

# Labs for Foundations of Applied Mathematics

## Volume 2 Algorithm Design and Optimization

Jeffrey Humpherys & Tyler J. Jarvis, managing editors





# List of Contributors

E. Evans  
*Brigham Young University*  
R. Evans  
*Brigham Young University*  
J. Grout  
*Drake University*  
J. Humpherys  
*Brigham Young University*  
T. Jarvis  
*Brigham Young University*  
J. Whitehead  
*Brigham Young University*  
J. Adams  
*Brigham Young University*  
J. Bejarano  
*Brigham Young University*  
Z. Boyd  
*Brigham Young University*  
M. Brown  
*Brigham Young University*  
A. Carr  
*Brigham Young University*  
C. Carter  
*Brigham Young University*  
T. Christensen  
*Brigham Young University*  
M. Cook  
*Brigham Young University*  
R. Dorff  
*Brigham Young University*  
B. Ehlert  
*Brigham Young University*  
M. Fabiano  
*Brigham Young University*

K. Finlinson  
*Brigham Young University*  
J. Fisher  
*Brigham Young University*  
R. Flores  
*Brigham Young University*  
R. Fowers  
*Brigham Young University*  
A. Frandsen  
*Brigham Young University*  
R. Fuhriman  
*Brigham Young University*  
S. Giddens  
*Brigham Young University*  
C. Gigena  
*Brigham Young University*  
M. Graham  
*Brigham Young University*  
F. Glines  
*Brigham Young University*  
C. Glover  
*Brigham Young University*  
M. Goodwin  
*Brigham Young University*  
R. Grout  
*Brigham Young University*  
D. Grundvig  
*Brigham Young University*  
E. Hannesson  
*Brigham Young University*  
J. Hendricks  
*Brigham Young University*  
A. Henriksen  
*Brigham Young University*

I. Henriksen  
*Brigham Young University*

C. Hettinger  
*Brigham Young University*

S. Horst  
*Brigham Young University*

K. Jacobson  
*Brigham Young University*

J. Leete  
*Brigham Young University*

J. Lytle  
*Brigham Young University*

R. McMurray  
*Brigham Young University*

S. McQuarrie  
*Brigham Young University*

D. Miller  
*Brigham Young University*

J. Morrise  
*Brigham Young University*

M. Morrise  
*Brigham Young University*

A. Morrow  
*Brigham Young University*

R. Murray  
*Brigham Young University*

J. Nelson  
*Brigham Young University*

E. Parkinson  
*Brigham Young University*

M. Probst  
*Brigham Young University*

M. Proudfoot  
*Brigham Young University*

D. Reber  
*Brigham Young University*

H. Ringer  
*Brigham Young University*

C. Robertson  
*Brigham Young University*

M. Russell  
*Brigham Young University*

R. Sandberg  
*Brigham Young University*

C. Sawyer  
*Brigham Young University*

M. Stauffer  
*Brigham Young University*

J. Stewart  
*Brigham Young University*

S. Suggs  
*Brigham Young University*

A. Tate  
*Brigham Young University*

T. Thompson  
*Brigham Young University*

M. Victors  
*Brigham Young University*

J. Webb  
*Brigham Young University*

R. Webb  
*Brigham Young University*

J. West  
*Brigham Young University*

A. Zaitzeff  
*Brigham Young University*

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

# 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

<b>Preface</b>	<b>iii</b>
<b>I Labs</b>	<b>1</b>
1 Linked Lists	3
2 Binary Search Trees	13
3 Nearest Neighbor Search	29
4 Breadth-first Search	43
5 Markov Chains	55
6 The Discrete Fourier Transform	67
7 Convolution and Filtering	77
8 Introduction to Wavelets	85
9 Polynomial Interpolation	103
10 Gaussian Quadrature	115
11 One-dimensional Optimization	121
12 Gradient Descent Methods	129
13 The Simplex Method	139
14 OpenGym AI	147
15 CVXOPT	153
16 Interior Point 1: Linear Programs	163
17 Interior Point 2: Quadratic Programs	173

<b>18</b>	<b>Dynamic Programming</b>	<b>183</b>
<b>19</b>	<b>Policy Function Iteration</b>	<b>193</b>
<b>II</b>	<b>Appendices</b>	<b>201</b>
<b>A</b>	<b>Getting Started</b>	<b>203</b>
<b>B</b>	<b>Installing and Managing Python</b>	<b>211</b>
<b>C</b>	<b>NumPy Visual Guide</b>	<b>215</b>
	<b>Bibliography</b>	<b>219</b>



# Part I Labs



# 1

## 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 1.1), while the nodes of a *doubly linked list* have two references: one for the previous node, and one for the next node (see Figure 1.2). 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.
```

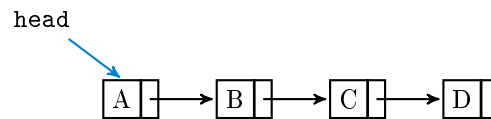


Figure 1.1: A singly linked list. Each node has a reference to the next node in the list. The head attribute is always assigned to the first node.

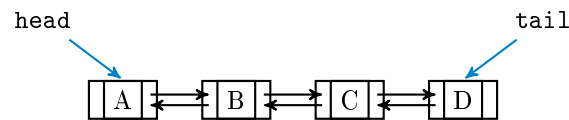


Figure 1.2: A doubly linked list. Each node has a reference to the node before it and a reference to the node after it. In addition to the head attribute, this list has a tail attribute that is always assigned to the last node.

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 1.3). There are two cases for appending that must be considered separately in the implementation: either the list is empty, or the list is nonempty.

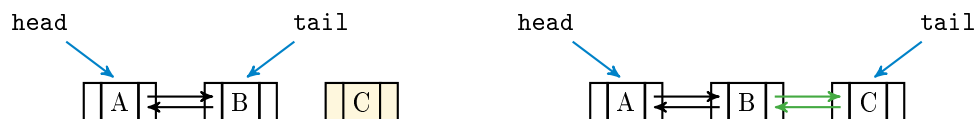


Figure 1.3: Appending a new node to the end of a nonempty doubly linked list. The green arrows are the new connections. Note that the `tail` attribute is reassigned from B to C.

```

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.)

## 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 1.4). 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 1.4: 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 1.5: 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 1.4.

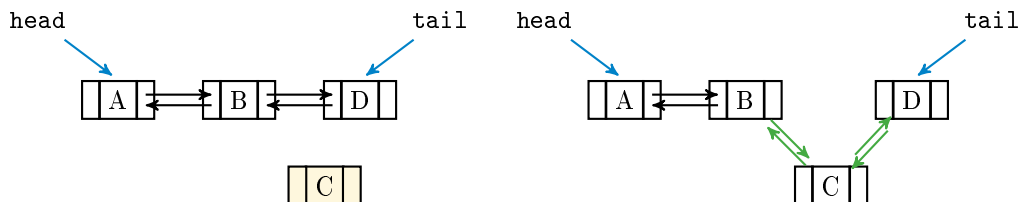


Figure 1.6: Insertion for doubly linked lists.

**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 1.6 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 1.1 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 1.1: 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 do not believe that  
I can solve these problems.  
Programming is hard, but  
I am a mathematician.

I am a mathematician.  
Programming is hard, but  
I can solve these problems.  
I do not believe that  
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.

# 2

## 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): # Calculate g(2), g(3), ..., g(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)
```

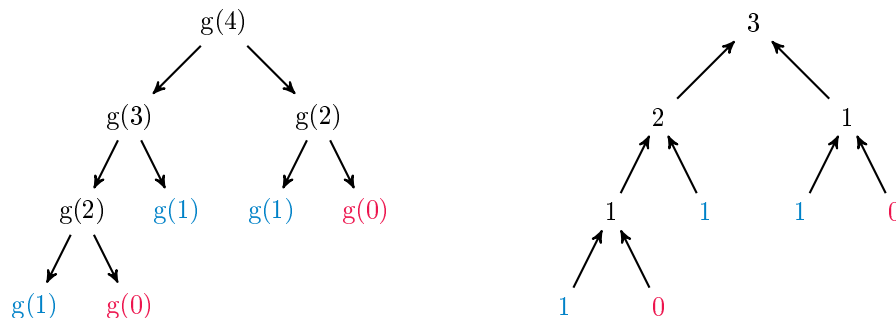


Figure 2.1: To calculate  $g(n)$  recursively, call  $g(n-1)$  and  $g(n-2)$ , down to the base cases  $g(0)$  and  $g(1)$ . As the recursion unwinds, the values from the base cases are passed up to previous calls and combined, eventually giving the value for  $g(n)$ .

**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.

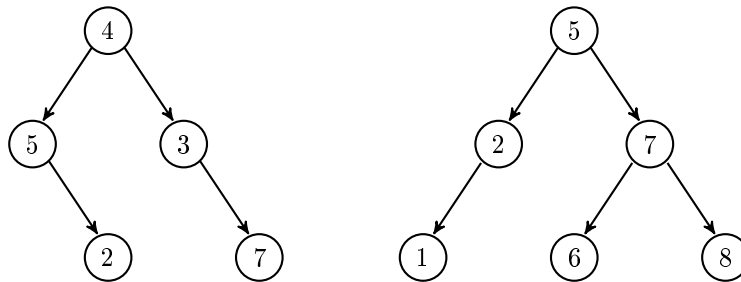


Figure 2.2: Both of these graphs are trees, but the tree on the left is not a binary search tree because 5 is to the left of 4. Swapping 5 and 3 in the graph on the left would result in a BST.

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.

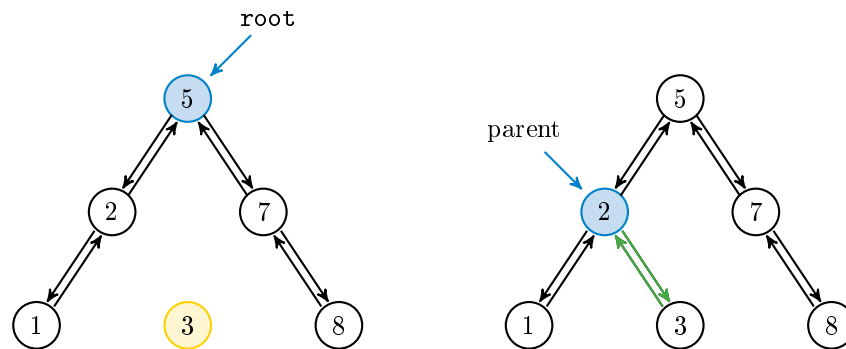


Figure 2.3: To insert 3 to the BST on the left, start at the root and recurse down the tree as if searching for 3: since  $3 < 5$ , step left to 2; since  $2 < 3$ , step right. However, 2 has no right child, so 2 becomes the parent of a new node containing 3.

**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 2.2 and 2.3.

## 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`.

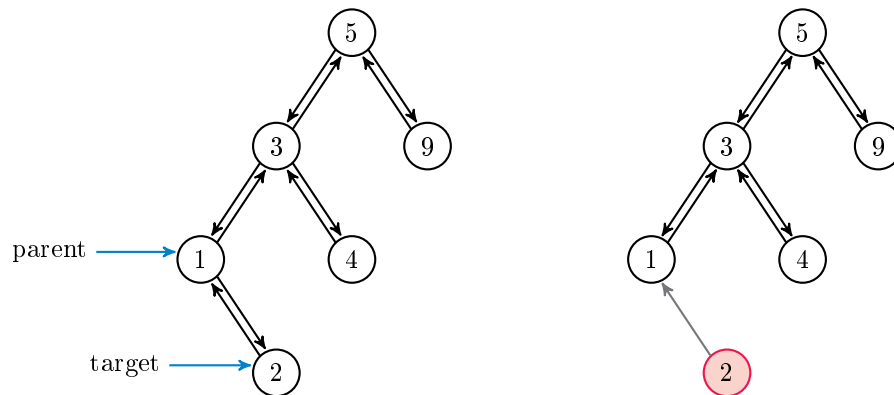


Figure 2.4: To remove 2, get a reference to its parent. Then set the parent's `right` attribute to `None`. Even though 2 still points to 1, 2 is deleted since nothing in the tree points to it.

## 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.

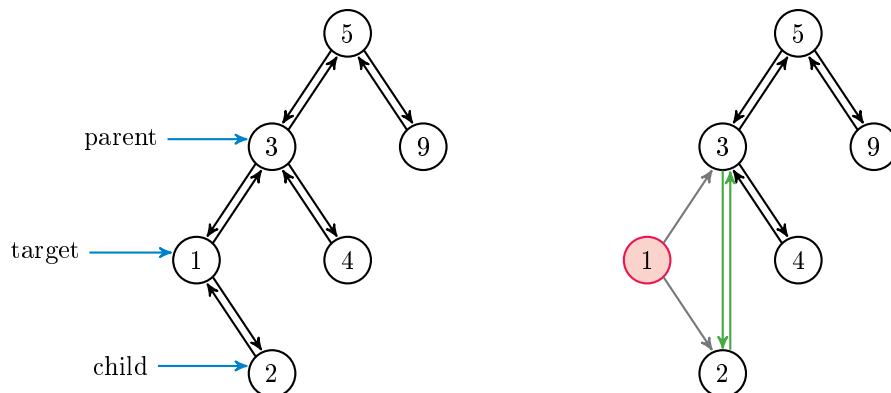


Figure 2.5: To remove 1, locate its parent (3) and its child (2). Set the parent's `left` attribute to the child and the child's `prev` attribute to the parent. Even though 1 still points to other nodes, it is deleted since nothing in the tree points to it.

### 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 its 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.

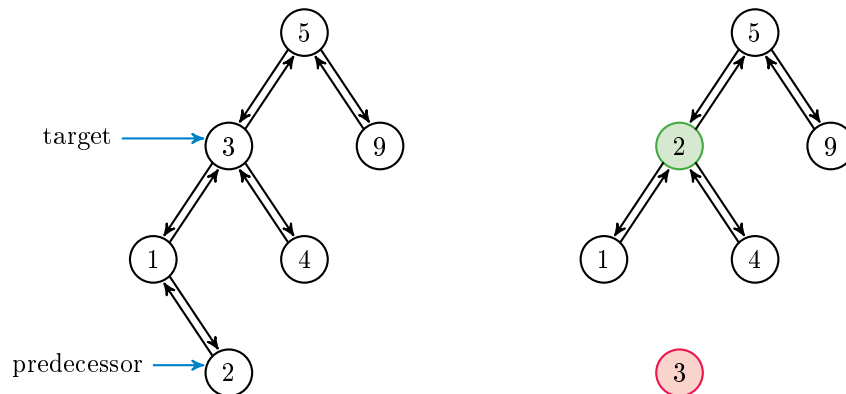


Figure 2.6: To remove 3, locate its immediate predecessor 2 by stepping left to 1, then right as far as possible. Since it is a leaf node, the predecessor can be deleted using the process in Figure 2.4. Delete the predecessor, and replace the value of the target with the predecessor's value. If the predecessor has a left child, it can be deleted with the procedure from Figure 2.5.

### 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 2.4–2.6. 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.

## 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 2.7). 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.

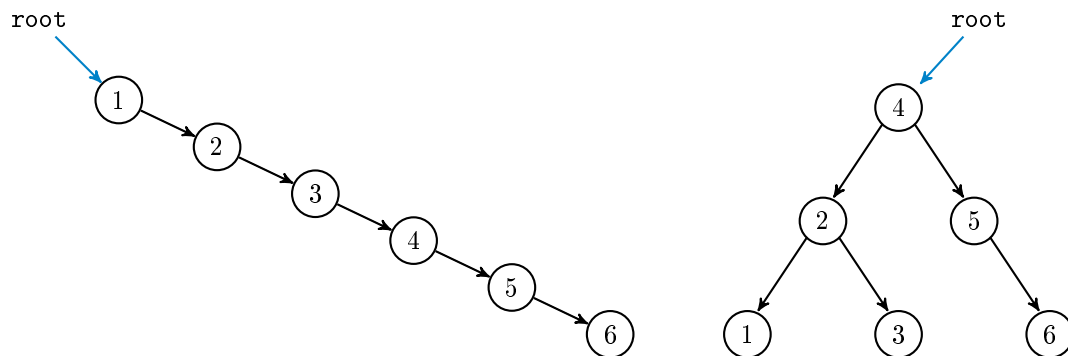


Figure 2.7: On the left, the unbalanced BST resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. On the right, the balanced AVL tree that results from the same insertion. After each insertion, the AVL tree rebalances if necessary.

**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)
                  / \   '[3]'      The nodes of the BST are printed
                  (2) (5)  '[2, 5]'  by depth levels. Edges and empty
                  /   \    '[1, 4, 6]'' nodes are not printed.
                 (1) (4) (6)
        """
        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_right(n)
```



# 3

## 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*.

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\| = \sqrt{\sum_{i=1}^k (x_i - z_i)^2} \quad (3.1)$$

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.

$$\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \quad d^* = \min_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \quad (3.2)$$

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 (3.2) 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 (3.2) 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 (3.1) 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  $\mathbf{x} = [x_0, x_1, \dots, x_{k-1}]^T \in \mathbb{R}^k$ , 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 3.1 for an example where  $k = 3$ .

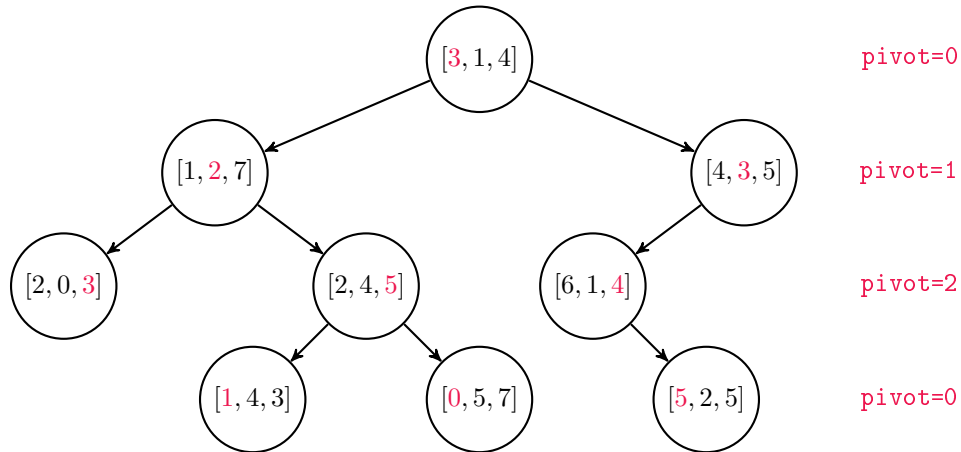


Figure 3.1: A  $k$ -d tree with  $k = 3$ . The root  $[3, 1, 4]$  has an *pivot* of 0, so  $[1, 2, 7]$  is to the left of the root because  $1 < 3$ , and  $[4, 3, 5]$  is to the right since  $3 \leq 4$ . Similarly, the node  $[2, 4, 5]$  has an *pivot* of 2, so  $[1, 4, 3]$  is to its left since  $4 < 5$  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.

## 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 3.2.

```

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):
                # Base case 2: data found!
                return current
            elif data[current.pivot] < current.value[current.pivot]:
                # Recursively search left.
                return _step(current.left)
            else:
                # Recursively search right.
                return _step(current.right)

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

```

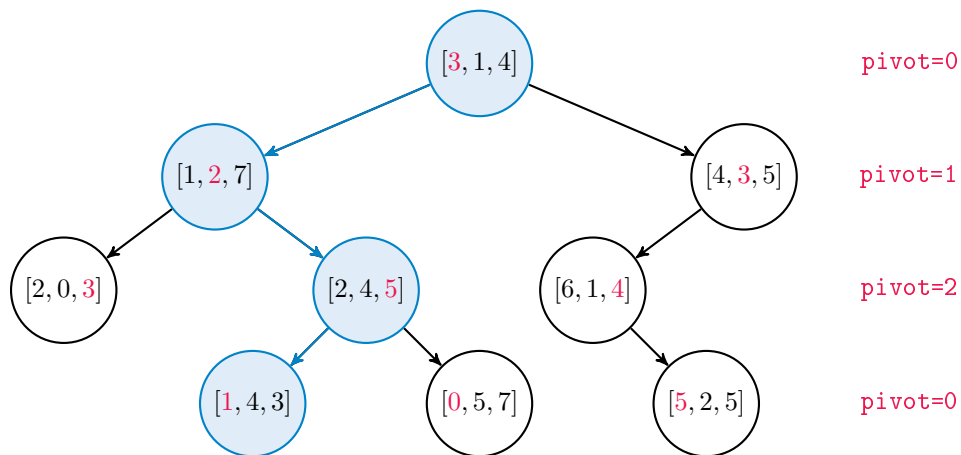
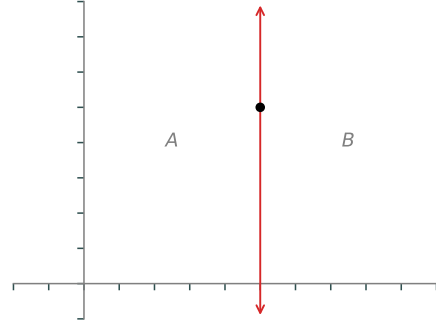
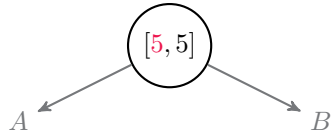
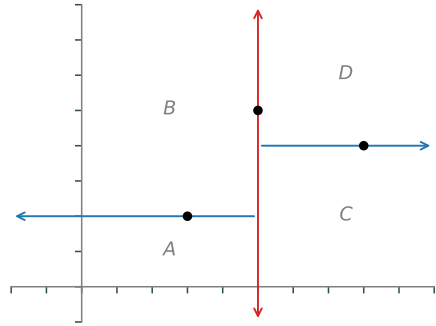
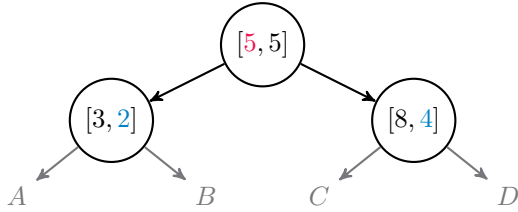


Figure 3.2: To locate the node containing  $[1, 4, 3]$ , start by comparing  $[1, 4, 3]$  to the root  $[3, 1, 4]$ . The root has an pivot of 0, so compare the first component of the data to the first component of the root: since  $1 < 3$ , step left. Next,  $[1, 4, 3]$  must be to the right of  $[1, 2, 7]$  because  $2 \leq 4$ . Similarly,  $[1, 4, 3]$  must be to the left of  $[2, 4, 5]$  as  $3 < 5$ .

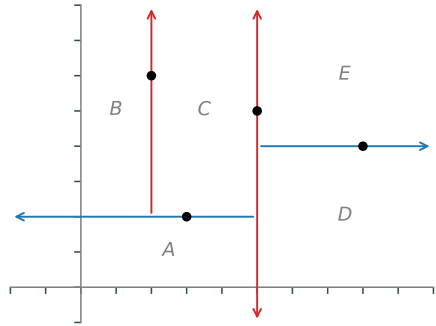
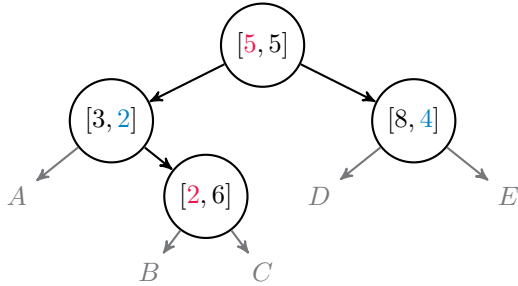




(a) Insert  $[5, 5]$  as the root. The root always has an **pivot** of 0, so nodes to the left of the root contain points from  $A = \{(x, y) \in \mathbb{R}^2 : x < 5\}$ , and nodes on the right branch have points in  $B = \{(x, y) \in \mathbb{R}^2 : 5 \leq x\}$ .



(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 3.3: 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 3.2. 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 3.3 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 `KDTNode` 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 `KDTNode` 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`.

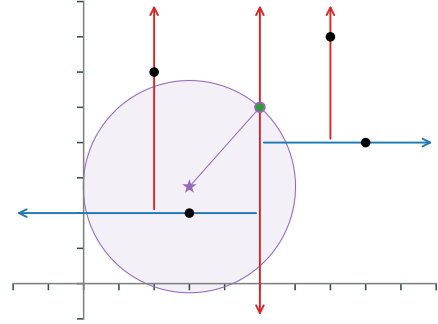
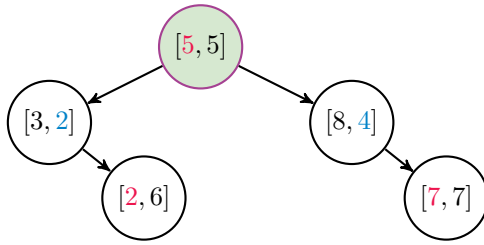
To test your method, use the `__str__()` method provided in the Additional Materials section. Try constructing the trees in Figures 3.1 and 3.3. Also check that the provided `find()` method works as expected.

### Nearest Neighbor Search with K-D Trees

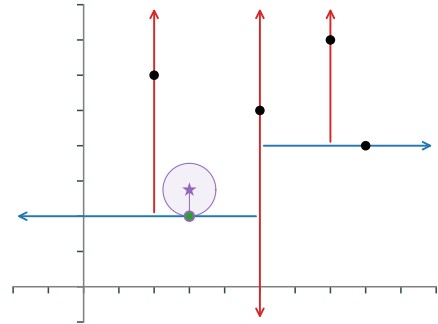
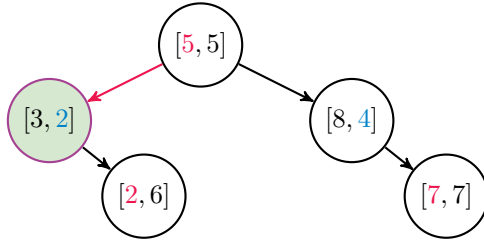
Given a target  $\mathbf{z} \in \mathbb{R}^k$  and a  $k$ -d tree containing a set  $X \subset \mathbb{R}^k$  of  $m$  points, the nearest neighbor problem can be solved by traversing the tree in a manner that is similar to the `find()` or `insert()` methods from the previous section. The advantage of this strategy over an exhaustive search is that not every  $\mathbf{x} \in X$  has to be compared to  $\mathbf{z}$  via (3.1); 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.

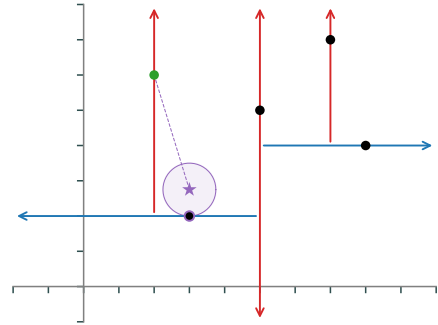
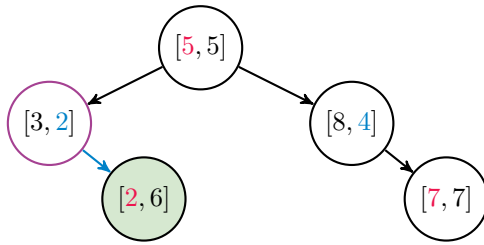
Next, backtrack along the search path and determine if the non-explored branch needs to be searched. To do this, check that the sphere of radius  $d^*$  centered at  $\mathbf{z}$  does not intersect with the separating hyperplane defined by the current node. That is, if the separating hyperplane is further than  $d^*$  from  $\mathbf{z}$ , then no points on the other side of the hyperplane can possibly be the nearest neighbor. See Figure 3.4 for an example and Algorithm 3.1 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 3.4: 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 3.1** *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                                     ▷ 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                                     ▷ 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                                             ▷ Search to the left.
11:      nearest,  $d^* \leftarrow$  KDSEARCH(current.left, nearest,  $d^*$ )
12:      if  $z_i + d^* \geq x_i$  then                                     ▷ Search to the right if needed.
13:        nearest,  $d^* \leftarrow$  KDSEARCH(current.right, nearest,  $d^*$ )
14:    else                                                           ▷ Search to the right.
15:      nearest,  $d^* \leftarrow$  KDSEARCH(current.right, nearest,  $d^*$ )
16:      if  $z_i - d^* \leq x_i$  then                                     ▷ 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 3.1 to solve (3.2). 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 3.5.

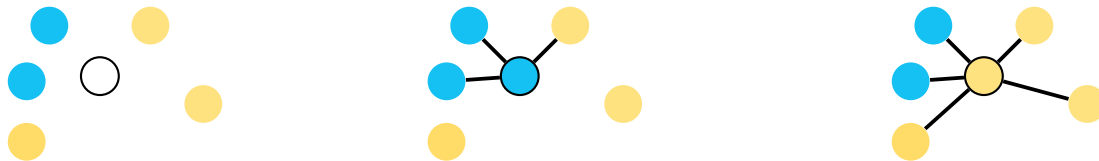


Figure 3.5: To classify the center node, determine its  $k$ -nearest neighbors and pick the most common label of the neighbors. If  $k = 3$ , the  $k$  nearest points are two blues and a yellow, so the center node is labeled blue. For  $k = 5$ , the  $k$  nearest points consists of two blues and three 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  $\mathbf{y}$  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  $\mathbf{z}$  with  $k$  entries. Query the `KDTree` for the `n_neighbors` elements of  $X$  that are nearest to  $\mathbf{z}$  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 3.5), 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 that 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.

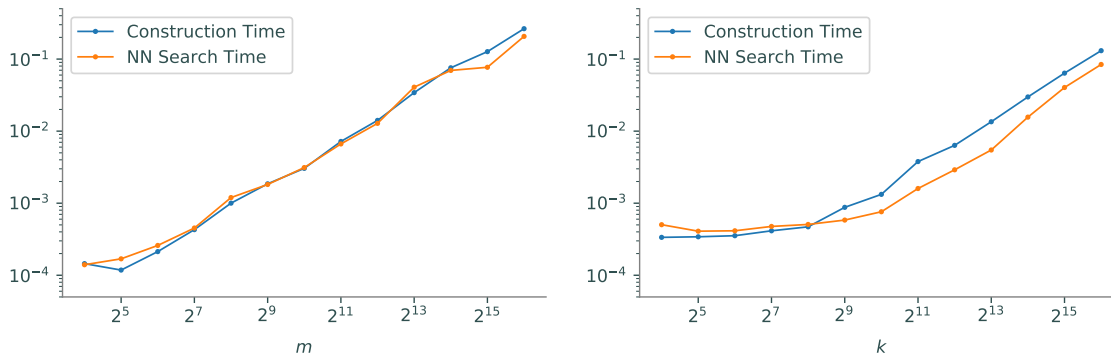
## Additional Material

### 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.



(a) Fixing  $k$  and increasing  $m$  leads to consistent growth in execution time.

(b) For fixed  $m$ , the times takes a sharp upturn around  $k = 2^9$  relative to previous growth rates.

Figure 3.6: 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]
             / \
          [3,2] [8,4]
           \   \
        [2,6]  [7,5]

        """
        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)
```



# 4

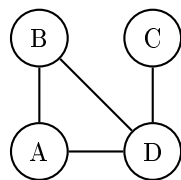
## Breadth-first Search

**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).



	A	B	C	D
A	0	1	0	1
B	1	0	0	1
C	0	0	0	1
D	1	1	1	0

{A : {B, D},  
B : {A, D},  
C : {D},  
D : {A, B, D}}

Figure 4.1: 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 4.1: 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,
KeyError: 'emu'              # resulting in an exception.

>>> 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 4.2: 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 4.1. 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 be 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.

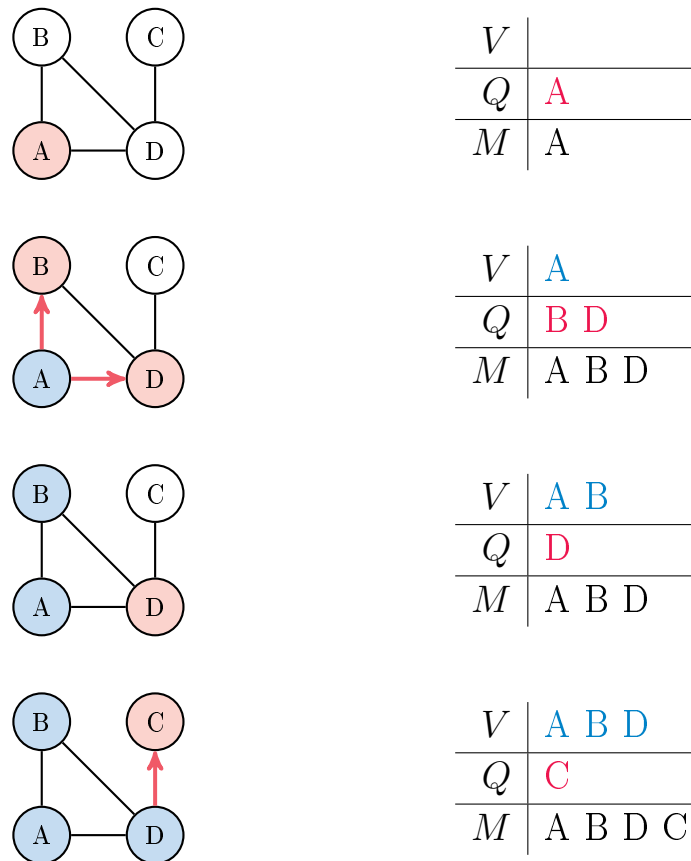


Figure 4.2: To start a BFS from node A to node C, put A in the visit queue  $Q$  and mark it by adding it to the set  $M$ . Pop A off the queue and “visit” it by adding A to the visited list  $V$  and the neighboring nodes B and D to  $Q$ . Then visit B, but do not add anything to  $Q$  because all of 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 to the target with a BFS must be the shortest path from the source node to the target node.

Examine again the graph in Figures 4.1 and 4.2. 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.

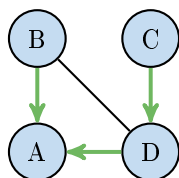


Figure 4.3: In the BFS from Figure 4.2, nodes B and D were marked while visiting node A, and node C was marked while visiting node D (this is same as reversing the red arrows in Figure 4.2). Thus the “visit path” from C to A is  $C \rightarrow D \rightarrow A$ , so the shortest path from A to C is [A, 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 4.3: 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 4.4.

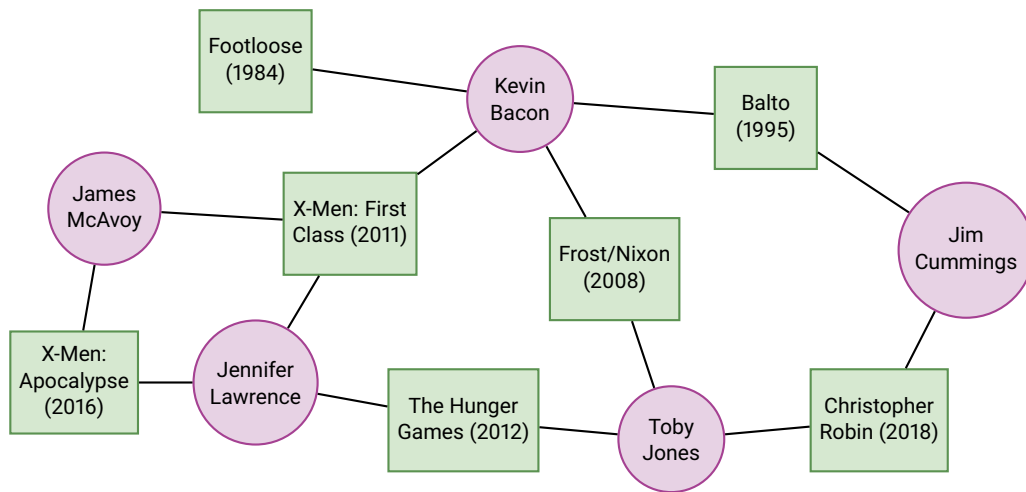


Figure 4.4: A subset of the graph in `movie_data.txt`. Each of these actors have a Bacon number of 1 because they have all been in a movie with Kevin Bacon. Every actor in *The Hunger Games* has a Bacon number of at most 2 because of the paths through Jennifer Lawrence or Toby Jones.

#### NOTE

The movie/actor graph of Problem 4 and Figure 4.4 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 <b>True</b> if there is a path between two specified nodes.
<code>shortest_path()</code>	Return <b>one</b> shortest path between nodes.
<code>shortest_path_length()</code>	Return the length of the shortest path between nodes.
<code>all_shortest_paths()</code>	Yield <b>all</b> shortest paths between nodes.

Table 4.4: 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')
{'A': 0, 'D': 1, 'B': 1, 'C': 2}
```

# Path lengths are defined by the  
# 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 movies are just intermediate steps between actors.

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()`.

# 5

## 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

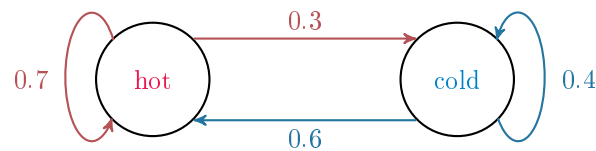
A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the  $(i, j)$ th entry of the matrix is the probability of moving from state  $i$  to state  $j$ . Both representations 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.

$$\begin{array}{cc}
 & \begin{array}{cc} \text{hot today} & \text{cold today} \end{array} \\
 \begin{array}{c} \text{hot tomorrow} \\ \text{cold tomorrow} \end{array} & \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix}
 \end{array}$$

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

$$\begin{array}{cc}
 & \begin{array}{cccc} \text{hot} & \text{mild} & \text{cold} & \text{freezing} \end{array} \\
 \begin{array}{c} \text{hot} \\ \text{mild} \\ \text{cold} \\ \text{freezing} \end{array} & \begin{bmatrix} 0.5 & 0.3 & 0.1 & 0 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0 & 0.1 & 0.2 & 0.2 \end{bmatrix}
 \end{array}
 .$$

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  $\mathbf{x} = [0.8 \ 0.2]^T$  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  $\mathbf{x} = [0 \ 1]^T$  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  $\mathbf{x}_5 = [0.8 \ 0.2]^T$ , 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

$$\lim_{k \rightarrow \infty} A^k = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix}. \quad (5.1)$$

In addition, for any initial state distribution vector  $\mathbf{x}_0 = [a, b]^T$  (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,  $\mathbf{x}_k \rightarrow \mathbf{x} = [2/3 \ 1/3]^T$  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  $\mathbf{x} = [2/3 \ 1/3]^T$  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 (5.1). 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,

$\$start$	I	am	Sam	...	ham.	$\$stop$
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

$$\begin{array}{c}
 \text{\textcolor{green}{\$start}} \quad \text{I} \quad \text{am} \quad \text{Sam} \quad \quad \text{ham.} \quad \text{\textcolor{red}{\$stop}} \\
 \begin{array}{c} \text{\textcolor{green}{\$start}} \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \vdots \\ \text{ham.} \\ \text{\textcolor{red}{\$stop}} \end{array} \left[ \begin{array}{cccccc} 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3 & 0 & 0 & 2 & \dots & 0 & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 \end{array} \right].
 \end{array}$$

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:

$$\begin{array}{c}
 \text{\textcolor{green}{\$start}} \quad \text{I} \quad \text{am} \quad \text{Sam} \quad \quad \text{ham.} \quad \text{\textcolor{red}{\$stop}} \\
 \begin{array}{c} \text{\textcolor{green}{\$start}} \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \vdots \\ \text{ham.} \\ \text{\textcolor{red}{\$stop}} \end{array} \left[ \begin{array}{cccccc} 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3/4 & 0 & 0 & 2/3 & \dots & 0 & 0 \\ 0 & 1/5 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1/3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \end{array} \right].
 \end{array}$$

The 3/4 indicates that 3 out of 4 times, the sentences in the training set start with the word “I”. Similarly, the 2/3 and 1/3 says that “Sam” is followed by “I” 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 5.1** 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 "**\textcolor{red}{\\$start}**" and "**\textcolor{red}{\\$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 "**\textcolor{red}{\\$start}**" and append "**\textcolor{red}{\\$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 5.1. 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 `"$start"` and `"$top"` 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/>.



# 6

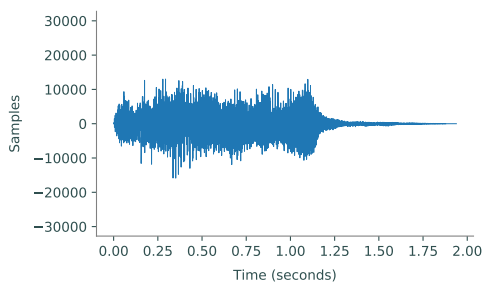
## The Discrete Fourier Transform

**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.*

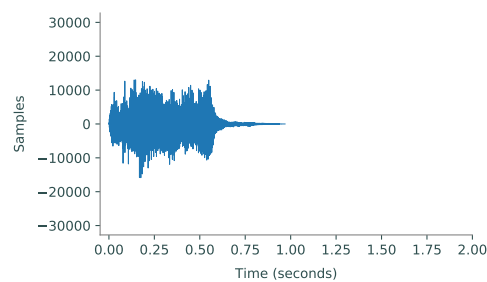
### Digital Audio Signals

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 6.1: 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 6.1. 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:

$$\frac{\text{num samples}}{\text{sample rate}} = \frac{\text{num samples}}{\text{num samples/second}} = \text{second}. \quad (6.1)$$

**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 (6.1) 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 6.1a.

## 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.

$$\text{np.int16} \left( \left( \frac{\text{original samples}}{\max(|\text{original samples}|)} \right) \times 32767 \right) = \text{scaled samples} \quad (6.2)$$

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 (6.2) 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,

$$f(x) = \sin(2\pi x k),$$

where  $k$  is the desired frequency. According to (6.1), 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  $A \rightarrow C \rightarrow E$ , 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  $\mathbf{f} = [f_0 \ f_1 \ \cdots \ f_{n-1}]^T$  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

$$\mathbf{f} = \sum_{k=0}^{n-1} c_k \mathbf{w}_n^{(k)}, \quad (6.3)$$

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  $\mathbf{c} = [c_0 \ c_1 \ \cdots \ c_{n-1}]^T$  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

$$\mathbf{w}_n^{(k)} = [\omega_n^0 \ \omega_n^{-k} \ \cdots \ \omega_n^{-(n-1)k}]^T, \quad (6.4)$$

where  $\omega_n = e^{2\pi i/n}$ .

The DFT is then represented by the change of basis matrix

$$F_n = \frac{1}{n} [\mathbf{w}_n^0 \ \mathbf{w}_n^1 \ \mathbf{w}_n^2 \ \cdots \ \mathbf{w}_n^{n-1}] = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{bmatrix}, \quad (6.5)$$

and we can take the DFT of  $f$  by calculating

$$\mathbf{c} = F_n \mathbf{f}. \quad (6.6)$$

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 (6.6), `scipy.fftpack.fft()` uses the formula

$$\mathbf{c} = nF_n \mathbf{f},$$

where  $F_n$  is as given (6.5). 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 6.6 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 (6.6) 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 6.1** 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  $nF_n \mathbf{g}$                                 ▷ Use the function from Problem 5 for small enough  $\mathbf{g}$ .
6:     else
7:        $\text{even} \leftarrow \text{SPLIT}(\mathbf{g}_{::2})$                   ▷ Get the DFT of every other entry of  $\mathbf{g}$ , starting from 0.
8:        $\text{odd} \leftarrow \text{SPLIT}(\mathbf{g}_{1::2})$                   ▷ 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                      ▷ Calculate the exponential parts of the sum.
11:         $z_k \leftarrow e^{-2\pi i k / n}$ 
12:       $m \leftarrow n // 2$                                 ▷ Get the middle index for  $\mathbf{z}$  ( $//$  is integer division).
13:      return  $[\text{even} + \mathbf{z}_{::m} \odot \text{odd}, \text{even} + \mathbf{z}_{m::} \odot \text{odd}]$  ▷ Concatenate two arrays of length  $m$ .
14:    return SPLIT( $\mathbf{f}$ ) /  $\text{size}(\mathbf{f})$ 

```

---



Note that the base case in lines 4–5 of Algorithm 6.1 results from setting  $n = 1$  in (6.6), 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 6.1 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 6.1 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.)

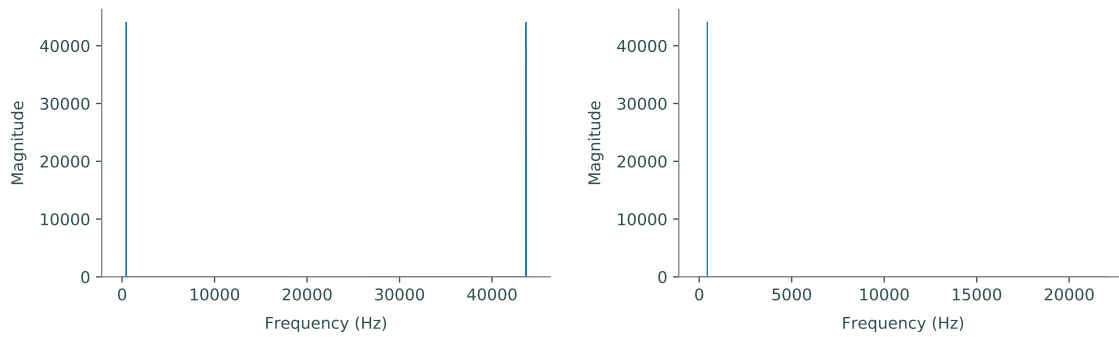
Test your implementation on random arrays against `scipy.fftpack.fft()`, scaling your output  $\mathbf{c}$  to match SciPy’s output  $\hat{\mathbf{c}}$ . Time your function from Problem 5, this function, and SciPy’s function on an array with 8192 entries.

(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 6.2a 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 6.2b.



(a) The DFT of an A tone with symmetries.

(b) The DFT of an A tone without symmetries.

Figure 6.2: 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-symmetrical 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.

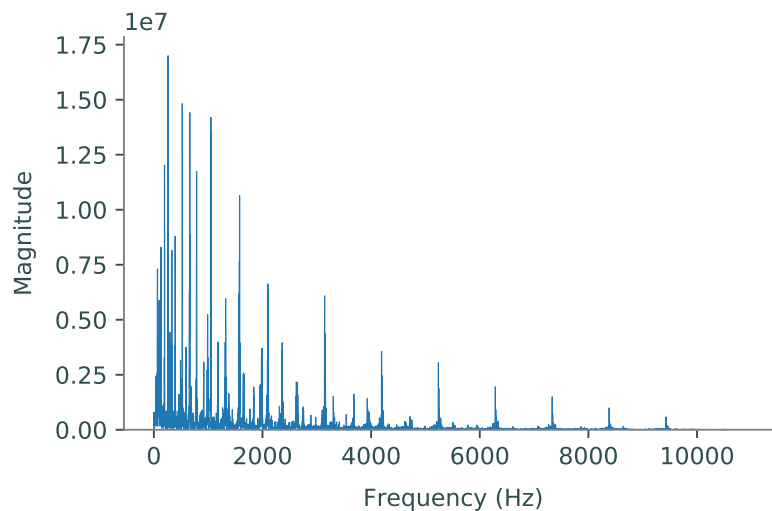


Figure 6.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency present in the sound wave. Since the sample rate of `tada.wav` is 22050 Hz, the plot of its DFT without symmetries only goes up to 11025 Hz, half of its sample rate.

## 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:

$$\frac{k}{\text{num samples}} \times \frac{\text{num samples}}{\text{second}} = \frac{k}{\text{second}}. \quad (6.7)$$

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 6.2b and 6.2b), and use (6.7) 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 6.2a and 6.4.

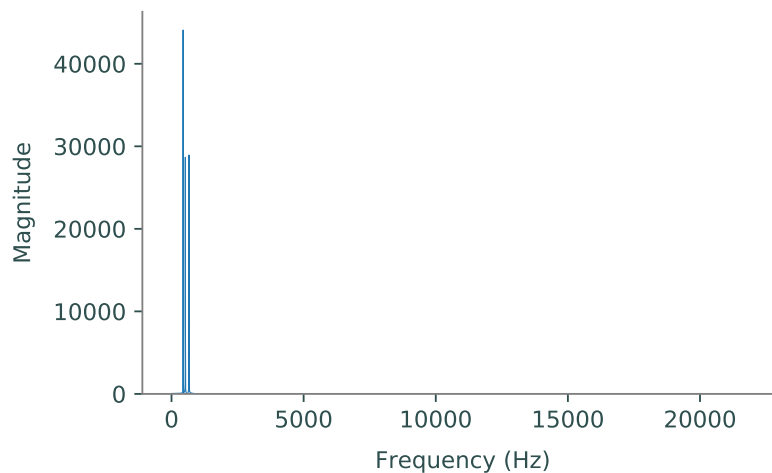


Figure 6.4: The DFT of the A minor chord.

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 6.4. 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 (6.7) 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.)

# 7

## Convolution and Filtering

**Lab Objective:** *The Fourier transform reveals information in the frequency domain about signals and images that might not be apparent in the usual time (sound) or spatial (image) domain. In this lab, we use the discrete Fourier transform to efficiently convolve sound signals and filter out some types of unwanted noise from both sounds and images. This lab is a continuation of the Discrete Fourier Transform lab and should be completed in the same Jupyter Notebook.*

### Convolution

Mixing two sounds signals—a common procedure in signal processing and analysis—is usually done through a *discrete convolution*. Given two periodic sound sample vectors  $\mathbf{f}$  and  $\mathbf{g}$  of length  $n$ , the discrete convolution of  $\mathbf{f}$  and  $\mathbf{g}$  is a vector of length  $n$  where the  $k$ th component is given by

$$(\mathbf{f} * \mathbf{g})_k = \sum_{j=0}^{n-1} f_{k-j} g_j, \quad k = 0, 1, 2, \dots, n-1. \quad (7.1)$$

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples; even a one-minute signal has 2,646,000 samples if it is recorded at the standard rate of 44,100 samples per second (44,100 Hz). The naïve method of using the sum in (7.1)  $n$  times is  $O(n^2)$ , which is often too computationally expensive for convolutions of this size.

Fortunately, the discrete Fourier transform (DFT) can be used compute convolutions efficiently. The *finite convolution theorem* states that the Fourier transform of a convolution is the element-wise product of Fourier transforms:

$$F_n(\mathbf{f} * \mathbf{g}) = n(F_n \mathbf{f}) \odot (F_n \mathbf{g}). \quad (7.2)$$

In other words, convolution in the time domain is equivalent to component-wise multiplication in the frequency domain. Here  $F_n$  is the DFT on  $\mathbb{R}^n$ ,  $*$  is discrete convolution, and  $\odot$  is component-wise multiplication. Thus, the convolution of  $\mathbf{f}$  and  $\mathbf{g}$  can be computed by

$$\mathbf{f} * \mathbf{g} = nF_n^{-1}((F_n \mathbf{f}) \odot (F_n \mathbf{g})), \quad (7.3)$$

where  $F_n^{-1}$  is the *inverse discrete Fourier transform* (IDFT). The fast Fourier transform (FFT) puts the cost of (7.3) at  $O(n \log n)$ , a huge improvement over the naïve method.

## NOTE

Although individual samples are real numbers, results of the IDFT may have small complex components due to rounding errors. These complex components can be safely discarded by taking only the real part of the output of the IDFT.

```
>>> import numpy
>>> from scipy.fftpack import fft, ifft # Fast DFT and IDFT functions.

>>> f = np.random.random(2048)
>>> f_dft_idft = ifft(fft(f)).real      # Keep only the real part.
>>> np.allclose(f, f_dft_idft)         # Check that IDFT(DFT(f)) = f.
True
```

## ACHTUNG!

SciPy uses a different convention to define the DFT and IDFT than this and the previous lab, resulting in a slightly different form of the convolution theorem. Writing SciPy's DFT as  $\hat{F}_n$  and its IDFT as  $\hat{F}_n^{-1}$ , we have  $\hat{F}_n = nF_n$ , so (7.3) becomes

$$\mathbf{f} * \mathbf{g} = \hat{F}_n^{-1}((\hat{F}_n \mathbf{f}) \odot (\hat{F}_n \mathbf{g})), \quad (7.4)$$

without a factor of  $n$ . Use (7.4), not (7.3), when using `fft()` and `ifft()` from `scipy.fftpack`.

## Circular Convolution

The definition (7.1) and the identity (7.3) require  $\mathbf{f}$  and  $\mathbf{g}$  to be periodic vectors. However, the convolution  $\mathbf{f} * \mathbf{g}$  can always be computed by simply treating each vector as periodic. The convolution of two raw sample vectors is therefore called the *periodic* or *circular convolution*. This strategy mixes sounds from the end of each signal with sounds at the beginning of each signal.

## Problem 1.

Implement the `__mul__()` 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 circular convolution of the samples from `A` and `B`. If the samples from `A` and `B` are not the same length, append zeros to the shorter array to make them the same length before convolving. Use `scipy.fftpack` and (7.4) to compute the convolution, and raise a `ValueError` if the sample rates from `A` and `B` are not equal.

A circular convolution creates an interesting effect on a signal when convolved with a segment of white noise: the sound loops seamlessly from the end back to the beginning. To see this, generate two seconds of white noise (at the same sample rate as `tada.wav`) with the following code.

```
>>> rate = 22050          # Create 2 seconds of white noise at a given rate.
>>> white_noise = np.random.randint(-32767, 32767, rate*2, dtype=np.int16)
```

Next, convolve `tada.wav` with the white noise. Finally, use the `>>` operator to append the convolution result to itself. This final signal sounds the same from beginning to end, even though it is the concatenation of two signals.

## Linear Convolution

Although circular convolutions can give interesting results, most common sound mixtures do not combine sounds at the beginning of one signal with sounds at the end of another. Whereas circular convolution assumes that the samples represent a full period of a periodic function, *linear convolution* aims to combine non-periodic discrete signals in a way that prevents the beginnings and endings from interacting. Given two samples with lengths  $n$  and  $m$ , the simplest way to achieve this is to pad both samples with zeros so that they each have length  $n + m - 1$ , compute the convolution of these larger arrays, and take the first  $n + m - 1$  entries of that convolution.

### Problem 2.

Implement the `__pow__()` 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 linear convolution of the samples from `A` and `B`. Raise a `ValueError` if the sample rates from `A` and `B` are not equal.

Because `scipy.fftpack` performs best when the length of the inputs is a power of 2, start by computing the smallest  $2^a$  such that  $2^a \geq n + m - 1$ , where  $a \in \mathbb{N}$  and  $n$  and  $m$  are the number of samples from `A` and `B`, respectively. Append zeros to each sample so that they each have  $2^a$  entries, then compute the convolution of these padded samples using (7.4). Use only the first  $n + m - 1$  entries of this convolution as the samples of the returned `SoundWave` object.

To test your method, read `CGC.wav` and `GCG.wav`. Time (separately) the convolution of these signals with `SoundWave.__pow__()` and with `scipy.signal.fftconvolve()`. Compare the results by listening to the original and convolved signals.

**Problem 3.** Clapping in a large room with an echo produces a sound that resonates in the room for up to several seconds. This echoing sound is referred to as the *impulse response* of the room, and is a way of approximating the acoustics of a room. When the sound of a single instrument in a carpeted room is convolved with the impulse response from a concert hall, the new signal sounds as if the instrument is being played in the concert hall.

The file `chopin.wav` contains a short clip of a piano being played in a room with little or no echo, and `balloon.wav` is a recording of a balloon being popped in a room with a substantial echo (the impulse). Use your method from Problem 2 or `scipy.signal.fftconvolve()` to compute the linear convolution of `chopin.wav` and `balloon.wav`.

## Filtering Frequencies with the DFT

The DFT also provides a way to clean a signal by altering some of its frequencies. Consider `noisy1.wav`, a noisy recording of a short voice clip. The time-domain plot of the signal only shows that the signal has a lot of static. On the other hand, the signal's DFT suggests that the static may be the result of some concentrated noise between about 1250–2600 Hz. Removing these frequencies could result in a much cleaner signal.

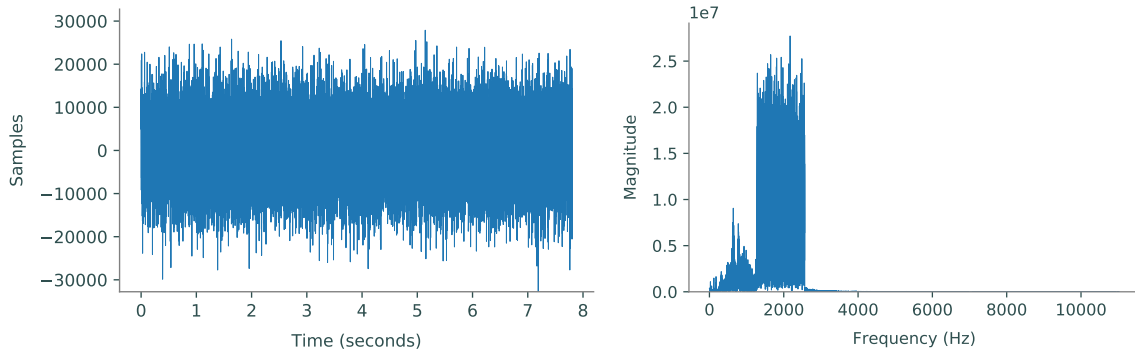


Figure 7.1: The time-domain plot (left) and DFT (right) of `noisy1.wav`.

To implement this idea, recall that the  $k$ th entry of the DFT array  $\mathbf{c} = F_n \mathbf{f}$  corresponds to the frequency  $v = kr/n$  in Hertz, where  $r$  is the sample rate and  $n$  is the number of samples. Hence, the DFT entry  $c_k$  corresponding to a given frequency  $v$  in Hertz has index  $k = vn/r$ , rounded to an integer if needed. In addition, since the DFT is symmetric,  $c_{n-k}$  also corresponds to this frequency. This suggests a strategy for filtering out an unwanted interval of frequencies  $[v_{\text{low}}, v_{\text{high}}]$  from a signal:

1. Compute the integer indices  $k_{\text{low}}$  and  $k_{\text{high}}$  corresponding to  $v_{\text{low}}$  and  $v_{\text{high}}$ , respectively.
2. Set the entries of the signal's DFT from  $k_{\text{low}}$  to  $k_{\text{high}}$  and from  $n - k_{\text{high}}$  to  $n - k_{\text{low}}$  to zero, effectively removing those frequencies from the signal.
3. Take the IDFT of the modified DFT to obtain the cleaned signal.

Using this strategy to filter `noisy1.wav` results in a much cleaner signal. However, any “good” frequencies in the affected range are also removed, which may decrease the overall sound quality. The goal, then, is to remove only as many frequencies as necessary.

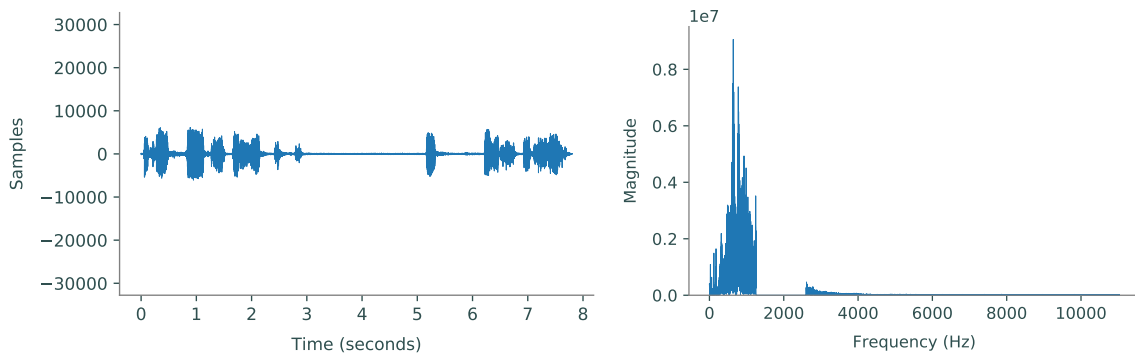


Figure 7.2: The time-domain plot (left) and DFT (right) of `noisy1.wav` after being cleaned.



**Problem 4.** Add a method to the `SoundWave` class that accepts two frequencies  $v_{\text{low}}$  and  $v_{\text{high}}$  in Hertz. Compute the DFT of the stored samples and zero out the frequencies in the range  $[v_{\text{low}}, v_{\text{high}}]$  (remember to account for the symmetry DFT). Take the IDFT of the altered array and store it as the sample array.

Test your method by cleaning `noisy1.wav`, then clean `noisy2.wav`, which also has some artificial noise that obscures the intended sound.

(Hint: plot the DFT of `noisy2.wav` to determine which frequencies to eliminate.)

A digital audio signal made of a single sample vector with is called *monaural* or *mono*. When several sample vectors with the same sample rate and number of samples are combined into a matrix, the overall signal is called *stereophonic* or *stereo*. This allows multiple speakers to each play one *channel*—one of the original sample vectors—simultaneously. “Stereo” usually means there are two channels, but there may be any number of channels (5.1 surround sound, for instance, has five).

Most stereo sounds are read as  $n \times m$  matrices, where  $n$  is the number of samples and  $m$  is the number of channels (i.e., each column is a channel). However, some functions, including Jupyter’s embedding tool `IPython.display.Audio()`, receive stereo signals as  $m \times n$  matrices (each row is a channel). Be aware that both conventions are common.

**Problem 5.** During the 2010 World Cup in South Africa, large plastic horns called vuvuzelas were blown excessively throughout the games. Broadcasting organizations faced difficulties with their programs due to the incessant noise level. Eventually, audio filtering techniques were used to cancel out the sound of the vuvuzela, which has a frequency of around 200–500 Hz.

The file `vuvuzela.wav`<sup>a</sup> is a stereo sound with two channels. Use your function from Problem 4 to clean the sound clip by filtering out the vuvuzela frequencies in each channel. Recombine the two cleaned samples.

<sup>a</sup>See [https://www.youtube.com/watch?v=g\\_0NoBKWCT8](https://www.youtube.com/watch?v=g_0NoBKWCT8).

## The Two-dimensional Discrete Fourier Transform

The DFT can be easily extended to any number of dimensions. Computationally, the problem reduces to performing the usual one-dimensional DFT iteratively along each of the dimensions. For example, to compute the two-dimensional DFT of an  $m \times n$  matrix, calculate the usual DFT of each of the  $n$  columns, then take the DFT of each of the  $m$  rows of the resulting matrix. Calculating the two-dimensional IDFT is done in a similar fashion, but in reverse order: first calculate the IDFT of the rows, then the IDFT of the resulting columns.

```
>>> from scipy.fftpack import fft2, ifft2

>>> A = np.random.random((10,10))
>>> A_dft = fft2(A) # Calculate the 2d DFT of A.
>>> A_dft_ifft = ifft2(A_dft).real # Calculate the 2d IDFT.
>>> np.allclose(A, A_dft_ifft)
True
```

Just as the one-dimensional DFT can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove “noise” in images. The procedure is similar to the filtering technique in Problems 4 and 5: take the two-dimensional DFT of the image matrix, modify certain entries of the DFT matrix to remove unwanted frequencies, then take the IDFT to get a cleaner version of the original image. This strategy makes the fairly strong assumption that the noise in the image is periodic and corresponds to certain frequencies. While this may seem like an unlikely scenario, it does actually occur in many digital images—for an example, try taking a picture of a computer screen with a digital camera.

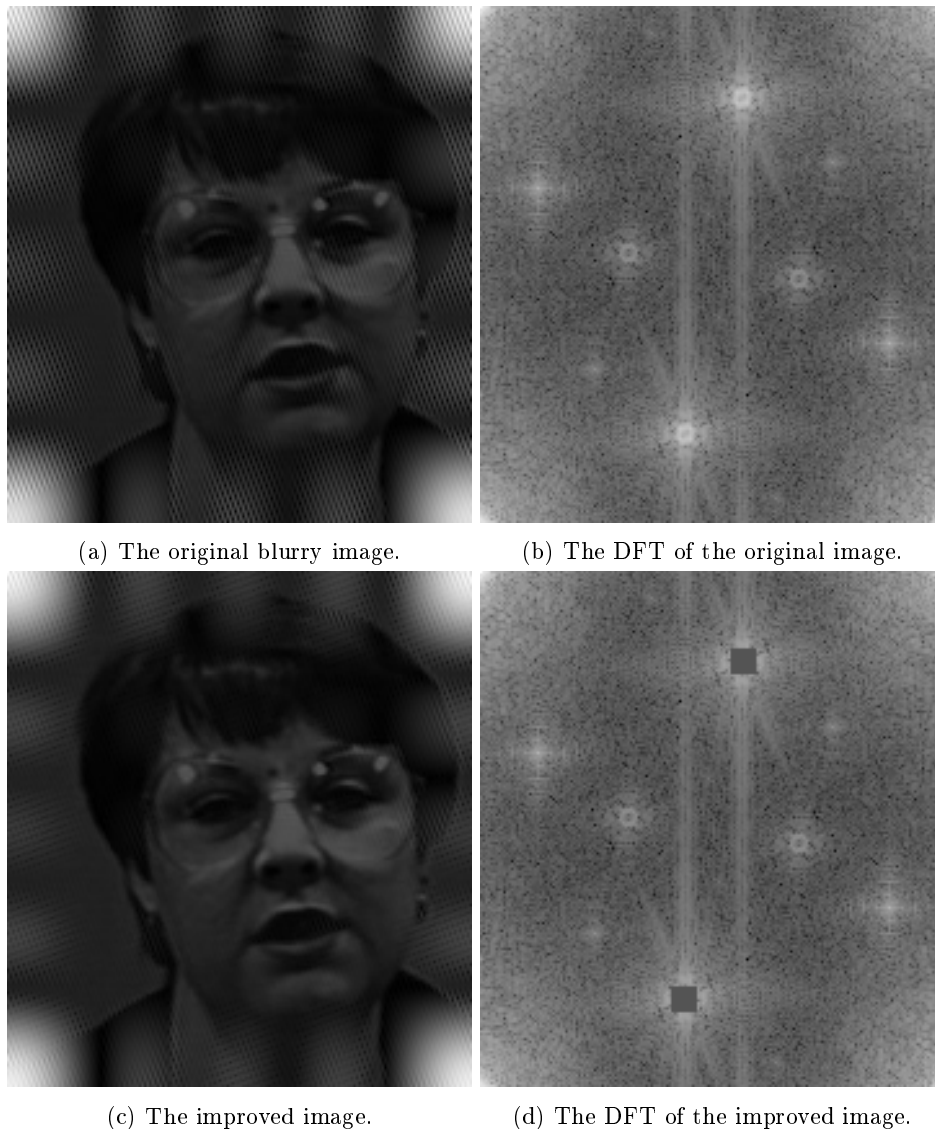


Figure 7.3: To remove noise from an image, take the DFT of the image and replace the abnormalities with values more consistent with the rest of the DFT. Notice that the new image is less noisy, but only slightly. This is because only some of the abnormalities in the DFT were changed; in order to further decrease the noise, we would need to further alter the DFT.

To begin cleaning an image with the DFT, take the two-dimensional DFT of the image matrix. Identify *spikes*—abnormally high frequency values that may be causing the noise—in the image DFT by plotting the log of the magnitudes of the Fourier coefficients. With `cmap="gray"`, spikes show up as bright spots. See Figures 7.3a–7.3b.

```
# Read the image.
>>> import imageio
>>> im = imageio.read("noisy_face.png")

# Plot the log magnitude of the image's DFT.
>>> im_dft = fft2(image)
>>> plt.imshow(np.log(np.abs(im_dft)), cmap="gray")
>>> plt.show()
```

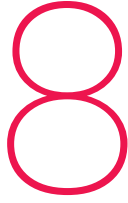
Instead of setting spike frequencies to zero (as was the case for sounds), replace them with values that are similar to those around them. There are many ways to do this, but one convention is to simply “patch” each spike by setting portions of the DFT matrix to some set value, such as the mean of the DFT array. See Figure 7.3d.

Once the spikes have been covered, take the IDFT of the modified DFT to get a (hopefully cleaner) image. Notice that Figure 7.3c still has noise present, but it is a slight improvement over the original. However, it often suffices to remove some of the noise, even if it is not possible to remove it all with this method.

**Problem 6.** The file `license_plate.png` contains a noisy image of a license plate. The bottom right corner of the plate has a sticker with information about the month and year that the vehicle registration was renewed. However, in its current state, the year is not clearly legible.

Use the two-dimensional DFT to clean up the image enough so that the year in the bottom right corner is legible. This may require a little trial and error.





# 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.

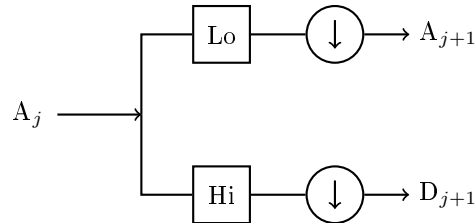
The two primary operations of the algorithm are the discrete convolution and downsampling, denoted  $*$  and  $DS$ , respectively. First, a signal is convolved with both filters. The resulting arrays will be twice the size of the original signal because the frequency of the sample will have changed by a factor of 2. To remove this redundant information, the resulting arrays are *downsampled*. 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 8.1.

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 difference filter. Given an array  $\mathbf{x}$ , the difference filter produces an array  $\mathbf{y}$  where  $y_n$  is the difference between  $x_n$  and  $x_{n-1}$ . 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 these 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:

$$L = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

$$H = \begin{bmatrix} -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$



Key:  $\square$  = convolve     $\downarrow$  = downsample

Figure 8.1: 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 8.1.

---

**Algorithm 8.1** 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:    $A_0 \leftarrow X$  ▷ Initialization.
3:   for  $i = 0 \dots n - 1$  do
4:      $D_{i+1} \leftarrow DS(A_i * H)$  ▷ High-pass filter and downsample.
5:      $A_{i+1} \leftarrow DS(A_i * L)$  ▷ 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 8.1. 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 8.2.  
(Hint: Use array broadcasting).



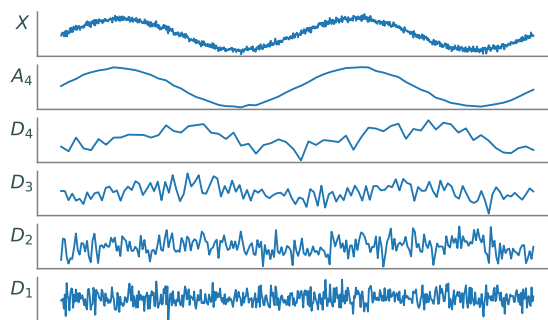


Figure 8.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture 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 8.1 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 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  $D_n, D_{n-1}, \dots, D_1$ ), so your code should perform the process given above  $n$  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 8.1 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 downsamples 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 8.3.

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  $LL$ ,  $LH$ ,  $HL$ , and  $HH$  are obtained.  $LL$  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 vertical, horizontal, and diagonal directions. The arrays  $LL$ ,  $LH$ ,  $HL$ , and  $HH$  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 8.4, with an example of an image decomposition given in Figure 8.5.

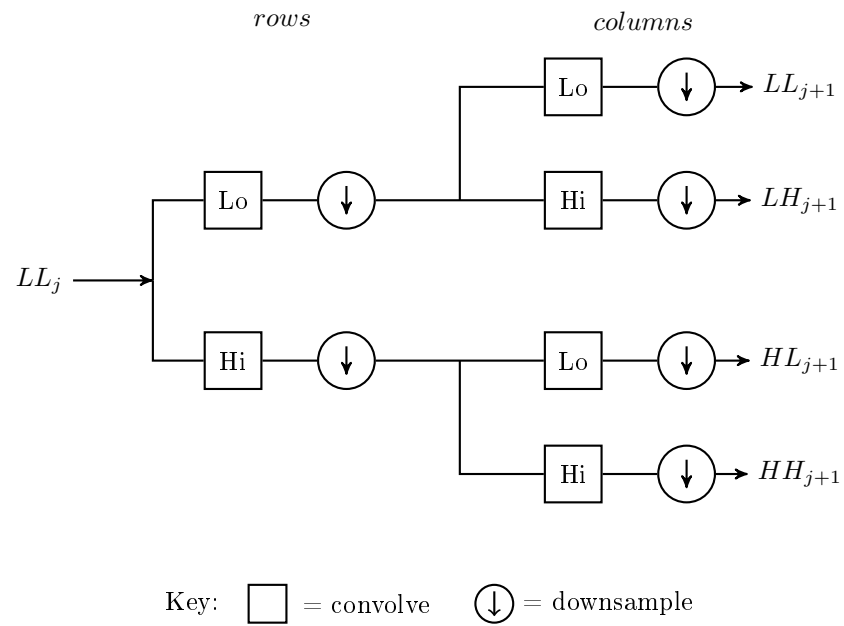


Figure 8.3: The two-dimensional discrete wavelet transform implemented as a filter bank.

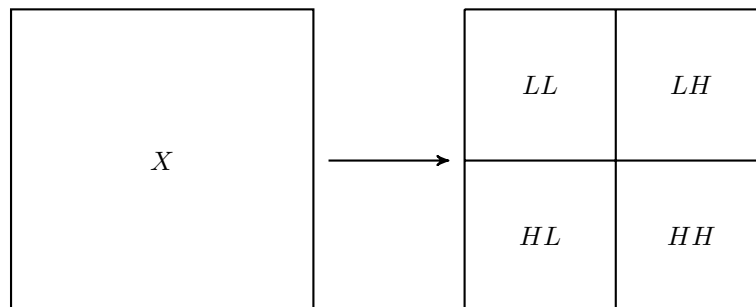


Figure 8.4: The subband pattern for one step in the 2-dimensional wavelet transform.

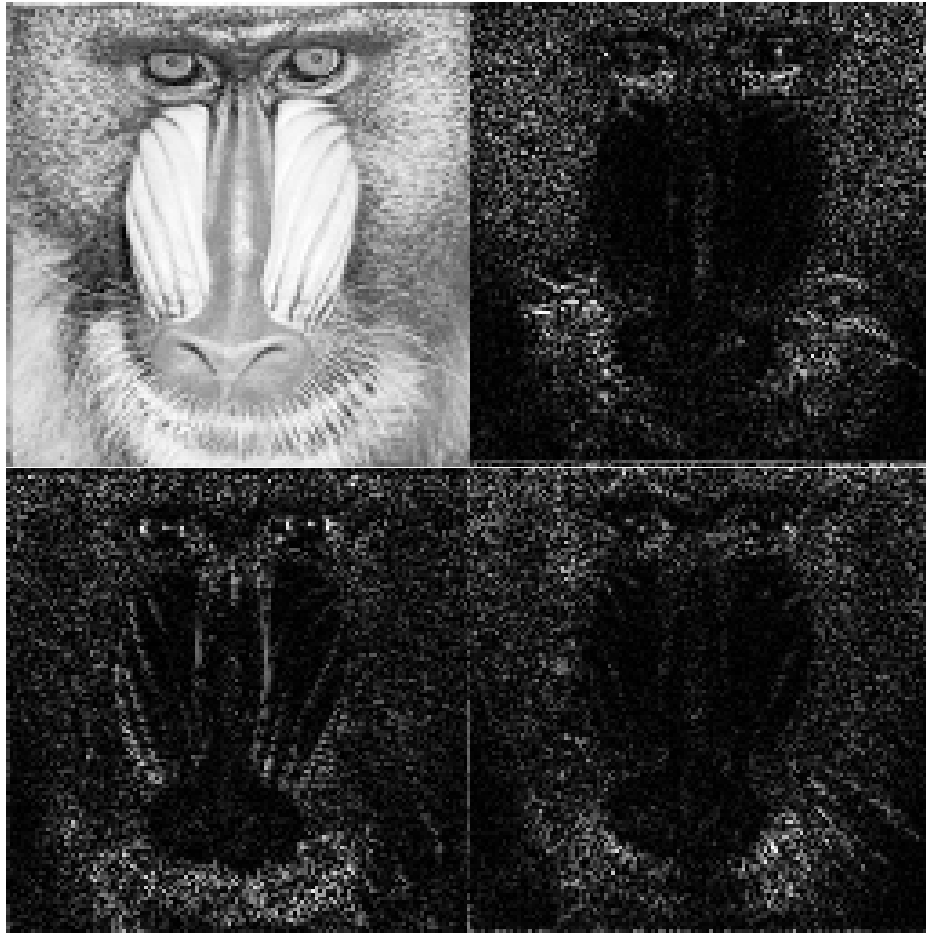


Figure 8.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband ( $LL$ ) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.

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', 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 list. The first entry of the list is the *LL*, or approximation, subband. The second entry of the list is a tuple containing the remaining subbands, *LH*, *HL*, and *HH* (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']
```

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
>>> racoon = 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 8.5.
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.

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 ( $LH$ ,  $HL$ , and  $HH$ ). 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 are thresholded, so the first entry of the input coefficient list 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  $\tau = 40$ .

## 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 8.6.

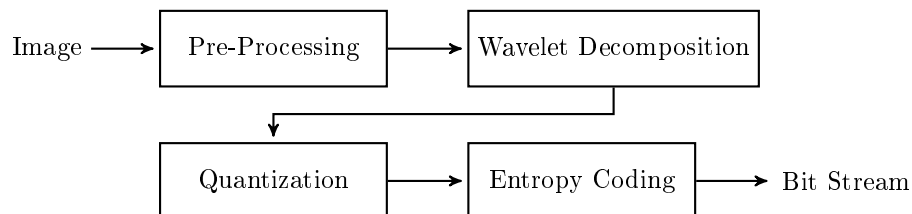


Figure 8.6: Wavelet Image Compression Schematic

## 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 8.7. 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.

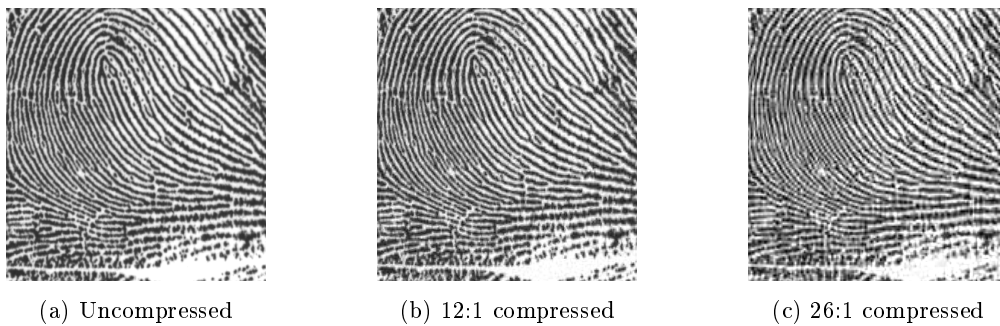


Figure 8.7: Fingerprint scan at different levels of compression. Original image source: <http://www.nist.gov/itl/iad/ig/wsq.cfm>.

### 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).

**Problem 5.** Implement the preprocessing step as well as its inverse by implementing the class methods `pre_process()` and `post_process()`. Each method accepts a NumPy array (the image) and returns the processed image as a NumPy array. In the `pre_process()` method, calculate the values of  $m$  and  $s$  given above and store them in the class attributes `_m` and `_s`.



### 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 8.8. Each of these subbands should be placed into a list in the same ordering as in Figure 8.8 (another way to consider this ordering is the approximation subband followed by each level of detail coefficients  $[LL_5, LH_5, HL_5, HH_5, LH_4, HL_4, \dots, HH_1]$ ).

**Problem 6.** 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()`.

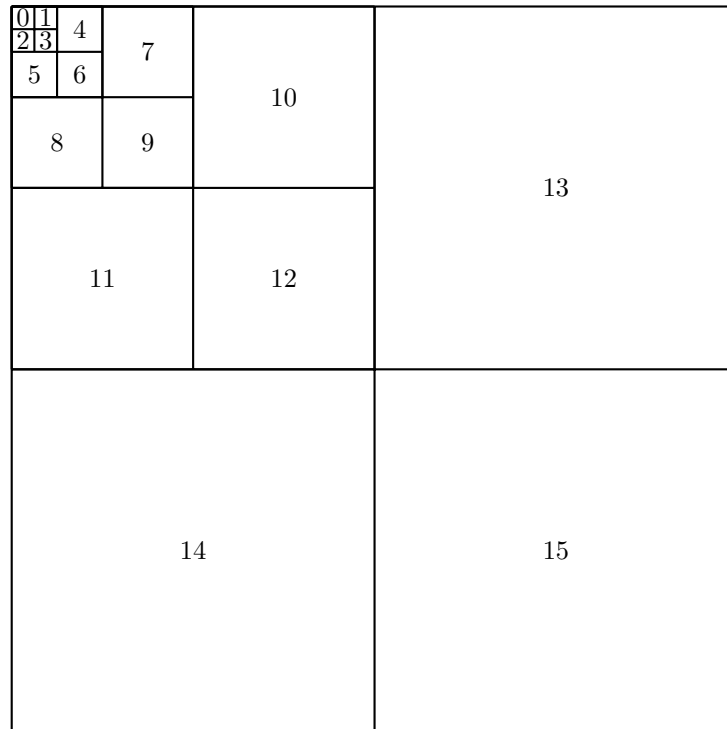


Figure 8.8: Subband Pattern for simplified WSQ algorithm.

### 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.

Selecting appropriate values for these parameters 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 7.** 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
```

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 8.** 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! You can 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', 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 8.9. 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 *LL*, *LH* and *HL* subbands of the original approximation subband creating 46 additional subbands. Finally, the subband corresponding to the top left of Figure 8.9 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									
2	3	4	7	8	19	20	23	24	52	53
5	6	9	10	21	22	25	26			
11	12	15	16	27	28	31	32			
13	14	17	18	29	30	33	34			
35	36	39	40	51				54	55	
37	38	41	42							
43	44	47	48							
45	46	49	50							
56				57				60	61	
58				59				62	63	

Figure 8.9: True subband pattern for WSQ algorithm.

# 9

## 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

$$L_j(x) = \frac{\prod_{k=1, k \neq j}^n (x - x_k)}{\prod_{k=1, k \neq j}^n (x_j - x_k)} \quad (9.1)$$

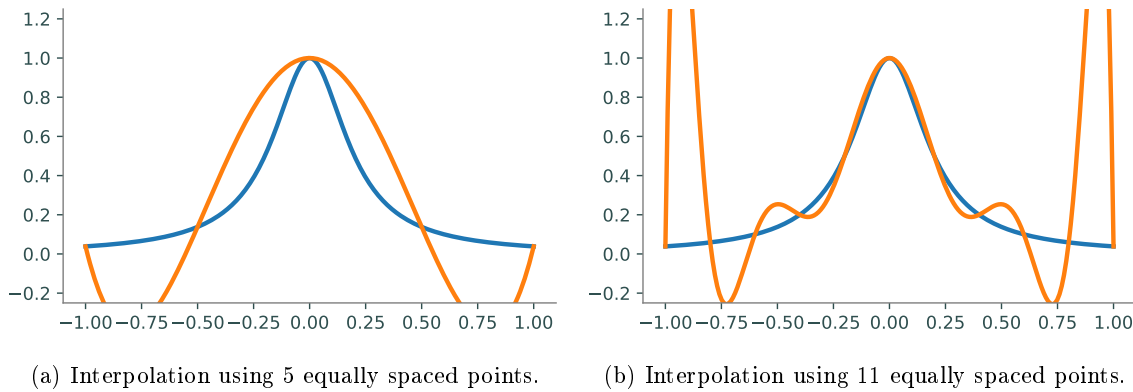


Figure 9.1: Interpolations of Runge's function  $f(x) = \frac{1}{1+25x^2}$  with equally spaced interpolating points.

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 9.1) .
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:

$$p(x) = \sum_{j=1}^n y_j L_j(x) \quad (9.2)$$

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 9.2. 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 9.1.

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

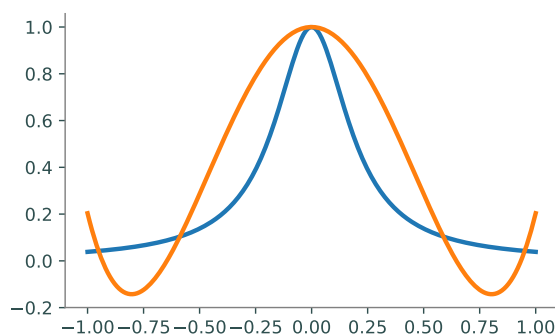
Scipy includes a Barycentric interpolator class. This class includes the same functionality as the class described in Problems 3 and 4 in addition to the ability to update the  $y$ -values of the interpolation points. The following code will produce a figure similar to Figure 9.1b.

```
>>> from scipy.interpolate import BarycentricInterpolator

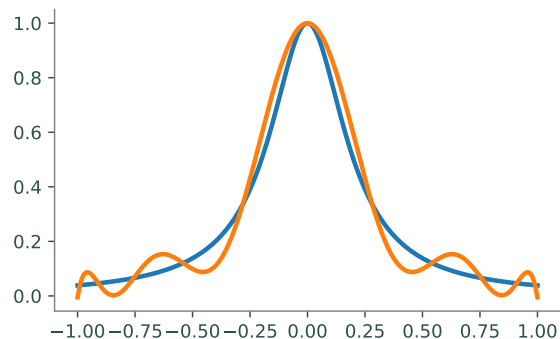
>>> f = lambda x: 1/(1+25 * x**2) # Function to be interpolated.
# Obtain the Chebyshev extremal points on [-1,1].
>>> n = 11
>>> pts = np.linspace(-1, 1, n)
>>> domain = np.linspace(-1, 1, 200)

>>> 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 9.2: 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 9.1

## 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 9.1 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 9.3. 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  $n$ .

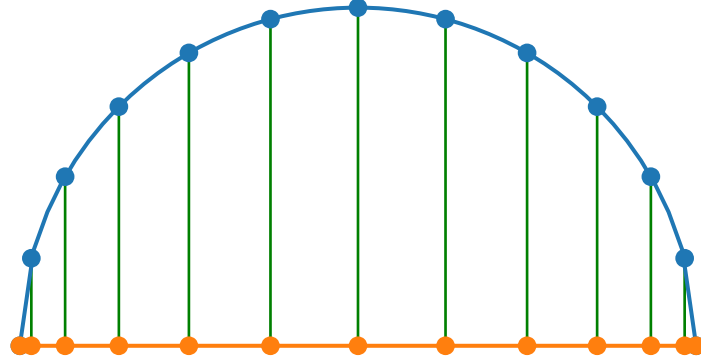


Figure 9.3: The Chebyshev extremal points. The  $n$  points where the Chebyshev polynomial of degree  $n$  reaches its local extrema. These points are also the projection onto the x-axis of  $n$  equally spaced points around the unit circle.

## 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 9.1** 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  $a_0 + xu_1 - u_2$ 

```

---



# 10

## 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.

$$J = \begin{bmatrix} \alpha_1 & \sqrt{\beta_1} & 0 & \dots & 0 \\ \sqrt{\beta_1} & \alpha_2 & \sqrt{\beta_2} & \dots & 0 \\ 0 & \sqrt{\beta_2} & \alpha_3 & \ddots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \dots & & \sqrt{\beta_{n-1}} & \sqrt{\beta_{n-1}} \\ 0 & \dots & & \sqrt{\beta_{n-1}} & \alpha_n \end{bmatrix}$$

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	$\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$
$w_i$	$\frac{322-13\sqrt{70}}{900}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{128}{225}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{322-13\sqrt{70}}{900}$

<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.

## 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

$$\int_{-1}^1 f(x)dx = \int_{-1}^1 g(x)w(x)dx \approx \sum_{i=1}^n g(x_i)w_i. \quad (10.1)$$

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 (10.1) 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()`.

```
>>> 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 (10.1). This results in the final quadrature formula

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 h(x)dx = \frac{b-a}{2} \int_{-1}^1 g(x)w(x)dx \approx \frac{b-a}{2} \sum_{i=1}^n g(x_i)w_i, \quad (10.2)$$

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 (10.2) 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 -inf 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

$$\int_{-1}^1 \int_{-1}^1 h(x, y) dx dy = \int_{-1}^1 \int_{-1}^1 g(x, y) w(x) w(y) dx dy \approx \sum_{i=1}^n \sum_{j=1}^n w_i w_j g(z_i, z_j). \quad (10.3)$$

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

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x) dx dy = \frac{(b_1 - a_1)(b_2 - a_2)}{4} \int_{-1}^1 \int_{-1}^1 h(x, y) dx dy. \quad (10.4)$$

Combining (10.3) and (10.4) gives the final 2-D Gaussian quadrature formula. Compare it to (10.2).

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} f(x) dx dy \approx \frac{(b_1 - a_1)(b_2 - a_2)}{4} \sum_{i=1}^n \sum_{j=1}^n w_i w_j g(z_i, z_j) \quad (10.5)$$

**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 (10.5) 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.





# 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 11.1). 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.

$$\tilde{a} = b - \frac{b-a}{\varphi} \qquad \tilde{b} = a + \frac{b-a}{\varphi}$$

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.

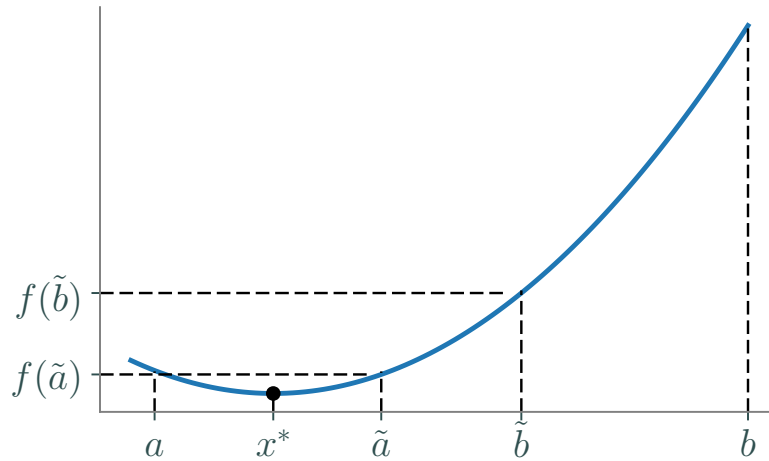


Figure 11.1: The unimodal  $f : [a, b] \rightarrow \mathbb{R}$  can be minimized with a golden section search. For the first iteration,  $f(\tilde{a}) < f(\tilde{b})$ , so  $x^* \in [a, \tilde{b}]$ . New values of  $\tilde{a}$  and  $\tilde{b}$  are then calculated from this new, smaller interval.

---

**Algorithm 11.1** 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 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
14:        break            $\triangleright$  Stop iterating if the approximation stops changing enough.
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 **tol**, and a maximum number of iterations **maxiter**. Use Algorithm 11.1 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

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{11.1}$$

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 (11.1) 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  $q'(x) = 0$ .

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \quad \implies \quad x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (11.1) using  $x_{k+1}$  for  $x$ . See Figure 11.2.

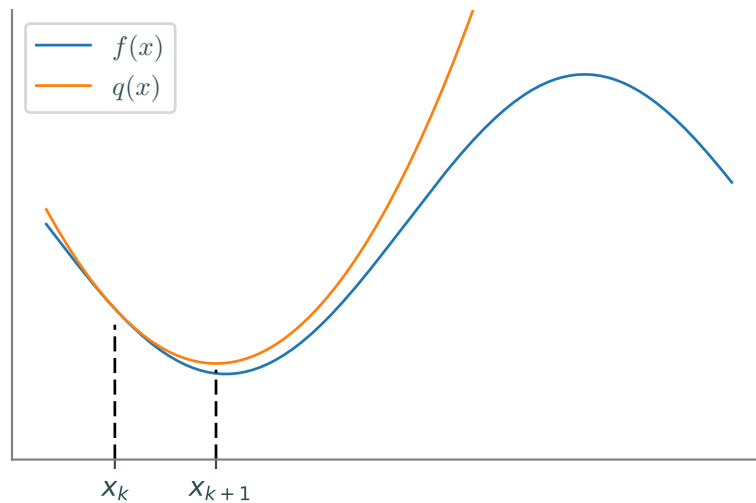


Figure 11.2: A quadratic approximation of  $f$  at  $x_k$ . The minimizer  $x_{k+1}$  of  $q$  is close to the minimizer of  $f$ .

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 `tol`, and a maximum number of iterations `maxiter`. Implement Newton's method using (11.1) 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.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \quad (11.2)$$

Inserting (11.2) into (11.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1} f'(x_k) - x_k f'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \quad (11.3)$$

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 (11.3) 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)
-3.2149595174761636
```

## 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.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (11.4)$$

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 (11.2) 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 11.3.

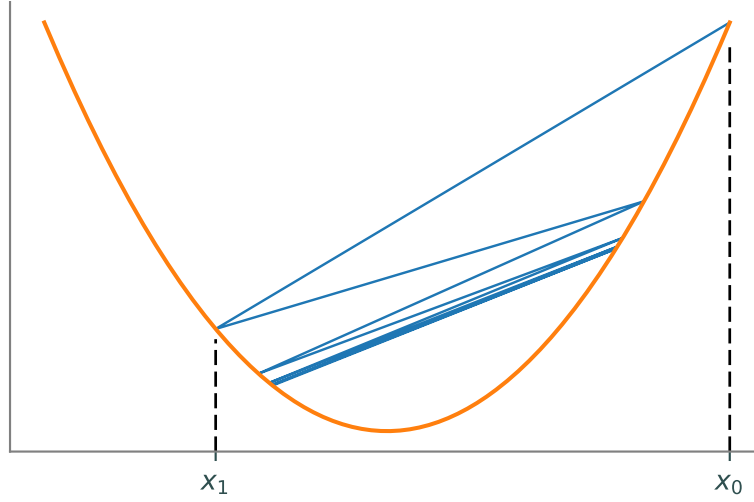


Figure 11.3: If the step size  $\alpha_k$  is too large, a descent method may repeatedly overstep the minimizer.

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*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k \quad (11.5)$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^\top \mathbf{p}_k \quad (11.6)$$

where  $0 < c_1 < c_2 < 1$  (for the best results, choose  $c_1 \ll c_2$ ). The condition (11.5) 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,

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small  $\alpha_k$  will always satisfy (11.5) since  $Df(\mathbf{x}_k)^\top \mathbf{p}_k < 0$  (as  $\mathbf{p}_k$  is a descent direction). The condition (11.6), 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 (11.6) to ensure that  $\alpha_k$  is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k| \leq c_2 |Df(\mathbf{x}_k)^\top \mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (11.6):

$$f(\mathbf{x}_k) + (1 - c) \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k,$$

where  $0 < c < 1$ . These conditions are very similar to the Wolfe conditions (the right inequality is (11.5)), but they do not require the calculation of the directional derivative  $Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k$ .

## 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 (11.5). This is usually sufficient, but if it finds  $\alpha$ 's that are too small, the algorithm can be modified to satisfy (11.6) or one of its variants.

---

### Algorithm 11.2 Backtracking using the Armijo Rule

---

```

1: procedure BACKTRACKING( $f, Df, \mathbf{x}_k, \mathbf{p}_k, \alpha, \rho, c$ )
2:    $\text{Dfp} \leftarrow Df(\mathbf{x}_k)^\top \mathbf{p}_k$  ▷ Compute these values only once.
3:    $\text{fx} \leftarrow f(\mathbf{x}_k)$ 
4:   while ( $f(\mathbf{x}_k + \alpha \mathbf{p}_k) > \text{fx} + c\alpha \text{Dfp}$ ) 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 11.2. 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 autograd import numpy as anp
>>> from autograd 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 = anp.array([150., .03, 40.])          # Current minimizer guessss.
>>> p = anp.array([-0.5, -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.))

```





# 12 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  $f$ .

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (12.1)$$

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 (12.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top \quad (12.2)$$

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 (12.2) 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.

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^\top)$$

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.

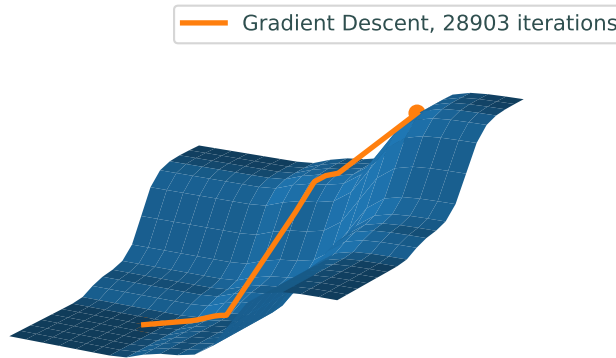


Figure 12.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

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

$$\mathbf{0} = Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b} \quad \Rightarrow \quad Q\mathbf{x} = \mathbf{b},$$

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 12.2. 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.

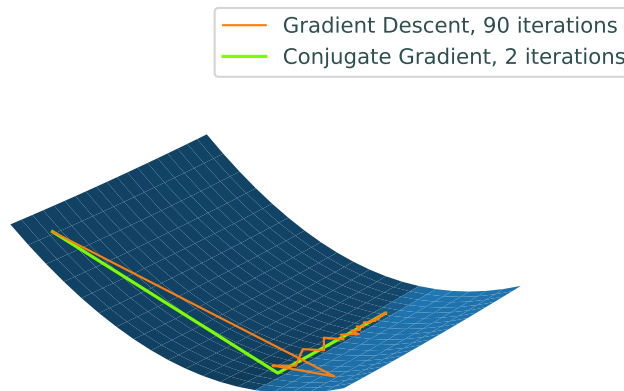


Figure 12.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

**Algorithm 12.1**


---

```

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 12.1 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.

```

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

# Generate Q, b, and the initial guess x0.
>>> n = 10
>>> 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  $\alpha_k = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ .
- 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 12.2

---

```

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:    $\alpha_0 \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_0 + \alpha \mathbf{d}_0)$ 
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:     $\alpha_k \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ 
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 defaultin to  $1e^{-5}$ , and a maximum number of iterations defaultin to 100. Use Algorithm 12.2 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])
```

## 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:

$$\mathbf{x}^* = \min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2,$$

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

$$\mathbf{x}^* = (A^\top A)^{-1} A^\top \mathbf{b}.$$

In other words, the minimizer solves the linear system

$$A^\top A \mathbf{x} = A^\top \mathbf{b}. \quad (12.3)$$

Since  $A$  has full column rank, it is invertible,  $A^\top A$  is positive definite, and for any non-zero vector  $\mathbf{z}$ ,  $A\mathbf{z} \neq 0$ . Therefore,  $\mathbf{z}^\top A^\top A \mathbf{z} = \|A\mathbf{z}\|^2 > 0$ . As  $A^\top A$  is positive definite, conjugate gradient can be used to solve Equation 12.3.

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.

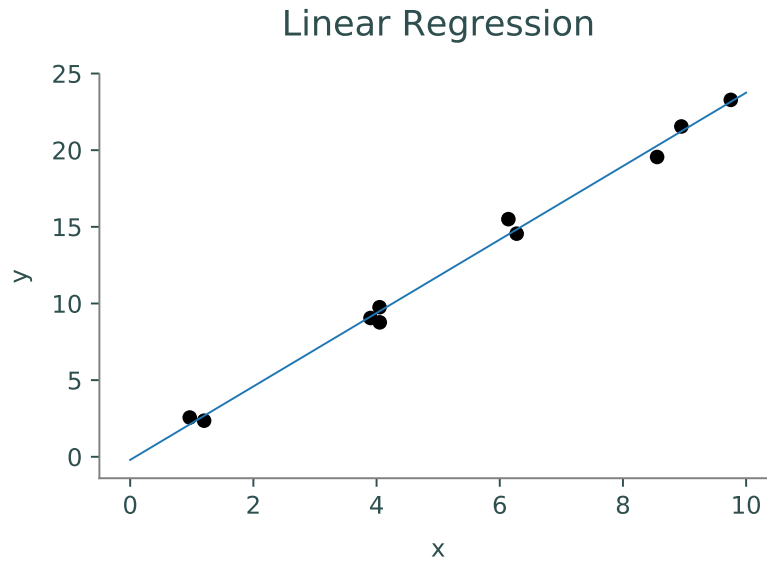


Figure 12.3: Solving the linear regression problem results in a best-fit hyperplane.

**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,

$$\mathbb{P}(y_i = 1 \mid x_{i,1}, \dots, x_{i,n}) = p_i,$$

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.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^m \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \quad (12.4)$$

**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  $\beta_0 \in \mathbb{R}^2$ . Define the negative log likelihood function as given in (12.4), then minimize it (with respect to  $\beta$ ) 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  $y = 1$ .

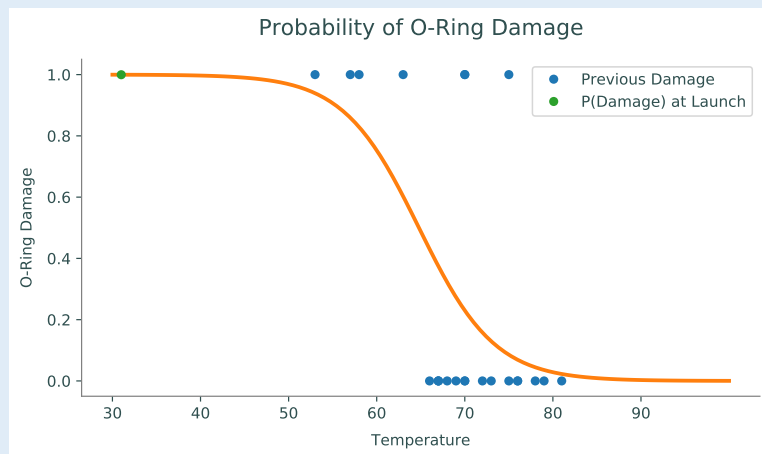
This class does not need an explicit constructor. You may assume that `predict()` will be called after `fit()`.



**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  $\beta_0 = [20, -1]^T$ . 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.





# 13 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.*

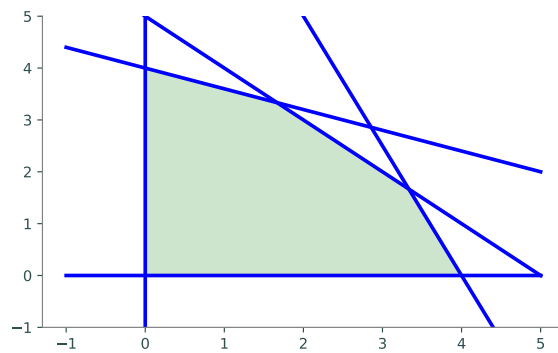
## Standard Form

The Simplex Algorithm accepts a linear constrained optimization problem, also called a *linear program*, in the form given below:

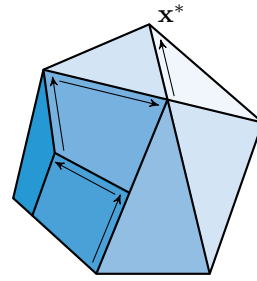
$$\begin{array}{ll}\text{minimize} & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} & A\mathbf{x} \preceq \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0}\end{array}$$

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 13.1). 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.



(a) The feasible region for a linear program with 2-dimensional constraints.



(b) The feasible region for a linear program with 3-dimensional constraints.

Figure 13.1: 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  $A\mathbf{x} \preceq \mathbf{b}$  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  $A\mathbf{x} \preceq \mathbf{b}$  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. One way is to mimic the dictionary that is often used when performing the Simplex Algorithm by hand. To do this we will set 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

$$\bar{A} = \begin{bmatrix} A & I_m \end{bmatrix},$$

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

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^T \\ \mathbf{b} & -\bar{A} \end{bmatrix} \quad (13.1)$$

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 ( $D$ ) as a NumPy array.

## 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:

$$D = \begin{bmatrix} 0 & \boxed{-3} & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}$$

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 a 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 13.1 (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, we first divide the pivot row by -3, and then zero out the two entries above the pivot element and the single entry below it:

$$\begin{aligned} & \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix} \rightarrow \\ & \begin{bmatrix} -5 & 0 & -1 & 0 & 1 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & -4/3 & 1 & -1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix} \rightarrow \\ & \begin{bmatrix} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & 4/3 & -1 & 1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 1/3 & 0 & -5/3 & 0 & 4/3 & -1 \end{bmatrix}. \end{aligned}$$

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. The dependent variables all have the value 0 in the objective function or first row of our dictionary array. The independent variables have values given by the first column of the dictionary. Specifically, the independent 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  
 $(-5.2, \{0: 1.6, 1: .2, 2: .6\}, \{3: 0, 4: 0\})$ .

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  $d_i$ .

Report the number of units that should be produced for each product. *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 Cube 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.

$$\begin{array}{llllll}
 \max & 2^{n-1}x_1 & +2^{n-2}x_2 & +\cdots & +2x_{n-1} & +x_n \\
 \text{subject to} & x_1 & & & & \leq 5 \\
 & 4x_1 & +x_2 & & & \leq 25 \\
 & 8x_1 & +4x_2 & +x_3 & & \leq 125 \\
 & \vdots & & & & \vdots \\
 & 2^n x_1 & +2^{n-1}x_2 & +\cdots & +4x_{n-1} & +x_n \leq 5
 \end{array}$$

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.

# 14 OpenGym AI

**Lab Objective:** *OpenGym AI is a module designed to learn and apply reinforcement learning. The purpose of this lab is to learn the variety of functionalities available in OpenGym AI and to implement them in various environments. Applying reinforcement learning techniques into OpenGym AI will take place in future labs.*

OpenGym AI 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 OpenGym AI, run the following code.

```
>>> pip install gym
```

## Environments

Each environment in OpenGym AI can be thought of as a different scenario where reinforcement learning can be applied. A catalog of all available environments can be found using the following code:

```
>>> from gym import envs
>>> print(envs.registry.all())
dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0), EnvSpec(ReversedAddition↵
-v0), EnvSpec(ReversedAddition3-v0), EnvSpec(DuplicatedInput-v0), EnvSpec(↵
Reverse-v0), EnvSpec(CartPole-v0),...]
```

Note that some of these environments require additional software. To learn more about each environment, its scenario and necessary software, visit [gym.openai.com/envs](http://gym.openai.com/envs).

To begin working in an environment, identify the name of the environment and initialize the environment. Once initialized, make sure to reset the environment. Resetting the environment is necessary to begin using the environment by putting everything in the correct starting position. For example, the environment "**NChain-v0**" presents a scenario where a player is traversing a chain with  $n$  states. Restarting the environment puts the player at the beginning of the chain. Once the environment is complete, make sure to close the environment. Closing the environment tells the computer to stop running the environment as to not run the environment in the background.

```
>>> import gym

>>> # Get NChain-v0 environment
env = gym.make('NChain-v0')

>>> # Reset the environment
>>> env.reset()
0

>>> # Close the environment
>>> env.close()
```

## Action Space

Once reset, the player in the environment can then perform actions from the action space. In "**NChain-v0**", the action space has 2 actions; move forward one state or return to the beginning of the chain. To perform an action, use the function `step`, which accepts the action as a parameter and returns an observation (more on those later). If the action space is discrete, then actions are defined as integers 0 through  $n$ , where  $n$  is the number of actions. The action each integer represents can be found in the documentation of each environment.

```
>>> # Determine the number of actions available
>>> env.action_space
Discrete(2)

>>> # Reset environment and perform a random action
>>> env.step(env.action_space.sample())
(1, 0, False, {})
```

However, not all action spaces are discrete. Consider the environment "**GuessingGame-v0**". The purpose of this game is to guess within 1% of a random number in the interval  $[-1000, 1000]$  in 200 guesses. Since the number is not required to be an integer, and each action is guessing a number, it does not make sense for the action space to be discrete. Rather this action space should be an interval. In OpenGym AI, this action space is described as an  $n$ -dimensional array `Box(n,)`. This means a feasible action is an  $n$ -dimensional vector. To identify the range of the box, use the attributes `high` and `low`. Thus, in the environment "**GuessingGame-v0**", the action space will be a 1-dimensional box with range  $[-1000, 1000]$ .

```
>>> # Get Guessing Game environment
env = gym.make("GuessingGame-v0")

>>> # Check size of action space
>>> env.action_space
Box(1,)

>>> # Check range of action space
>>> env.action_space.high
```

```
array([10000.], dtype=float32)

>>> env.action_space.low
array([-10000.], dtype=float32)
```

## Observation Space

The observation space contains all possible observations given an action. For example, in "**NChain-v0**", an observation would be the position of the player on the chain and in "**GuessingGame-v0**", the observation would be whether the guess is higher than, equal to, or lower than the target. The observation from each action can be found in the tuple returned by `step`. This tuple tells us the following information:

1. **observation**: The current state of the environment. For example, in "**GuessingGame-v0**", 0 indicates the guess is too high, 1 indicates the guess is on target, and 2 indicates the guess is too low.
2. **reward**: The reward given from the observation. In most environments, maximizing the total reward increases performance. For example, the reward for each observation is 0 in "**GuessingGame-v0**" unless the observation is within 1% of the target.
3. **done**: A boolean indicating if the observation terminates the environment.
4. **info**: Various information that may be helpful when debugging.

Consider the code below.

```
>>> env = gym.make("GuessingGame-v0")

>>> # Make a random guess
>>> env.step(env.action_space.sample())
(1, 0, False, {'guesses': 1, 'number': 524.50509074})
```

This tuple can be interpreted as follows:

1. The guess was too high.
2. The guess was not within 1% of the target.
3. The environment is not terminated.
4. Information that may help debugging (the number of guesses made so far and the target number).

**Problem 1.** The game Blackjack<sup>a</sup> is a card game where the player receives two cards from a facecard 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 facecard is considered 10. At 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. 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, the player loses.

The environment `"Blackjack-v0"` is an OpenGym AI environment that plays blackjack. The actions in the action space are 0 to stop drawing and 1 to draw another card. The observation (first entry in the tuple returned by `step`) is a tuple containing the total sum of the players hand, the first card of the computer's hand, and whether the player has an ace. The reward (second entry in the tuple returned by `step`) is 1 if the player wins, -1 if the player loses, and 0 if there is a draw.

Write a function `random_blackjack()` that accepts an integer  $n$ . Initialize `"Blackjack-v0"`  $n$  times and each time take random actions until the game is terminated. Return the percentage of games the player wins.

<sup>a</sup>For more on how to play Blackjack, see <https://en.wikipedia.org/wiki/Blackjack>.

## 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 OpenAI Gym are very well documented. Documentation for any given environment can be found through `gym.openai.com/envs` by clicking on the github link in the environment.

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 an OpenGym AI environment.

In addition to documentation, certain environments can be understood better through visualization. For example, the environment `"Acrobot-v1"` displays an inverted pendulum. Visualizing the environment allows the user to see the movement of the inverted pendulum as forces are applied to it. This can be done with the function `render()`. When using `render()`, ALWAYS use a try-finally block to close the environment. This ensures that the video rendering ends no matter what.

```
>>> # Get environment
>>> env = gym.make("Acrobot-v1")

>>> # Take random actions and visualize each action
>>> try:
>>>     env.reset()
>>>     done = False
>>>     while not done:
>>>         env.render()
>>>         obs, reward, done, info = env.step(env.action_space.sample())
>>>         if done:
>>>             break
>>> finally:
```

```
>>> env.close()
```



Figure 14.1: Rendering of "Acrobot-v1"

**Problem 2.** Write a function `blackjack()` which runs a naive algorithm to win blackjack. The function should receive an integer `n`. If the players hand is less than or equal to `n`, the player should draw another card. If the players hand is more than `n`, they should stop playing. Run the algorithm 10000 times and return the average reward. What value of `n` wins most on average?

## 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 "GuessingGame-v0". Each observation tells us whether the guess was too high or too low. After each observation, the interval where the target lies continues to get smaller. By choosing the midpoint of the current interval where the target lies, the true target can be identified much faster.

**Problem 3.** The environment "CartPole-v0" presents a cart with a vertical pole. The goal of the environment is to keep the pole vertical as long as possible. Write a function `cartpole()` which initializes the environment and keeps the pole vertical as long as possible based on the velocity of the tip of the pole. Render the environment at each step and return the time before the environment terminates. The time should be at least 2 seconds and on average be about 3 seconds.

(Hint: Use the documentation of the environment to determine the meaning of each action and observation. It can be found at <https://github.com/openai/gym/wiki/CartPole-v0>.)

**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 the action 2, can be driven backward with the action 0, and will be put in neutral with the action 1. Note that the car cannot immediately get up the hill because of gravity. 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. Each observation is a 2-dimensional array, containing the (x,y) position of the car. Using the position of the car, write a function `car()` that solves the `"MountainCar-v0"` environment. Render the environment at each step and return the time before the environment terminates. The time should be less than 3 seconds.

## Q-Learning

While naive methods like the ones above can be useful, reinforcement is a much better approach for using OpenAI Gym. 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.

A simple reinforcement method is called *Q-learning*. While the details of 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.

**Problem 5.** Write a function `taxi()` which initializes the environment `"Taxi-v2"`. 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 (see <https://gym.openai.com/envs/Taxi-v2/>). First, randomly act until the environment is done and calculate the reward. Then use `find_qtable()` to get the optimal Q-table of the environment. Set `alpha=.1`, `gamma=.6`, and `epsilon=.1`. Use the qtable to move through the environment and calculate the reward. Return the average reward of the random moves and the average reward of the Q-learning over 10000 iterations.

(Hint: Use the documentation found at [https://github.com/openai/gym/blob/master/gym/envs/toy\\_text/taxi.py](https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py) to understand the environment better).



# 15 CVXOPT

**Lab Objective:** *CVXOPT 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 CVXOPT 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

$$\begin{array}{ll}\text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & G\mathbf{x} \preceq \mathbf{h} \\ & A\mathbf{x} = \mathbf{b}.\end{array}$$

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  $y_i \in \mathbf{y}$ .

Define vector  $\mathbf{s} \succeq \mathbf{0}$  such that the constraint  $G\mathbf{x} + \mathbf{s} = \mathbf{h}$ . This vector is known as a *slack variable*. Since  $\mathbf{s} \succeq \mathbf{0}$ , the constraint  $G\mathbf{x} + \mathbf{s} = \mathbf{h}$  is equivalent to  $G\mathbf{x} \preceq \mathbf{h}$ .

With a slack variable, a new form of the linear program is found:

$$\begin{array}{ll}\text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & G\mathbf{x} + \mathbf{s} = \mathbf{h} \\ & A\mathbf{x} = \mathbf{b} \\ & \mathbf{s} \succeq \mathbf{0}.\end{array}$$

This is the formulation used by CVXOPT. It requires that the matrix  $A$  has full row rank, and that the block matrix  $[G \ A]^T$  has full column rank.

Consider the following example:

$$\begin{array}{ll}\text{minimize} & -4x_1 - 5x_2 \\ \text{subject to} & x_1 + 2x_2 \leq 3 \\ & 2x_1 + x_2 = 3 \\ & x_1, x_2 \geq 0\end{array}$$

Recall that all inequalities must be less than or equal to, so that  $G\mathbf{x} \preceq \mathbf{h}$ . Because the final two constraints are  $x_1, x_2 \geq 0$ , they need to be adjusted to be  $\leq$  constraints. This is easily done by multiplying by  $-1$ , resulting in the constraints  $-x_1, -x_2 \leq 0$ . If we define

$$G = \begin{bmatrix} 1 & 2 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 3 \end{bmatrix}$$

then we can express the constraints compactly as

$$\begin{array}{l} G\mathbf{x} \preceq \mathbf{h}, \\ A\mathbf{x} = \mathbf{b}, \end{array} \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

By adding a slack variable  $\mathbf{s}$ , we can write our constraints as

$$G\mathbf{x} + \mathbf{s} = \mathbf{h},$$

which matches the form discussed above.

To solve the problem using CVXOPT, initialize the arrays  $\mathbf{c}$ ,  $G$ ,  $\mathbf{h}$ ,  $A$ , and  $\mathbf{b}$  and pass them to the appropriate function. CVXOPT uses its own data type for an array or matrix. While similar to the NumPy array, it does have a few differences, especially when it comes to initialization. Below, we initialize CVXOPT matrices for  $\mathbf{c}$ ,  $G$ ,  $\mathbf{h}$ ,  $A$ , and  $\mathbf{b}$ . We then use the CVXOPT function for linear programming `solvers.lp()`, which accepts  $\mathbf{c}$ ,  $G$ ,  $\mathbf{h}$ ,  $A$ , and  $\mathbf{b}$  as arguments.

```
>>> from cvxopt import matrix, solvers

>>> c = matrix([-4., -5.])
>>> G = matrix([[1., -1., 0.],[2., 0., -1.]])
>>> h = matrix([ 3., 0., 0.])
>>> A = matrix([[2.],[1.]])
>>> b = matrix([3.])

>>> sol = solvers.lp(c, G, h, A, b)
      pcost      dcost      gap    pres    dres    k/t
0: -8.5714e+00 -1.4143e+01  4e+00  0e+00  3e-01  1e+00
1: -8.9385e+00 -9.2036e+00  2e-01  3e-16  1e-02  3e-02
2: -8.9994e+00 -9.0021e+00  2e-03  3e-16  1e-04  3e-04
3: -9.0000e+00 -9.0000e+00  2e-05  1e-16  1e-06  3e-06
4: -9.0000e+00 -9.0000e+00  2e-07  1e-16  1e-08  3e-08
Optimal solution found.
>>> print(sol['x'])
[ 1.00e+00]
[ 1.00e+00]
>>> print(sol['primal objective'])
-8.999999939019435
>>> print(type(sol['x']))
<class 'cvxopt.base.matrix'>
```

**ACHTUNG!**

CVXOPT matrices only accept floats. Other data types will raise a `TypeError`.

Additionally, CVXOPT matrices are initialized column-wise rather than row-wise (as in the case of NumPy). Alternatively, we can initialize the arrays first in NumPy (a process with which you should be familiar), and then simply convert them to the CVXOPT matrix data type.

```
>>> import numpy as np

>>> c = np.array([-4., -5.])
>>> G = np.array([[1., 2.], [-1., 0.], [0., -1]])
>>> h = np.array([3., 0., 0.])
>>> A = np.array([[2., 1.]])
>>> b = np.array([3.])

# Convert the arrays to the CVXOPT matrix type.
>>> c = matrix(c)
>>> G = matrix(G)
>>> h = matrix(h)
>>> A = matrix(A)
>>> b = matrix(b)
```

In this lab we will initialize non-trivial matrices first as NumPy arrays for consistency.

**NOTE**

Although it is often helpful to see the progress of each iteration of the algorithm, you may suppress this output by first running,

```
solvers.options['show_progress'] = False
```

The function `solvers.lp()` returns a dictionary containing useful information. For now, we will only focus on the value of `x` and the primal objective value (i.e. the minimum value achieved by the objective function).

**ACHTUNG!**

Note that the minimizer `x` returned by the `solvers.lp()` function is a `cvxopt.base.matrix` object. `np.ravel()` is a NumPy function that takes an object and returns its values as a flattened NumPy array. Use `np.ravel()` to return all minimizers in this lab as flattened NumPy arrays.

**Problem 1.** Solve the following convex optimization problem:

$$\begin{aligned} & \text{minimize} && 2x_1 + x_2 + 3x_3 \\ & \text{subject to} && x_1 + 2x_2 \geq 3 \\ & && 2x_1 + 10x_2 + 3x_3 \geq 10 \\ & && x_1 \geq 0 \\ & && x_2 \geq 0 \\ & && x_3 \geq 0 \end{aligned}$$

Return the minimizer  $\mathbf{x}$  and the primal objective value.

(Hint: make the necessary adjustments so that all inequality constraints are  $\leq$  rather than  $\geq$ ).

## $l_1$ Norm

The  $l_1$  norm is defined

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|.$$

A  $l_1$  minimization problem is minimizing a vector's  $l_1$  norm, while fitting certain constraints. It can be written in the following form:

$$\begin{aligned} & \text{minimize} && \|\mathbf{x}\|_1 \\ & \text{subject to} && A\mathbf{x} = \mathbf{b}. \end{aligned}$$

This problem can be converted into a linear program by introducing an additional vector  $\mathbf{u}$  of length  $n$ . Define  $\mathbf{u}$  such that  $|x_i| \leq u_i$ . Thus,  $-u_i - x_i \leq 0$  and  $-u_i + x_i \leq 0$ . These two inequalities can be added to the linear system as constraints. Additionally, this means that  $\|\mathbf{x}\|_1 \leq \|\mathbf{u}\|_1$ . So minimizing  $\|\mathbf{u}\|_1$  subject to the given constraints will in turn minimize  $\|\mathbf{x}\|_1$ . This can be written as follows:

$$\begin{aligned} & \text{minimize} && \begin{bmatrix} \mathbf{1}^\top & \mathbf{0}^\top \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} \\ & \text{subject to} && \begin{bmatrix} -I & I \\ -I & -I \\ -I & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} \preceq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \\ & && \begin{bmatrix} \mathbf{0} & A \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} = \mathbf{b}. \end{aligned}$$

Solving this gives values for the optimal  $\mathbf{u}$  and the optimal  $\mathbf{x}$ , but we only care about the optimal  $\mathbf{x}$ .

**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. Remember to first discard the unnecessary  $u$  values from the minimizer.

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.

## 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 15.3. 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 15.1: Number of pianos available at each supply center

Demand Center	Number of pianos needed
4	5
5	8

Table 15.2: 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 15.3: Cost of transporting one piano from a supply center to a demand center

A system of constraints is defined for the variables  $p_1, p_2, p_3, p_4, p_5$ , and  $p_6$ . First, there cannot be a negative number of pianos so the variables must be nonnegative. Next, the Tables 15.1 and 15.2 define the following three supply constraints and two demand constraints:

$$\begin{aligned}
 p_1 + p_2 &= 7 \\
 p_3 + p_4 &= 2 \\
 p_5 + p_6 &= 4 \\
 p_1 + p_3 + p_5 &= 5 \\
 p_2 + p_4 + p_6 &= 8
 \end{aligned}$$

The objective function is the number of pianos shipped from each location multiplied by the respective cost (found in Table 15.3):

$$4p_1 + 7p_2 + 6p_3 + 8p_4 + 8p_5 + 9p_6.$$

## 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.

Recall the variables are nonnegative, so  $p_1, p_2, p_3, p_4, p_5, p_6 \geq 0$ . Thus,  $G$  and  $\mathbf{h}$  constrain the variables to be non-negative. Because CVXOPT uses the format  $G\mathbf{x} \preceq \mathbf{h}$ , we see that this inequality must be multiplied by  $-1$ . So,  $G$  must be a  $6 \times 6$  identity matrix multiplied by  $-1$ , and

$\mathbf{h}$  is a column vector of zeros. Since the supply and demand constraints are equality constraints, they are  $A$  and  $\mathbf{b}$ . Initialize these arrays and solve the linear program by entering the code below.

```
>>> c = matrix(np.array([4., 7., 6., 8., 8., 9.]))
>>> G = matrix(-1*np.eye(6))
>>> h = matrix(np.zeros(6))
>>> A = matrix(np.array([[1.,1.,0.,0.,0.,0.],
                        [0.,0.,1.,1.,0.,0.],
                        [0.,0.,0.,0.,1.,1.],
                        [1.,0.,1.,0.,1.,0.],
                        [0.,1.,0.,1.,0.,1.])))
>>> b = matrix(np.array([7., 2., 4., 5., 8.]))
>>> sol = solvers.lp(c, G, h, A, b)
      pcost      dcost      gap    pres    dres    k/t
0:  8.9500e+01  8.9500e+01  2e+01  2e-16  2e-01  1e+00
1:  8.7023e+01  8.7044e+01  3e+00  1e-15  3e-02  2e-01
Terminated (singular KKT matrix).
>>> print(sol['x'])
[ 4.31e+00]
[ 2.69e+00]
[ 3.56e-01]
[ 1.64e+00]
[ 3.34e-01]
[ 3.67e+00]
>>> print(sol['primal objective'])
87.023
```

Notice that some problems occurred. First, CVXOPT alerted us to the fact that the algorithm terminated prematurely (due to a singular matrix). Second, the minimizer and solution obtained do not consist of integer entries.

So what went wrong? Recall that the matrix  $A$  is required to have full row rank, but we can easily see that the rows of  $A$  are linearly dependent. We rectify this by converting the last row of the equality constraints into two *inequality* constraints, so that the remaining equality constraints define a new matrix  $A$  with linearly independent rows.

This is done as follows:

Suppose we have the equality constraint

$$x_1 + 2x_2 - 3x_3 = 4.$$

This is equivalent to the pair of inequality constraints

$$\begin{aligned}x_1 + 2x_2 - 3x_3 &\leq 4, \\x_1 + 2x_2 - 3x_3 &\geq 4.\end{aligned}$$

The linear program requires only  $\leq$  constraints, so we obtain the pair of constraints

$$\begin{aligned}x_1 + 2x_2 - 3x_3 &\leq 4, \\-x_1 - 2x_2 + 3x_3 &\leq -4.\end{aligned}$$

Apply this process to the last equality constraint of the transportation problem. Then define a new matrix  $G$  with several additional rows (to account for the new inequality constraints), a new vector  $\mathbf{h}$  with more entries, a smaller matrix  $A$ , and a smaller vector  $\mathbf{b}$ .

**Problem 3.** Solve the transportation problem by converting the last equality constraint into an inequality constraint. 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

$$\begin{aligned}\text{minimize} \quad & \frac{1}{2}\mathbf{x}^\top Q\mathbf{x} + \mathbf{r}^\top \mathbf{x} \\ \text{subject to} \quad & G\mathbf{x} \preceq \mathbf{h} \\ & A\mathbf{x} = \mathbf{b},\end{aligned}$$

where  $Q$  is a positive semidefinite symmetric matrix. In this formulation, we require again that  $A$  has full row rank and that the block matrix  $[Q \ G \ A]^\top$  has full column rank.

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

$$\begin{aligned}\frac{1}{2}\mathbf{x}^\top Q\mathbf{x} + \mathbf{r}^\top \mathbf{x} &= \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^\top \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} d \\ e \end{bmatrix}^\top \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \frac{1}{2}ax_1^2 + bx_1x_2 + \frac{1}{2}cx_2^2 + dx_1 + ex_2\end{aligned}$$

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 CVXOPT function for quadratic programming `solvers.qp()`.

```
>>> Q = matrix(np.array([[4., 2.], [2., 2.]])
>>> r = matrix([1., -1.])
>>> sol=solvers.qp(Q, r)
>>> print(sol['x'])
[-1.00e+00]
[ 1.50e+00]
>>> print sol['primal objective']
-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.)

**Problem 5.** The  $l_2$  minimization problem is to

$$\begin{array}{ll} \text{minimize} & \|\mathbf{x}\|_2 \\ \text{subject to} & \mathbf{Ax} = \mathbf{b}. \end{array}$$

This problem is equivalent to a quadratic program, since  $\|\mathbf{x}\|_2 = \mathbf{x}^T \mathbf{x}$ . Write a function that accepts a matrix  $A$  and vector  $\mathbf{b}$  and solves the  $l_2$  minimization problem. Return the minimizer  $\mathbf{x}$  and the primal objective value.

To test your function, use the matrix  $A$  and vector  $\mathbf{b}$  from Problem 2. The minimizer is approximately  $\mathbf{x} = [0.966, 2.169, 0.809, 0.888]^T$  and the minimum primal objective value is approximately 7.079.

## 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 one night 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 15.4, 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:  $s$ , number of servings
- 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	Serving Size $s$	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 15.4: 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 CVXOPT to identify how much of each food item a college student should eat to minimize cost spent each day. Return the minimizing vector and the total amount of money spent.

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 CVXOPT at <http://cvxopt.org/index.html>.

<sup>1</sup>[urlhttps://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/pdv.html](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>

# 16 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 16.1 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).

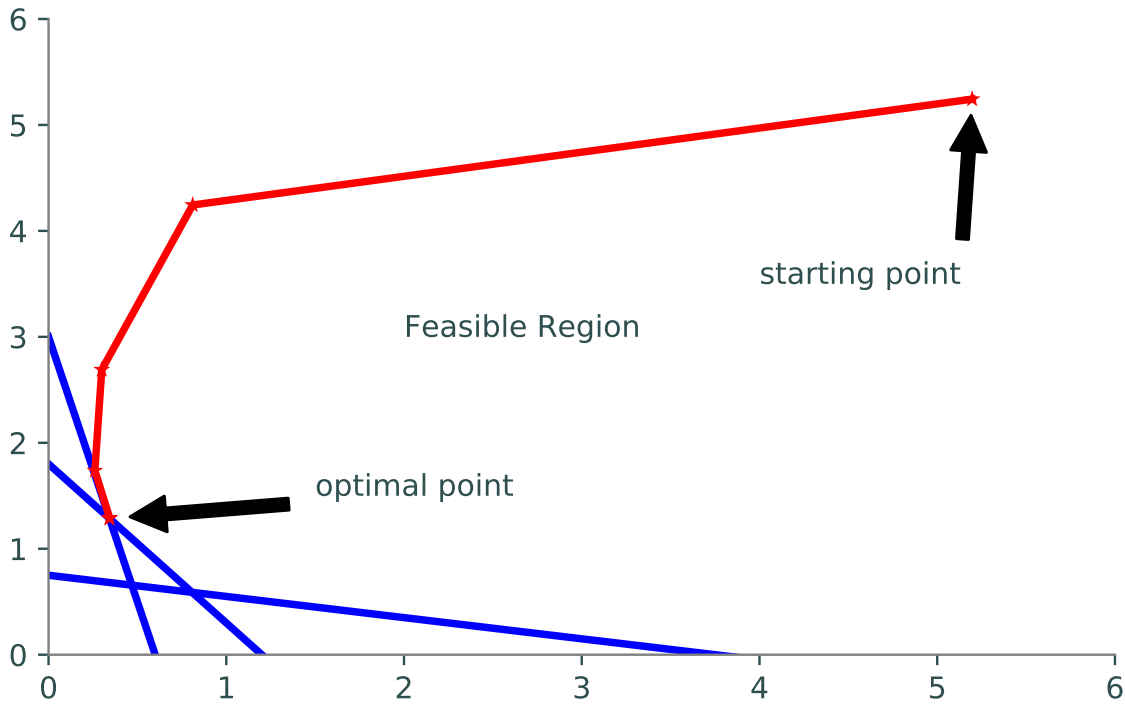


Figure 16.1: A path traced by an Interior Point algorithm.

## 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:

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && A\mathbf{x} = \mathbf{b} \\ & && \mathbf{x} \succeq \mathbf{0}. \end{aligned}$$

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:

$$\begin{aligned} & \text{maximize} && \mathbf{b}^T \boldsymbol{\lambda} \\ & \text{subject to} && A^T \boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c} \\ & && \boldsymbol{\mu}, \boldsymbol{\lambda} \succeq \mathbf{0}, \end{aligned}$$

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

$$\begin{aligned} A^T \boldsymbol{\lambda} + \boldsymbol{\mu} &= \mathbf{c} \\ A\mathbf{x} &= \mathbf{b} \\ x_i \mu_i &= 0, \quad i = 1, 2, \dots, n, \\ \mathbf{x}, \boldsymbol{\mu} &\succeq \mathbf{0}. \end{aligned}$$

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:

$$F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := \begin{bmatrix} A^T \boldsymbol{\lambda} + \boldsymbol{\mu} - \mathbf{c} \\ A\mathbf{x} - \mathbf{b} \\ M\mathbf{x} \end{bmatrix} = \mathbf{0},$$

$$(\mathbf{x}, \boldsymbol{\mu} \succeq \mathbf{0}),$$

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.

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  $(\mathbf{x}^{*T}, \boldsymbol{\lambda}^{*T}, \boldsymbol{\mu}^{*T})$ , 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:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \\ \Delta \boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \quad (16.1)$$

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 16.1 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:

$$\nu = \frac{\mathbf{x}^\top \boldsymbol{\mu}}{n}$$

The idea is to use Newton's method to identify a direction that strictly decreases  $\nu$ . Thus instead of solving Equation 16.1, we solve:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \\ \Delta \boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma \nu \mathbf{e} \end{bmatrix} \quad (16.2)$$

where  $\mathbf{e} = (1, 1, \dots, 1)^\top$  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  $(\Delta \mathbf{x}^\top, \Delta \boldsymbol{\lambda}^\top, \Delta \boldsymbol{\mu}^\top)$  by solving Equation 16.2. 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:

$$\begin{aligned} \alpha_{\max} &= \min\{-\mu_i / \Delta \mu_i \mid \Delta \mu_i < 0\} \\ \delta_{\max} &= \min\{-x_i / \Delta x_i \mid \Delta x_i < 0\} \end{aligned}$$

If all values of  $\Delta \mu$  are nonnegative, let  $\alpha_{\max} = 1$ . Likewise, if all values of  $\Delta x$  are nonnegative, let  $\delta_{\max} = 1$ . Next, we back off from these maximum step lengths slightly:

$$\begin{aligned} \alpha &= \min(1, 0.95\alpha_{\max}) \\ \delta &= \min(1, 0.95\delta_{\max}). \end{aligned}$$

These are our final step lengths. Thus, the next point in the iteration is given by:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + \delta \Delta \mathbf{x}_k \\ (\boldsymbol{\lambda}_{k+1}, \boldsymbol{\mu}_{k+1}) &= (\boldsymbol{\lambda}_k, \boldsymbol{\mu}_k) + \alpha (\Delta \boldsymbol{\lambda}_k, \Delta \boldsymbol{\mu}_k). \end{aligned}$$

<sup>1</sup> $\nu$  is the Greek letter for  $n$ , pronounced “nu.”

**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  $\beta_{\max}$  (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 naïvely 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  $\mathbf{c}^T \mathbf{x}^*$ .

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():
    """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:
        m (int >= n): number of desired constraints.
        n (int): dimension of space in which to optimize.
    Outputs:
        A ((m,n+m) ndarray): Constraint matrix.
        b ((m,) ndarray): Constraint vector.
        c ((n+m,) ndarray): Objective function with m trailing 0s.
        x ((n,) ndarray): The first 'n' terms of the solution to the LP.
    """
    A = np.random.random((m,n))*20 - 10
    A[A[:,-1]<0] *= -1
    x = np.random.random(n)*10
    b = np.zeros(m)
    b[:n] = A[:n,:] @ x
    b[n:] = A[n:,:] @ x + np.random.random(m-n)*10
    c = np.zeros(n+m)
    c[:n] = A[:n,:].sum(axis=0)/n
    A = np.hstack((A, np.eye(m)))
    return A, b, -c, x
```

```
>>> m, n = 7, 5
>>> A, b, c, x = randomLP(m, n)
>>> point, value = interiorPoint(A, b, c)
>>> np.allclose(x, point[:n])
True
```

## 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:

$$y_i = \beta^T \mathbf{x}_i + b, \quad i = 1, 2, \dots, m,$$



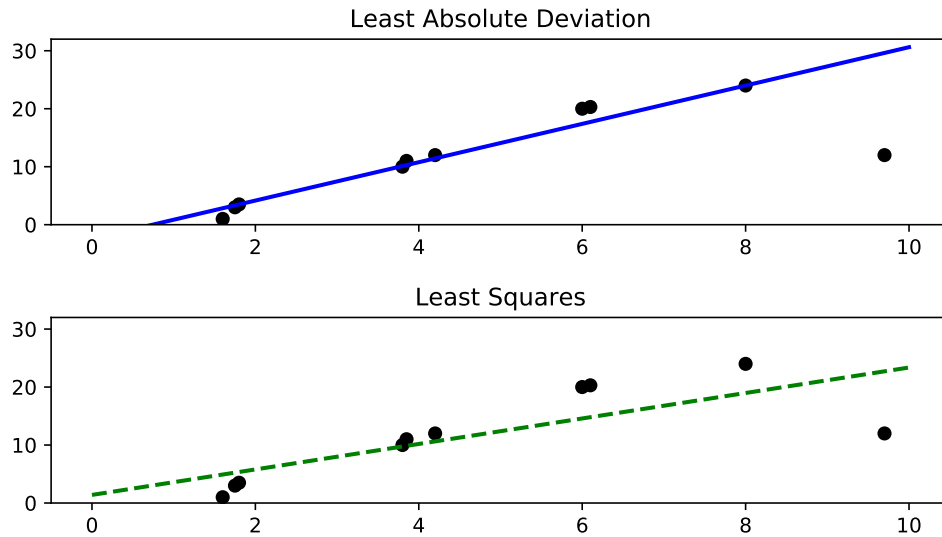


Figure 16.2: Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

where  $\beta \in \mathbb{R}^n$  and  $b \in \mathbb{R}$ . The error between the data and the proposed linear model is given by

$$\sum_{i=1}^n |\beta^\top \mathbf{x}_i + b - y_i|,$$

and we seek to choose the parameters  $\beta, b$  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 16.2.

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  $\beta^\top \mathbf{x}_i + b - y_i$  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.

## 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  $|\beta^\top \mathbf{x}_i + b - y_i|$ , and we require this variable to satisfy  $u_i \geq |\beta^\top \mathbf{x}_i + b - y_i|$ . This constraint is not yet linear, but we can split it into an equivalent set of two linear constraints:

$$\begin{aligned} u_i &\geq \beta^\top \mathbf{x}_i + b - y_i, \\ u_i &\geq y_i - \beta^\top \mathbf{x}_i - b. \end{aligned}$$

The  $u_i$  are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^m u_i \\ \text{subject to} \quad & u_i \geq \beta^\top \mathbf{x}_i + b - y_i, \\ & u_i \geq y_i - \beta^\top \mathbf{x}_i - b. \end{aligned}$$

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:

$$\begin{aligned} u_i - \beta^\top \mathbf{x}_i - b - s_{2i-1} &= -y_i, \\ u_i + \beta^\top \mathbf{x}_i + b - s_{2i} &= y_i, \\ s_{2i-1}, s_{2i} &\geq 0. \end{aligned}$$

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:

$$\begin{aligned} u_i - \beta_1^\top \mathbf{x}_i + \beta_2^\top \mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i, \\ u_i + \beta_1^\top \mathbf{x}_i - \beta_2^\top \mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i, \\ \beta_1, \beta_2 &\succeq \mathbf{0}; u_i, b_1, b_2, s_{2i-1}, s_{2i} \geq 0. \end{aligned}$$

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:

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{v} \\ & \text{subject to} && A\mathbf{v} = \mathbf{y}, \\ & && \mathbf{v} \succeq \mathbf{0}, \end{aligned}$$

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  $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^T$ . 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)
```

# 17

## Interior Point 2: Quadratic Programs

**Lab Objective:** *Interior point methods originated as an alternative to the Simplex method for solving linear optimization problems. However, they can also be adapted to treat convex optimization problems in general. In this lab we implement a primal-dual Interior Point method for convex quadratic constrained optimization and explore applications in elastic membrane theory and finance.*

### Quadratic Optimization Problems

A *quadratic constrained optimization problem* differs from a linear constrained optimization problem only in that the objective function is quadratic rather than linear. We can pose such a problem as follows:

$$\begin{aligned} &\text{minimize} && \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ &\text{subject to} && A \mathbf{x} \succeq \mathbf{b}, \\ &&& G \mathbf{x} = \mathbf{h}. \end{aligned}$$

We will restrict our attention to quadratic programs involving positive semidefinite quadratic terms (in general, indefinite quadratic objective functions admit many local minima, complicating matters considerably). Such problems are called *convex*, since the objective function is convex. To simplify the exposition, we will also only allow inequality constraints (generalizing to include equality constraints is not difficult). Thus, we have the problem

$$\begin{aligned} &\text{minimize} && \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ &\text{subject to} && A \mathbf{x} \succeq \mathbf{b} \end{aligned}$$

where  $Q \in \mathbb{R}^{n \times n}$  is a positive semidefinite matrix,  $A \in \mathbb{R}^{m \times n}$ ,  $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^m$ .

The Lagrangian function for this problem is:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} - \boldsymbol{\mu}^T (A \mathbf{x} - \mathbf{b}), \quad (17.1)$$

where  $\boldsymbol{\mu} \in \mathbb{R}^m$  is the Lagrange multiplier.

We also introduce a nonnegative slack vector  $\mathbf{y} \in \mathbb{R}^m$  to change the inequality  $A\mathbf{x} - \mathbf{b} \succeq \mathbf{0}$  into the equality  $A\mathbf{x} - \mathbf{b} - \mathbf{y} = \mathbf{0}$ .

Then the complete set of KKT conditions are:

$$\begin{aligned} Q\mathbf{x} - A^\top \boldsymbol{\mu} + \mathbf{c} &= \mathbf{0}, \\ A\mathbf{x} - \mathbf{b} - \mathbf{y} &= \mathbf{0}, \\ y_i \mu_i &= 0, \quad i = 1, 2, \dots, m, \\ \mathbf{y}, \boldsymbol{\mu} &\succeq \mathbf{0}. \end{aligned}$$

## Quadratic Interior Point Method

The Interior Point method we describe here is an adaptation of the method we used with linear programming. Define  $Y = \text{diag}(y_1, y_2, \dots, y_m)$ ,  $M = \text{diag}(\mu_1, \mu_2, \dots, \mu_m)$ , and let  $\mathbf{e} \in \mathbb{R}^m$  be a vector of all ones. Then the roots of the function

$$F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q\mathbf{x} - A^\top \boldsymbol{\mu} + \mathbf{c} \\ A\mathbf{x} - \mathbf{y} - \mathbf{b} \\ Y M \mathbf{e} \end{bmatrix} = \mathbf{0}, \quad (\mathbf{y}, \boldsymbol{\mu}) \succeq \mathbf{0}$$

satisfy the KKT conditions. The derivative matrix of this function is given by

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q & 0 & -A^\top \\ A & -I & 0 \\ 0 & M & Y \end{bmatrix},$$

and the duality measure  $\nu$  for this problem is

$$\nu = \frac{\mathbf{y}^\top \boldsymbol{\mu}}{m}.$$

## Search Direction

We calculate the search direction for this algorithm in the spirit of Newton's Method; this is the same way that we did in the linear programming case. That is, we solve the system:

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{y} \\ \Delta \boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma \nu \mathbf{e} \end{bmatrix}, \quad (17.2)$$

where  $\sigma \in [0, 1)$  is the centering parameter.

**Problem 1.** Create a function `qInteriorPoint()`. It should accept the arrays  $Q, \mathbf{c}, A$ , and  $\mathbf{b}$ , a tuple of arrays `guess` giving initial estimates for  $\mathbf{x}, \mathbf{y}$ , and  $\boldsymbol{\mu}$  (this will be explained later), along with the keyword arguments `niter=20` and `tol=1e-16`.

In this function, calculate the search direction. Create  $F$  and  $DF$  as described above, and calculate the search direction  $(\Delta \mathbf{x}^\top, \Delta \mathbf{y}^\top, \Delta \boldsymbol{\mu}^\top)$  by solving Equation 17.2. Use  $\sigma = \frac{1}{10}$  for the centering parameter.

(Hint: What are the dimensions of  $F$  and  $DF$ ?)

## Step Length

Now that we have our search direction, we select a step length. We want to step nearly as far as possible without violating the nonnegativity constraints. However, we back off slightly from the maximum allowed step length because an overly greedy step at one iteration may prevent a descent step at the next iteration. Thus, we choose our step size

$$\alpha = \max\{a \in (0, 1] \mid \tau(\mathbf{y}, \boldsymbol{\mu}) + a(\Delta\mathbf{y}, \Delta\boldsymbol{\mu}) \succeq \mathbf{0}\},$$

where  $\tau \in (0, 1)$  controls how much we back off from the maximal step length. For now, choose  $\tau = 0.95$ . In general,  $\tau$  can be made to approach 1 at each successive iteration. This may speed up convergence in some cases.

We wish to step nearly as far as possible without violating the problem's constraints, as to remain in the interior of the feasible region. First, we calculate the maximum allowable step lengths for  $\boldsymbol{\mu}$  and  $\mathbf{y}$ .

$$\begin{aligned}\beta_{\max} &= \min\{-\mu_i / \Delta\mu_i \mid \Delta\mu_i < 0\} \\ \delta_{\max} &= \min\{-y_i / \Delta y_i \mid \Delta y_i < 0\}\end{aligned}$$

If all of the entries of  $\Delta\boldsymbol{\mu}$  are nonnegative, we let  $\beta_{\max} = 1$ . Likewise, if all the entries of  $\Delta\mathbf{y}$  are nonnegative, let  $\delta_{\max} = 1$ . Next, we back off from these maximum step lengths slightly:

$$\begin{aligned}\beta &= \min(1, \tau\beta_{\max}) \\ \delta &= \min(1, \tau\delta_{\max}) \\ \alpha &= \min(\beta, \delta)\end{aligned}$$

This  $\alpha$  is our final step length. Thus, the next point in the iteration is given by:

$$(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \boldsymbol{\mu}_k) + \alpha(\Delta\mathbf{x}_k, \Delta\mathbf{y}_k, \Delta\boldsymbol{\mu}_k).$$

This completes one iteration of the algorithm.

## Initial Point

The starting point  $(\mathbf{x}_0, \mathbf{y}_0, \boldsymbol{\mu}_0)$  has an important effect on the convergence of the algorithm. The code listed below will calculate an appropriate starting point:

```
def startingPoint(G, c, A, b, guess):
    """
    Obtain an appropriate initial point for solving the QP
    .5 x\trp Gx + x\trp c s.t. Ax >= b.
    Parameters:
        G -- symmetric positive semidefinite matrix shape (n,n)
        c -- array of length n
        A -- constraint matrix shape (m,n)
        b -- array of length m
        guess -- a tuple of arrays (x, y, l) of lengths n, m, and m, resp.
```

```

Returns:
    a tuple of arrays (x0, y0, l0) of lengths n, m, and m, resp.
"""
m,n = A.shape
x0, y0, l0 = guess

# initialize linear system
N = np.zeros((n+m+m, n+m+m))
N[:n,:n] = G
N[:n, n+m:] = -A.T
N[n:n+m, :n] = A
N[n:n+m, n:n+m] = -np.eye(m)
N[n+m:, n:n+m] = np.diag(l0)
N[n+m:, n+m:] = np.diag(y0)
rhs = np.empty(n+m+m)
rhs[:n] = -(G.dot(x0) - A.T.dot(l0)+c)
rhs[n:n+m] = -(A.dot(x0) - y0 - b)
rhs[n+m:] = -(y0*l0)

sol = la.solve(N, rhs)
dx = sol[:n]
dy = sol[n:n+m]
dl = sol[n+m:]

y0 = np.maximum(1, np.abs(y0 + dy))
l0 = np.maximum(1, np.abs(l0+dl))

return x0, y0, l0

```

Notice that we still need to provide a tuple of arrays `guess` as an argument. Do your best to provide a reasonable guess for the array  $\mathbf{x}$ , and we suggest setting  $\mathbf{y}$  and  $\boldsymbol{\mu}$  equal to arrays of ones. We summarize the entire algorithm below.

- 
- 1: **procedure** INTERIOR POINT METHOD FOR QP
  - 2:   Choose initial point  $(\mathbf{x}_0, \mathbf{y}_0, \boldsymbol{\mu}_0)$ .
  - 3:   **while**  $k < \text{niters}$  and  $\nu < \text{tol}$ : **do**
  - 4:     Calculate the duality measure  $\nu$ .
  - 5:     Solve 17.2 for the search direction  $(\Delta \mathbf{x}_k, \Delta \mathbf{y}_k, \Delta \boldsymbol{\mu}_k)$ .
  - 6:     Calculate the step length  $\alpha$ .
  - 7:      $(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \boldsymbol{\mu}_k) + \alpha(\Delta \mathbf{x}_k, \Delta \mathbf{y}_k, \Delta \boldsymbol{\mu}_k)$ .
- 

**Problem 2.** Complete the implementation of `qInteriorPoint()`. Return the optimal point  $\mathbf{x}$  as well as the final objective function value.



Test your algorithm on the simple problem

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2 \\ & \text{subject to} && -x_1 - x_2 \geq -2, \\ & && x_1 - 2x_2 \geq -2, \\ & && -2x_1 - x_2 \geq -3, \\ & && x_1, x_2 \geq 0. \end{aligned}$$

In this case, we have for the objective function matrix  $Q$  and vector  $\mathbf{c}$ ,

$$Q = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -2 \\ -6 \end{bmatrix}.$$

The constraint matrix  $A$  and vector  $\mathbf{b}$  are given by:

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

Use  $\mathbf{x} = [.5, .5]$  as the initial guess. The correct minimizer is  $\left[\frac{2}{3}, \frac{4}{3}\right]$ .

(Hint: You may want to print out the duality measure  $\nu$  to check the progress of the iteration).

## NOTE

The Interior Point methods presented in this and the preceding labs are only special cases of the more general Interior Point algorithm. The general version can be used to solve many convex optimization problems, provided that one can derive the corresponding KKT conditions and duality measure  $\nu$ .

## Application: Optimal Elastic Membranes

The properties of elastic membranes (stretchy materials like a thin rubber sheet) are of interest in certain fields of mathematics and various sciences. A mathematical model for such materials can be used by biologists to study interfaces in cellular regions of an organism or by engineers to design tensile structures. Often we can describe configurations of elastic membranes as a solution to an optimization problem. As a simple example, we will find the shape of a large circus tent by solving a quadratic constrained optimization problem using our Interior Point method.

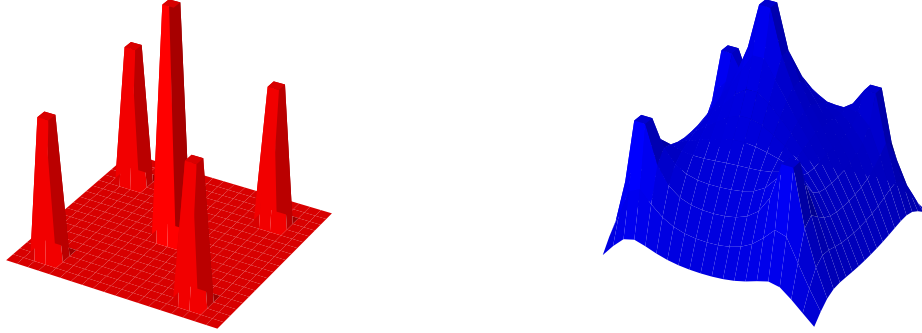


Figure 17.1: Tent pole configuration (left) and optimal elastic tent (right).

Imagine a large circus tent held up by a few poles. We can model the tent by a square two-dimensional grid, where each grid point has an associated number that gives the height of the tent at that point. At each grid point containing a tent pole, the tent height is constrained to be at least as large as the height of the tent pole. At all other grid points, the tent height is simply constrained to be greater than zero (ground height). In Python, we can store a two-dimensional grid of values as a simple two-dimensional array. We can then flatten this array to give a one-dimensional vector representation of the grid. If we let  $\mathbf{x}$  be a one-dimensional array giving the tent height at each grid point, and  $L$  be the one-dimensional array giving the underlying tent pole structure (consisting mainly of zeros, except at the grid points that contain a tent pole), we have the linear constraint:

$$\mathbf{x} \succeq L.$$

The theory of elastic membranes claims that such materials tend to naturally minimize a quantity known as the *Dirichlet energy*. This quantity can be expressed as a quadratic function of the membrane. Since we have modeled our tent with a discrete grid of values, this energy function has the form

$$\frac{1}{2}\mathbf{x}^\top H\mathbf{x} + \mathbf{c}^\top \mathbf{x},$$

where  $H$  is a particular positive semidefinite matrix closely related to Laplace's Equation,  $\mathbf{c}$  is a vector whose entries are all equal to  $-(n-1)^{-2}$ , and  $n$  is the side length of the grid. Our circus tent is therefore given by the solution to the quadratic constrained optimization problem:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\mathbf{x}^\top H\mathbf{x} + \mathbf{c}^\top \mathbf{x} \\ \text{subject to} & \mathbf{x} \succeq L. \end{array}$$

See Figure 17.1 for an example of a tent pole configuration and the corresponding tent.

We provide the following function for producing the Dirichlet energy matrix  $H$ .

```

from scipy.sparse import spdiags
def laplacian(n):
    """Construct the discrete Dirichlet energy matrix H for an n x n grid."""
    data = -1*np.ones((5, n**2))
    data[2,:] = 4
    data[1, n-1::n] = 0
    data[3, ::n] = 0
    diags = np.array([-n, -1, 0, 1, n])
    return spdiags(data, diags, n**2, n**2).toarray()

```

Now we initialize the tent pole configuration for a grid of side length  $n$ , as well as initial guesses for  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mu$ .

```

# Create the tent pole configuration.
>>> L = np.zeros((n,n))
>>> L[n//2-1:n//2+1,n//2-1:n//2+1] = .5
>>> m = [n//6-1, n//6, int(5*(n/6.))-1, int(5*(n/6.))]
>>> mask1, mask2 = np.meshgrid(m, m)
>>> L[mask1, mask2] = .3
>>> L = L.ravel()

# Set initial guesses.
>>> x = np.ones((n,n)).ravel()
>>> y = np.ones(n**2)
>>> mu = np.ones(n**2)

```

We leave it to you to initialize the vector  $\mathbf{c}$ , the constraint matrix  $A$ , and to initialize the matrix  $H$  with the `laplacian()` function. We can solve and plot the tent with the following code:

```

>>> from matplotlib import pyplot as plt
>>> from mpl_toolkits.mplot3d import axes3d

# Calculate the solution.
>>> z = qInteriorPoint(H, c, A, L, (x,y,mu))[0].reshape((n,n))

# Plot the solution.
>>> domain = np.arange(n)
>>> X, Y = np.meshgrid(domain, domain)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(111, projection='3d')
>>> ax1.plot_surface(X, Y, z, rstride=1, cstride=1, color='r')
>>> plt.show()

```

**Problem 3.** Solve the circus tent problem with the tent pole configuration given above, for grid side length  $n = 15$ . Plot your solution.

## Application: Markowitz Portfolio Optimization

Suppose you have a certain amount of money saved up, with no intention of consuming it any time soon. What will you do with this money? If you hide it somewhere in your living quarters or on your person, it will lose value over time due to inflation, not to mention you run the risk of burglary or accidental loss. A safer choice might be to put the money into a bank account. That way, there is less risk of losing the money, plus you may even add to your savings through interest payments from the bank. You could also consider purchasing bonds from the government or stocks from various companies, which come with their own sets of risks and returns. Given all of these possibilities, how can you invest your money in such a way that maximizes the return (i.e. the wealth that you gain over the course of the investment) while still exercising caution and avoiding excessive risk? Economist and Nobel laureate Harry Markowitz developed the mathematical underpinnings and answer to this question in his work on modern portfolio theory.

A *portfolio* is a set of investments over a period of time. Each investment is characterized by a financial asset (such as a stock or bond) together with the proportion of wealth allocated to the asset. An asset is a random variable, and can be described as a sequence of values over time. The variance or spread of these values is associated with the risk of the asset, and the percent change of the values over each time period is related to the return of the asset. For our purposes, we will assume that each asset has a positive risk, i.e. there are no *riskless* assets available.

Stated more precisely, our portfolio consists of  $n$  risky assets together with an allocation vector  $\mathbf{x} = (x_1, \dots, x_n)^\top$ , where  $x_i$  indicates the proportion of wealth we invest in asset  $i$ . By definition, the vector  $\mathbf{x}$  must satisfy

$$\sum_{i=1}^n x_i = 1.$$

Suppose the  $i$ th asset has an expected rate of return  $\mu_i$  and a standard deviation  $\sigma_i$ . The total return on our portfolio, i.e. the expected percent change in our invested wealth over the investment period, is given by

$$\sum_{i=1}^n \mu_i x_i.$$

We define the risk of this portfolio in terms of the covariance matrix  $Q$  of the  $n$  assets:

$$\sqrt{\mathbf{x}^\top Q \mathbf{x}}.$$

The covariance matrix  $Q$  is always positive semidefinite and captures the variance and correlations of the assets.

Given that we want our portfolio to have a prescribed return  $R$ , there are many possible allocation vectors  $\mathbf{x}$  that make this possible. It would be wise to choose the vector minimizing the risk. We can state this as a quadratic program:

$$\begin{aligned} & \text{minimize} && \frac{1}{2} \mathbf{x}^\top Q \mathbf{x} \\ & \text{subject to} && \sum_{i=1}^n x_i = 1 \\ & && \sum_{i=1}^n \mu_i x_i = R. \end{aligned}$$

Note that we have slightly altered our objective function for convenience, as minimizing  $\frac{1}{2}\mathbf{x}^\top Q\mathbf{x}$  is equivalent to minimizing  $\sqrt{\mathbf{x}^\top Q\mathbf{x}}$ . The solution to this problem will give the portfolio with least risk having a return  $R$ . Because the components of  $\mathbf{x}$  are not constrained to be nonnegative, the solution may have some negative entries. This indicates short selling those particular assets. If we want to disallow short selling, we simply include nonnegativity constraints, stated in the following problem:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}\mathbf{x}^\top Q\mathbf{x} \\ & \text{subject to} && \sum_{i=1}^n x_i = 1 \\ & && \sum_{i=1}^n \mu_i x_i = R \\ & && \mathbf{x} \succeq \mathbf{0}. \end{aligned}$$

Each return value  $R$  can be paired with its corresponding minimal risk  $\sigma$ . If we plot these risk-return pairs on the risk-return plane, we obtain a hyperbola. In general, the risk-return pair of any portfolio, optimal or not, will be found in the region bounded on the left by the hyperbola. The positively-sloped portion of the hyperbola is known as the *efficient frontier*, since the points there correspond to optimal portfolios. Portfolios with risk-return pairs that lie to the right of the efficient frontier are inefficient portfolios, since we could either increase the return while keeping the risk constant, or we could decrease the risk while keeping the return constant. See Figure 17.2.

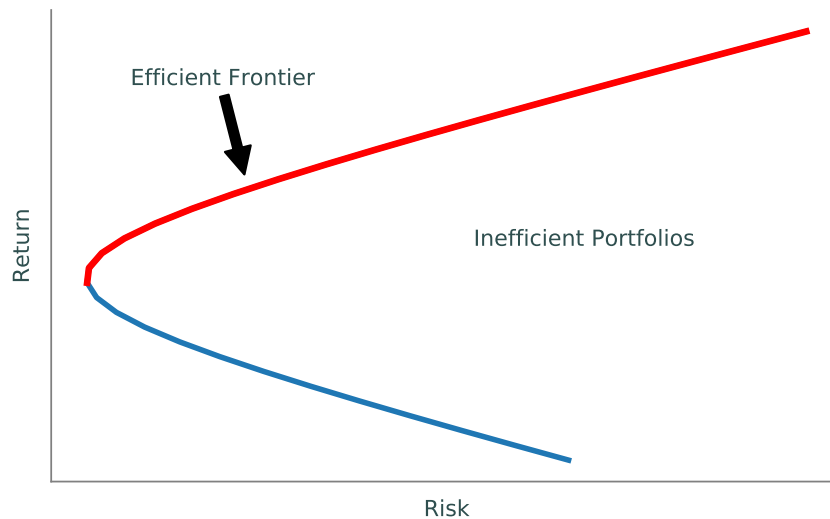


Figure 17.2: Efficient frontier on the risk-return plane.

One weakness of this model is that the risk and return of each asset is in general unknown. After all, no one can predict the stock market with complete certainty. There are various ways of estimating these values given past stock prices, and we take a very straightforward approach. Suppose for each asset, we have  $k$  previous return values of the asset. That is, for asset  $i$ , we have the data vector

$$\mathbf{y}^i = [y_1^i, \dots, y_k^i]^\top.$$

We estimate the expected rate of return for asset  $i$  by simply taking the average of  $y_1, \dots, y_k$ , and we estimate the variance of asset  $i$  by taking the variance of the data. We can estimate the covariance matrix for all assets by taking the covariance matrix of the vectors  $y^1, \dots, y^n$ . In this way, we obtain estimated values for each  $\mu_i$  and  $Q$ .

**Problem 4.** The text file `portfolio.txt` contains historical stock data for several assets (U.S. bonds, gold, S&P 500, etc). In particular, the first column gives the years corresponding to the data, and the remaining eight columns give the historical returns of eight assets over the course of these years. Use this data to estimate the covariance matrix  $Q$  as well as the expected rates of return  $\mu_i$  for each asset. Assuming that we want to guarantee an expected return of  $R = 1.13$  for our portfolio, find the optimal portfolio both with and without short selling.

Since the problem contains both equality and inequality constraints, use the QP solver in `CVXOPT` rather than your `qInteriorPoint()` function.

Hint: Use `numpy.cov()` to compute  $Q$ .

# 18 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, when hiring a secretary, how many people should you interview before hiring the current interviewer? Or how many people should you date before you get married? These problems try to determine at what 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. That 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 we choose the best partner when we have passed over the first  $t-1$  candidates with an optimal policy. In other words, we have dated  $t-1$  people and want to know the probability that the  $t^{\text{th}}$  person is the one we should marry. Note that the probability that the  $t^{\text{th}}$  person is not the best candidate is  $\frac{t-1}{t}$  and the probability that they are is  $\frac{1}{t}$ . If the  $t^{\text{th}}$  person is not the best out of the first  $t$ , then probability they are the best overall is 0 and the probability they are not is  $V(t)$ . If the  $t^{\text{th}}$  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,

$$V(t-1) = \frac{t-1}{t} \max\{0, V(t)\} + \frac{1}{t} \max\left\{\frac{t}{N}, V(t)\right\} = \max\left\{\frac{t-1}{t}V(t) + \frac{1}{N}, V(t)\right\}. \quad (18.1)$$

Notice that (18.1) 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 (18.1) 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

$$\begin{aligned} V(4) &= 0, \\ V(3) &= \max \left\{ \frac{3}{4}V(4) + \frac{1}{4}, 0 \right\} = .25, \\ V(2) &= \max \left\{ \frac{2}{3}V(3) + \frac{1}{4}, .25 \right\} = .4166, \\ V(1) &= \max \left\{ \frac{1}{4}, .4166 \right\} = .4166. \end{aligned}$$

In this case, the maximum expected value is .4166 and the stopping point is  $t = 2$ . It is also useful to look at the optimal stopping percentage of people to date before getting married. In this case, it is  $2/4 = .5$ .

**Problem 1.** Write a function that accepts a number of candidates  $N$ . Calculate the expected values of choosing candidate  $t$  for  $t = 0, 1, \dots, N-1$ .

Return the highest expected value  $V(t_0)$  and the optimal stopping point  $t_0$ .  
(Hint: Since Python starts indices at 0, the first candidate is  $t = 0$ .)

Check your answer for  $N = 4$  with the example detailed above.

**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 .367.

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

$$\sum_{t=0}^T \beta^t u(c_t).$$

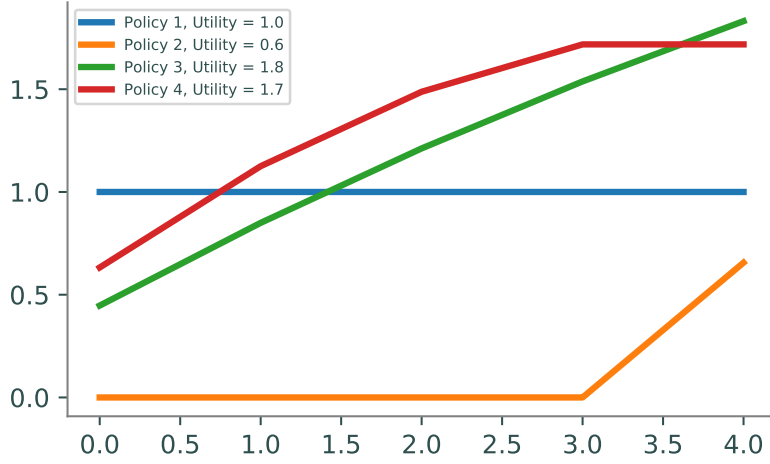


Figure 18.1: Plots for various policies with  $u(x) = \sqrt{x}$  and  $\beta = 0.9$ . Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same 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.

$$\begin{aligned} \max_{\mathbf{c}} \quad & \sum_{t=0}^T \beta^t u(c_t) \\ \text{subject to} \quad & \sum_{t=0}^T c_t = W \\ & c_t \geq 0. \end{aligned} \tag{18.2}$$

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  $b$ .

$$\begin{aligned} V(a, b, W) = \max_{\mathbf{c}} \quad & \sum_{t=a}^b \beta^t u(c_t) \\ \text{subject to} \quad & \sum_{t=a}^b c_t = W \\ & c_t \geq 0. \end{aligned}$$

$V(0, T, W)$  gives how much utility we gain in  $T$  days and is the same as Equation 18.2.

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.

$$V(t, T, W_t) = \max_{W_{t+1}} (u(W_t - W_{t+1}) + \beta V(t, T - 1, W_{t+1})) \quad (18.3)$$

where  $u(W_t - W_{t+1})$  is the value gained from eating  $W_t - W_{t+1}$  cake at time  $t$ .

Let  $\mathbf{w} = [0 \quad \frac{1}{N} \quad \dots \quad \frac{N-1}{N} \quad 1]^\top$ . 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  $w = [0, .25, .5, .75, 1]^\top$ , 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 18.3 is a new value function problem with  $a = t, b = T - 1$ , and  $W = W_{t+1}$ , making 18.3 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 (18.3),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, -1, w_j)) = u(w_i). \quad (18.4)$$

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 (18.4) as follows,

$$A_{i(T-1)} = V(0, 1, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, 0, w_j)) = \max_{w_j} (u(w_i - w_j) + \beta A_{jT}). \quad (18.5)$$

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  $t = T - 1$ ,

$$CV_{ij}^{T-1} = u(w_i - w_j) + \beta A_{jT}. \quad (18.6)$$

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 (18.5) and (18.6) shows

$$CV_{ij}^t = u(w_i - w_j) + \beta A_{j(t+1)}. \quad A_{it} = \max_j (CV_{ij}^t). \quad (18.7)$$

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 18.2: 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 (18.7),
- 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 .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 .25. So we eat 1 slice of cake at  $t = 1$ . We continue this pattern to find the optimal policy  $\mathbf{c} = [.25 \ .25 \ .25 \ .25]$ .

#### 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.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} \\ \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.5} \\ \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.25} & \sqrt{0.75} \\ \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} + \beta^3\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.5} & \sqrt{1} \end{bmatrix}$$

**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.)

**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  $u$ .





# 19 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  $\epsilon$  error of the solution. These methods are significantly faster than computing the exact solution using dynamic programming. We demonstrate two iterative methods, value iteration (VI) and policy iteration (PI), and use them to solve a deterministic Markov decision process.*

## Dynamic Optimization

Many dynamic optimization problem 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,  $\mathbb{T} = 0, 1, \dots, T$ .  $S$  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 less a reward is worth in the future. 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 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  $p(s_t, a_t, s_{t+1}) \leq 1$ , given that there can be multiple possible next states for taking a given action in a given state.

The dynamic optimization problem is

$$\max_{\mathbf{a}} \sum_{t=0}^T \beta^t u(s_t, a_t) \quad (19.1)$$

$$\text{subject to } s_{t+1} = g(s_t, a_t) \forall t. \quad (19.2)$$

The cake eating problem described in the previous lab 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.

$$P[s][a] = [(p(s, a, \bar{s}), \bar{s}, u(s, a, \bar{s}), is\_terminal), \dots]$$

There is a tuple for each  $\bar{s} \in N_{s,a}$  in the list. The final entry in the tuple,  $is\_terminal$ , indicates if the  $\bar{s}$  is a stopping point.

## 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  $Up = 3$ .

### ACHTUNG!

It is important to remember that the actions do not correspond to the states the robot is in after the action. When the robot is in state 0 and takes action 1, he is then in state 2.

$A_s$  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  $g(s_t, a_t) = s_{t+1}$  can be explicitly defined for each  $s, a$  where  $s_{t+1}$  is the new state after moving.

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 2,  $-1$  if the robot moves into 0 from 1, and  $1$  when it reaches the end, 3. We define the reward function to be  $u(s_t, a_t, s_{t+1}) = u(s_{t+1})$ . Since this is a deterministic model,  $p(s_t, a_t, s_{t+1}) = p(s_{t+1}) = 1, \forall s, a$ .

0	1
2	3

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.

$$\begin{array}{ll} P[0][0] = [(0, 0, 0, \text{False})] & P[2][0] = [(0, 2, -1, \text{False})] \\ P[0][1] = [(1, 2, -1, \text{False})] & P[2][1] = [(0, 2, -1, \text{False})] \\ P[0][2] = [(1, 1, 0, \text{False})] & P[2][2] = [(1, 3, 1, \text{True})] \\ P[0][3] = [(0, 0, 0, \text{False})] & P[2][3] = [(1, 0, 0, \text{False})] \\ P[1][0] = [(1, 0, -1, \text{False})] & P[3][0] = [(0, 0, 0, \text{True})] \\ P[1][1] = [(1, 3, 1, \text{True})] & P[3][1] = [(0, 0, 0, \text{True})] \\ P[1][2] = [(0, 0, 0, \text{False})] & P[3][2] = [(0, 0, 0, \text{True})] \\ P[1][3] = [(0, 0, 0, \text{False})] & P[3][3] = [(0, 0, 1, \text{True})] \end{array}$$

We define the *value function*  $V(s)$  to be the maximum possible reward of starting in state  $s$ . Then using Bellman's optimality equation,

$$V(s) = \max_{a \in A_s} \{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V(\bar{s})) \}. \quad (19.3)$$

The summation occurs when it is a stochastic Markov process. For example, if the robot is in the top left corner and moves right, we could have that the probability the robot actually moves right is  $.5$ . In this case,  $P[0][2] = [(.5, 1, 0, \text{False}), (.5, 2, -1, \text{False})]$ . This 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:

$$[T(f)](s) = \max_{a \in A_s} \{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta f(\bar{s})) \} \quad (19.4)$$

It can be shown that 19.1 is the fixed "point" of our operator  $T$ . A result of contraction mappings is that there exists a unique solution to 19.4.

$$V_{k+1}(s_i) = [T(V_k)](s_i) = \max_{a \in A_s} \{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V_k(\bar{s})) \} \quad (19.5)$$

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. We calculate  $V_1(s)$  from the robot example above.

$$\begin{aligned} V_1(0) &= \max_{a \in A_0} \{ \sum_{\bar{s} \in N_{0,a}} p(\bar{s}) * (u(\bar{s}) + V_0(\bar{s})) \} \\ &= \max \{ p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2)) \} \\ &= \max \{ 1(-1 + 0), 1(0 + 0) \} \\ &= \max \{ -1, 0 \} \\ &= 0 \\ V_1(1) &= \max \{ p(0) * (u(0) + V_0(0)), p(2) * (u(2) + V_0(2)) \} \\ &= \max \{ 1(0 + 0), 1(1 + 0) \} \\ &= 1 \end{aligned}$$

Calculating  $V_1(2)$  and  $V_1(3)$  gives  $V_1 = [0, 1, 1, 0]$ . Repeating the process,  $V_2 = [1, 1, 1, 0]$ , which is the solution. It means that maximum reward the robot can achieve by starting on square  $i$  is  $V_2(i)$ .

**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  $\beta \in (0, 1)$ , the tolerance amount  $\varepsilon$ , and the maximum number of iterations `maxiter`. 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 he can expect, but not how to get it. The policy vector,  $\mathbf{c}$ , is found by using the policy function:  $\pi : \mathbb{R} \rightarrow \mathbb{R}$ .  $\pi(s)$  is the action we should take while in state  $s$  to maximize reward. We can modify the Bellman equation using  $V^*(s)$  to find  $\pi$ :

$$\pi(s) = \operatorname{argmax}_{a \in A_s} \{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V^*(\bar{s})) \} \quad (19.6)$$

Using value iteration, we found  $V^* = [1, 1, 1, 0]$  in the example above. We find  $\pi(0)$  by looking at actions 1 and 2 (since actions 0 and 3 have probability 0).

$$\begin{aligned} \pi(0) &= \operatorname{argmax}_{1,2} \{ p(1) * (u(1) + V^*(1)), p(2) * (u(2) + V^*(2)) \} \\ &= \operatorname{argmax} \{ 1 * (-1 + 1), 1 * (0 + 1) \} \\ &= \operatorname{argmax} \{ 0, 1 \} \\ &= 2. \end{aligned}$$

So when the robot is in state 0, he should take action 2, moving *Right*. This avoids the  $-1$  penalty for moving *Down* into square 2. Similarly,

$$\pi(1) = \operatorname{argmax}_{0,1} \{ 1 * (0 + 1), 1 * (1 + 1) \} = \operatorname{argmax} \{ 1, 2 \} = 1.$$

The policy corresponding to the optimal reward is  $[2, 1, 2, 0]$ . The robot should move to square 3 if possible, avoiding 2 because it has a negative reward. Since 3 is terminal, it does not matter what  $\pi(3)$  is. We set it to 0 for convenience.

### 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 to 1. 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 (19.1) 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 policy iteration.

That is, given a specific policy function  $\pi_k$ , we can modify (19.1) by assuming that the policy function is the optimal choice. This process, called *policy evaluation*, evaluates the value function for a given policy.

$$V_{k+1}(s) = \max_{a \in [A_s]} \{ \sum_{\bar{s} \in N_{s,a}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s}))) \} = \sum_{\bar{s} \in N_{s, \pi(s)}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s}))) \quad (19.7)$$

The last equality occurs because in state  $s$ , the robot should choose the action that maximizes reward, which is  $\pi(s)$  by definition.

**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  $\varepsilon$ . 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. Called *policy improvement*, this step is the same method used in value iteration to find the policy.

Given an initial guess for our policy function,  $\pi_0$ , we calculate the corresponding value function using (19.7), and then use (19.6) to improve our policy function. The algorithm for policy function iteration can be summarized as follows:

---

**Algorithm 19.1** Policy Iteration

---

```

1: procedure POLICY ITERATION FUNCTION( $P, nS, nA, \beta, tol, \text{maxiter}$ )
2:    $\pi_0 \leftarrow [\pi_0(w_0), \pi_0(w_1), \dots, \pi_0(w_N)]$   $\triangleright$  Common choice is  $\pi_0(w_i) = w_{i-1}$  with  $\pi_0(0) = 0$ 
3:   for  $k = 1, 2, \dots, \text{maxiter}$  do  $\triangleright$  Iterate only maxiter times at most.
4:     for  $s \in S$  do  $\triangleright$  Policy evaluation
5:        $V_{k+1}(s) = \sum_{\bar{s} \in N_{s, \pi(s)}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})))$   $\triangleright$  compute_policy_v.
6:     for  $s \in S$  do  $\triangleright$  Policy improvement.
7:        $\pi_{k+1}(s) = \operatorname{argmax}_{a \in A_s} \{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V_{k+1}(\bar{s})) \}$   $\triangleright$  extract_policy.
8:     if  $\|\pi_{k+1} - \pi_k\| < \varepsilon$  then
9:       break  $\triangleright$  Stop iterating if the policy doesn't change enough.
10:  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  $\varepsilon$ , and the maximum number of iterations **maxiter**. 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 1 and 2.

## The Frozen Lake Problem

For the rest of the lab, we will be using OpenAI Gym environments. They can be installed using conda or pip.

```
$ pip install gym[all]
```

In the Frozen Lake problem, you and your friends tossed a frisbee onto a mostly frozen lake. The lake is divided into an  $N \times N$  grid where the top left hand corner is the start, the bottom right hand corner is the end, and the other squares are either frozen or holes. To retrieve the frisbee, you must successfully navigate around the melted ice without falling. The possible actions are left, right, up, and down. Since ice is slippery, you won't always move in the intended direction. Hence, this is a stochastic Markov process, i.e.  $p(s_t, a_t, s_{t+1}) < 1$ . If you fall, your reward is 0. If you succeed, your reward is 1. There are two scenarios:  $N = 4$  and  $N = 8$ .

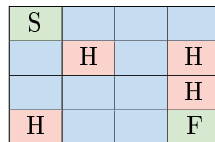


Figure 19.1: Diagram of the 4x4 scenario. The green  $S$  represents the starting point and the green  $F$  represents the frisbee. Red squares marked  $H$  are holes. Blue squares are pieces of the frozen lake.

This problem can be found in two environments in OpenAI Gym. To run the  $4 \times 4$  scenario, use `env_name='FrozenLake-v0'`. For the  $8 \times 8$  scenario, use `env_name='FrozenLake8x8-v0'`.

### Using Gym

To use gym, we import it and create an environment based on the built-in gym environment. The FrozenLake environment has 3 important attributes,  $P$ ,  $nS$ , and  $nA$ .  $P$  is the same dictionary we used in the previous problems.  $nS$  and  $nA$  are the number of states and actions respectively. We can calculate the optimal policy with value iteration or policy iteration using these attributes. Since the ice is slippery, this policy will not always result in a reward of 1.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name = 'FrozenLake-v0'
env = gym.make(env_name).env
# Find number of states and actions
number_of_states = env.nS
number_of_actions = env.nA
```

**Problem 5.** Write a function that runs `value_iteration` and `policy_iteration` on FrozenLake. It should accept a boolean `basic_case` defaulting to `True` and an integer `n` defaulting to 1000 that indicates how many times to run the simulation. If `basic_case` is `True`, run the 4x4 scenario, If not, run the 8x8 scenario. Calculate the value function and policy for the environment using both value iteration and policy iteration. Return the policies generated by value iteration and the policy and value function generated by policy iteration. Set the mean total rewards to 0 and return them as well.

The gym environments have built-in functions that allow us to simulate each step of the environment. Before running a scenario in gym, always put it in the starting state by calling `env.reset()`. To simulate moving, call `env.step`.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name = 'FrozenLake-v0'
env = gym.make(env_name).env
# Put environment in starting state
obs = env.reset()
# Take a step in the optimal direction and update variables
obs, reward, done, _ = env.step(int(policy[obs]))
```

The step function returns four values: observation, reward, done, info. The observation is an environment-specific object representing the observation of the environment. For FrozenLake, this is the current state. When we step, or take an action, we get a new observation, or state, as well as the reward for taking that action. If we fall into a hole or reach the frisbee, the simulation is over so we are done. When we are done, the boolean `done` is `True`. The info value is a dictionary of diagnostic information. It will not be used in this lab.

**Problem 6.** Write a function `run_simulation()` that takes in the environment `env`, a policy `policy`, a boolean `render`, and a discount factor  $\beta$ . Calculate the total reward for the policy for one simulation using `env.reset` and `env.step()`. Stop the simulation when `done` is `True`. (Hint: When calculating reward, use  $\beta^k$ .)

Modify `frozen_lake()` to call `run_simulation()` for both the value iteration policy and the policy iteration policy  $M$  times. Return the policy generated by value iteration, the mean total reward for the policy generated by value iteration, the value function generated by policy iteration, the policy generated by policy iteration, and the mean total reward for the policy generated by policy iteration.

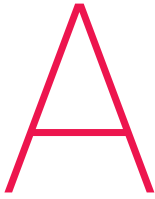




Part II

# Appendices





# Getting Started

The labs in this curriculum aim to introduce computational and mathematical concepts, walk through implementations of those concepts in Python, and use industrial-grade code to solve interesting, relevant problems. Lab assignments are usually about 5–10 pages long and include code examples (yellow boxes), important notes (green boxes), warnings about common errors (red boxes), and about 3–7 exercises (blue boxes). Get started by downloading the lab manual(s) for your course from <http://foundations-of-applied-mathematics.github.io/>.

## Submitting Assignments

### Labs

Every lab has a corresponding specifications file with some code to get you started and to make your submission compatible with automated test drivers. Like the lab manuals, these materials are hosted at <http://foundations-of-applied-mathematics.github.io/>.

Download the .zip file for your course, unzip the folder, and move it somewhere where it won't get lost. This folder has some setup scripts and a collection of folders, one per lab, each of which contains the specifications file(s) for that lab. See **Student-Materials/wiki/Lab-Index** for the complete list of labs, their specifications and data files, and the manual that each lab belongs to.

### ACHTUNG!

Do **not** move or rename the lab folders or the enclosed specifications files; if you do, the test drivers will not be able to find your assignment. Make sure your folder and file names match **Student-Materials/wiki/Lab-Index**.

To submit a lab, modify the provided specifications file and use the file-sharing program specified by your instructor (discussed in the next section). The instructor will drop feedback files in the lab folder after grading the assignment. For example, the Introduction to Python lab has the specifications file `PythonIntro/python_intro.py`. To complete that assignment, modify `PythonIntro/python_intro.py` and submit it via your instructor's file-sharing system. After grading, the instructor will create a file called `PythonIntro/PythonIntro_feedback.txt` with your score and some feedback.

## Homework

Non-lab coding homework should be placed in the `_Homework/` folder and submitted like a lab assignment. Be careful to name your assignment correctly so the instructor (and test driver) can find it. The instructor may drop specifications files and/or feedback files in this folder as well.

## Setup

### ACHTUNG!

We strongly recommend using a Unix-based operating system (Mac or Linux) for the labs. Unix has a true bash terminal, works well with git and python, and is the preferred platform for computational and data scientists. It is possible to do this curriculum with Windows, but expect some road bumps along the way.

There are two ways to submit code to the instructor: with git (<http://git-scm.com/>), or with a file-syncing service like Google Drive. Your instructor will indicate which system to use.

## Setup With Git

*Git* is a program that manages updates between an online code repository and the copies of the repository, called *clones*, stored locally on computers. If git is not already installed on your computer, download it at <http://git-scm.com/downloads>. If you have never used git, you might want to read a few of the following resources.

- Official git tutorial: <https://git-scm.com/docs/gittutorial>
- Bitbucket git tutorials: <https://www.atlassian.com/git/tutorials>
- GitHub git cheat sheet: [services.github.com/.../github-git-cheat-sheet.pdf](https://services.github.com/.../github-git-cheat-sheet.pdf)
- GitLab git tutorial: <https://docs.gitlab.com/ce/gitlab-basics/start-using-git.html>
- Codecademy git lesson: <https://www.codecademy.com/learn/learn-git>
- Training video series by GitHub: <https://www.youtube.com/playlist?list=PLg7.../>

There are many websites for hosting online git repositories. Your instructor will indicate which web service to use, but we only include instructions here for setup with Bitbucket.

1. *Sign up.* Create a Bitbucket account at <https://bitbucket.org>. If you use an academic email address (ending in `.edu`, etc.), you will get free unlimited public and private repositories.
2. *Make a new repository.* On the Bitbucket page, click the `+` button from the menu on the left and, under **CREATE**, select **Repository**. Provide a name for the repository, mark the repository as **private**, and make sure the repository type is **Git**. For **Include a README?**, select **No** (if you accidentally include a **README**, delete the repository and start over). Under **Advanced settings**, enter a short description for your repository, select **No forks** under forking, and select **Python** as the language. Finally, click the blue **Create repository** button. Take note of the URL of the webpage that is created; it should be something like <https://bitbucket.org/<name>/<repo>>.

3. *Give the instructor access to your repository.* On your newly created Bitbucket repository page (<https://bitbucket.org/<name>/<repo>> or similar), go to **Settings** in the menu to the left and select **User and group access**, the second option from the top. Enter your instructor's Bitbucket username under **Users** and click **Add**. Select the blue **Write** button so your instructor can read from and write feedback to your repository.
4. *Connect your folder to the new repository.* In a shell application (Terminal on Linux or Mac, or Git Bash (<https://gitforwindows.org/>) on Windows), enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder # cd means 'change directory'.

# Make sure you are in the right place.
$ pwd                # pwd means 'print working directory'.
/path/to/folder
$ ls *.md             # ls means 'list files'.
README.md            # This means README.md is in the working directory.

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://<name>@bitbucket.org/<name>/<repo>.git

# Record your credentials.
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

For example, if your Bitbucket username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is on the desktop, enter the following commands.

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

# Connect this folder to the online repository.
$ git init
$ git remote add origin https://greek314@bitbucket.org/greek314/acmev1.git

# Record credentials.
$ git config --local user.name "archimedes"
```

```
$ git config --local user.email "greek314@example.com"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

At this point you should be able to see the files on your repository page from a web browser. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line.

```
$ git remote set-url origin https://<name>@bitbucket.org/<name>/<repo>.git
```

5. *Download data files.* Many labs have accompanying data files. To download these files, navigate to your clone and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folder for you. You can also find individual data files through `Student-Materials/wiki/Lab-Index`.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash download_data.sh
```

6. *Install Python package dependencies.* The labs require several third-party Python packages that don't come bundled with Anaconda. Run the following command to install the necessary packages.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash install_dependencies.sh
```

7. (Optional) *Clone your repository.* If you want your repository on another computer after completing steps 1–4, use the following commands.

```
# Navigate to where you want to put the folder.
$ cd ~/Desktop/or/something/

# Clone the folder from the online repository.
$ git clone https://<name>@bitbucket.org/<name>/<repo>.git <foldername>

# Record your credentials in the new folder.
$ cd <foldername>
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Download data files to the new folder.
$ bash download_data.sh
```

## Setup Without Git

Even if you aren't using git to submit files, you must install it (<http://git-scm.com/downloads>) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 5 and 6 of the previous section to download the data files and install package dependencies.

## Using Git

Git manages the history of a file system through *commits*, or checkpoints. Use `git status` to see the files that have been changed since the last commit. These changes are then moved to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. Save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

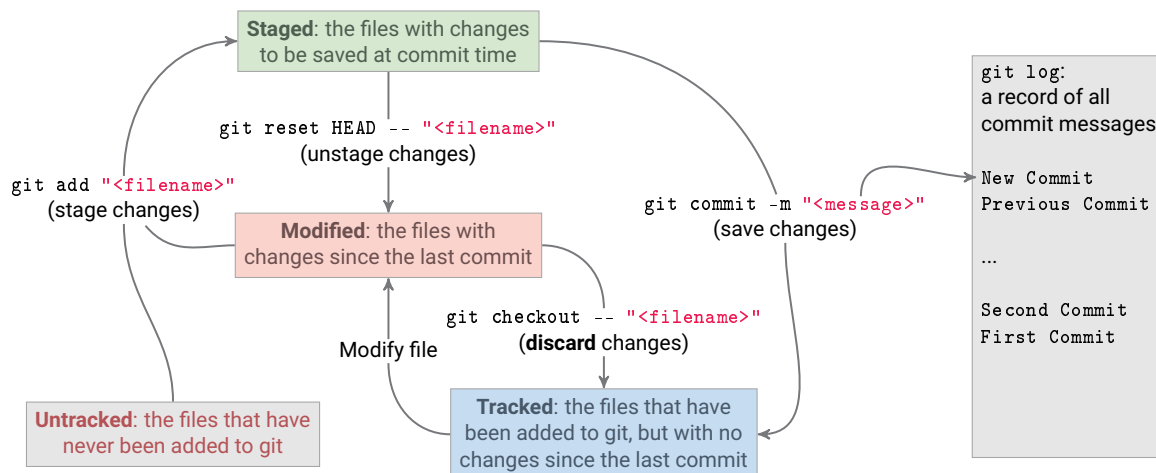


Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

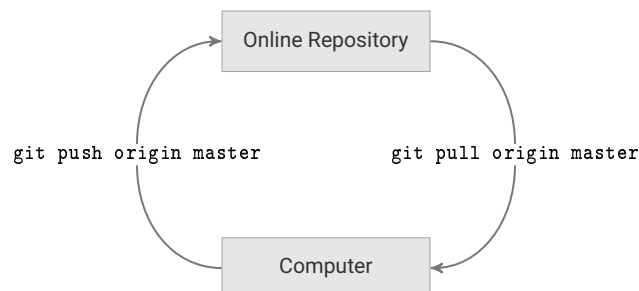


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
<code>git status</code>	Display the staging area and untracked changes.
<code>git pull origin master</code>	Pull changes from the online repository.
<code>git push origin master</code>	Push changes to the online repository.
<code>git add &lt;filename(s)&gt;</code>	Add a file or files to the staging area.
<code>git add -u</code>	Add all modified, tracked files to the staging area.
<code>git commit -m "&lt;message&gt;"</code>	Save the changes in the staging area with a given message.
<code>git checkout -- &lt;filename&gt;</code>	Revert changes to an unstaged file since the last commit.
<code>git reset HEAD -- &lt;filename&gt;</code>	Remove a file from the staging area.
<code>git diff &lt;filename&gt;</code>	See the changes to an unstaged file since the last commit.
<code>git diff --cached &lt;filename&gt;</code>	See the changes to a staged file since the last commit.
<code>git config --local &lt;option&gt;</code>	Record your credentials ( <code>user.name</code> , <code>user.email</code> , etc.).

Table A.1: Common git commands.

**NOTE**

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in *vim* ([https://en.wikipedia.org/wiki/Vim\\_\(text\\_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))), is asking you to enter a message (or use the default message) to create a *merge commit* that will reconcile both changes. To close this screen and create the merge commit, type `:wq` and press `enter`.

**Example Work Sessions**

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master           # Pull updates.
### Make changes to a file.
$ git add -u                      # Track changes.
$ git commit -m "Made some changes." # Commit changes.
$ git push origin master          # Push updates.
```



```

# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ git pull origin master
From https://bitbucket.org/username/repo
 * branch                master      -> FETCH_HEAD
Already up-to-date.

### Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://username@bitbucket.org/username/repo.git
   5742a1b..fed9b34  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

```



# B

## Installing and Managing Python

**Lab Objective:** *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

### Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to <https://www.anaconda.com/download/>.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see <https://docs.anaconda.com/anaconda/install/>. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

#### ACHTUNG!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

### Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see <https://xkcd.com/349/>).

## Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See <https://docs.anaconda.com/anaconda/packages/pkg-docs> for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
<code>conda install &lt;package-name&gt;</code>	Install the specified package.
<code>conda update &lt;package-name&gt;</code>	Update the specified package.
<code>conda update conda</code>	Update <code>conda</code> itself.
<code>conda update anaconda</code>	Update <b>all</b> packages included in Anaconda.
<code>conda --help</code>	Display the documentation for <code>conda</code> .

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda           # Make sure that conda is up to date.
$ conda install matplotlib     # Attempt to install matplotlib.
$ conda update matplotlib      # Attempt to update matplotlib.
```

See <https://conda.io/docs/user-guide/tasks/manage-pkgs.html> for more examples.

### NOTE

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

### ACHTUNG!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

## Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
<code>pip install package-name</code>	Install the specified package.
<code>pip install --upgrade package-name</code>	Update the specified package.
<code>pip freeze</code>	Display the version number on all installed packages.
<code>pip --help</code>	Display the documentation for pip.

See [https://pip.pypa.io/en/stable/user\\_guide/](https://pip.pypa.io/en/stable/user_guide/) for more complete documentation.

## Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

### Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: <https://atom.io/>
- Sublime Text: <https://www.sublimetext.com/>
- Notepad++ (Windows): <https://notepad-plus-plus.org/>
- Geany: <https://www.geany.org/>
- Vim: <https://www.vim.org/>
- Emacs: <https://www.gnu.org/software/emacs/>

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                                # List the files in the current directory.
hello_world.py
$ cat hello_world.py                # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py             # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

#### NOTE

While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:

- Windows subsystem for linux: [docs.microsoft.com/en-us/windows/wsl/](https://docs.microsoft.com/en-us/windows/wsl/).
- Git bash: <https://gitforwindows.org/>.

## Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See <https://github.com/jupyter/jupyter/wiki/> for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Jupyter Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and  $\text{\LaTeX}$ , and can embed images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

## Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: <http://jupyterlab.readthedocs.io/en/stable/>
- PyCharm: <https://www.jetbrains.com/pycharm/>
- Spyder: <http://code.google.com/p/spyderlib/>
- Eclipse with PyDev: <http://www.eclipse.org/>, <https://www.pydev.org/>

See <https://realpython.com/python-ides-code-editors-guide/> for a good overview of these (and other) workflow tools.



# 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.

$$\begin{aligned}
 A &= \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} & B &= \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \\
 \\ 
 \text{np.hstack}((A,B,A)) &= \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix} \\
 \\ 
 \text{np.vstack}((A,B,A)) &= \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}
 \end{aligned}$$

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()`.

$$\begin{aligned}
 x &= [\times \quad \times \quad \times \quad \times] & y &= [* \quad * \quad * \quad *] \\
 \\ 
 \text{np.hstack}((x,y,x)) &= [\times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times] \\
 \\ 
 \text{np.vstack}((x,y,x)) &= \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} & \text{np.column_stack}((x,y,x)) &= \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}
 \end{aligned}$$

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.



$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\mathbf{A} + \mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\mathbf{A} + \mathbf{x}.\text{reshape}((1,-1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

## 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`.

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

$$\mathbf{A}.\text{sum}(\text{axis}=0) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\mathbf{A}.\text{sum}(\text{axis}=1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$



# Bibliography

- [ADH<sup>+</sup>01] David Ascher, Paul F Dubois, Konrad Hinsén, 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.
- [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.
- [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.