The Cake Eating Problem

Imagine you are given a cake. How do you eat it to maximize your enjoyment? Some people may prefer to eat all of their cake at once and not save any for later. Others may prefer to eat a little bit at a time. If we are to consume a cake of size  $W$  over  $T + 1$  time periods, then our consumption at each step is represented as a vector

$$\mathbf{c} = [c_0 \quad c_1 \quad \cdots \quad c_T]^\top,$$

where

$$\sum_{i=0}^T c_i = W.$$

This vector is called a *policy vector* and describes how much cake is eaten at each time period. The enjoyment of eating a slice of cake is represented by a utility function. For some amount of consumption  $c_i \in [0, W]$ , the utility gained is given by  $u(c_i)$ .

For this lab, we assume the utility function satisfies  $u(0) = 0$ , that  $W = 1$ , and that  $W$  is cut into  $N$  equally-sized pieces so that each  $c_i$  must be of the form  $\frac{i}{N}$  for some integer  $0 \leq i \leq N$ .

## Discount Factors

A person or firm typically has a time preference for saving or consuming. For example, a dollar today can be invested and yield interest, whereas a dollar received next year does not include the accrued interest. Since cake gets stale as it gets older, we assume that cake in the present yields more utility than cake in the future. We can model this by multiplying future utility by a discount factor  $\beta \in (0, 1)$ . For example, if we were to consume  $c_0$  cake at time 0 and  $c_1$  cake at time 1, with  $c_0 = c_1$  then the utility gained at time 0 is larger than the utility at time 1:

$$u(c_0) > \beta u(c_1).$$

The total utility for eating the cake is

$$\sum_{t=0}^T \beta^t u(c_t).$$

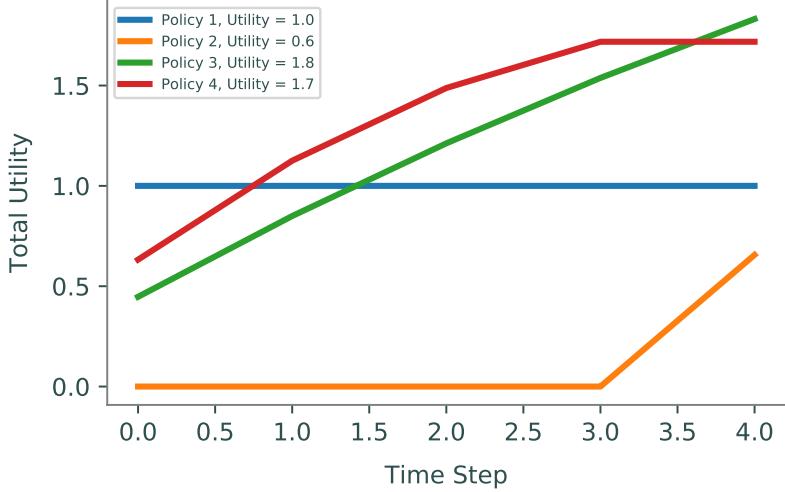


Figure 24.1: Plots for various policies with  $u(x) = \sqrt{x}$  and  $\beta = 0.9$ . Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same amount of cake at each step, while policy 4 begins by eating .4 of the cake, then .3, .2, and .1.

## The Value Function

The cake eating problem is an optimization problem where we maximize utility.

$$\begin{aligned} & \max_{\mathbf{c}} \sum_{t=0}^T \beta^t u(c_t) \\ & \text{subject to } \sum_{t=0}^T c_t = W \\ & \quad c_t \geq 0. \end{aligned} \tag{24.2}$$

One way to solve it is with the value function. The value function  $V(a, b, W)$  gives the utility gained from following an optimal policy from time  $a$  to time  $b$ .

$$\begin{aligned} V(a, b, W) &= \max_{\mathbf{c}} \sum_{t=a}^b \beta^t u(c_t) \\ &\text{subject to } \sum_{t=a}^b c_t = W \\ &\quad c_t \geq 0. \end{aligned}$$

$V(0, T, W)$  gives how much utility we gain in  $T$  days and is the same as Equation 24.2.

Let  $W_t$  represent the total amount of cake left at time  $t$ . Observe that  $W_{t+1} \leq W_t$  for all  $t$ , because our problem does not allow for the creation of more cake. Notice that  $V(t+1, T, W_{t+1})$  can be represented by  $\beta V(t, T-1, W_{t+1})$ , which is the value of eating  $W_{t+1}$  cake later. Then we can express the value function as the sum of the utility of eating  $W_t - W_{t+1}$  cake now and  $W_{t+1}$  cake later.

$$V(t, T, W_t) = \max_{W_{t+1}} (u(W_t - W_{t+1}) + \beta V(t, T-1, W_{t+1})) \quad (24.3)$$

where  $u(W_t - W_{t+1})$  is the value gained from eating  $W_t - W_{t+1}$  cake at time  $t$ .

Let  $\mathbf{w} = [0 \quad \frac{1}{N} \quad \dots \quad \frac{N-1}{N} \quad 1]^T$ . We define the *consumption matrix*  $C$  by  $C_{ij} = u(w_i - w_j)$ . Note that  $C$  is an  $(N+1) \times (N+1)$  lower triangular matrix since we assume  $j \leq i$ ; we can't consume more cake than we have. The consumption matrix will help solve the value function by calculating all possible value of  $u(W_t - W_{t+1})$  at once. At each time  $t$ ,  $W_t$  can only have  $N+1$  values, which will be represented as  $w_i = \frac{i}{N}$ , which is  $i$  pieces of cake remaining. For example, if  $N=4$ , then  $w = [0, .25, .5, .75, 1]^T$ , and  $w_3 = 0.75$  represents having three pieces of cake left. In this case, we get the following consumption matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ u(0.25) & 0 & 0 & 0 & 0 \\ u(0.5) & u(0.25) & 0 & 0 & 0 \\ u(0.75) & u(0.5) & u(0.25) & 0 & 0 \\ u(1) & u(0.75) & u(0.5) & u(0.25) & 0 \end{bmatrix}.$$

**Problem 3.** Write a function that accepts the number of equal sized pieces  $N$  that divides the cake and a utility function  $u(x)$ . Assume  $W = 1$ . Create a partition vector  $\mathbf{w}$  whose entries correspond to possible amounts of cake. Return the consumption matrix.

## Solving the Optimization Problem

Initially we do not know how much cake to eat at  $t=0$ : should we eat one piece of cake ( $w_1$ ), or perhaps all of the cake ( $w_N$ )? It may not be obvious which option is best and that option may change depending on the discount factor  $\beta$ . Instead of asking how much cake to eat at some time  $t$ , we ask how valuable  $w_i$  cake is at time  $t$ . As mentioned above,  $V(t, T-1, W_{t+1})$  in 24.3 is a new value function problem with  $a=t$ ,  $b=T-1$ , and  $W=W_{t+1}$ , making 24.3 a recursion formula. By using the optimal value of the value function in the future,  $V(t, T-1, W_{t+1})$ , we can determine the optimal value for the present,  $V(t, T, W_t)$ .  $V(t, T, W_t)$  can be solved by trying each possible  $W_{t+1}$  and choosing the one that gives the highest utility.

The  $(N+1) \times (T+1)$  matrix  $A$  that solves the value function is called the *value function matrix*.  $A_{ij}$  is the value of having  $w_i$  cake at time  $j$ .  $A_{0j} = 0$  because there is never any value in having  $w_0$  cake, i.e.  $u(w_0) = u(0) = 0$ .

We start at the last time period. Since there is no value in having any cake left over when time runs out, the decision at time  $T$  is obvious: eat the rest of the cake. The amount of utility gained from having  $w_i$  cake at time  $T$  is given by  $u(w_i)$ . So  $A_{iT} = u(w_i)$ . Written in the form of (24.3),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, -1, w_j)) = u(w_i). \quad (24.4)$$

This happens because  $V(0, -1, w_j) = 0$ . As mentioned, there is no value in saving cake so this equation is maximized when  $w_j = 0$ . All possible values of  $w_i$  are calculated so that the value of having  $w_i$  cake at time  $T$  is known.

### ACHTUNG!

Given a time interval from  $t = 0$  to  $t = T$  the utility of waiting until time  $T$  to eat  $w_i$  cake is actually  $\beta^T u(W_i)$ . However, through backwards induction, the problem is solved backwards by beginning with  $t = T$  as an isolated state and calculating its value. This is why the value function above is  $V(0, 0, W_i)$  and not  $V(T, T, W_i)$ .

For example, the following matrix results with  $T = 3$ ,  $N = 4$ , and  $\beta = 0.9$ .

$$\begin{bmatrix} 0 & 0 & 0 & u(0) \\ 0 & 0 & 0 & u(0.25) \\ 0 & 0 & 0 & u(0.5) \\ 0 & 0 & 0 & u(0.75) \\ 0 & 0 & 0 & u(1) \end{bmatrix}.$$

**Problem 4.** Write a function that accepts a stopping time  $T$ , a number of equal sized pieces  $N$  that divides the cake, a discount factor  $\beta$ , and a utility function  $u(x)$ . Return the value function matrix  $A$  for  $t = T$  (the matrix should have zeros everywhere except the last column). Return a matrix of zeros for the policy matrix  $P$ .

Next, we use the fact that  $A_{jT} = V(0, 0, w_j)$  to evaluate the  $T - 1$  column of the value function matrix,  $A_{i(T-1)}$ , by modifying (24.4) as follows,

$$A_{i(T-1)} = V(0, 1, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, 0, w_j)) = \max_{w_j} (u(w_i - w_j) + \beta A_{jT}). \quad (24.5)$$

Remember that there is a limited set of possibilities for  $w_j$ , and we only need to consider options such that  $w_j \leq w_i$ . Instead of doing these one by one for each  $w_i$ , we can compute the options for each  $w_i$  simultaneously by creating a matrix. This information is stored in an  $(N + 1) \times (N + 1)$  matrix known as the *current value matrix*, or  $CV^t$ , where the  $(ij)$ th entry is the value of eating  $w_i - w_j$  pieces of cake at time  $t$  and saving  $j$  pieces of cake until the next period. For  $t = T - 1$ ,

$$CV_{ij}^{T-1} = u(w_i - w_j) + \beta A_{jT}. \quad (24.6)$$

The largest entry in the  $i$ th row of  $CV^{T-1}$  is the optimal value that the value function can attain at  $T - 1$ , given that we start with  $w_i$  cake. The maximal values of each row of  $CV^{T-1}$  become the column of the value function matrix,  $A$ , at time  $T - 1$ .

### ACHTUNG!

The notation  $CV^t$  does not mean raising the matrix to the  $t$ th power; rather, it indicates what time period we are in. All of the  $CV^t$  could be grouped together into a three-dimensional matrix,  $CV$ , that has dimensions  $(N + 1) \times (N + 1) \times (T + 1)$ . Although this is possible, we will not use  $CV$  in this lab, and will instead only consider  $CV^t$  for any given time  $t$ .

The following matrix is  $CV^2$  where  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$ . The maximum value of each row, circled in red, is used in the  $3^{rd}$  column of  $A$ . Remember that  $A$ 's column index begins at 0, so the  $3^{rd}$  column represents  $j = 2$ .

$$CV^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0.45 & 0 & 0 & 0 \\ 0.707 & 0.95 & 0.636 & 0 & 0 \\ 0.866 & 1.157 & 1.136 & 0.779 & 0 \\ 1 & 1.316 & 1.343 & 1.279 & 0.9 \end{bmatrix}$$

Now that the column of  $A$  corresponding to  $t = T - 1$  has been calculated, we repeat the process for  $T - 2$  and so on until we have calculated each column of  $A$ . In summary, at each time step  $t$ , find  $CV^t$  and then set  $A_{it}$  as the maximum value of the  $i$ th row of  $CV^t$ . Generalizing (24.5) and (24.6) shows

$$CV_{ij}^t = u(w_i - w_j) + \beta A_{j(t+1)}. \quad A_{it} = \max_j (CV_{ij}^t). \quad (24.7)$$

The full value function matrix corresponding to the example is below. The maximum value in the value function matrix is the maximum possible utility to be gained.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 0.95 & 0.95 & 0.95 & 0.707 \\ 1.355 & 1.355 & 1.157 & 0.866 \\ 1.7195 & 1.562 & 1.343 & 1 \end{bmatrix}.$$

Figure 24.2: The value function matrix where  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$ . The bottom left entry indicates the highest utility that can be achieved is 1.7195.

**Problem 5.** Complete your function from Problem 4 so it returns the entire value function matrix. Starting from the next to last column, iterate backwards by

- calculating the current value matrix for time  $t$  using (24.7),
- finding the largest value in each row of the current value matrix, and
- filling in the corresponding column of  $A$  with these values.

(Hint: Use `axis` arguments.)

## Solving for the Optimal Policy

With the value function matrix constructed, the optimization problem is solved in some sense. The value function matrix contains the maximum possible utility to be gained. However, it is not immediately apparent what policy should be followed by only inspecting the value function matrix  $A$ . The  $(N + 1) \times (T + 1)$  policy matrix,  $P$ , is used to find the optimal policy. The  $(ij)$ th entry of the policy matrix indicates how much cake to eat at time  $j$  if we have  $i$  pieces of cake. Like  $A$  and  $CV$ ,  $i$  and  $j$  begin at 0.

The last column of  $P$  is calculated similarly to last column of  $A$ .  $P_{iT} = w_i$ , because at time  $T$  we know that the remainder of the cake should be eaten. Recall that the column of  $A$  corresponding to  $t$  was calculated by the maximum values of  $CV^t$ . The column of  $P$  for time  $t$  is calculated by taking  $w_i - w_j$ , where  $j$  is the smallest index corresponding to the maximum value of  $CV^t$ ,

$$P_{it} = w_i - w_j.$$

$$\text{where } j = \{ \min\{j\} \mid CV_{ij}^t \geq CV_{ik}^t \forall k \in [0, 1, \dots, N] \}$$

Recall  $CV^2$  in our example with  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$  above.

$$CV^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0.45 & 0 & 0 & 0 \\ 0.707 & 0.95 & 0.636 & 0 & 0 \\ 0.866 & 1.157 & 1.136 & 0.779 & 0 \\ 1 & 1.316 & 1.343 & 1.279 & 0.9 \end{bmatrix}$$

To calculate  $P_{12}$ , we look at the second row ( $i = 1$ ) in  $CV^2$ . The maximum, .5, occurs at  $CV_{10}^2$ , so  $j = 0$  and  $P_{12} = w_1 - w_0 = .25 - 0 = .25$ . Similarly,  $P_{42} = w_4 - w_2 = 1 - .5 = .5$ . Continuing in this manner,

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.5 \\ 0.25 & 0.25 & 0.5 & 0.75 \\ 0.25 & 0.5 & 0.5 & 1 \end{bmatrix}$$

Given that the rows of  $P$  are the slices of cake available and the columns are the time intervals, we find the policy by starting in the bottom left corner,  $P_{N0}$ , where there are  $N$  slices of cake available and  $t = 0$ . This entry tells us what percentage of the  $N$  slices of cake we should eat. In the example, this entry is 0.25, telling us we should eat 1 slice of cake at  $t = 0$ . Thus, when  $t = 1$  we have  $N - 1$  slices of cake available, since we ate 1 slice of cake. We look at the entry at  $P_{(N-1)1}$ , which has value 0.25. So we eat 1 slice of cake at  $t = 1$ . We continue this pattern to find the optimal policy  $\mathbf{c} = [.25 \quad .25 \quad .25 \quad .25]$ .

### ACHTUNG!

The optimal policy will not always be a straight diagonal in the example above. For example, if the bottom left corner had value .5, then we should eat 2 pieces of cake instead of 1. Then the next entry we should evaluate would be  $P_{(N-2)1}$  in order to determine the optimal policy.

To verify the optimal policy found with  $P$ , we can use the value function matrix  $A$ . By expanding the entries of  $A$ , we can see that the optimal policy does give the maximum value.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} \\ \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.5} \\ \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.25} & \sqrt{0.75} \\ \underline{\sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} + \beta^3\sqrt{0.25}} & \sqrt{0.5} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.5} & \sqrt{1} \end{bmatrix}$$

**Problem 6.** Modify your function from Problem 4 to determine the policy matrix. Initialize the matrix as zeros and fill it in starting from the last column at the same time that you calculate the value function matrix.

(Hint: You may find `np.argmax()` useful.)

### UNIT TEST

There is a file called `test_dynamic_programming.py` that contains a prewritten unit test for Problem 1. There is a spot for you to add your own unit test for your function from Problem 6 to make sure it produces the correct matrices from the example. This will be graded.

**Problem 7.** Write a function `find_policy()` that will find the optimal policy for the stopping time  $T$ , a cake of size 1 split into  $N$  pieces, a discount factor  $\beta$ , and the utility function  $u$ .



# 25 Policy Function Iteration

**Lab Objective:** *Iterative methods can be powerful ways to solve dynamic optimization problems without computing the exact solution. Often we can iterate very quickly to the true solution, or at least within some  $\varepsilon$  error of the solution. These methods are significantly faster than computing the exact solution using dynamic programming. We demonstrate two iterative methods, value iteration and policy iteration, and we use them to solve a deterministic Markov decision process.*

## Dynamic Optimization

Many dynamic optimization problems take the form of a *Markov decision process*. A Markov decision process is similar to that of a Markov chain, but rather than determining state movement using only probabilities, state movement is determined based on probabilities, actions, and rewards. They are formulated as follows.

$\mathbb{T}$  is a set of discrete time periods. In this lab,  $\mathbb{T} = 0, 1, \dots, L$ , where  $L$  is the final time period.  $S$  is the set of possible states. The set of allowable actions for each state  $s$  is  $A_s$ .  $s_{t+1} = g(s_t, a_t)$  is a transition function that determines the state  $s_{t+1}$  at time  $t + 1$  based on the previous state  $s_t$  and action  $a_t$ . The reward  $u(s_t, a_t, s_{t+1})$  is the reward for taking action  $a$  while in state  $s$  at time  $t$  and the next state being state  $s_{t+1}$ .

The time discount factor  $\beta \in [0, 1]$  determines how much the reward function decreases in value with time. Looking at the cake eating problem described in the previous lab, the cake goes stale with each passing day. In other words, each day you do not finish the cake, the reward of eating a piece decreases.  $\beta$  accounts for this decrease in value.

Let  $N_{s,a}$  be the set of all possible next states when taking action  $a$  in state  $s$ .  $p(s_t, a_t, s_{t+1})$  is the probability of taking action  $a$  at time  $t$  while in state  $s$  and arriving at state  $s_{t+1} \in N_{s,a}$ . A deterministic Markov process has  $p(s_t, a_t, s_{t+1}) = 1 \forall s, a$ . This means that  $N_{s,a}$  has one element  $\forall s, a$ . A stochastic Markov process has  $p(s_t, a_t, s_{t+1}) \leq 1$ , which implies that there can be multiple possible next states for taking a given action in a given state. As a result,  $N_{s,a}$  has multiple elements for each  $s, a$ .

The dynamic optimization problem is

$$\max_{\mathbf{a}} \sum_{t=0}^L \beta^t u(s_t, a_t) \quad (25.1)$$

$$\text{subject to } s_{t+1} = g(s_t, a_t) \quad \forall t. \quad (25.2)$$

The cake eating problem follows this format where  $S$  consists of the possible amounts of remaining cake ( $\frac{i}{W}$ ),  $c_t$  is the amount of cake we can eat, and the amount of cake remaining  $s_{t+1} = g(s_t, a_t)$  is  $w_t - c_t$ , where  $w_t$  is the amount of cake we have left and  $c_t$  is the amount of cake we eat at time  $t$ . This is an example of a deterministic Markov process.

For this lab, we define a dictionary  $P$  to represent the decision process. This dictionary contains all of the information about the states, actions, probabilities, and rewards. Each dictionary key is a state-action combination and each dictionary value is a list of tuples.

$$P[s][a] = [(p(s, a, \bar{s}), \bar{s}, u(s, a, \bar{s}), is\_terminal), \dots]$$

Note the slight notation change from  $(s_t, a_t, s_{t+1})$  to  $(s, a, \bar{s})$ . In the dictionary,  $s$  is the current state,  $a$  is the action,  $\bar{s} \in N_{s,a}$  is the next state if action  $a$  is taken, and  $is\_terminal$  indicates if  $\bar{s}$  is a stopping point. In addition,  $p(s, a, \bar{s})$  is the probability of taking action  $a$  while in state  $s$ , and  $u(s, a, \bar{s})$  is the reward for taking action  $a$  while in state  $s$ .

## Moving on a Grid

Now consider an  $N \times N$  grid. Assume that a robot moves around the grid, one space at a time, until it reaches the lower right hand corner and stops. Each square is a state,  $S = \{0, 1, \dots, N^2 - 1\}$ , and the set of actions is  $\{Left, Down, Right, Up\}$ . For this lab,  $Left = 0$ ,  $Down = 1$ ,  $Right = 2$ , and  $Up = 3$ . If you take the action  $a = 1$ , then you move  $Down$  on the grid. Let  $N = 2$  and label the squares as displayed below. In this example, we define the reward to be -1 if the robot moves into state 2, -1 if the robot moves into state 0 from state 1, and 1 when it reaches the end state, state 3. We define the reward function to be  $u(s, a, \bar{s}) = u(\bar{s})$ . Since this is a deterministic model,  $p(s, a, \bar{s}) = p(\bar{s}) = 1$  for all possible  $s, a$ .

0	1
2	3

$A_s$  is the set of actions that keep the robot on the grid. If the robot is in the top left hand corner, the only allowed actions are  $Down$  and  $Right$  so  $A_0 = \{1, 2\}$ . The transition function  $g(s, a) = \bar{s}$  can be explicitly defined for each  $s, a$  where  $\bar{s}$  is the new state after moving.

All of this information is encapsulated in  $P$ . We define  $P[s][a]$  for all states and actions, even if they are not possible. This simplifies coding the algorithm but is not necessary.

```

P[0][0] = [(0, 0, 0, False)]    P[2][0] = [(0, 2, -1, False)]
P[0][1] = [(1, 2, -1, False)]    P[2][1] = [(0, 2, -1, False)]
P[0][2] = [(1, 1, 0, False)]     P[2][2] = [(1, 3, 1, True)]
P[0][3] = [(0, 0, 0, False)]     P[2][3] = [(1, 0, 0, False)]
P[1][0] = [(1, 0, -1, False)]    P[3][0] = [(0, 0, 0, True)]
P[1][1] = [(1, 3, 1, True)]      P[3][1] = [(0, 0, 0, True)]
P[1][2] = [(0, 0, 0, False)]     P[3][2] = [(0, 0, 0, True)]
P[1][3] = [(0, 0, 0, False)]     P[3][3] = [(0, 0, 1, True)]

```

For the sake of clarity, we will do a quick example using the above dictionary. We first assume that we start in state 0 corresponding to the 0 in the above grid. Next, we move *Down* the grid to state 2. This corresponds to taking action 1. To get the correct values from the dictionary, we look at  $P[s][a]$  or in this case  $P[0][1] = [(1, 2, -1, False)]$ . So, when we move *Down* from state 0 to state 2,  $p(\bar{s}) = 1$ ,  $u(\bar{s}) = -1$ , and  $\bar{s} = 2$ . As a final note, when the action is not possible  $p(\bar{s}) = 0$ , as shown in the dictionary above.

We define the *value function*  $V(s)$  to be the maximum possible reward of starting in state  $s$ . Then using Bellman's optimality equation,

$$V(s) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V(\bar{s})) \right\}. \quad (25.3)$$

The summation occurs when it is a stochastic Markov process. For example, if the robot is in the top left corner and we want it to move right, we could have the probability of the robot actually moving right as 0.5. In this case,  $P[0][2] = [(0.5, 1, 0, False), (0.5, 2, -1, False)]$ . This type of process will occur later in the lab.

## Value Iteration

In the previous lab, we used dynamic programming to solve for the value function. This was a recursive method where we calculated all possible values for each state and time period. *Value iteration* is another algorithm that solves the value function by taking an initial value function and calculating a new value function iteratively. Since we are not calculating all possible values, it is typically faster than dynamic programming.

### Convergence of Value Iteration

A function  $f$  that is a contraction mapping has a *fixed point*  $p$  such that  $f(p) = p$ . Blackwell's contraction theorem can be used to show that Bellman's equation is a “fixed point” (it actually acts more like a fixed function in this case) for an operator  $T : L^\infty(X; \mathbb{R}) \rightarrow L^\infty(X; \mathbb{R})$  where  $L^\infty(X; \mathbb{R})$  is the set of all bounded functions:

$$T[f](s) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta f(\bar{s})) \right\} \quad (25.4)$$

It can be shown that Equation 25.1 is the fixed “point” of our operator  $T$ . A result of contraction mappings is that there exists a unique solution to Equation 25.4, namely

$$V_{k+1}(s_i) = T[V_k](s_i) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V_k(\bar{s})) \right\} \quad (25.5)$$

where an initial guess for  $V_0(s)$  is used. As  $k \rightarrow \infty$ , it is guaranteed that  $(V_k(s)) \rightarrow V^*(s)$ . Because of the contraction mapping, if  $V_{k+1}(s) = V_k(s) \forall s$ , we have found the true value function,  $V^*(s)$ .

As an example, let  $V_0 = [0, 0, 0, 0]$  and  $\beta = 1$ , where each entry of  $V_0$  represents the maximum value at that state, and  $V_0(s) = V_0[s]$  if we are using the array or list form of the *value function*. We calculate  $V_1(s)$  from the robot example above. For  $V_1(0)$ , we choose the `max` of the possible outcomes, states 1 or 2, after moving. Thus we use  $P[0][2]$  for state 1 because moving from state 0 to state 1 requires going right, action 2.

$$\begin{aligned}
V_1(0) &= \max_{a \in A_0} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + V_0(\bar{s})) \right\} \\
&= \max\{p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2))\} \\
&= \max\{1(0 + 0), 1(-1 + 0)\} \\
&= \max\{0, -1\} \\
&= 0 \\
V_1(1) &= \max\{p(0) * (u(0) + V_0(0)), p(3) * (u(3) + V_0(3))\} \\
&= \max\{1(-1 + 0), 1(1 + 0)\} \\
&= 1 \\
V_1(2) &= \max\{p(0) * (u(0) + V_0(0)), p(3) * (u(3) + V_0(3))\} \\
&= \max\{1(0 + 0), 1(1 + 0)\} \\
&= 1 \\
V_1(3) &= \max\{p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2))\} \\
&= \max\{1(0 + 0), 1(0 + 0)\} \\
&= 0
\end{aligned}$$

This calculation gives  $V_1 = [0, 1, 1, 0]$ . Repeating the process yields  $V_2 = [1, 1, 1, 0]$ . Repeating a third time gives  $V_3 = [1, 1, 1, 0]$ , which is the same as  $V_2$ , so the process has converged. This means that the solution is  $[1, 1, 1, 0]$ . Thus, the total maximum reward the robot can achieve by starting on square  $i$  is the  $i$ th entry of the solution  $[1, 1, 1, 0]$ .

When implementing functions in this lab, instead of only looking at possible actions  $a \in A_s$ , we can consider all of the actions. This will not affect the results, because  $p(\bar{s}) = 0$  when an action is not possible. This simplifies the coding significantly. For example, when calculating  $V_{k+1}(s_i)$  consider the following lines of code.

```

sa_vector = np.zeros(nA)
for a in range(nA):
    for tuple_info in P[s][a]:
        # tuple_info is a tuple of (probability, next state, reward, done)
        p, s_, u, _ = tuple_info
        # sums up the possible end states and rewards with given action
        sa_vector[a] += (p * (u + beta * V_old[s_]))
#add the max value to the value function
V_new[s] = np.max(sa_vector)

```

**Problem 1.** Write a function called `value_iteration()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, a discount factor  $\beta \in (0, 1]$  defaulting to 1, the tolerance amount  $\varepsilon$  defaulting to `1e-8`, and the maximum number of iterations `maxiter` defaulting to 3,000. Perform value iteration until  $\|V_{k+1} - V_k\| < \varepsilon$  or  $k > \text{maxiter}$ . Return the final vector representing  $V^*$  and the number of iterations. Test your code on the example given above.

## Calculating the Policy

While knowing the maximum expected value is helpful, it is usually more important to know the policy that generates the most value. Value iteration tells the robot what reward it can expect, but not how to get it. The policy vector is found by using the policy function:  $\pi : \mathbb{R} \rightarrow \mathbb{R}$ .  $\pi(s)$  is the action that should be taken while in state  $s$  to maximize the reward. We can modify the Bellman equation using  $V^*(s)$ , which is the true value function we found in Problem 1, to find  $\pi$ :

$$\pi(s) = \operatorname{argmax}_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V^*(\bar{s})) \right\} \quad (25.6)$$

Using value iteration, we found  $V^* = [1, 1, 1, 0]$  in the example above. We find  $\pi(0)$  from the example above with  $\beta = 1$  by looking at actions 1 and 2 (since actions 0 and 3 have probability 0).

$$\begin{aligned} \pi(0) &= \operatorname{argmax}_{1,2} \{p(2) * (u(2) + V^*(2)), p(1) * (u(1) + V^*(1))\} \\ &= \operatorname{argmax}\{1 * (-1 + 1), 1 * (0 + 1)\} \\ &= \operatorname{argmax}\{0, 1\} \\ &= 2 \end{aligned}$$

So when the robot is in state 0, it should take action 2, moving *Right*. This avoids the -1 penalty for moving *Down* into state 2. Similarly,

$$\begin{aligned} \pi(1) &= \operatorname{argmax}_{0,1} \{1 * (-1 + 1), 1 * (1 + 0)\} \\ &= \operatorname{argmax}\{0, 1\} = 1 \\ \pi(2) &= \operatorname{argmax}_{2,3} \{1 * (1 + 0), 1 * (0 + 1)\} \\ &= \operatorname{argmax}\{1, 1\} = 2 \end{aligned}$$

Since state 3 is terminal, it doesn't matter what  $\pi(3)$  is, but we'll set it to 0 for convenience. Thus, the policy corresponding to the optimal reward is  $[2, 1, 2, 0]$ . The robot should move to state 3 if possible, avoiding state 2 because it has a negative reward.

### NOTE

Note that  $\pi$  gives the optimal action  $a$  to take at each state  $s$ . It does not give a sequence of actions to take in order to maximize the policy.

**Problem 2.** Write a function called `extract_policy()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, an array representing the value function, and a discount factor  $\beta \in (0, 1]$  defaulting as before. Return the policy vector corresponding to  $V^*$ . Test your code on the example with  $\beta = 1$ .

## Policy Iteration

For dynamic programming problems, it can be shown that value function iteration converges relative to the discount factor  $\beta$ . As  $\beta \rightarrow 1$ , the number of iterations increases dramatically. As mentioned earlier  $\beta$  is usually close to 1, which means this algorithm can converge slowly. In value iteration, we used an initial guess for the value function,  $V_0$  and used Equation 25.1 to iterate towards the true value function. Once we achieved a good enough approximation for  $V^*$ , we recovered the true policy function  $\pi^*$ . Instead of iterating on our value function, we can instead make an initial guess for the policy function,  $\pi_0$ , and use this to iterate toward the true policy function. We do so by taking advantage of the definition of the value function, where we assume that our policy function yields the most optimal result. This is called *policy iteration*.

That is, given a specific policy function  $\pi_k$ , we can modify Equation 25.1 by assuming that the policy function is the optimal choice. This process, called *policy evaluation*, evaluates the value function for a given policy.

$$V_{k+1}(s) = \max_{a \in [A_s]} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})) \right\} = \sum_{\bar{s} \in N_{s,\pi(s)}} p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})) \quad (25.7)$$

The last equality occurs because in state  $s$ , the robot should choose the action that maximizes the reward, which is  $\pi(s)$  by definition. Like value iteration, policy iteration iterates until the desired tolerance is reached. This ensures that we get the value function that optimizes the reward for each starting state.

**Problem 3.** Write a function called `compute_policy_v()` that accepts a dictionary  $P$  representing the decision process, the number of states, the number of actions, an array representing a policy, a discount factor  $\beta \in (0, 1]$ , and a tolerance amount  $\epsilon$ , defaulting as before. Use the policy evaluation process above to return the value function corresponding to the policy.

Test your code on the policy vector generated from `extract_policy()` for the example. The result should be the same value function array from `value_iteration()`.

Now that we have the value function for our policy, we can take the value function and find a better policy. This is called *policy improvement*. This step is the same method used in value iteration to find the policy. In other words, this step uses the `extract_policy()` method from Problem 2 with the newly computed value function.

*Policy function iteration* starts with an intial  $\pi_0$  and iterates using policy evalutaion and policy improvement successively until the desired tolerance is reached. The algorithm for policy function iteration, using two of the functions that you previously implemented, can be summarized as follows:

**Algorithm 1** Policy Iteration

---

```

1: procedure POLICY ITERATION FUNCTION( $P, nS, nA, \beta, tol, \text{maxiter}$ )
2:    $\pi_0 \leftarrow [\pi_0(s_0), \pi_0(s_1), \dots, \pi_0(s_N)]$                                  $\triangleright$  Initialize  $\pi$  as array of ones of length  $nS$ 
3:   for  $k = 0, 1, \dots, \text{maxiter}$  do                                          $\triangleright$  Iterate only  $\text{maxiter}$  times at most
4:      $V_{k+1} = \text{compute\_policy\_v}(\pi_k)$                                       $\triangleright$  Policy evaluation using compute_policy_v()
5:      $\pi_{k+1} = \text{extract\_policy}(V_{k+1})$                                       $\triangleright$  Policy improvement using extract_policy()
6:     if  $\|\pi_{k+1} - \pi_k\| < \varepsilon$  then                                          $\triangleright$  Stop iterating if the policy doesn't change enough
7:       break
8:   return  $V_{k+1}, \pi_{k+1}$ 

```

---

**Problem 4.** Write a function called `policy_iteration()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, a discount factor  $\beta \in (0, 1]$ , the tolerance amount  $\varepsilon$ , defaulting as before, and the maximum number of iterations `maxiter` defaulting to 200. Perform policy iteration until  $\|\pi_{k+1} - \pi_k\| < \varepsilon$  or  $k > \text{maxiter}$ . Return the final vector representing  $V_k$ , the optimal policy  $\pi_k$ , and the number of iterations. Test your code on the example given above and compare your answers to the results from Problems 1 and 2.  
(Hint: This is just the Policy Iteration algorithm, except you also return the number of iterations)

## The Frozen Lake Problem

For the rest of this lab, we will be using the Gymnasium environment '[FrozenLake-v1](#)'. Gymnasium can be installed using the following code.

```

>>> pip install gymnasium
>>> # You may also need to install these dependencies
>>> pip install gymnasium[all]
>>> pip install gymnasium[classic-control]

```

In the Frozen Lake problem, an elf attempts to cross a treacherous frozen lake to obtain a present. The lake is divided into an  $N \times N$  grid where the top left corner is the start, the bottom right corner is the end, and the other squares are either frozen or holes. To retrieve the present, the elf must successfully navigate around the melted ice without falling through. The possible actions are left, right, up, and down, but since the ice is slippery, the elf won't always move in the intended direction. Hence, this is a stochastic Markov process, i.e.  $p(s_t, a_t, s_{t+1}) < 1$ . The reward for falling is 0, and the reward for obtaining the present is 1. There are two scenarios with  $N = 4$  and  $N = 8$ .

## Using Gymnasium

The '[FrozenLake-v1](#)' environment has 3 important attributes, `P`, `observation_space.n`, and `action_space.n`. We can calculate the optimal policy of '[FrozenLake-v1](#)' with value iteration or policy iteration using these 3 attributes. Since the ice is slippery, this policy will not always result in a reward of 1.

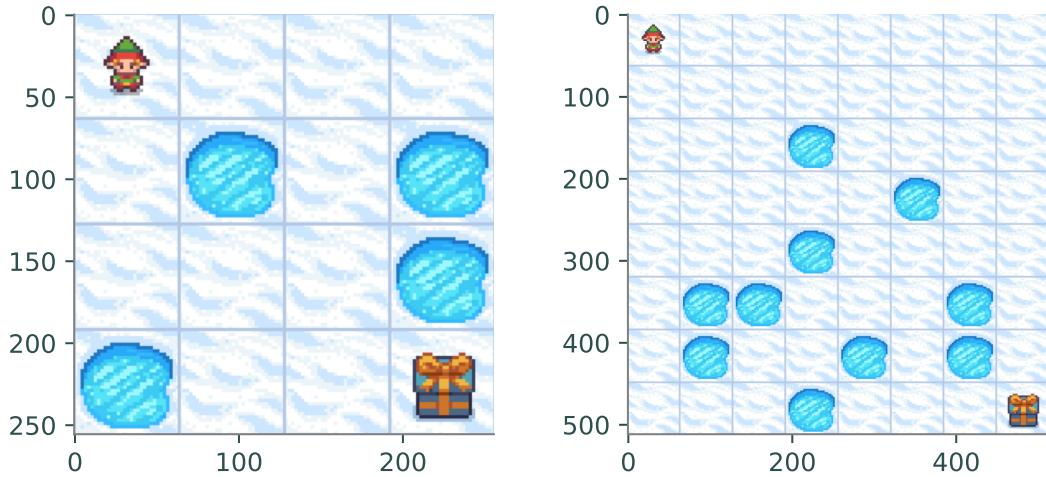


Figure 25.1: Default starting positions of the  $4 \times 4$  and  $8 \times 8$  versions of "FrozenLake-v1"

```
>>> import gymnasium as gym

>>> # Initialize environment for 4x4 scenario
>>> env = gym.make('FrozenLake-v1', desc=None, map_name='4x4', is_slippery=True)
>>> # Find number of states and actions
>>> env.obersevation_space.n
16
>>> env.action_space.n
4
>>> # Get the dictionary with all the states and actions
>>> dictionary_P = env.P

>>> env.close() # Always close the environment!
```

The attribute  $P$  is similar to the dictionary we used in the previous problems. As already mentioned,  $p(s_t, a_t, s_{t+1}) < 1$ , which means the set  $N_{s,a}$  has more than one value. If you did not implement the functions in this lab to account for this, they will not work as intended on this dictionary, which we will use for the remainder of this lab.

**Problem 5.** Write a function `frozen_lake()` that runs `value_iteration()`, `extract_policy()`, and `policy_iteration()` on FrozenLake. It should accept a boolean `basic_case` defaulting to `True`, an integer  $M$  defaulting to 1000 that indicates how many times to run the simulation, and a boolean `render` defaulting to `False`. If `basic_case` is `True`, run the  $4 \times 4$  scenario. If not, run the  $8 \times 8$  scenario. If `render` is `True`, render the environment by applying the argument `render_mode='human'` when initializing the environment. Close the environment at the end of the function. (Hint: Rendering this environment can take a long time, so only render it with small values of  $M$ .)

Calculate and return the policy generated by value iteration, the mean total rewards of value iteration (set to 0 for now), the policy iteration value function, the policy generated by policy iteration, and the mean total rewards of policy iteration (also set to 0 for now).

Gymnasium environments have built-in functions that allow us to simulate each step of the scenario. Before running a simulation in Gymnasium, always revert it to the starting position by calling the `reset()` function. The function `step()` moves the simulation to the next state.

```
>>> import gymnasium as gym
>>> # Initialize environment for 4x4 scenario
>>> env = gym.make('FrozenLake-v1', desc=None, map_name='4x4', is_slippery=True)

>>> # Put environment in starting state
>>> observation, info = env.reset()
>>> # Take a step in the optimal direction and update variables
>>> observation, reward, done, trunc, info = env.step(int(policy[observation]))

>>> env.close() # Always close the environment!
```

The function `step()` takes integers representing different actions and returns: `observation`, `reward`, `done`, `truncated`, and `info`. When we take an action, we get a new `observation`, or state, as well as the `reward` for taking that action. If the elf falls into a hole or reaches the present, the simulation terminates (`done=True`). The `truncated` and `info` values will not be used in this lab. For more information about this environment, visit [gymnasium.farama.org/environments/toy\\_text/frozen\\_lake/](https://gymnasium.farama.org/environments/toy_text/frozen_lake/).

**Problem 6.** Write a function `run_simulation()` that takes in an environment `env`, a policy `policy`, and a discount factor  $\beta$ . Calculate the total reward of the policy for one simulation of the environment (step through the environment until `done=True`). This function will be called by `frozen_lake()`, which both initializes and closes the environment, so do not call `close()` in this function. However, you should call `reset()` at the beginning of this function, to revert the environment back to its starting position.

(Hint: When calculating the reward, use  $\beta^k$  as shown in Equation 25.1.)

Next, modify `frozen_lake()` to call `run_simulation()` for both the value iteration policy and the policy iteration policy  $M$  times. Then, modify `frozen_lake()` to return the actual values of the mean total reward for both policies.

(Hint: Even though you run the simulation  $M$  times, you should only calculate the policies once, because each policy depends on the dictionary  $P$ , which does not change.)



Part II  
Appendices



# A

# NumPy Visual Guide

**Lab Objective:** NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to  $n$ -dimensional arrays.

## Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as “the  $a$ th entry up to (but not including) the  $b$ th entry.” Similarly, `[a:]` means “the  $a$ th entry to the end” and `[:b]` means “everything up to (but not including) the  $b$ th entry.”

$$A[1] = A[1,:] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[:,2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[1:,:2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[1:-1,1:-1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

$$B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\text{np.hstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\text{np.vstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = [ \times \quad \times \quad \times \quad \times ]$$

$$y = [ * \quad * \quad * \quad * ]$$

$$\text{np.hstack}((x, y, x)) = [ \times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times ]$$

$$\text{np.vstack}((x, y, x)) = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix}$$

$$\text{np.column_stack}((x, y, x)) = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

## Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> for more in-depth examples and broadcasting rules.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad x = [10 \quad 20 \quad 30]$$

$$A + x = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ + \\ 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$A + x.reshape((1, -1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

## Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

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

$$A.sum(axis=0) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [4 \quad 8 \quad 12 \quad 16]$$

$$A.sum(axis=1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [10 \quad 10 \quad 10 \quad 10]$$



# B

# Matplotlib Syntax and Customization Guide

**Lab Objective:** *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at <https://matplotlib.org/>.*

## Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.
2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
<code>add_subplot()</code>	Add a single subplot to the current figure
<code>axes()</code>	Add an axes to the current figure
<code>clf()</code>	Clear the current figure
<code>figure()</code>	Create a new figure or grab an existing figure
<code>gca()</code>	Get the current axes
<code>gcf()</code>	Get the current figure
<code>subplot()</code>	Add a single subplot to the current figure
<code>subplots()</code>	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

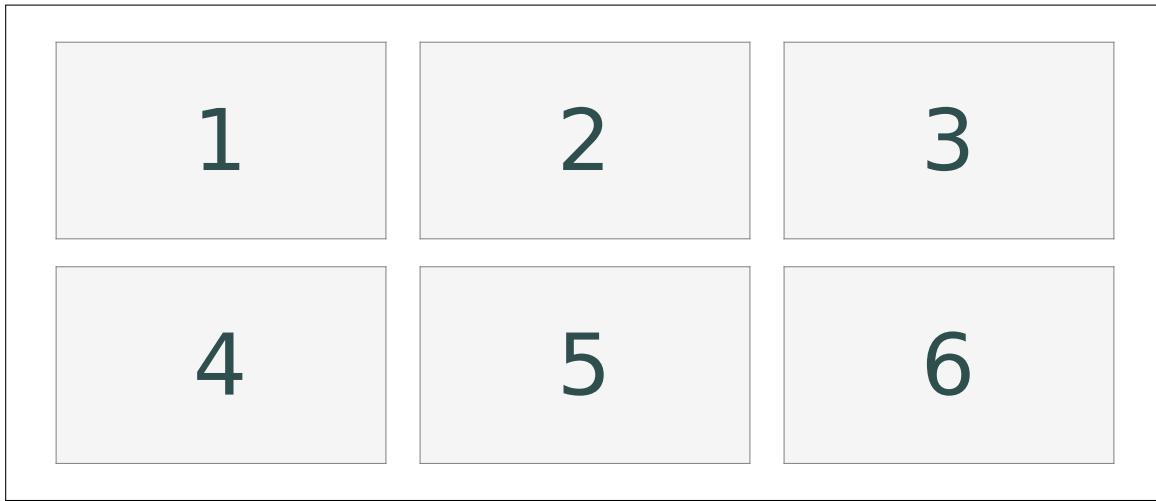


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current axes.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

## ACHTUNG!

Be careful not to mix up the following similarly-named functions:

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` or `ax.axis()` sets properties of the *x*- and *y*-axis in the current axes, such as the *x* and *y* limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

# Plot Customization

## Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at

[https://matplotlib.org/stable/gallery/style\\_sheets/style\\_sheets\\_reference.html](https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html).

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    #
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

## Plot layout

### Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the *x* and *y* ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin`, `xmax`, `ymin`, `ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the *x*- and *y*-scales equal (i.e. makes circles circular).

Function	Description
<code>axis()</code>	set the $x$ - and $y$ -limits of the plot
<code>grid()</code>	add gridlines
<code>xlim()</code>	set the limits of the $x$ -axis
<code>ylim()</code>	set the limits of the $y$ -axis
<code>xticks()</code>	set the location of the tick marks on the $x$ -axis
<code>yticks()</code>	set the location of the tick marks on the $y$ -axis
<code>xscale()</code>	set the scale type to use on the $x$ -axis
<code>yscale()</code>	set the scale type to use on the $y$ -axis
<code>ax.spines[side].set_position()</code>	set the location of the given spine
<code>ax.spines[side].set_color()</code>	set the color of the given spine
<code>ax.spines[side].set_visible()</code>	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is '`top`', '`bottom`', '`left`', or '`right`'. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments '`center`' and '`zero`', which place the spine in the center of the subplot or at an  $x$ - or  $y$ -coordinate of zero, respectively. The others are passed as a tuple `(position_type, amount)`:

- '`data`': place the spine at an  $x$ - or  $y$ -coordinate equal to `amount`.
- '`axes`': place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- '`outward`': places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of  $\sin(x)$ . The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```

>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()

```

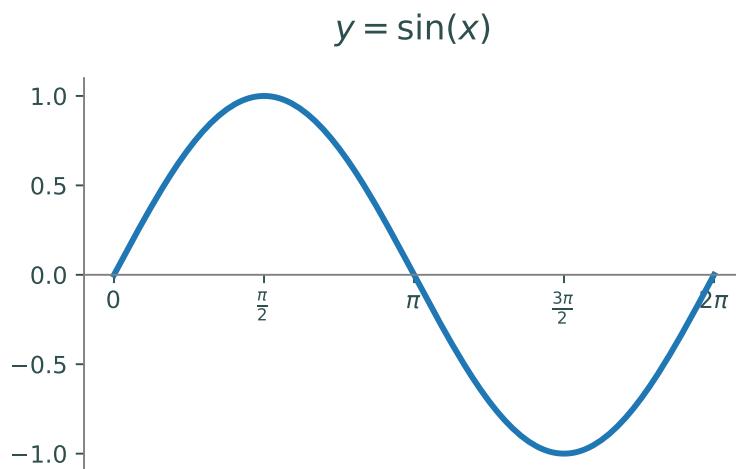


Figure B.2: Plot of  $y = \sin(x)$  with axes modified for clarity

### Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`. The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements. The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

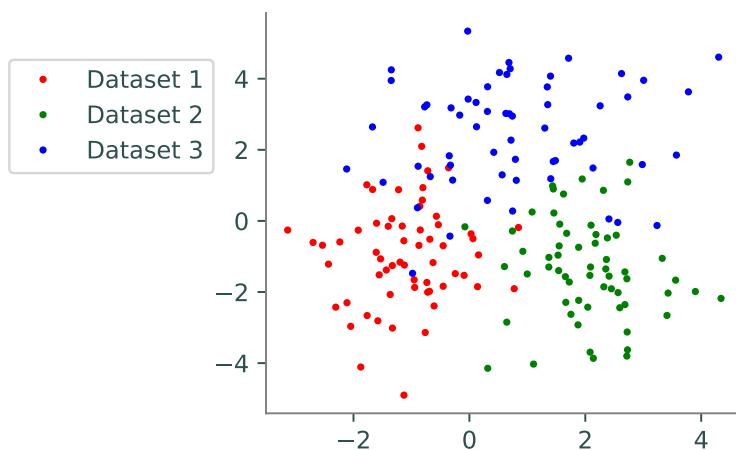


Figure B.3: Example of repositioning axes.

## Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specify colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"#0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at [https://matplotlib.org/stable/gallery/color/named\\_colors.html](https://matplotlib.org/stable/gallery/color/named_colors.html). If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color	Code	Color
'b'	blue	'y'	yellow
'g'	green	'k'	black
'r'	red	'w'	white
'c'	cyan	'CO' - 'C9'	Default colors
'm'	magenta		

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of 1.0 corresponds to fully opaque, and 0.0 corresponds to fully transparent. The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

## Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at [https://matplotlib.org/stable/gallery/color/colormap\\_reference.html](https://matplotlib.org/stable/gallery/color/colormap_reference.html). Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
<code>annotate()</code>	adds a commentary at a given point on the plot	<code>annotate('text',(x,y))</code>
<code>arrow()</code>	draws an arrow from a given point on the plot	<code>arrow(x,y,dx,dy)</code>
<code>colorbar()</code>	Create a colorbar	<code>colorbar()</code>
<code>legend()</code>	Place a legend in the plot	<code>legend(loc='best')</code>
<code>text()</code>	Add text at a given position on the plot	<code>text(x,y,'text')</code>
<code>title()</code>	Add a title to the plot	<code>title('text')</code>
<code>suptitle()</code>	Add a title to the figure	<code>suptitle('text')</code>
<code>xlabel()</code>	Add a label to the $x$ -axis	<code>xlabel('text')</code>
<code>ylabel()</code>	Add a label to the $y$ -axis	<code>ylabel('text')</code>

Table B.4: Text and annotation functions in Matplotlib

## Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with L<sup>A</sup>T<sub>E</sub>X, a system for creating technical documents.<sup>1</sup> To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be  $\frac{1}{2} \sin(x^2)$ :

```
>>> plt.title(r"\frac{1}{2}\sin(x^2)")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to '`best`', which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are '`upper right`', '`upper left`', '`lower left`', '`lower right`', '`center left`', '`center right`', '`lower center`', '`upper center`', and '`center`'. Alternately, a tuple of ( $x, y$ ) can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point  $(0,0)$  corresponds to the bottom-left of the current subplot, and  $(1,1)$  corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

---

<sup>1</sup>See <http://www.latex-project.org/> for more information.

```
# Create the legend
>>> plt.legend()
```

## Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker		vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Matplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:  
>>> plt.plot(x, y, 'r.')  
  
#To change the size:  
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)  
>>> plt.scatter(x, y, marker='+', s=12)
```

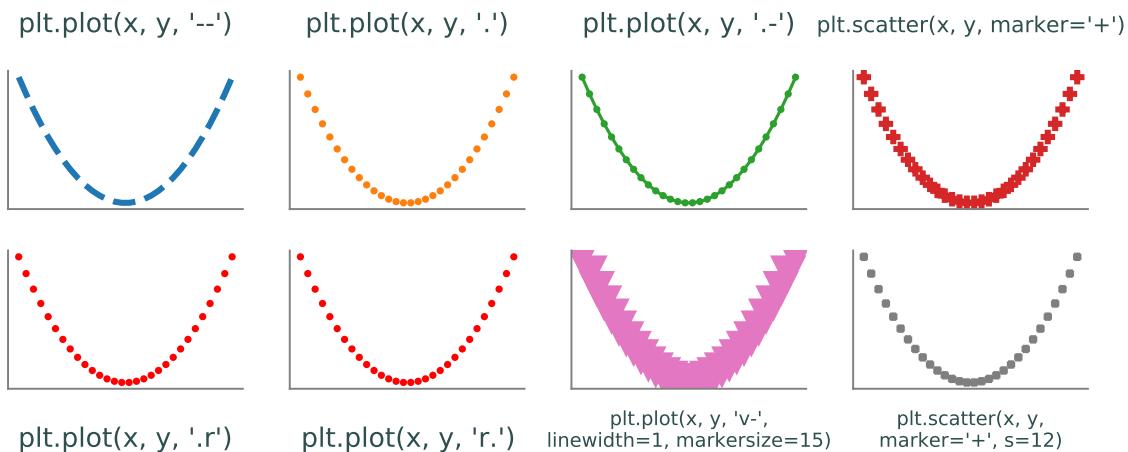


Figure B.4: Examples of setting line and marker styles.

## Plot Types

Matplotlib has functions for creating many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
<code>bar</code>	makes a bar graph	<code>bar(x,height)</code>
<code>barh</code>	makes a horizontal bar graph	<code>barh(y,width)</code>
<code>boxplots</code>	makes one or more boxplots	<code>boxplots(data)</code>
<code>contour</code>	makes a contour plot	<code>contour(X,Y,Z)</code>
<code>contourf</code>	makes a filled contour plot	<code>contourf(X,Y,Z)</code>
<code>imshow</code>	shows an image	<code>imshow(image)</code>
<code>fill</code>	plots lines with shading under the curve	<code>fill(x,y)</code>
<code>fill_between</code>	plots lines with shading between two given y values	<code>fill_between(x,y1, y2=0)</code>
<code>hexbin</code>	creates a hexbin plot	<code>hexbin(x,y)</code>
<code>hist</code>	plots a histogram from data	<code>hist(data)</code>
<code>pcolormesh</code>	makes a heatmap	<code>pcolormesh(X,Y,Z)</code>
<code>pie</code>	makes a pie chart	<code>pie(x)</code>
<code>plot</code>	plots lines and data on standard axes	<code>plot(x,y)</code>
<code>plot_surface</code>	plot a surface in 3-D space	<code>plot_surface(X,Y,Z)</code>
<code>polar</code>	plots lines and data on polar axes	<code>polar(theta,r)</code>
<code>loglog</code>	plots lines and data on logarithmic x and y axes	<code>loglog(x,y)</code>
<code>scatter</code>	plots data in a scatterplot	<code>scatter(x,y)</code>
<code>semilogx</code>	plots lines and data with a log scaled x axis	<code>semilogx(x,y)</code>
<code>semilogy</code>	plots lines and data with a log scaled y axis	<code>semilogy(x,y)</code>
<code>specgram</code>	makes a spectrogram from data	<code>specgram(x)</code>
<code>spy</code>	plots the sparsity pattern of a 2D array	<code>spy(Z)</code>
<code>triplot</code>	plots triangulation between given points	<code>triplot(x,y)</code>

Table B.6: Some basic plotting functions in Matplotlib.

## Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

## Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barch()`, but with argument names `y`, `width`, `height` and `align`.

## Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

## Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

## Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of  $z = (x^2 + y) \sin(y)$ . The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

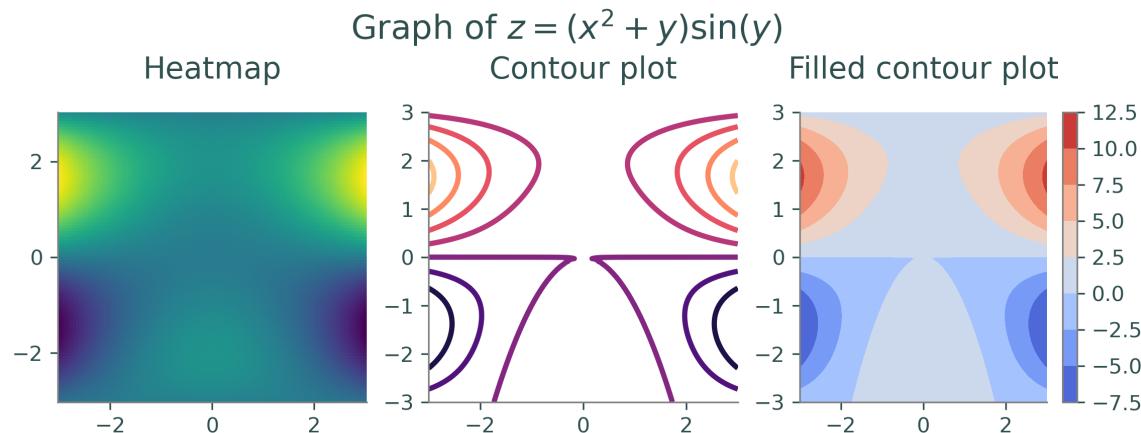


Figure B.5: Example of heatmaps and contour plots.

## Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D  $n \times m$  array for a grayscale image, or a 3-D  $n \times m \times 3$  array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range [0, 1]. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

## 3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.mplot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")
```

```
#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

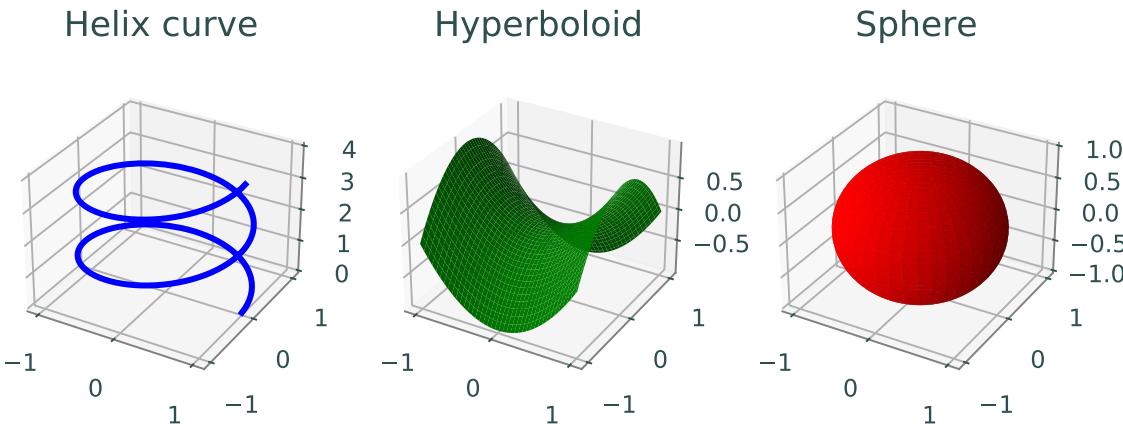


Figure B.6: Examples of 3-D plotting.

## Additional Resources

### rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the "`figure.dpi`" parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at [https://matplotlib.org/stable/api/matplotlib\\_configuration\\_api.html#matplotlib.RcParams](https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams).

### Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

## Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at <https://matplotlib.org/stable/tutorials/index.html>. It also has a large gallery of examples, found at <https://matplotlib.org/stable/gallery/index.html>. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

# Bibliography

- [ADH<sup>+</sup>01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.
- [BL04] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.
- [Gei60] Theodor Seuss Geisel. *Green eggs and ham*. Beginner Books, 1960.
- [Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
- [jup] Jupyter notebooks—a publishing format for reproducible computational workflows. pages 87–90.
- [KM72] Victor Klee and George J. Minty. How good is the simplex algorithm? In *Inequalities*, volume 3, pages 159–175. Academic Press, 1972.
- [Nas00] J.C. Nash. The (dantzig) simplex method for linear programming. *Computing in Science and Engineering*, 2(1):29–31, 2000.
- [Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.
- [Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.
- [PG07] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, may 2007.
- [VD10] Guido VanRossum and Fred L Drake. *The python language reference*. Python software foundation Amsterdam, Netherlands, 2010.
- [VHL06] Philipp Von Hilgers and Amy N Langville. The five greatest applications of markov chains. In *Proceedings of the Markov Anniversary Meeting, Boston Press, Boston, MA*. Citeseer, 2006.