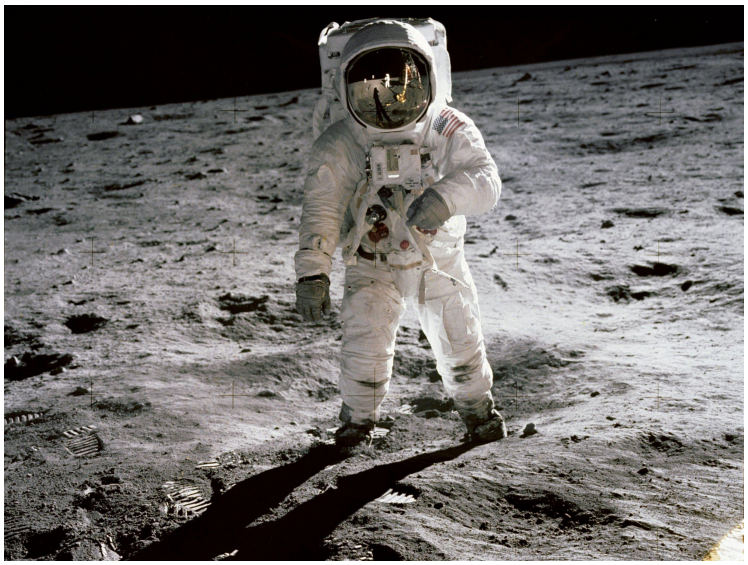
Labs for Foundations of Applied Mathematics

Volume II
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

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

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
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

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
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

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
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

Preface

This lab manual is designed to accompany the textbooks *Foundations of Applied Mathematics* by Humpherys and Jarvis.

Contents

Part I

Labs

1 Linked Lists

Lab Objective: *Analyzing and manipulating data are essential skills in scientific computing. Storing, retrieving, and rearranging data take time. As a dataset grows, so does the amount of time it takes to access and analyze it. To write effecient algorithms involving large data sets, it is therefore essential to be able to design or choose the data structures that are most optimal for a particular problem. In this lab we begin our study of data structures by constructing a generic linked list, then using it to implement a few common data structures.*

Introduction

Data structures are specialized objects for organizing data efficiently. There are many kinds, each with specific strengths and weaknesses, and different applications require different structures for optimal performance. For example, some data structures take a long time to build, but once built their data are quickly accessible. Others are built quickly, but are not as efficiently accessible. These strengths and weaknesses are determined by how the structure is implemented.

Python has several built-in data structure classes, namely `list`, `set`, `dict`, and `tuple`. Being able to use these structures is important, but selecting the correct data structure to begin with is often what makes or breaks a good program. In this lab we create a structure that mimics the built-in list class, but that has a different underlying implementation. Thus our class will be better than a plain Python list for some tasks, but worse for others.

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 '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.

Note

Often the data stored in a node is actually a *key* value. The key might be a memory address, a dictionary key, or the index of an array where the true desired information resides. For simplicity, in this and the following lab we store actual data in node objects, not references to data located elsewhere.

Linked Lists

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

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 in the list (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 '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
        self.prev = None
```

Now we create a new class, `LinkedList`, that will link `LinkedListNode` instances together by modifying each node's `next` and `prev` attributes. The list is empty initially, so we assign the `head` and `tail` attributes the placeholder value `None`.

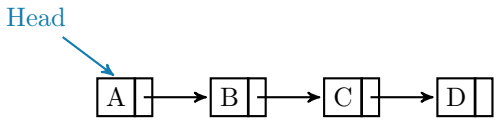


Figure 1.1: A singly linked list. Each node has a reference to the next node in the list. The head attribute is always assigned to the first node.

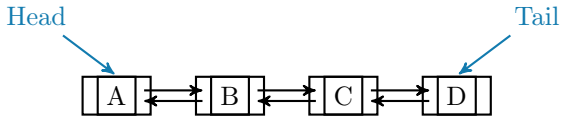


Figure 1.2: A doubly linked list. Each node has a reference to the node before it and a reference to the node after it. In addition to the head attribute, this list has a tail attribute that is always assigned to the last node.

We also need a method for adding data to the list. The `append()` makes a new node and adds it to the very end of the list. There are two cases to consider: appending to an empty list, and appending to a nonempty list. See Figure 1.3.

```
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 '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
```

Figure 1.3: Appending a new node to the end of a nonempty doubly linked list. The red arrows are the new connections. Note that the `tail` attribute is adjusted.

Achtung!

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

```
# This evaluates to True since the numerical values are the same.
>>> 7 == 7.0
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.

find()

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, we must use each successive node's `next` and `prev` attributes.

```
>>> my_list = LinkedList()
>>> my_list.append(2)
>>> my_list.append(4)
>>> my_list.append(6)

# To access each value, we 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
4
>>> my_list.head.next.next.value
6
```

```
>>> my_list.head.next.next is my_list.tail
True
>>> my_list.tail.prev.prev is my_list.head
True
```

Problem 2. Add a method called `find(self, data)` to the `LinkedList` class that returns the first node in the list containing `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 `current` is assigned to one of the nodes the list, what does the following line do?)

```
current = current.next
```

Magic Methods

Endowing data structures with magic methods makes it much easier to use it intuitively. Consider, for example, how a Python list responds to built-in functions like `len()` and `print()`. At the bare minimum, we should give our linked list the same functionality.

Problem 3. Add magic methods to the `LinkedList` class so it behaves more like the built-in Python list.

1. Write the `__len__()` method so that the length of a `LinkedList` instance is equal to the number of nodes in the list. To accomplish this, consider adding an attribute that tracks the current size of the list. It should be updated every time a node is successfully added or removed.

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, and strings are surrounded by single quotes. Note the difference between the string representations of the following lists:

```
>>> num_list = [1, 2, 3]
>>> str_list = ['1', '2', '3']
>>> print(num_list)
[1, 2, 3]
>>> print(str_list)
['1', '2', '3']
```

remove()

In addition to adding new nodes to the end of a list, it is also useful to remove nodes and insert new nodes at specified locations. To delete a node, all references to the node must be removed. Then Python will automatically delete the object, since 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`.

```
class LinkedList:
    # ...
    def remove(self, data):
        """Attempt to remove the first node containing '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 will delete the node (see Figure 1.4). However, 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. Account for the special cases of removing the first, last, or only node.

ACHTUNG!

Python keeps track of the variables in use and automatically deletes a variable if there is no access to it. In many other languages, leaving a reference to an object without explicitly deleting it could cause a serious memory leak. See `https://docs.python.org/2/library/gc.html` for more information on Python's auto-cleanup system.

insert()

Problem 5. Add a method called `insert(self, data, place)` to the `LinkedList` class that inserts a new node containing `data` immediately before the first node in the list containing `place`. Account for the special case of inserting before the first node.

See Figure 1.6 for an illustration. Note that since `insert()` places a new node before an existing node, it is not possible to use `insert()` to put a new node at the end of the list or in an empty list (use `append()` instead).

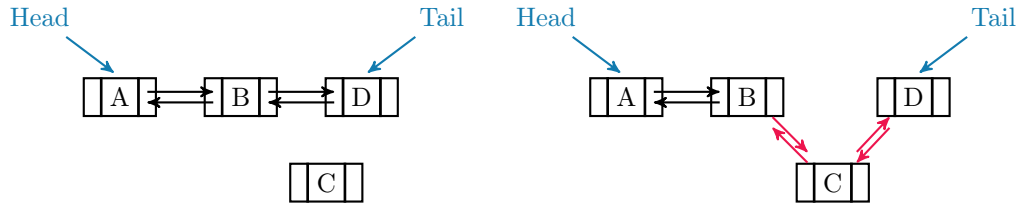


Figure 1.6: Insertion for Doubly linked Lists.

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. The three most common and basic restricted-access structures 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 line at the bank: the person at the front of the line is served next, 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.[1]
- **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()`.

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

1. Use inheritance to implement the following methods:
   - `pop(self)`: Remove the last node in the list and return its data.
   - `popleft(self)`: Remove the first node in the list and return its data.
   - `appendleft(self, data)`: Insert a new node containing `data` at the beginning of the list.

   The `LinkedList` class already implements the `append()` method.
2. Override the `remove()` method with the following:

```
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 the `insert()` method 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, as Python lists are not implemented as linked lists.

[1] `push()` and `pop()` for queues are sometimes called `enqueue()` and `dequeue()`, respectively)

The `collections` module in the standard library has a `deque` object, 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.

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 to the end of a deque. After reading the entire file, pop each entry off of the end of the deque one at a time, writing the result to a line of the second file.

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

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

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

Additional Material

Improvements to the Linked List Class

1. Add a keyword argument to the constructor so that if an iterable is input, 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]
```

2. Add new methods:
   - `count()`: return the number of occurrences of a specified value.
   - `reverse()`: reverse the ordering of the nodes (in place).
   - `rotate()`: rotate the nodes a given number of steps to the right (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.
   - `__iter__()`: support `for` loop iteration, the `iter()` built-in function, and the `in` statement.

Other Linked List

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

1. A *sorted list* adds new nodes strategically so that the data is always kept in order. A `SortedLinkedList` class that inherits from the `LinkedList` class should have a method called `add(self, data)` that 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.

   A linked list is **not** an ideal implementation for a sorted list (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.

2 Binary Search Trees

Lab Objective: *A tree is a linked list where each node in the list may refer to more than one other node. This structural flexibility makes trees more useful and efficient than regular linked lists in many applications. Many trees are most easily constructed recursively, so we begin with an overview of recursion. We then implement a recursively structured doubly linked Binary Search Tree. Finally, we compare the standard linked list, our Binary Search Tree, and an AVL tree to illustrate the relative strengths and weaknesses of each structure.*

Recursion

A *recursive* function is one that calls itself. When the function is executed, it continues calling itself until it reaches a specified *base case* where the solution to the problem is known. The function then exits without calling itself again, and each previous function call is resolved.

As a simple example, consider the function that sums all positive integers from 1 to some integer n. This function may be represented recursively:

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

$f(n)$ may be calculated by recursively calculating $f(n-1)$, which calculates $f(n-2)$, and so on. The recursion halts with the base case $f(1) = 1$.

```
def recursive_sum(n):
    """Calculate the sum of all positive integers in [1, n] recursively."""
    # Base Case: the sum of all positive integers in [1, 1] is 1.
    if n == 1:
        return 1

    # If the base case hasn't been reached, the function recurses by calling
    # itself on the next smallest integer. The result of that call, plus the
    # particular 'n' from this call, gives the result.
    else:
        return n + recursive_sum(n-1)
```

The computer calculates `recursive_sum(5)` with a sequence of function calls.

```
# To find recursive_sum(5), calculate recursive_sum(4).
# To find recursive_sum(4), calculate recursive_sum(3).
# This continues until the base case is reached.

recursive_sum(5)        # return 5 + recursive_sum(4)
    recursive_sum(4)        # return 4 + recursive_sum(3)
        recursive_sum(3)        # return 3 + recursive_sum(2)
            recursive_sum(2)        # return 2 + recursive_sum(1)
                recursive_sum(1)        # Base case: return 1.
```

Substituting the values that resulted from each call unwinds the recursion.

```
recursive_sum(5)        # return 5 + 10
    recursive_sum(4)        # return 4 + 6
        recursive_sum(3)        # return 3 + 3
            recursive_sum(2)        # return 2 + 1
                recursive_sum(1)        # Base case: return 1.
```

So `recursive_sum(5)` returns 15 (which is correct, since $1 + 2 + 3 + 4 + 5 = 15$).

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

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

The mathematical function itself is defined recursively, so it makes sense for an implementation to use recursion. Compare the following iterative method implementing g to its recursive equivalent.

```
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    fibonacci = []          # Initialize an empty list.
    fibonacci.append(0)     # Append 0 (the 0th Fibonacci number).
    fibonacci.append(1)     # Append 1 (the 1st Fibonacci number).
    for i in range(1, n):
        # Starting at the third entry, calculate the next number
        # by adding the last two entries in the list.
        fibonacci.append(fibonacci[-1] + fibonacci[-2])
    # When the entire list has been loaded, return the nth entry.
    return fibonacci[n]

def recursive_fib(n):
    """Calculate the nth Fibonacci number recursively."""
    # The base cases are the first two Fibonacci numbers.
    if n == 0:              # Base case 1: the 0th Fibonacci number is 0.
        return 0
    elif n == 1:            # Base case 2: the 1st Fibonacci number is 1.
        return 1
    # If this call isn't a base case, the function recurses by calling
    # itself to calculate the previous two Fibonacci numbers.
```

```
    else:
        return recursive_fib(n-1) + recursive_fib(n-2)
```

This time, the sequence of function calls is slightly more complicated because `recursive_fib()` calls itself twice at each step.

```
recursive_fib(5)        # The original call makes two additional calls:
    recursive_fib(4)        # this one...
        recursive_fib(3)
            recursive_fib(2)
                recursive_fib(1)        # Base case 2: return 1
                recursive_fib(0)        # Base case 1: return 0
            recursive_fib(1)        # Base case 2: return 1
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
    recursive_fib(3)        # ...and this one.
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
        recursive_fib(1)        # Base case 2: return 1
```

The sum of all of the base case results, from top to bottom, is $1+0+1+1+0+1+0+1=5$, so `recursive_fib(5)` returns 5 (correctly). The key to recursion is understanding the base cases correctly and making correct recursive calls.

Problem 1. The following code defines a simple class for singly linked lists.

```
class SinglyLinkedListNode:
    """Simple singly linked list node."""
    def __init__(self, data):
        self.value, self.next = data, None

class SinglyLinkedList:
    """A very simple 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 '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
```

Rewrite the following iterative function for finding data in a linked list using recursion. Use instances of the `SinglyLinkedList` class to test your function.

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

(Hint: define a second function to perform the actual recursion.)

ACHTUNG!

It is **not** usually better to rewrite an iterative method recursively. In Python, a function may only call itself 999 times. On the 1000^{th} call, a `RuntimeError` is raised to prevent a stack overflow. Whether or not recursion is appropriate depends on the problem to be solved and the algorithm used to solve it.

Trees

A *tree* data structure is a specialized linked list. Trees are more difficult to build than standard linked lists, but they are almost always more efficient. While the computational complexity of finding a node in a linked list is $O(n)$, a well-built, balanced tree will find a node with a complexity of $O(\log n)$. Some types of trees can be constructed quickly but take longer to retrieve data, while others take more time to build and less time to retrieve data.

The first node in a tree is called the *root*. The root node points to other nodes, called children. Each child node in turn points to its children. This continues on each branch until its end is reached. A node with no children is called a *leaf node*.

Mathematically, a tree is a directed graph with no cycles. Therefore a linked lists as a graph qualifies as a tree, albeit a boring one. The head node is the root node, and it has one child node. That child node also has one child node, which in turn has one child. The last node in the list is the only leaf node.

Other kinds of trees may be more complicated.

Binary Search Trees

A *binary search tree* (BST) data structure 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 data that is less than its parent node's data. The right child's data is greater.

The tree on the right in Figure 2.1 is an example of a of binary search tree. In practice, binary search tree nodes have attributes that keep track of their data, their children, and (in doubly linked trees) their parent.

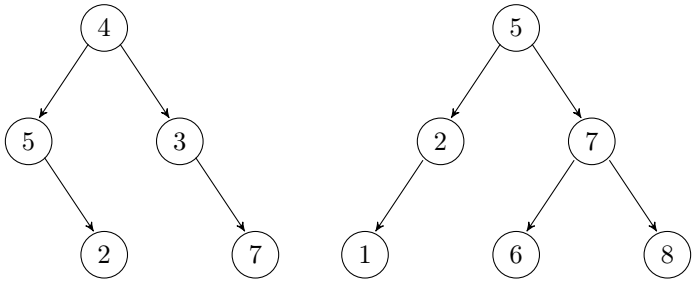


Figure 2.1: Both of these graphs are trees, but only the tree on the right is a binary search tree. How could the graph on the left be altered to make it a BST?

```
class BSTNode:
    """A Node class for Binary Search Trees. Contains some data, a
    reference to the parent node, and references to two child nodes.
    """
    def __init__(self, data):
        """Construct a new node and set the data 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      # This node's value will be less than self.value
        self.right = None     # This node's value will be greater than self.↩
            value
```

The actual binary search tree class has an attribute pointing to its root.

```
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
```

find()

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, if the data is not in the tree, an empty child is searched.

```
class BST:
    # ...
    def find(self, data):
        """Return the node containing 'data'. If there is no such node
```

```
        in the tree, or 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
            '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)
```

> **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.

insert()

To insert new data into a binary search tree, add a leaf node at the correct location. First, find the node that should be the parent of the new node. This parent node is found recursively, using a similar approach to the `find()` method. Then the new node is added as the left or right child of the parent. See Figure 2.2.

Problem 2. Implement the `insert()` method in the `BST` class.

1. Find the parent of the new node. Consider writing a recursive method, similar to `find()`, to do this. Determine whether the new node will be the parent's left or right child, then double-link the parent and the new child.
2. Do not allow for duplicates in the tree. Raise a `ValueError` if there is already a node in the tree containing the input data.

Be sure to consider the special case of inserting to an empty tree. To test your tree, use (but do not modify) the provided `BST.__str__()` method.

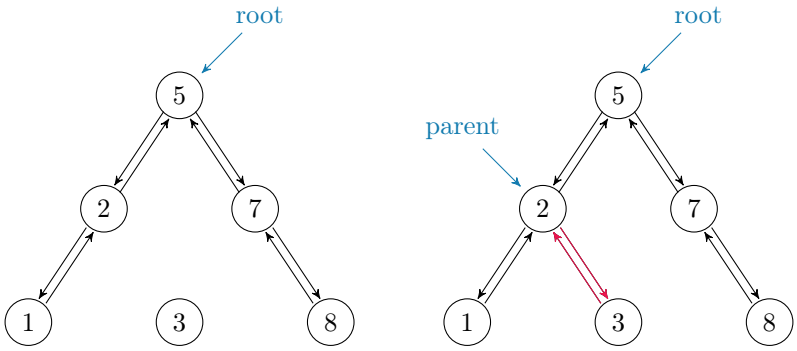


Figure 2.2: To insert a node containing 3 to the BST on the left, start at the root and recurse down the tree to find the node that should be 3's parent. Connect that parent to the child, then the child to its new parent.

remove()

Deleting nodes from a binary search tree is more difficult than finding or inserting. Insertion always creates a new leaf node, but removal may delete any kind of node. This leads to several different cases to consider.

Removing a Leaf Node

In Python, an object is automatically deleted if there are no references to it. Call the node to be removed the *target node*, and suppose it has no children. To remove the target, find the target's parent, then delete the parent's reference to the target. Then there are no references to the target, so the target node is deleted. Since the target is a leaf node, removing it does not affect the rest of the tree structure.

Removing a Node with One Child

If the target node has one or more children, be careful not to delete the children when the target is removed. Simply removing the target as if it were a leaf node would delete the entire subtree originating from the target.

To avoid deleting all of the target's descendents, point the target's parent to an appropriate successor. If the target has only one child, then that child is the successor. Connect the target's parent to the successor, and double-link by setting the successor's parent to be the target node's parent. Then, since the target has no references pointing to it, it is deleted. The target's successor, however, is pointed to by the target's parent, and so it remains in the tree.

Removing a Node with Two Children

Removal is more complicated if the target node has two children. To delete this kind of node, first find its immediate in-order successor. This successor is the node with the smallest value that is larger than the target's value. It may be found by moving to the right child of the target (so that it's value is greater than the target's value), and then to the left for as long as possible (so that it has the smallest such value). Note that because of how the successor is chosen, any in-order successor can only have at most one child.

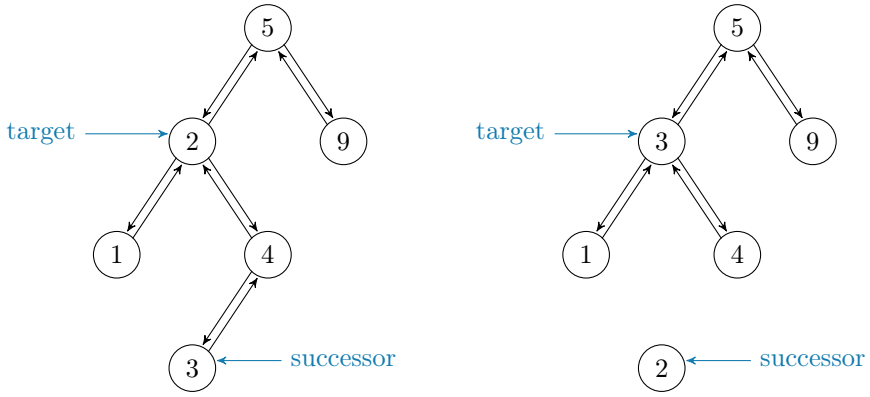


Figure 2.3: To remove the node containing 2 from the top left BST, locate the target and its in-order successor. Delete the successor, recording its value. Finally, replace the data in the target with the data that was in the successor.

Once the successor is found, the target and its successor must switch places in the graph, and then the target must be removed. This can be done by simply switching the values for the target and its successor. Then the node with the target data has at most one child, and may be deleted accordingly. If the successor was chosen appropriately, then the binary search tree structure and ordering will be maintained once the deletion is finished.

The easiest way to implement this is to use recursion. First, because the successor has at most one child, remove the successor node recursively by calling `remove()` on the successor's value. Then set the data stored in the target node as the successor's value. See Figure 2.3.

Removing the Root Node

In each of the above cases, we must also consider the subcase where the target is the root node. If the root has no children, resetting the root or calling the constructor will do. If the root has one child, that child becomes the new root of the tree. If the root has two children, the successor becomes the new root of the tree.

Problem 3. Implement the `remove()` method in the `BST` class. If the tree is empty, or if the target node is not in the tree, raise a `ValueError`. Test your solutions thoroughly, accounting for all possible cases:

1. The tree is empty (`ValueError`).
2. The target is not in the tree (`ValueError`).
3. The target is the root node:
   (a) the root is a leaf node, hence the only node in the tree.
   (b) the root has one child.
   (c) the root has two children.
4. The target is in the tree but is not the root:

(a) the target is a leaf node.

(b) the target has one child.

(c) the target has two children.

(Hints: **Before coding anything**, outline the entire function with comments and `if-else` blocks. Use the `find()` method wherever appropriate.)

AVL Trees

Binary search trees are a good way of organizing data so that it is quickly accessible. However, pathologies may arise when certain data sets are stored using a basic binary search tree. This is best demonstrated by inserting ordered data into a binary search tree. Since the data is already ordered, each node will only have one child, and the result is essentially a linked list.

```
# Sequentially adding ordered integers destroys the efficiency of a BST.
>>> unbalanced_tree = BST()
>>> for i in range(10):
...     unbalanced_tree.insert(i)
...
# The tree is perfectly flat, so it loses its search efficiency.
>>> print(unbalanced_tree)
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
```

Problems also arise when one branch of the tree becomes much longer than the others, leading to longer search times.

An *AVL tree* (named after Georgy Adelson-Velsky and Evgenii Landis) is a tree that prevents any one branch from getting longer than the others. It accomplishes this by recursively "balancing" the branches as nodes are added. See Figure 2.4. The AVL's balancing algorithm is beyond the scope of this project, but details and exercises on the algorithm can be found in Chapter 2 of the Volume II text.

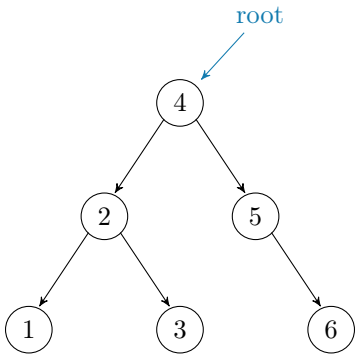


Figure 2.4: The balanced AVL tree resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. After each insertion the tree rebalances if necessary.

```
>>> balanced_tree = AVL()
>>> for i in range(10):
...     balanced_tree.insert(i)
...
# The AVL tree is balanced, so it retains (and optimizes) its search efficiency↩
    .
>>> print(balanced_tree)
[3]
[1, 7]
[0, 2, 5, 8]
[4, 6, 9]
```

Problem 4. Write a function to compare the build and search times of the data structures we have implemented so far.

Read the file `english.txt`, adding the contents of each line to a list of data. For various values of n, repeat the following:

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_search()` function 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 if appropriate.

Conclusion

Every data structure has advantages and disadvantages. Recognizing when an application may take advantage of a certain structure, especially when that structure is more complicated than a Python list or set, is an important skill. Choosing structures wisely often results in huge speedups and easier data maintenance.

Additional Material

Improvements to the BST

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

1. Add a keyword argument to the constructor so that if an iterable is input, each element of the iterable is immediately added to the tree. This makes it possible to cast other iterables as a `BST`, like 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 using a depth-first search. (Hint: examine the provided `__str__()` method carefully.)

Other Trees

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 Chapter 2 of the Volume II text for details.
2. The nodes of a *Red-Black Tree* are labeled either red or black. The tree satisfies the following rules.
   (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 will be close to the root.

3 Nearest Neighbor Search

Lab Objective: *The nearest neighbor problem is an optimization problem that arises in applications such as computer vision, pattern recognition, internet marketing, and data compression. Solving the problem efficiently requires the use of a k-d tree, a variation of the binary search tree. In this lab we implement a k-d tree, use it to solve the nearest neighbor problem, then apply SciPy's k-d tree object to a handwriting recognition algorithm.*

The Nearest Neighbor Problem

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. This is called the nearest neighbor search problem, and it has many applications.

In general, suppose that X is a collection of data, called a *training set*. Let y be any point (often called the *target* point) in the same space as the data in X. The nearest neighbor search problem determines the point in X that is closest to y. For example, in the post office problem, the set X could be addresses or latitude and longitude data for each post office in the city. Then y would be the data that represents your new home, and the task is to find the closest post office in X to y.

Problem 1. Roughly speaking, a function that measures the distance between two points in a set is called a *metric*.[a] The *Euclidean metric* measures the distance between two points in $\mathbb{R}^n$ with the familiar distance formula:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} = \|\mathbf{x}-\mathbf{y}\|_2$$

Write a function that accepts two 1-dimensional NumPy arrays and returns the Euclidean distance between them. Raise a `ValueError` if the arrays don't have the same number of entries. (Hint: NumPy already has some functions to help do this quickly.)

[a] Metrics and metric spaces are examined in detail in Chapter 5 of Volume I.

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, measure the mileage, and then choose the post office that is the closest. This is called an *exhaustive search*. More precisely, measure the distance of y to each point in X, and choose the point in X with the smallest distance from y. However, this method is inefficient, and only feasible for relatively small training sets.

Problem 2. Write a function that solves the nearest neighbor search problem by exhaustively checking all of the distances between a given point and each point in a data set. The function should take in a set of data points (as an $m \times k$ NumPy array, where each row represents one of the m points in the data set) and a single target point (as a 1-dimensional NumPy array with k entries). Return the point in the training set that is closest to the target point and its distance from the target.

The complexity of this algorithm is $O(mk)$, where k is the number of dimensions and m is the number of data points.

K-D Trees

A *k-d tree* is a special kind of binary search tree for high dimensional data (i.e., more dimensions than one). While a binary search tree excludes regions of the number line from a search until the search point is found, a k-d tree works on regions of $\mathbb{R}^k$. In other words, a regular binary search tree partitions $\mathbb{R}$, but a k-d tree partitions $\mathbb{R}^k$. So long as the data in the tree meets certain dimensionality requirements, similar efficiency gains may be made.

Recall that to search for a value in a binary search tree, start at the root, and if the value is less than the root, proceed down the left branch of the tree. If it is larger, proceed down the right branch. By doing this, a subset of values (and therefore the subtree containing those values) is excluded from the search. By eliminating this subset from consideration, there are far fewer points to search and the efficiency of the search is greatly increased.

Like a binary search tree, a k-d tree starts with a root node with a depth, or level, of 0. At the i^{th} level, the nodes to the left of a parent have a strictly lower value in the i^{th} dimension. Nodes to the right have a greater or equal value in the i^{th} dimension. At the next level, do the same for the next dimension. For example, consider data in $\mathbb{R}^3$. The root node partitions the data according to the first dimension. The children of the root partition according to the second dimension, and the grandchildren along the third. See Figures 3.1 and 3.2 for examples in $\mathbb{R}^2$ and $\mathbb{R}^3$.

As with any other data structure, the first task is to construct a node class to store data. A `KDTNode` is similar to a `BSTNode`, except it has another attribute called `axis`. The `axis` attribute indicates the dimension of $\mathbb{R}^k$ to compare points.

Problem 3. Import the `BSTNode` class from the previous lab using the following code:

```
import sys
sys.path.insert(1, "../Trees")
from trees import BSTNode
```

Write a `KDTNode` class that inherits from `BSTNode`. Modify the constructor so that a `KDTNode` can only hold a NumPy array (of type `np.ndarray`). If any other data type is given, raise a `TypeError`. Create an `axis` attribute (set it to `None` or 0 for now).

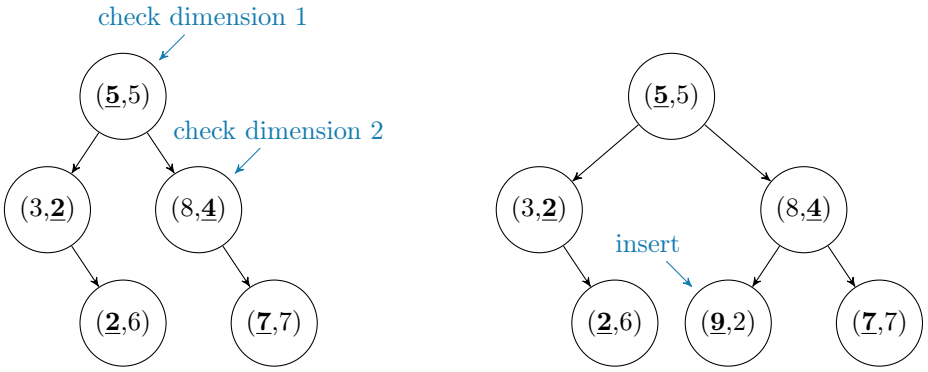


Figure 3.1: These trees illustrate an insertion of (9, 2) into a k-d tree in $\mathbb{R}^2$ loaded with the points (5, 5), (8, 4), (3, 2), (7, 7), and (2, 6), in that order. To insert the point (9, 2), find the node that will be the new node's parent. Start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), move into the right subtree of the root node, thus excluding all points (x, y) with $x < 5$. Next, compare (9, 2) to the root's right child, (8, 4). Since the y-coordinate of (9, 2) is less than the y-coordinate of (8, 4), move to the left of (8,4), thus excluding all points (x, y) with $y > 4$. Since (8,4) does not have a left child, insert (9, 2) as the left child of (8,4).

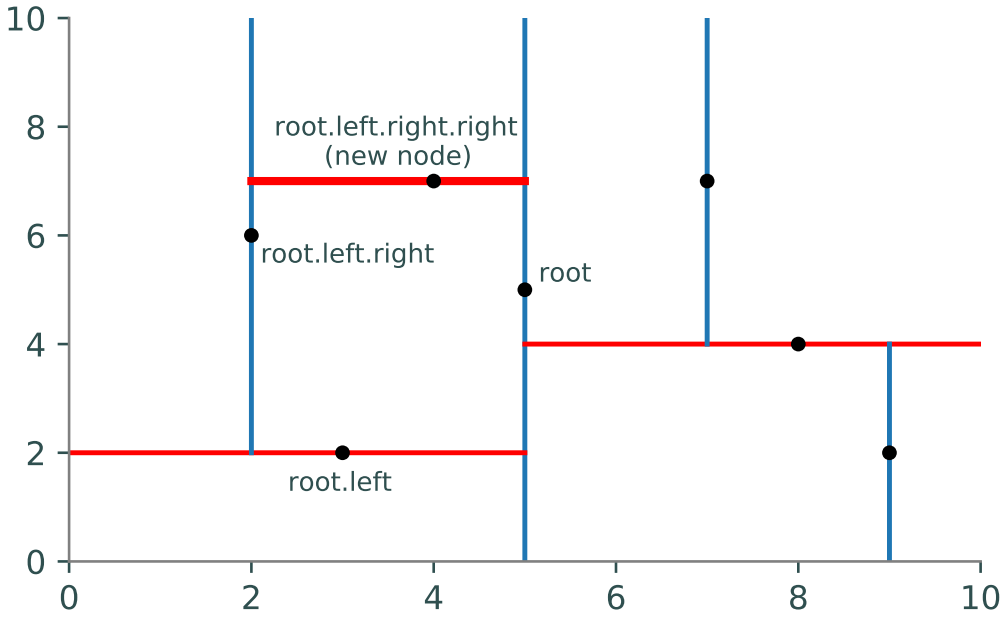


Figure 3.2: This figure is another illustration of the k-d tree from Figure 3.1 with the point (9,2) already inserted. To insert the point (4, 7), start at the root. Since the x-coordinate of (4, 7) is less than the x-coordinate of (5, 5), move into the region to the left of the middle blue line, to the root's left child, (3, 2). The y-coordinate of (4, 7) is greater than the y-coordinate of (3, 2), so move above the red line on the left, to the right child (2, 6). Now return to comparing the x-coordinates, and since $4 > 2$ and (2, 6) has no right child, install (4, 7) as the right child of (2, 6).

The major difference between a k-d tree and a binary search tree is how the data is compared at each depth level. Though the nearest neighbor problem does not need to use a `find()` method, the k-d tree version of `find()` is provided as an instructive example.

In the `find()` method, every comparison in the recursive `_step()` function compares the data of `target` and `current` based on the `axis` attribute of `current`. This way, if each existing node in the tree has the correct `axis`, the correct comparisons are made when descending through the tree.

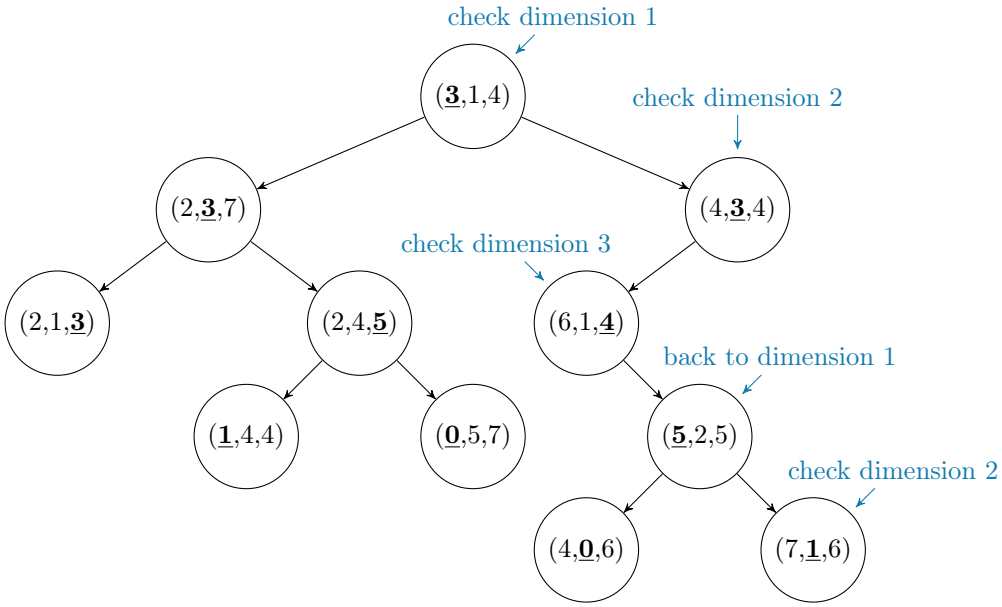


Figure 3.3: To find the point (7, 1, 6), start at the root. Since the x-coordinate of (7, 1, 6) is greater than the x-coordinate of (3, 1, 4), move to the right subtree of the root node, thus excluding all points (x, y, z) with $x < 7$. Next, compare (7, 1, 6) to the root's right child, (4, 3, 4). Since the y-coordinate of (7, 1, 6) is less than the y-coordinate of (4, 3, 4), move to the left subtree of (4, 3, 4), thus excluding all points (x, y, z) with $y > 1$. Continue in this manner until (7, 1, 6) is found.

```
import numpy as np
# Import the BST class from the previous lab.
import sys
sys.path.insert(1, "../Trees")
from trees import BST

class KDT(BST):
    """A k-dimensional binary search tree object.
    Used to solve the nearest neighbor problem efficiently.

    Attributes:
        root (KDTNode): The root node of the tree. Like all nodes in the tree,
            the root houses data as a NumPy array.
        k (int): The dimension of the tree (the 'k' of the k-d tree).
    """
```

```
    def find(self, data):
        """Return the node containing 'data'. If there is no such node in the
            tree, or 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
            '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")
            elif np.allclose(data, current.value):
                return current                      # Base case 2: data found!
            elif data[current.axis] < current.value[current.axis]:
                return _step(current.left)          # Recursively search left.
            else:
                return _step(current.right)         # Recursively search right.

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

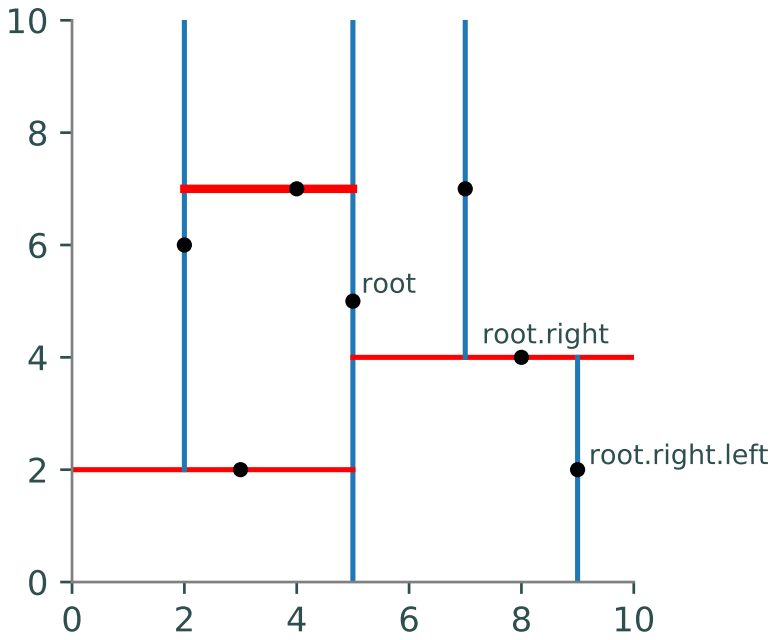


Figure 3.4: The above graph is another illustration of the k-d tree from Figure 3.1. To find the point (9, 2), start at the root. Since the x-coordinate of (9, 2) is greater than the x-coordinate of (5, 5), move into the region to the right of the middle blue line, thus excluding all points (x, y) with $x < 5$. Next, compare (9, 2) to the root's right child, (8, 4). Since the y-coordinate of (9, 2) is less than the y-coordinate of (8, 4), move below the red line on the right, thus excluding all points (x, y) with $y > 4$. The point (9, 2) is now found, since it is the left child of (8, 4).

Problem 4. Finish implementing the `KDT` class.

1. Override the `insert()` method. To insert a new node, find the node that should be the parent of the new node by recursively descending through the tree as in the `find()` method (see Figure 3.2 for a geometric example). Do not allow duplicate nodes in the tree. Note that the `k` attribute will be initialized when a k-d tree object is instantiated.

   The `axis` attribute of the new node will be one more than that axis of the parent node. If the last dimension of the data has been reached, start `axis` over at 0.

2. To prevent a user from altering the tree, disable the `remove()` method. Raise a `NotImplementedError` if the method is called, and allow it to receive any number of arguments. (Disabling the `remove()` method ensures that the k-d tree remains the same after it is created. The same k-d tree is used multiple times with different target points to solve the nearest neighbor search problem.)

Using a k-d tree to solve the nearest neighbor search problem requires some care. At first glance, it appears that a procedure similar to `find()` or `insert()` will immediately yield the result. However, this is not always the case (see Figure 3.5).

To correctly find the nearest neighbor, keep track of the target point, the current search node, current best point, and current minimum distance. Start at the root node. Then the current search node and current best point will be root, and the current minimum distance will be the Euclidean distance from `root` to `target`. Then proceed recursively as in the `find()` method. As closer points (nearer neighbors) are found, update the appropriate variables accordingly.

Once the bottom of the tree is reached, a "close" neighbor has been found. However, this is not guaranteed to be the closest point. One way to ensure that this is the closest point is to draw a hypersphere with a radius of the current minimum distance around the candidate nearest neighbor. If this hypersphere does not intersect any of the hyperplanes that split the k-d tree, then this is indeed the closest point.

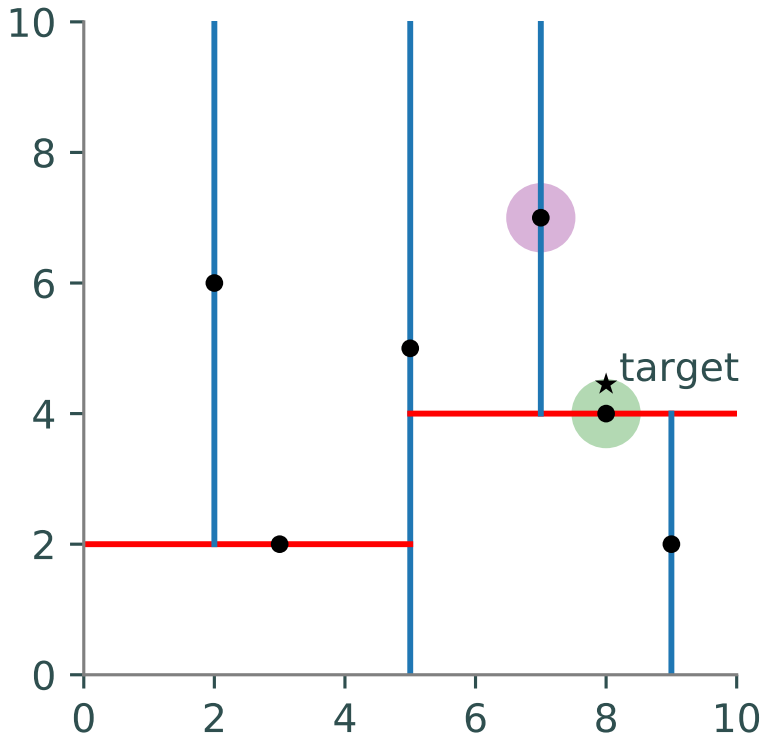


Figure 3.5: To find the point in the k-d tree of Figure 3.2 that is closest to (8, 4.5), first record the distance from the root to the target as the current minimum distance (about 3.04). Then travel down the tree to the right. The right child, (8, 4), is only 0.5 units away from the target (the green circle), so update the minimum distance. Since (8, 4) is not a leaf in the tree, the algorithm could continue down to the left child, (7, 7). However, this leaf node is much further from the target (the purple circle). To ensure that algorithm terminates correctly, check to see if the hypersphere of radius 0.5 around the current node (the green circle) intersects with any other hyperplanes. Since it does not, stop descending down the tree and conclude (correctly) that (8, 4) is the nearest neighbor.

While the correct hypersphere cannot be easily drawn, there is an equivalent procedure that has a straightforward implementation in Python. Before deciding to descend in one direction, add the minimum distance to the i^{th} entry of the target point's data, where i is the `axis` of the candidate nearest neighbor. If this sum is greater that the i^{th} entry of the current search node, then the hypersphere would necessarily intersect one of the hyperplanes drawn by the tree (why?).

The algorithm is summarized below.

Algorithm 3.1 k-d tree nearest neighbor search

```
Given a set of data and a target, build a k-d tree out of the data set.
procedure SEARCH(current, neighbor, dist)
    if current is None then                                   ▷ Base case.
        return neighbor, dist
    index ← current.axis
    if metric(current, target) < dist then
        neighbor ← current                                    ▷ Update the best estimate.
        dist ← metric(current, target)
    if target[index] < current.value[index] then              ▷ Recurse left.
        neighbor, dist ← Search(current.left, neighbor, dist)
        if target[index] + dist ≥ current.value[index] then
            neighbor, dist ← Search(current.right, neighbor, dist)
    else                                                      ▷ Recurse right.
        neighbor, dist ← Search(current.right, neighbor, dist)
        if target[index] - dist ≤ current.value[index] then
            neighbor, dist ← Search(current.left, neighbor, dist)
    return neighbor, dist
Start SEARCH() at the root of the tree.
```

Problem 5. Use Algorithm 3.1 to write a function that solves the nearest neighbor search problem by searching through your `KDT` object. The function should take in a NumPy array of data and a NumPy array representing the target point. Return the nearest neighbor in the data set and the distance from the nearest neighbor to the target point, as in Problem 2 (be sure to return a NumPy array for the neighbor).

To test your function, use SciPy's built-in `KDTree` object. This structure behaves like the `KDT` class, but its operations are heavily optimized. 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))
>>> target = np.random.random(5)
>>> tree = KDTree(data)

# Query the tree and print the minimum distance.
```

```
>>> min_distance, index = tree.query(target)
>>> print(min_distance)
0.309671532426

# Print the nearest neighbor by indexing into the tree's data.
>>> print(tree.data[index])
[ 0.68001084  0.02021068  0.70421171  0.57488834  0.50492779]
```

k-Nearest Neighbors

Previously in the lab, a k-d tree was used to find the nearest neighbor of a target point. A more general problem is to find the k nearest neighbors to a point for some k (using some metric to measure "distance" between data points). The k-nearest neighbors algorithm is a machine learning model. In machine learning, a set of data points has a corresponding set of *labels*, or classifications, that specifies the category of a specific data point in the training set. A machine learning algorithm takes unlabelled data and learns how to classify it. For example, suppose a data set contains the incomes and debt levels of n individuals. Along with this data, there is a set of n data points that state whether an individual has filed for bankruptcy; these points are the labels. The goal of a machine learning model would be to correctly predict whether a new individual would go bankrupt.

Classification

In classification, the k nearest neighbors of a new point are found, and each neighbor "votes" to decide what label to give the new point. The "vote" of each neighbor is its label, or output class. The output class with the highest number of votes determines the label of the new point. See Figure 3.6 for an illustration of the algorithm.

Consider the bankruptcy example. If the 10 nearest neighbors to a new individual are found and 8 of them went bankrupt, then the algorithm would predict that the individual would also go bankrupt. On the other hand, if 7 of the nearest neighbors had not filed for bankruptcy, the algorithm would predict that the individual was at low risk for bankruptcy.

Handwriting Recognition

The problem of recognizing handwritten letters and numbers with a computer has many applications. A computer image may be thought of as a vector in $\mathbb{R}^n$, where n is the number of pixels in the image and the entries represent how bright each pixel is. If two people write the same number, the vectors representing a scanned image of those numbers are expected to be close in the Euclidean metric. This insight means that given a training set of scanned images along with correct labels, the label of a new scanned image can be confidently inferred.

Problem 6. Write a class that performs the k-nearest neighbors algorithm. The constructor should accept a NumPy array of data (the training set) and a NumPy array of corresponding labels for that data. Use SciPy's `KDTree` class to build a k-d tree from the training set in the constructor.

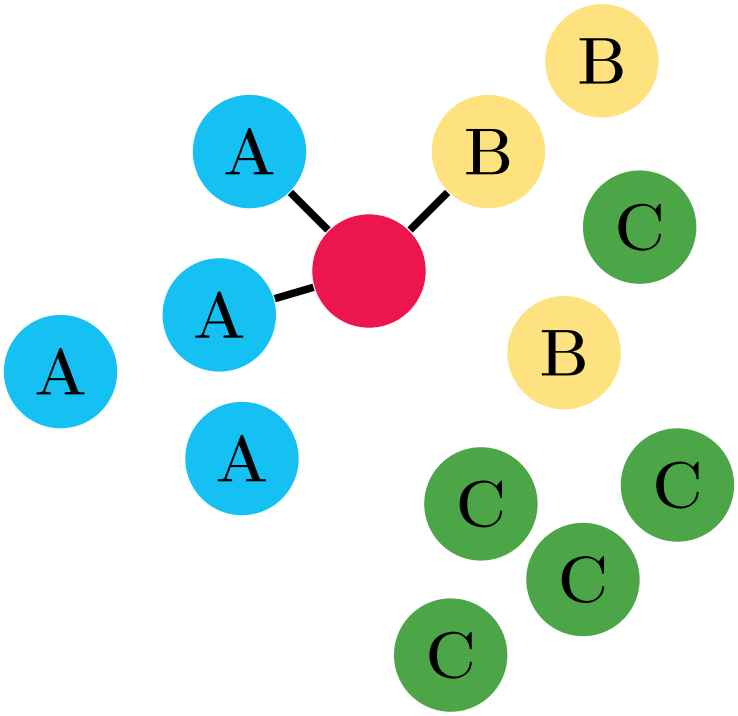


Figure 3.6: In this example, the red node is the new point that needs to be classified. Its three nearest neighbors are two type A nodes and one type B node. The red node is classified as type A since that is the most common label of its three nearest neighbors.

Write a method for this class that accepts a NumPy array of new data points and an integer `k`. Use the k-nearest neighbors algorithm to predict the label of the new data points. Return a NumPy array of the predicted labels. In the case of ties between labels, choose the label with the smaller identifier (hint: try `scipy.stats.mode()`).

To test your classifier, use the data file `PostalData.npz`. This is a collection of labeled handwritten digits that the United States Postal Service has made available to the public. This data set may be loaded by using the following command:

```
points, testpoints, labels, testlabels = np.load('PostalData.npz').items()
```

The first entry of each array is a name, so `points[1]` and `labels[1]` are the actual points and labels to use. Each point is an image that is represented by a flattened 28×28 matrix of pixels (so each image is a NumPy array of length 784). The corresponding label indicates the number written and is represented by an integer.

Use `labels[1]` and `points[1]` to initialize the classifier. Use `testpoints[1]` for predicting output classes. Choose a random data point from `testpoints[1]`. Plot this point with the following code:

```
import matplotlib.pyplot as plt
plt.imshow(img.reshape((28,28)), cmap="gray")
```

```
plt.show()
```

where `img` is the random data point (a NumPy array of length 784). Your plot should look similar to Figure 3.7.

Use your classifier to predict the label of this data point. Does the output match the number you plotted? Compare the output of your classifier to the corresponding label in `testlabels[1]`. They should be the same.

A similar classification process is used by the United States Postal Service to automatically determine the zip code written or printed on a letter.

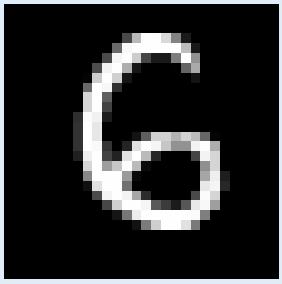


Figure 3.7: The number 6 taken from the data set.

Additional Material

sklearn

The `sklearn` package contains powerful tools for solving the nearest neighbor problem. To start nearest neighbors classification, import the `neighbors` module from `sklearn`. This module has a class for setting up a k-nearest neighbors classifier.

```
# Import the neighbors module
>>> from sklearn import neighbors

# Create an instance of a k-nearest neighbors classifier.
# 'n_neighbors' determines how many neighbors to find.
# 'weights' may be 'uniform' or 'distance.'  The 'distance' option
#     gives nearer neighbors a more weighted vote.
# 'p=2' instructs the class to use the Euclidean metric.
>>> nbrs = neighbors.KNeighborsClassifier(n_neighbors=8, weights='distance', p↩
    =2)
```

The `nbrs` object has two useful methods for classification. The first, `fit`, takes arrays of data (the training set) and labels and puts them into a k-d tree. This is used to find the k-nearest neighbors, much like the `KDT` class implemented previously in the lab.

```
# 'points' is some NumPy array of data
# 'labels' is a NumPy array of labels describing the data in points.
>>> nbrs.fit(points, labels)
```

The second method, `predict`, does a k-nearest neighbor search on the k-d tree and uses the result to attach a label to unlabeled points.

```
# 'testpoints' is an array of unlabeled points.
# Perform the search and calculate the accuracy of the classification.
>>> prediction = nbrs.predict(testpoints)
>>> nbrs.score(testpoints, testlabels)
```

More information about this module can be found at `http://scikit-learn.org/stable/modules/neighbors.html`.

4 Breadth-First Search

Lab Objective: *Graph theory has many practical applications. A graph may represent a complex system or network, and analyzing the graph often reveals critical information about the network. In this lab we learn to store graphs as adjacency dictionaries, implement a breadth-first search to identify the shortest path between two nodes, then use the NetworkX package to solve the so-called "Kevin Bacon problem."*

Graphs in Python

Computers can represent mathematical graphs using various kinds of data structures. In previous labs, we stored graphs as trees and linked lists. For non-tree graphs, perhaps the most common data structure is an adjacency matrix, where each row of the matrix corresponds to a node in the graph and the entries of the row indicate which other nodes the current node is connected to. For more on adjacency matrices, see chapter 2 of the Volume II text.

Another common graph data structure is an *adjacency dictionary*, a Python dictionary with a key for each node in the graph. The dictionary values are lists containing the nodes that are connected to the key. For example, the following is an adjacency dictionary for the graph in Figure 4.1:

```
# Python dictionaries are used to store adjacency dictionaries.
>>> adjacency_dictionary = {'A':['B', 'C', 'D', 'E'], 'B':['A', 'C'],
'C':['B', 'A', 'D'], 'D':['A', 'C'], 'E':['A']}

# The nodes are stored as the dictionary keys.
>>> print(adjacency_dictionary.keys())
['A', 'C', 'B', 'E', 'D']

# The values are the nodes that the key is connected to.
>>> print(adjacency_dictionary['A'])
>>> ['B', 'C', 'D', 'E']            # A is connected to B, C, D, and E.
```

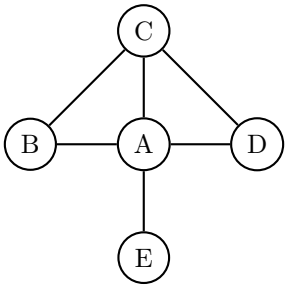


Figure 4.1: A simple graph with five vertices.

Problem 1. Implement the `__str__()` method in the provided `Graph` class. Print each node in the graph, followed by a list of the neighboring nodes separated by semicolons.
(Hint: consider using the `join()` method for strings.)

```
>>> my_dictionary = {'A':['C', 'B'], 'C':['A'], 'B':['A']}
>>> graph = Graph(my_dictionary)
>>> print(graph)
A: C; B
C: A
B: A
```

Breadth-First Search

Many graph theory problems are solved by finding the shortest path between two nodes in the graph. To find the shortest path, we need a way to strategically search the graph. Two of the most common searches are depth-first search (DFS) and breadth-first search (BFS). In this lab, we focus on BFS.

A BFS traverses a graph as follows: begin at a starting node. If the starting node is not the target node, explore each of the starting node's neighbors. If none of the neighbors are the target, explore the neighbors of the starting node's neighbors. If none of those neighbors are the target, explore each of their neighbors. Continue the process until the target is found.

As an example, we will do a programmatic BFS on the graph in Figure 4.1 one step at a time. Suppose that we start at node C and we are searching for node E.

```
# Start at node C
>>> start = 'C'
>>> current = start

# The current node is not the target, so check its neighbors
>>> adjacency_dictionary[current]
['B', 'A', 'D']

# None of these are E, so go to the first neighbor, B
>>> current = adjacency_dictionary[start][0]
>>> adjacency_dictionary[current]
```

```
['A', 'C']

# None of these are E either, so move to the next neighbor
# of the starting node, which is A
>>> current = adjacency_dictionary[start][1]
>>> adjacency_dictionary[current]
['B', 'C', 'D', 'E']

# The last entry of this list is our target node, and the search terminates.
```

You may have noticed that some problems in the previous approach that would arise in a more complicated graph. For example, what prevents us from entering a cycle? How can we algorithmically determine which nodes to visit as we explore deeper into the graph?

Implementing Breadth-First Search

We solve these problems using a queue. Recall that a *queue* is a type of limited-access list. Data is inserted to the back of the queue, but removed from the front. Refer to the end of the Data Structures I lab for more details.

A queue is helpful in a BFS to keep track of the order in which we will visit the nodes. At each level of the search, we add the neighbors of the current node to the queue. The `collections` module in the Python standard library has a `deque` object that we will use as our queue.

```
# Import the deque object and start at node C
>>> from collections import deque
>>> current = 'C'

# The current node is not the target, so add its neighbors to the queue.
>>> visit_queue = deque()
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
deque(['B', 'A', 'D'])

# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
deque(['A', 'D'])

# This isn't the node we're looking for, but we may want to explore its
# neighbors later. They should be explored after the other neighbors
# of the first node, so add them to the end of the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
```

```
deque(['A', 'D', 'A', 'C'])
```

We have arrived at a new problem. The nodes A and C were added to the queue to be visited, even though C has already been visited and A is next in line.

We can prevent these nodes from being added to the queue again by creating a set of nodes to contain all nodes that have already been visited, or that are marked to be visited. Checking set membership is very fast, so this additional data structure has minimal impact on the program's speed (and is faster than checking the deque).

In addition, we keep a list of the nodes that have actually been visited to track the order of the search. By checking the set at each step of the algorithm, the previous problems are avoided.

```
>>> current = 'C'
>>> marked = set()
>>> visited = list()
>>> visit_queue = deque()

# Visit the start node C.
>>> visited.append(current)
>>> marked.add(current)

# Add the neighbors of C to the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     # Since each neighbor will be visited, add them to marked as well.
...     marked.add(neighbor)
...
# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
['A', 'D']

# Visit B. Since it isn't the target, add B's neighbors to the queue.
>>> visited.append(current)
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     marked.add(neighbor)
...

# Since C is visited and A is in marked, the queue is unchanged.
>>> print(visit_queue)
deque(['A', 'D'])
```

Problem 2. Implement the `traverse()` method in the `Graph` class using a BFS. Start from a specified node and proceed until all nodes in the graph have been visited. Return the list of visited nodes. If the starting node is not in the adjacency dictionary, raise a `ValueError`.

Shortest Path

In a BFS, as few neighborhoods are explored as possible before finding the target. Therefore, the path taken to get to the target must be the shortest path.

Examine again the graph in Figure 4.1. The shortest path from C to E is start at C, go to A, and end at E. During a BFS, A is visited because it is one of C's neighbors, and E is visited because it is one of A's neighbors. If we knew programmatically that A was the node that visited E, and that C was the node that visited A, we could retrace our steps to reconstruct the search path.

To implement this idea, initialize a new dictionary before starting the BFS. When a node is added to the visit queue, add a key-value pair to the dictionary where the key is the node that is visited and the value is the node that is visiting it. When the target node is found, step back through the dictionary until arriving at the starting node, recording each step.

Problem 3. Implement the `shortest_path()` method in the `Graph` class using a BFS. Begin at a specified starting node and proceed until a specified target is found. Return a list containing the node values in the shortest path from the start to the target (including the endpoints). If either of the inputs are not in the adjacency graph, raise a `ValueError`.

Network X

NetworkX is a Python package for creating, manipulating, and exploring large graphs. It contains a graph object constructor as well as methods for adding nodes and edges to the graph. It also has methods for recovering information about the graph and its structure.

A node can be an int, a string, or a Python object, and an edge can be weighted or unweighted. There are several ways to add nodes and edges to the graph, some of which are listed below.

Function	Description
`add_node()`	Add a single node to the graph.
`add_nodes_from()`	Add a list of nodes to the graph.
`add_edge()`	Add an edge between two nodes.
`add_edges_from()`	Add a list of edges to the graph.

```
# Create a new graph object using networkX
>>> import networkx as nx
>>> nx_graph = nx.Graph()

>>> nx_graph.add_node('A')
>>> nx_graph.add_node('B')
>>> nx_graph.add_edge('A', 'B')
>>> nx_graph.add_edge('A', 'C') # Node 'C' is added to the graph
```

```
>>> nx_graph.add_edges_from([('A', 'D'), ('A', 'E'), ('B', 'C')])

# Graph edges can also have assigned weights.
>>> nx_graph.add_edge('C', 'D', weight=0.5)

# Access the nodes and edges.
>>> print(nx_graph.nodes())
['A', 'C', 'B', 'E', 'D']

>>> print(nx_graph.edges())
[('A', 'C'), ('A', 'B'), ('A', 'E'), ('A', 'D'), ('C', 'B'), ('C', 'D')]

>>> print(nx_graph.get_edge_data('C', 'D'))
{'weight': 0.5}

# Small graphs can be visualized with nx.draw().
>>> from matplotlib import pyplot as plt
>>> nx.draw(nx_graph)
>>> plt.show()
```

Problem 4. Write a function that accepts an adjacency dictionary. Create a `networkx` object, load it with the graph information from the dictionary, and return it.

The Kevin Bacon Problem

The 6 Degrees of Kevin Bacon is a well-known parlor game. The game is played by naming an actor, then trying to find a chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Captain America: The First Avenger* with Peter Stark, who was in *X-Men: First Class* with Kevin Bacon. Thus Samuel L. Jackson has a *Bacon number* of 2. Any actors who have been in a movie with Kevin Bacon have a Bacon number of 1.

Problem 5. Write a `BaconSolver` class to solve the Kevin Bacon problem.

The file `movieData.txt` contains data from about 13,000 movies released over the course of several years. A single movie is listed on each line, followed by a sequence of actors that starred in it. The movie title and actors' names are separated by a '/' character. The actors are listed by last name first, followed by their first name.

Implement the constructor of `BaconSolver`. Accept a filename to pull data from and generate a dictionary where each key is a movie title and each value is a list of the actors that appeared in the movie. Store the collection of values in the dictionary (the actors) as a class attribute, avoiding duplicates. Convert the dictionary to a NetworkX graph and store it as another class attribute. Note that in the graph, actors only have movies as neighbors, and movies only have actors as neighbors.
(Hint: recall the `split()` method for strings.)

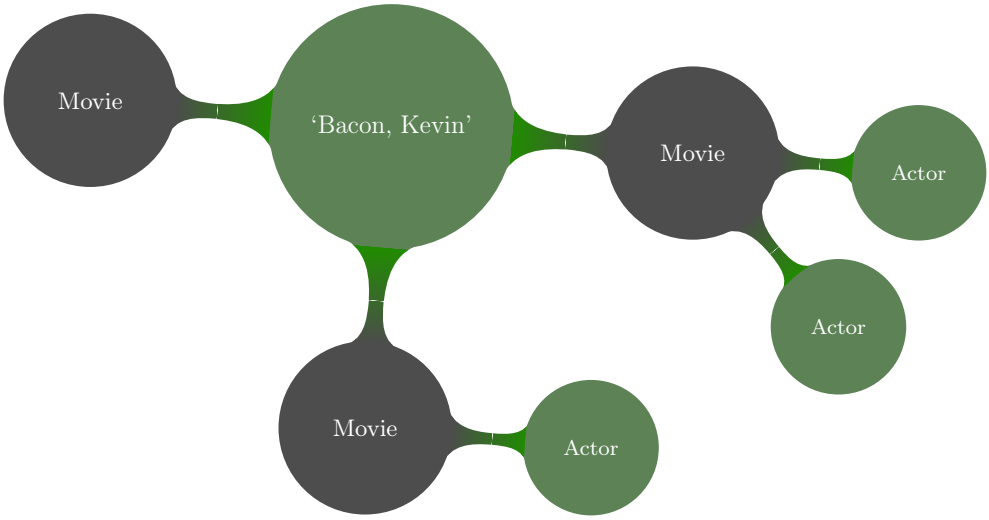


Figure 4.2: A small subgraph demonstrating the structure of the bacon graph.

Achtung!

Because of the size of the dataset, **do not** attempt to visualize the graph with `nx.draw()`. The visualization tool in NetworkX is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research.

NetworkX is equipped with a variety of methods to aid in the analysis of graphs. A selected few are listed below.

Function	Description
`adjacency_matrix()`	Returns the adjacency matrix as a SciPy sparse matrix.
`dijkstra_path()`	Returns the shortest path from a source to a target in a weighted graph.
`has_path()`	Returns `True` if the graph has a path from a source to a target.
`prim_mst()`	Returns a minimum spanning tree for a weighted graph.
`shortest_path()`	Returns the shortest path from a source to a target.

```
# Verify nx_graph has a path from 'C' to 'E'
>>> nx.has_path(nx_graph, 'C', 'E')
True

# The shortest_path method is implemented with a
# bidirectional BFS (starting from both ends)
>>> nx.shortest_path(nx_graph, 'C', 'E')
['C', 'A', 'E']
```

Problem 6. Use NetworkX to implement the `path_to_bacon()` method. Accept start and target values (actors' names) and return a list with the shortest path from the start to the target. Set Kevin Bacon as the default target. If either of the inputs are not contained in the stored collection of dictionary values (if either input is not an actor's name), raise a `ValueError`.

```
>>> movie_graph = BaconSolver("movieData.txt")
>>> movie_graph.path_to_bacon("Jackson, Samuel L.")
['Jackson, Samuel L.', 'Captain America: The First Avenger', 'Stark,
Peter', 'X-Men: First Class', 'Bacon, Kevin']
```

Problem 7. Implement the `bacon_number()` method in the `BaconSolver` class. Accept start and target values and return the number of actors in the shortest path from start to target. Note that this is different than the number of entries in the shortest path list, since movies do not contribute to an actor's Bacon number.

Also implement the `average_bacon()` method. Compute the average Bacon number across all of the actors in the stored collection of actors. Exclude any actors that are not connected to Kevin Bacon (their theoretical Bacon number would be infinity). Return the average Bacon number and the number of actors not connected at all to Kevin Bacon.

As an aside, the prolific Paul Erdős is considered the Kevin Bacon of the mathematical community. Someone with an "Erdős number" of 2 co-authored a paper with someone who co-authored a paper with Paul Erdős.

Additional Material

Depth-First Search

A *depth-first search* (DFS) takes the opposite approach of a breadth-first search. 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. Then the algorithm back tracks to the previous node and checks its second neighbor. A DFS is sometimes more useful than a BFS, but a BFS is usually better[1] at finding the shortest path.

Consider adding a keyword argument to the `traverse()` method of the `Graph` class that specifies whether to use a BFS (the default) or a DFS. This can be done by changing a single line of the BFS code.

Improvements to the BaconSolver Class

Consider adding a `plot_bacon()` method in the `BaconSolver` class that produces a simple histogram displaying the frequency of the Bacon numbers in the data set.

[1] `https://xkcd.com/761/`

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 apply a Markov chain to a natural language processing problem.*

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 die 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.

Finite Markov chains have an associated *transition matrix* that stores the information about the 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.

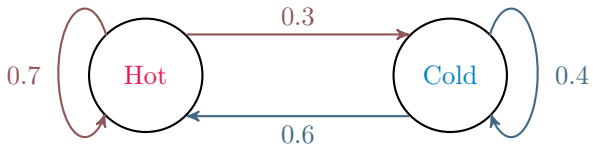
> **Note**
>
> A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the (i, j)th entry of the matrix is the probability of moving from state i to state j. Both representations are common, but in this lab we exclusively use column stochastic transition matrices for consistency.

Consider a very simple weather model where the probability of being hot or cold depends on the weather of the previous day. If the probability that tomorrow is hot given that today is hot is 0.7, and the probability that tomorrow is cold given that today is cold is 0.4, then by assigning hot to the 0th row and column, and cold to the 1^{st} row and column, this Markov chain has the following transition matrix.

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

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. Transition matrices for Markov chains are efficiently stored as NumPy arrays. Write a function that accepts an integer n and returns the transition matrix for a random Markov chain with n states.
(Hint: use array broadcasting to avoid looping.)

Simulating State Transitions

A single draw from a *binomial distribution* with parameters n and p indicates the number of successes out of n independent experiments, each with probability p of success. The classic example is a series of coin flips, where p is the probability that the coin lands heads side up. NumPy's `random` module has an efficient tool, `binomial()`, for drawing from a binomial distribution.

```
>>> import numpy as np

# Draw from the binomial distribution with n = 1 and p = .5 (flip 1 coin).
>>> np.random.binomial(1, .5)
0                           # The coin flip resulted in tails.
```

Consider again the simple weather model and suppose that today is hot. The column that corresponds to "hot"in the transition matrix is $[0.7, 0.3]$. To determine whether tomorrow is hot or cold, draw from the binomial distribution with $n = 1$ and $p = 0.3$. If the draw is 1, which has 30% likelihood, then tomorrow is cold. If the draw is 0, which has 70% likelihood, then tomorrow is hot. The following function implements this idea.

```
def forecast():
    """Forecast tomorrow's weather given that today is hot."""
    transition = np.array([[0.7, 0.6], [0.3, 0.4]])
```

```
    # Sample from a binomial distribution to choose a new state.
    return np.binomial(1, transition[1, 0])
```

Problem 2. Modify `forecast()` so that it accepts an integer parameter `days` and runs a simulation of the weather for the number of days given. Return a list containing the day-by-day weather predictions (0 for hot, 1 for cold). Assume the first day is hot, but do not include the data from the first day in the list of predictions. The resulting list should therefore have `days` entries.

Larger Chains

The `forecast()` function makes one random draw from a binomial distribution to simulate a state change. Larger Markov chains require draws from a *multinomial distribution*, a multivariate generalization of the binomial distribution. A draw from a multinomial distribution parameters n and $(p_1,\ p_2,\ \ldots,\ p_k)$ indicates which of k outcomes occurs in n different experiments. In this case the classic example is a series of dice rolls, with 6 possible outcomes of equal probability.

```
>>> die_probabilities = np.array([1./6, 1./6, 1./6, 1./6, 1./6, 1./6])

# Draw from the multinomial distribution with n = 1 (roll a single die).
>>> np.random.multinomial(1, die_probabilities)
array([0, 0, 0, 1, 0, 0])     # The roll resulted in a 4.
```

Problem 3. Let the following matrix be the transition matrix for a Markov chain modeling weather with four states: hot, mild, cold, and freezing.

	hot	mild	cold	freezing
hot	0.5	0.3	0.1	0
mild	0.3	0.3	0.3	0.3
cold	0.2	0.3	0.4	0.5
freezing	0	0.1	0.2	0.2

Write a new function that accepts an integer parameter and runs the same kind of simulation as `forecast()`, but that uses this new four-state transition matrix. This time, assume that the first day is mild. Return a list containing the day-to-day results (0 for hot, 1 for mild, 2 for cold, and 3 for freezing).

General State Distributions

For a Markov chain with n states, the probability of being in each of the states can be encoded by a single $n \times 1$ vector $\mathbf{x}$, called a *state distribution vector*. The entries of $\mathbf{x}$ must be nonnegative and sum to 1. Then the ith entry x_i of $\mathbf{x}$ is the probability of being in state i. For example, the state distribution vector $\mathbf{x} = [0.8,\ 0.2]^\mathsf{T}$ corresponding to the 2-state weather model of Problem 2 indicates that there is a 80% chance that today is hot and a 20% chance that today is cold. On the other hand, the vector $\mathbf{x} = [0, 1]^\mathsf{T}$ implies that today is, with 100% certainty, cold.

If A is an $n \times n$ transition matrix for a Markov chain and $\mathbf{x}$ is a 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 $\mathbf{x}_5 = [0.8,\ 0.2]^\mathsf{T}$, 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 following significant relation.

$$\mathbf{x}_k = A\mathbf{x}_{k-1} = A(A\mathbf{x}_{k-2}) = A(A(A\mathbf{x}_{x-3})) = \cdots = A^k\mathbf{x}_0$$

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, something curious happens to 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 $k \to \infty$, the entries of A^k converge, written as follows.

$$\lim_{k\to\infty} A^k = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix}. \tag{5.1}$$

In addition, for any initial state distribution vector $\mathbf{x}_0 = [a,\ b]^\mathsf{T}$, $a + b = 1$,

$$\lim_{k\to\infty} \mathbf{x}_k = \lim_{k\to\infty} A^k\mathbf{x}_0 = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 2(a+b)/3 \\ (a+b)/3 \end{bmatrix} = \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix}.$$

Thus as $k \to \infty$, $\mathbf{x}_k \to \mathbf{x} = [2/3,\ 1/3]^\mathsf{T}$, 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 $\mathbf{x} = [2/3,\ 1/3]^\mathsf{T}$ 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 with corresponding eigenvalue $\lambda = 1$.

Every 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.[1] In this case, $\lim_{k\to\infty} A^k$ 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. Consider the following example.

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad A^k = \begin{cases} A & \text{if } k \text{ is odd} \\ I & \text{if } k \text{ is even} \end{cases}$$

In this case as $k \to \infty$, 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.

Problem 4. Write a function that accepts an $n \times n$ transition matrix A, a convergence tolerance ϵ, and a maximum number of iterations N. Generate a random state distribution vector $\mathbf{x}_0$ and calculate $\mathbf{x}_{k+1} = A\mathbf{x}_k$ until $\|\mathbf{x}_{k-1} - \mathbf{x}_k\| < \epsilon$. If k exceeds N, 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, use Problem 1 to generate a random transition matrix A. Verify that $A\mathbf{x} = \mathbf{x}$ and that the columns of A^k approach $\mathbf{x}$ as $k \to \infty$. 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 function to validate the results of Problems 2 and 3:

1. Calculate the steady state distributions corresponding to the transition matrices for each simulation.
2. Run each simulation for a large number of days and verify that the results match the steady state distribution (for example, check that approximately 2/3 of the days are hot for the smaller weather model).

[1] This is a consequence of the *Perron-Frobenius theorem*, which is presented in conjunction with spectral calculus in Volume I.

> **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 version of the power method, together with a discussion of its convergence conditions, is discussed in another lab.

Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study natural languages. 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 Markov chains are, by themselves, insufficient to model 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. Thus a Markov chain simulating transitions between English words is completely unaware of context or even of previous words in a sentence. For example, a Markov chain's current state may be the word "continuous." Then the chain would say that the next word in the sentence is more likely to be "function" rather than "raccoon." However, without the context of the rest of the sentence, even two likely words stringed together may result in gibberish.

We restrict ourselves to a subproblem of modeling the English of a specific file. The transition probabilities of the resulting Markov chain reflect the sort of English that the source authors speak. Thus the Markov chain built from *The Complete Works of William Shakespeare* differs greatly from, say, the Markov chain built from a collection of academic journals. We call the source collection of works in the next problems the *training set*.

Making the Chain

With the weather models of the previous sections, we chose a fixed number of days to simulate. However, English sentences are of varying length, so we do not know beforehand how many words to choose (how many state transitions to make) before ending the sentence. To capture this feature, we include two extra states in our Markov model: a *start state* (\$tart) marking the beginning of a sentence, and a *stop state* (\$top) marking the end. Thus a training set with N unique words has an $(N+2) \times (N+2)$ transition matrix.

The start state 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 transition to the stop state. The stop state is called an *absorbing state* because once we reach it, we cannot transition back to another state.

After determining the states in the Markov chain, we need to determine the transition probabilities between the states and build the corresponding transition matrix. Consider the following small training set from Dr. Seuss as an example.

```
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.
```

If we include punctuation (so "ham?" and "ham." are counted as distinct words) and do not alter the capitalization (so "Do" and "do" are also different), there are 15 unique words in this training set:

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 17×17. Each state must be assigned a row and column index in the transition matrix. As easy way to do this is to assign the states an index based on the order that they appear in the training set. Thus our states and the corresponding indices will be as follows:

\$start	I	am	Sam	...	ham.	\$top
0	1	2	3	...	15	16

The start state should transition to the words "I" and "Do", and the words "am.", "ham?", and "ham." should each transition to the stop state. We first count the number of times that each state transitions to another state:

	\$start	I	am	Sam		ham.	\$top
\$start	0	0	0	0	...	0	0
I	3	0	0	2	...	0	0
am	0	1	0	0	...	0	0
Sam	0	0	1	1	...	0	0
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
ham.	0	0	0	0	...	0	0
\$top	0	0	0	0	...	1	1

Now divide each column by its sum so that each column sums to 1.

	\$start	I	am	Sam		ham.	\$top
\$start	0	0	0	0	...	0	0
I	3/4	0	0	2/3	...	0	0
am	0	1/5	0	0	...	0	0
Sam	0	0	1	1/3	...	0	0
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
ham.	0	0	0	0	...	0	0
\$top	0	0	0	0	...	1	1

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 tell us 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. Finally, to avoid a column of zeros, we place a 1 in the bottom right hand corner of the matrix (so the stop state always transitions to itself).

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.

```
procedure MakeTransitionMatrix
    Count the number of unique words in the training set.
    Initialize a square array of zeros of the appropriate size to be the transition
        matrix (remember to account for the start and stop states).
    Initialize a list of states, beginning with "$tart".
    for each sentence in the training set do
        Split the sentence into a list of words.
        Add each new word in the sentence to the list of states.
        Convert the list of words into a list of indices indicating which row and
            column of the transition matrix each word corresponds to.
        Add 1 to the entry of the transition matrix corresponding to
            transitioning from the start state to the first word of the sentence.
        for each consecutive pair (x, y) of words in the list of words do
            Add 1 to the entry of the transition matrix corresponding to
                transitioning from state x to state y.
        Add 1 to the entry of the transition matrix corresponding to
            transitioning from the last word of the sentence to the stop state.
    Make sure the stop state transitions to itself.
    Normalize each column by dividing by the column sums.
```

Problem 5. Write a class called `SentenceGenerator`. 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.

You may assume that the file has one complete sentence written on each line, and your implementation may be either column- or row-stochastic.

Problem 6. Add a method to the `SentenceGenerator` class called `babble()`. Begin at the start state and use the strategy from Problem 3 to repeatedly transition through the object's Markov chain. Keep track of the path through the chain and the corresponding sequence of words. When the stop state is reached, stop transitioning to terminate the simulation. Return the resulting sentence as a single 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

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 your `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 example 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 $tartParagraph state that always transitions to $tartSentence and a $topParagraph state that is sometimes transitioned to from $topSentence.
- Let the states be individual letters instead of individual words. Be sure to include a state for the spaces between words. We will explore this particular state space choice more in Volume III together with hidden Markov models.
- 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, $i = 1, \ldots, n$, 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. For example, `nltk.sent_tokenize()` reads a single string and splits it up into sentences.

```
>>> 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

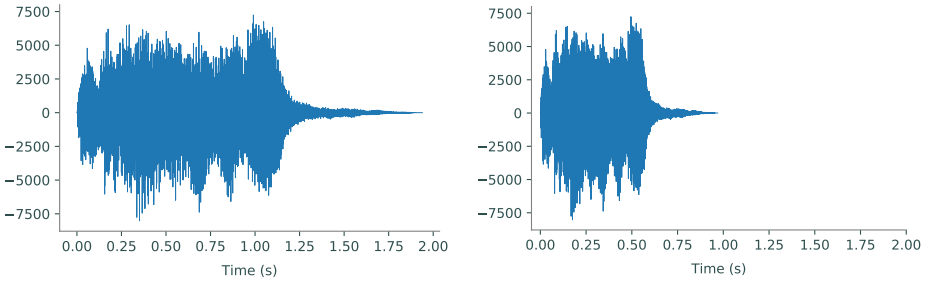
Lab Objective: *The analysis of periodic functions has many applications in pure and applied mathematics, especially in settings dealing with sound waves. The Fourier transform provides a way to analyze such periodic functions. In this lab, we use Python to work with digital audio signals and implement the discrete Fourier transform. We use the Fourier transform to detect frequencies present in a given sound wave. We recommend this lab be done in a Jupyter Notebook.*

Sound Waves

Sound is how vibrations are perceived in matter. These vibrations travel in waves. Sound waves have two important characteristics that determine what is heard, or whether or not it can be heard. The first characteristic is *frequency*, which is a measurement of the number of vibrations in a certain time period, and determines the pitch of the sound. Only certain frequencies are perceptible to the human ear. The second characteristic is *intensity* or *amplitude*, and determines the volume of the sound. Sound waves correspond physically to continuous functions, but computers can approximate sound waves using discrete measurements. Indeed, discrete measurements can be made indistinguishable to the human ear from a truly continuous wave. Usually, sound waves are of a sinusoidal nature (with some form of decay); the frequency is related to the wavelength, and the intensity to the wave amplitude.

Digital Audio Signals

Computer use *digital audio signals* to approximate sound waves. These signals have two key components that relate to the frequency and amplitude of sound waves: samples and sampling rate. A sample is a measurement of the amplitude of a sound wave at a specific instant in time. The sampling rate corresponds to the sound frequency.



(a) The plot of `tada.wav`.

(b) Compressed plot of `tada.wav`

Figure 6.1: The 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 (in this case, 22050), while the sample rate of the plot on the right has been doubled, resulting in a compression of the actual sound when played back. The compressed sound will be shorter and have a higher pitch. Similarly, if this set of samples were played back with a lower sample rate will be stretched and have a lower pitch.

To see why the sample rate is necessary, consider an array with samples from a sound wave. If how frequently those samples were collected is unknown, then the sound wave can be arbitrarily stretched or compressed to make a variety of sounds. See Figure **??** for an illustration of this principle.

However, if the rate at which a set of samples is taken is known, the wave can be reconstructed exactly as it was recorded. If the sample rate is unknown, then the frequencies will be unknown. In most applications, this sample rate will be measured in the number of samples taken per second, Hertz (Hz). 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 in a `wav` file, use SciPy's `read()` function that returns the file's sample rate and samples.

```
# Read from the sound file.
>>> from scipy.io import wavfile
>>> rate, wave = wavfile.read('tada.wav')
```

It is sometimes useful to visualize a sound wave by plotting time against the amplitude of the sound, as in Figure **??**. This type of plotting plots in the *time domain*. The amplitude of the sound at a given time is just the value of the sample at that time. Note that 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.

Problem 1. Make a class called `SoundWave` for storing digital audio signals. Write the constructor and have it accept a sample rate (integer) and an array of samples (NumPy array), which it stores as class attributes. Then, write a method called `plot()` that generates the graph of the sound wave. Use the sample rate to label the x-axis in terms of seconds. Finally, construct a `SoundWave` object using the data in `tada.wav` and display its plot. Your plot should look like Figure 6.1a.

Scaling

Writing a sound wave to a file is slightly more complicated than reading from a file. Use `wavfile.write()`, specifying the name of the new file, the sample rate, and the array of samples.

```
# Write a random signal sampled at a rate of 44100 Hz to my_sound.wav.
>>> wave = np.random.randint(-32767, 32767, 30000)
>>> samplerate = 44100
>>> wavfile.write('my_sound.wav', samplerate, wave)
```

In order for the `wavfile.write()` function to accurately write an array of samples to a file, the samples must be of an appropriate type. There are four types of sample values that the function will accept: 32-bit floating point, 32-bit integers, 16-bit integers, and 8-bit unsigned integers. If a different type of samples are passed into the function, it's possible that the function will write to a file, but the sound will likely be distorted in some way. This lab works with only 16-bit integer types, unless otherwise specified.

```
# The type of the elements of an array is stored in an attribute called dtype.
>>> x = np.array([1, 2, 3])
>>> y = np.array([1.0, 2.0, 3.0])
>>> print(x.dtype)
dtype('int64')
>>> print(y.dtype)
dtype('float64')
```

A 16-bit integer is an integer between -32767 and 32767. If an array of samples does not already have all of its elements as 16-bit integers, it will need to be scaled in such a way so that is does before it can be written to a file. In addition, it is ideal to scale the samples so that they cover as much of the 16-bit integer range as possible. This will ensure the most accurate representation of the sound.

```
# Generate random samples between -0.5 and 0.5.
>>> samples = np.random.random(30000)-.5
>>> print(samples.dtype)
dtype('float64')
# Scale the wave so that the samples are between -32767 and 32767.
>>> samples *= 32767*2
# Cast the samples as 16-bit integers.
>>> samples = np.int16(samples)
>>> print(samples.dtype)
```

```
dtype('int16')
```

In the above example, the original samples are all less than 0.5 in absolute value. Multiplying the original samples by 32767 and 2 scales the samples appropriately. In general, to get the scaled samples from the original, multiply by 32767 and divide by the greatest magnitude present in the original sample. Note also that for various reasons, samples may sometimes contain complex values. In this case, scale and export only the real part.

Problem 2. Add a method to the `SoundWave` class called `export()` that accepts a file name and generates a `.wav` file of that name from the sample rate and the array of samples. If the array of samples is not already in `int16` format, scale it appropriately before writing to the output file. Use your method to create a differently named file that contains the same sound as `tada.wav`. Display the two sounds to make sure the method works correctly.

> NOTE
>
> To display a sound in a Jupyter Notebook, first import IPython, then use `IPython.display.Audio()`. This function can accept either the name of a `.wav` file present in the same directory, or the keyword arguments `rate` and `data`, which represent the sample rate and the array of samples, respectively. The function will generate a music player that can be played within the Jupyter Notebook.

Creating Sounds in Python

In order to generate a sound in Python, sample the corresponding sinusoidal wave. The example below generates a sound with a frequency of 500 Hertz for 10 seconds.

```
>>> samplerate = 44100
>>> frequency = 500.0
>>> duration = 10.0          # Length in seconds of the desired sound.
```

Recall the the function $\sin(x)$ has a period of 2π. To create sounds, however, the desired period of a wave is 1, corresponding to 1 second. Thus, sample from the function

$$\sin(2\pi x k)$$

where k is the desired frequency.

```
# The lambda keyword is a shortcut for creating a one-line function.
>>> wave_function = lambda x: np.sin(2*np.pi*x*frequency)
```

To generate a sound wave, use the following three steps: First, generate the points at which to sample the wave. Next, sample the wave by passing the sample points into `wave_function()`. Then, use the `SoundWave` class to plot the sound wave or write it to a file.

```
# Calculate the sample points and the sample values.
```

```
>>> sample_points = np.linspace(0, duration, int(samplerate*duration))
>>> samples = wave_function(sample_points)

# Use the SoundWave class to write the sound to a file.
>>> sound = SoundWave(samplerate, samples)
>>> sound.export("example.wav")
```

Problem 3. Write a function that accepts a frequency and a duration. Follow the pattern above to generate and return an instance of the SoundWave class with the given frequency and duration. Use a sample rate of 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
A	440
B	493.88
C	523.25
D	587.33
E	659.25
F	698.46
G	783.99
A	880

The "A" note occurs at a frequency of 440 Hertz. Use your function to generate and display an "A" note being played for 2 seconds.

Problem 4.

1. A chord is a conjunction of several notes played together. You can create a chord in Python by adding several sound waves together. Write the `__add__()` magic method in the SoundWave so that it adds together the samples of two `SoundWave` objects and returns the resulting `SoundWave` object. Note this is only valid if the two sample arrays are the same length. Raise a `ValueError` if the arrays are not the same length.
2. Generate and display a minor chord (made up of the "A", "C", and "E" notes).
3. Add a method called `append()` to the `SoundWave` class that accepts a `SoundWave` object and appends the additional samples from the new object to the end of the samples from the current object. Note this only makes sense if the sample rates of the two objects are the same. Raise a `ValueError` if the sample rates of the two objects are not equal.
4. Finally, generate and display a sound that changes over time.

Discrete Fourier Transform

Under the right conditions, a continuous periodic function may be represented as a sum of sine waves:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \sin kx$$

where the constants c_k are called the *Fourier coefficients*.

Such a transform also exists for discrete periodic functions. Whereas the frequencies present in the continuous case are multiples of a sine wave with a period of 1, the discrete case is somewhat different. The Fourier coefficients in the discrete case represent the amplitudes of sine waves whose periods are multiples of a "fundamental frequency". The fundamental frequency is a sine wave with a period length equal to the amount of time of the sound wave.

The k^{th} coefficient of a sound wave $\{x_0, .., x_{N-1}\}$ is calculated with the following formula:

$$c_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i k n}{N}} \tag{6.1}$$

where i is the square root of -1. This process is done for each k from 0 to $N-1$, where N is the number of sample points. Thus, there are just as many Fourier coefficients as samples from the original signal. The discrete Fourier transform (DFT) is particularly useful when dealing with sound waves. The applications will be discussed further later on in the lab.

Problem 5. Write a function called `naive_DFT()` that accepts a NumPy array and computes the discrete Fourier transform of the array using Equation 6.1. Return the array of calculated coefficients.

SciPy has several methods for calculating the DFT of an array. Use `scipy.fft()` or `scipy.fftpack.fft()` to check your implementation by using your method and the SciPy method to calculate the DFT and comparing the results using `np.allclose()`. The naive method is significantly slower than SciPy's implementation, so test your function only on small arrays. When you have your method working, try to optimize it so that you can calculate each coefficient c_k in just one line of code.

Fast Fourier Transform

Calculating the DFT of a large set of samples using only (6.1) can be incredibly slow. Fortunately, when the size of the samples is a power of 2, the DFT can be implemented as a recursive algorithm by separating the computation of the DFT into its even and odd indices. This method of calculating the DFT is called the *Fast Fourier Transform* (FFT) due to its remarkably improved run time. The following algorithm is a simple implementation of the FFT.

Algorithm 6.1

```
procedure FFT(x)
    N ← size(x)
    if N = 1 then
        return DFT(x)                          ▷ Use the DFT function you wrote for Problem 5.
    else
        even ← FFT(x_{::2})
        odd ← FFT(x_{1::2})
        k ← arange(N)                          ▷ Use np.arange.
        factor ← exp(−2πik/N)
                                               ▷ Note * is component-wise multiplication.
        return concatenate((even + factor_{:N/2} * odd, even + factor_{N/2:} * odd))
```

This algorithm performs significantly better than the naive implementation using only (6.1). However, this simplified version will only work if the size of the input samples is exactly a power of 2. SciPy's FFT functions manage to get around this by padding the sample array with zeros until the size is a power of 2, then executing the remainder of the algorithm from there. In addition, SciPy's functions use various other tricks to speed up the algorithm even further.

Problem 6. Write a function that accepts a NumPy array and computes the discrete Fourier transform of the array using Algorithm 6.1. Return the array of calculated coefficients.

To verify your method works, generate an array of random samples of a size that is a power of 2 (preferably size 1024 or larger) and use `np.allclose()` as in the previous problem to make sure the outputs are the same. Then, compare the runtimes of your DFT method, your FFT method, and one of the SciPy methods and print the results.

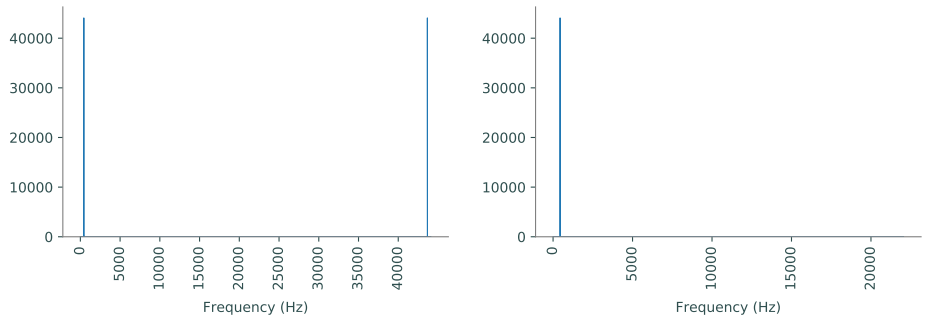
Hint: Concatenating vectors can be done with `np.concatenate`.

Plotting the DFT

The graph of the Fourier transform of a sound file is useful in a variety of applications. While the graph of the sound in the time domain gives information about the amplitude of a sound wave at a given time, the graph of the discrete Fourier transform shows which frequencies are present in the signal. Plotting a sound's DFT is referred to as plotting in the *frequency domain*. Often, this information is of greater importance.

Frequencies present in the sound have non-zero coefficients. The magnitude of these coefficients corresponds to how influential the frequency is in the signal. For example, in the case of the "A" note in Problem 3 the sound contained only one frequency. The graph of the DFT of this sound is Figure 6.2a. Note that in this plot, there are two spikes, despite there being only one frequency present in the sound. This is due to symmetries inherent in the DFT. For the purposes of this lab, ignore the second half of the plot. From now on, show plots of only the first half of the DFT, as in Figure 6.2b.

On the other hand, the DFT of a more complicated sound wave will have many frequencies present. Some of these frequencies correspond to the different tones present in the signal. See Figure 6.3 for an example.



(a) The DFT of the "A" note (with symmetries).

(b) The DFT of the "A" note (without symmetries).

Figure 6.2: Plots of the DFT (with and without symmetries). Notice that the x-axis of the symmetrical plot goes up to 44100, or the sample rate of the sound wave being plotted, while the x-axis of the other plot goes up to half of that. Also notice that the spikes occur at 440.0 Hz and 43660.0 Hz (which is $44100.0 - 440.0$).

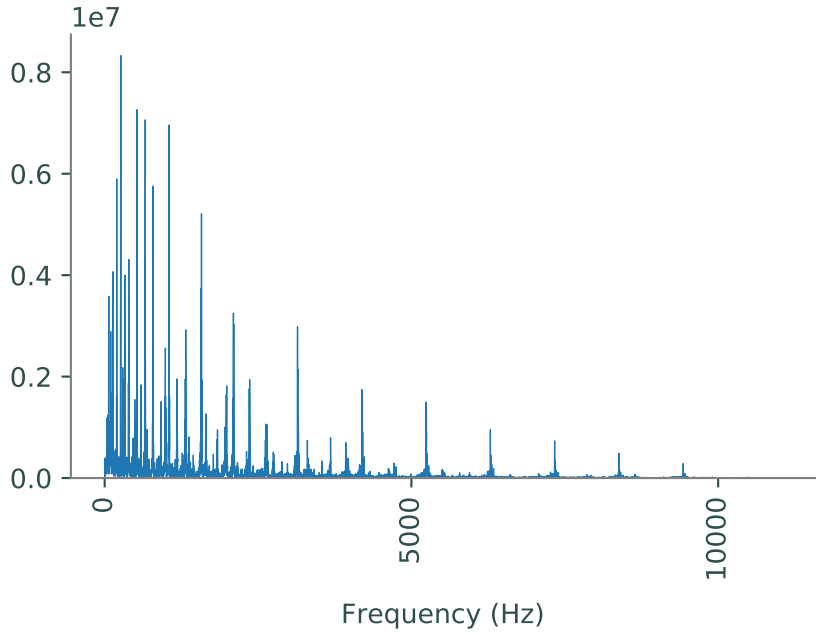


Figure 6.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency present in the signal. Since the sample rate of `tada.wav` is only 22050 instead of the usual 44100, the plot of its DFT without symmetries will only go up to 11025, half of its sample rate.

Fixing the x-axis

Plotting the DFT of a signal without any other considerations results in an x-axis that corresponds to the index of the coefficients in the DFT, not their frequencies. As mentioned earlier, the "fundamental frequency" for the DFT corresponds to a sine wave whose period is the same as the length of the signal. Thus, if unchanged, the x-axis gives the number of times a particular sine wave cycles throughout the whole signal. To label the x-axis with the frequencies measured in Hertz, or cycles per second, the units must be converted. Fortunately, the bitrate is measured in samples per second. Therefore, dividing the frequency (given by the index) by the number of samples and multiplying by the sample rate, results in cycles per second, or Hertz.

$$\frac{\text{cycles}}{\text{samples}} \times \frac{\text{samples}}{\text{second}} = \frac{\text{cycles}}{\text{second}}$$

```
# Calculate the DFT and the x-values that correspond to the coefficients. Then
# convert the x-values so that they measure frequencies in Hertz.
>>> dft = abs(sp.fft(samples))       # Ignore the complex part.
>>> N = dft.shape[0]
>>> x_vals = np.linspace(1, N, N)
>>> x_vals = x_vals * samplerate / N # Convert x_vals to frequencies
```

Problem 7. Write a method in the `SoundWave` class called `plot_dft()` that plots the frequencies present in a sound wave on the x-axis and the magnitude of those frequencies on the y-axis. Only display the first half of the plot (as in Figure 6.2b). Use one of SciPy's FFT implementations to calculate the DFT.

Display the DFT plots of the 'A' note and of the minor chord.

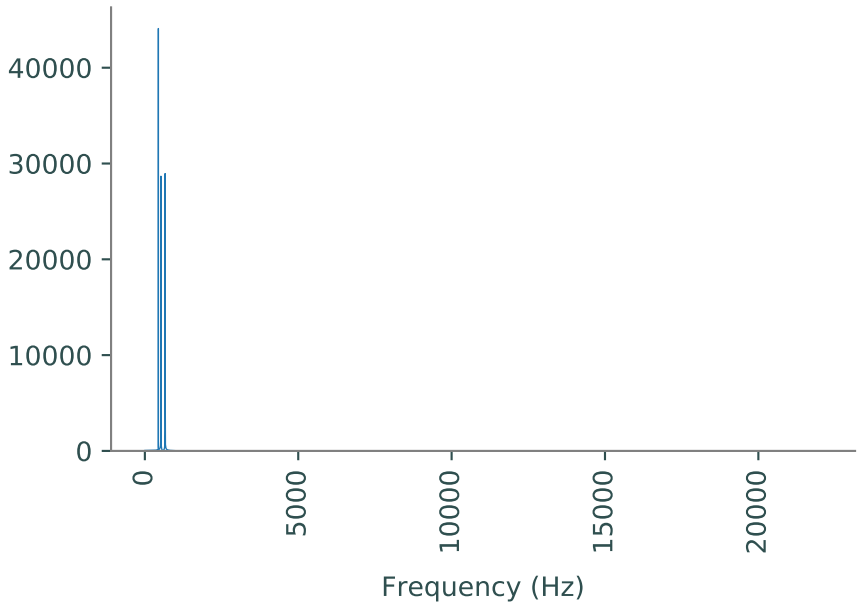


Figure 6.4: The DFT of the minor chord.

Conclusion

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 has many uses, and 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 minor chord as an example. The plot of its DFT looks like Figure 6.4. This graph shows that there are three main frequencies present in the sound. It remains to determine what exactly those frequencies are. There are many valid ways to do this. One possibility is to determine which indices of the array of DFT coefficients have the largest values, then scale these indices the same as the x-axis of the plot to determine to which frequencies these values correspond. This task is explored further in the next problem.

Problem 8. The file `mystery_sound.wav` contains an unknown chord. Use what you have learned about the DFT to determine the individual notes present in the sound.

Hints: The function `np.argmax()` may be useful. Also, remember that the DFT is symmetric, meaning the last half of the array of DFT coefficients will need to be ignored.

7 Filtering and Convolution

Lab Objective: *The Fourier transform reveals information in the frequency domain about signals and images that might not be apparent in the time (sound) or spatial domain (image). It provides an extremely powerful tool to analyze difficult problems simply. In this lab, we learn how to use the Fourier transform to effectively convolve sound signals and filter out unwanted noise.*

Convolution

Mixing two sounds signals is a common method used in signal processing and analysis. This is done through a discrete *convolution*. Given two periodic sound samples f and g of length n, the convolution of these two samples is an n-dimensional vector where the kth component is given by

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

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples. For example, a signal that is one minute long would have 2646000 samples if the signal was sampled at 44100 per second (which is the standard rate). Therefore, convolving samples using the naïve method of convolution (7.1) is very computationally expensive and often infeasible.

Fortunately, the Fourier transform can calculate convolutions quickly. The Convolution Theorem states that

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g), \tag{7.2}$$

where $\mathcal{F}$ is the discrete Fourier transform, $*$ is convolution, and $\cdot$ is component-wise multiplication. Thus, the convolution of two arrays is

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)), \tag{7.3}$$

where $\mathcal{F}^{-1}$ is the inverse discrete Fourier transform.

Recall that although the samples used are real numbers, taking the inverse Fourier transform of the samples may have small complex components due to rounding errors. To avoid this error, remember to take the real part of the inverse Fourier transform.

Circular Convolution

The Convolution Theorem requires that the samples be periodic in the time domain. Due to the mathematical properties of the Fourier transform, the result of this convolution is a *circular convolution*. This means that the convolution is assumed to be periodic and the sounds at the end of the signal will tend to mix with sounds at the beginning of the signal. A circular convolution creates an interesting effect on a signal when convoluted with a segment of white noise; the sound will loop seamlessly from the end back to the beginning.

Problem 1. Use SciPy's `sp.fft()` and `sp.ifft()` to create a `SoundWave` object that is the circular convolution of `tada.wav` with two seconds of white noise. Note that the length of the two samples must be the same, so pad an array of zeros to the end of the `tada.wav` sample to match the length of the noise. You may use `np.append()` to add an array of zeros to the end of a sample.

Make a sample of white noise by using NumPy's `random` module:

```
# Create 2 seconds of mono white noise.
samplerate = 22050
noise = np.int16(np.random.randint(-32767, 32767, samplerate*2))
```

To test your result, use the `append()` method in the `SoundWave` class to add multiple copies of the signal consecutively. Your new signal should have a continuous, seamless transition.

Linear Convolution

Although circular convolution can have unique results, common mixtures of sounds do not have sound at the beginning of a signal to mix with the sound at the end of the signal. This form of convolution is called a *linear convolution*. The linear convolution of two samples with lengths N and M has length $N + M - 1$. The simplest way to achieve this length when using the Convolution Theorem is to pad zeros at the end of both of the samples to have at least lengths $N + M - 1$ and return only the first $N + M - 1$ samples of the convolution.

Problem 2. Write a function that accepts two arrays of sound samples and returns the linear convolution of the samples using the Fourier transform. Make sure to pad the right amount of zeros to the end of both samples to avoid circular convolution and return the correct length of sample.

Print out and compare the time it takes to compute the convolution of the signals of `AEA.wav` and `EAE.wav` using the method you have written with SciPy's `sp.signal.fftconvolve()` function and a naive convolution function using Equation (7.1) given below.

```
def naive_convolve(sample1,sample2):
    sig1 = np.append(sample1, np.zeros(len(sample2)-1))
    sig2 = np.append(sample2, np.zeros(len(sample1)-1))

    final = np.zeros_like(sig1)
    rsig1 = sig1[::-1]
```

```
    for k in range(len(sig1)-1):
        final[i+1] = np.sum((np.append(rsig1[i:],rsig1[::-i][:i]))*sig2)
    return final
```

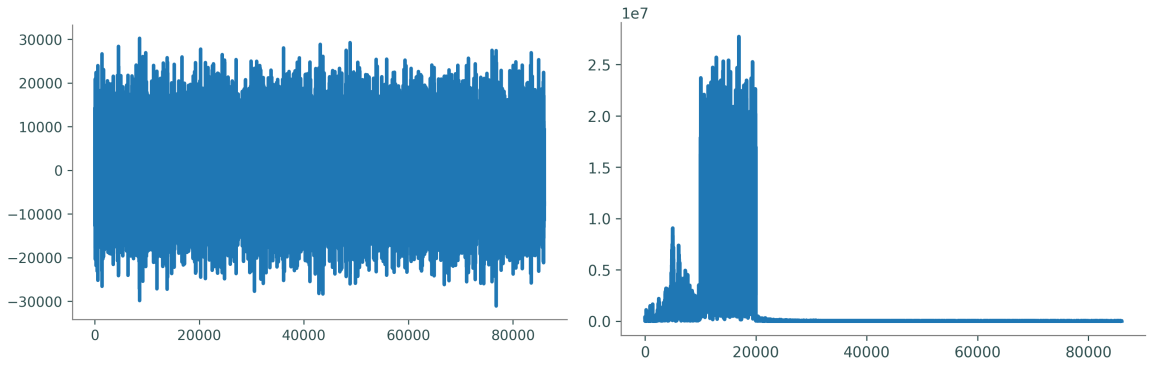
To test the convolution method, listen to the signal created with the fast Fourier transformation convolution. Compare your audio with the convolution created with SciPy's `signal.fftconvolve()`.

Suppose there is a recording of a musical piece played in a small, carpeted room with essentially no acoustics (little or no echo). The discrete linear convolution can mix this signal with an echoing sound to make the piece sound like it were played in a large concert hall with echo.

When a balloon is popped in a large room with echo, the sound resonate 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 individual sounds of an instrument in a carpeted room is convoluted with the impulse response from a concert hall, the new signal will sound as if the instrument is being played in the concert hall.

Problem 3. The `chopin.wav` file is a signal with piano being played in a dead room with little or no acoustics, and the `balloon.wav` file is a recording of a balloon pop in an echoic room. Use SciPy's `signal.fftconvolve()` to take the convolution of `chopin.wav` with `balloon.wav`.

Listen to the new signal, there should be echo in the piano recording.



(a) Signal in the time domain.

(b) Signal in the frequency domain.

Figure 7.1: The `Noisysignal1.wav` signal

Cleaning up Noise

The Fourier transform also reveals important information about images and signals in the frequency domain. For example, Figure 7.1a plots a noisy recording of a voice over time. This plot has a lot of static and does not reveal a lot of information about the signal.

The Fourier transform of the signal in Figure 7.1b, however, reveals that the static in the time domain is the result of some concentrated noise between 1250 Hz to 2500 Hz. Remove this noise at those frequencies to remove the noise of the signal.

Recall that the plot of a signal in the frequency domain is the plot of the discrete Fourier transform (DFT) with the x-axis scaled to reveal the amplitude at each frequency. This was done by multiplying the domain of the DFT by the sampling rate and dividing by the number of samples in the signal.

Hence, to remove the high amplitudes at certain frequencies between between 1250 to 2500 Hz in the signal, find the indices of the DFT array that correspond to those frequencies and set them to zero. In order to find the correct indices of the DFT array corresponding to these frequencies, multiply the frequency by the number of samples and dividing by the sampling rate. Make sure that the index is an integer.

```
# Find the indices of the DFT that corresponds with the frequencies 1250 and ↩
    2500.
>>> low = 1250*len(samples)//rate
>>> high = 2500*len(samples)//rate
```

The plot in Figure 7.1b has also been cut in half since the DFT is symmetric. Therefore, if the coefficient at index j is set to 0, then the coefficient at index $N - j$ must be set to 0 as well, where N is the number of discrete Fourier samples.

```
# Set the chosen coefficients between low and high to 0.
>>> fft_sig[low:high]=0
>>> fft_sig[-high:-low]=0
```

Then calculate the inverse Fourier transform to get a new, clean signal.

The plot of this new signal in the time domain now reveals individual syllables as they are spoken. See Figure 7.2.

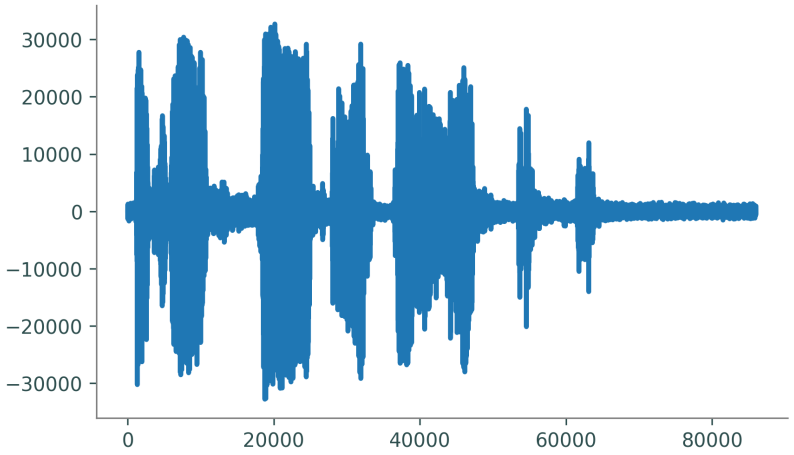


Figure 7.2: The plot of `Noisysignal1.wav` in the time domain after being cleaned.

Problem 4. Write a function that accepts a sound sample, a sample rate and low and high frequencies that define a range of frequencies to remove using the technique described above. It should return an array of samples that has the indicated frequencies removed.

Listen to `Noisysignal2.wav`, which just sounds like random noise. Use the `SoundWave` class to plot the discrete Fourier transform of this signal to see at what frequencies is the noise present. Remove the noise using the function you have written and create a `SoundWave` object from the new signal.

It may be helpful to plot the DFT of the new signal to fine-tune the low and high frequencies chosen to filter out.

Listen to the filtered signal and see if you can recognize the famous person speaking.

When a digital audio signal is played on a computer, the signal is sent to a speaker, which vibrates, producing sound waves. When multiple speakers are used, they can all produce the same signal or they can produce different signal. When there is only one signal, the sound is *monoaural*, or *mono*. When speakers produce more than one signal, the overall signal is *stereophonic*, or *stereo*. Usually stereo means two, but there may be any number of signals (5.1 surround sound, for instance, has 5).

All of the sounds used in this lab so far were mono; hence, the samples were only one dimensional arrays. More commonly, signals are multi-dimensional, and can be treated similarly to mono signals.

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 noise level of these vuvuzelas. To solve this problem, audio filtering techniques were used to cancel out the sound of the vuvuzela which has a frequency of around 200-500 Hz.

Listen to `vuvuzela.wav`[a] and notice the low humming sound of the vuvuzelas in the background. Use the function written previously to create a new `SoundWave` object that removed the vuvuzela noise. Note that the sound file is a stereo sound with two sound signals. The first and second column of the array sample corresponds to the signals for the left and right speaker respectively. Filter out the frequencies of the vuvuzela in each signal. Then combine the two samples back to its original form.

Listen to the resulted signal to see whether the vuvuzela horns has successfully been filtered out.

[a] A clip of `https://www.youtube.com/watch?v=g_0NoBKWCT8`.

The 2-Dimensional FFT

The Fourier transform can be readily extended to any number of dimensions. Computationally, the problem reduces to performing the one-dimensional Fourier transform iteratively along each of the dimensions. This lab focuses only on the Fourier transform of two-dimensional matrices. The Fourier transform of two-dimensional matrices is useful for image denoising, image compression, edge-detection, image enhancement, and more.

Given a matrix A, first calculate the one-dimensional Fourier transform of each column, storing the result column-wise in an array the same shape as A. Then calculate the Fourier transform of each row of this resulting array. This yields the two-dimensional Fourier transform of A. Calculating the two-dimensional inverse Fourier transform is done in a similar fashion, but in the opposite order: first calculate the inverse Fourier transform of the rows, then the columns.

The NumPy module has functions that perform these operations.

```
>>> A = np.array([[5, 3, 1], [4, 2, 7], [8, 9, 3]])
# Calculate the Fourier transform of A.
>>> fft = np.fft.fft2(A)
# Calculate the inverse Fourier transform.
>>> ifft = np.fft.ifft2(fft)
>>> np.allclose(A, ifft)
True
```

Images

Just as the one-dimensional Fourier transform can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove noise in images. By taking the two-dimensional Fourier transform of a noisy, or blurry, image as described above, we can determine the frequencies that are causing the blur and alter them. Taking the inverse Fourier transform of this produces a less-blurry version of the original image. Note that in order for this process to work perfectly, the noise must be periodic and all problem areas in the Fourier transform must be changed correctly. As a result, it is very difficult to completely remove all noise from an image using only the Fourier transform, but depending on the situation, it may be capable of removing enough noise to be useful.

Below is an example of how this process works. The following code refers to figure **??**. First plot the original blurry image.

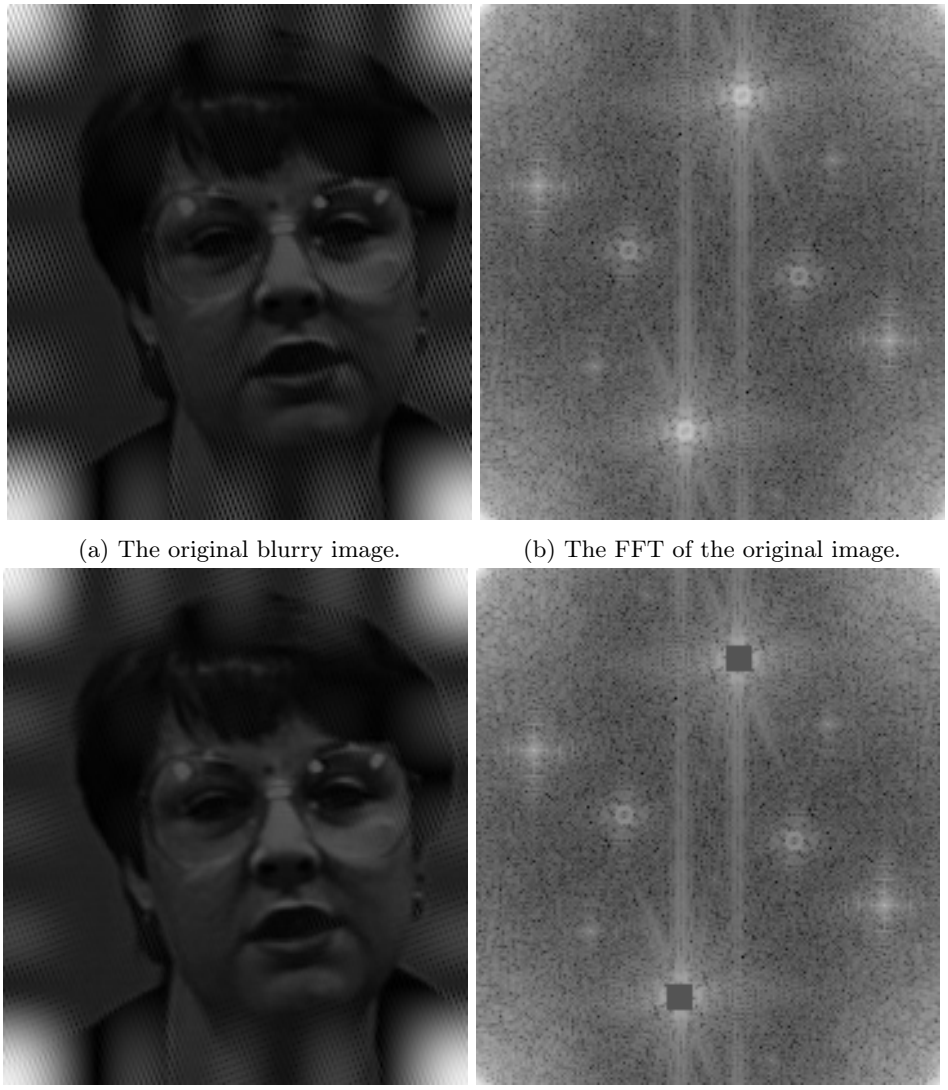
```
# Plot the blurry image (figure 1.3(a)).
>>> from scipy.misc import imread
>>> image = imread("face.png", True)
>>> plt.imshow(image, cmap='gray')
>>> plt.show()
```

Now plot the Fourier transform of this image in order to see where the spikes in frequency are. To effectively visualize this plot, plot the log of the absolute value of the result.

```
# Plot the Fourier transform of the blurry image (figure 1.3(b)).
>>> fft = np.fft.fft2(image)
>>> plt.imshow(np.log(np.abs(fft)), cmap='gray')
>>> plt.show()
```

Notice the spikes in the plot. In order to reduce the noise in the plot, replace these abnormally high values with values that are more similar to those around them. There are many ways to do this, but one possibility is to simply "patch" it by covering the area with a small matrix of values similar to other values in the plot.

```
# Cover the spikes in the Fourier transform (figure 1.3(d)).
>>> fft[30:40, 97:107] = np.ones((10,10)) * fft[33][50]
>>> fft[-39:-29, -106:-96] = np.ones((10,10)) * fft[33][50]
>>> plt.imshow(np.log(np.abs(fft)),cmap='gray')
>>> plt.show()
```



(a) The original blurry image.

(b) The FFT of the original image.

(c) The improved image.

(d) The FFT of the improved image.

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

Finally, take the inverse Fourier transform of this to get an image similar to the original, but with less noise. Notice that this new image still has noise present, but is less noisy than the original. Once again, use the absolute value of this result to plot it.

```
# Plot the new image (figure 1.3(c)).
>>> new_image = np.abs(np.fft.ifft2(fft))
>>> plt.imshow(new_image, cmap='gray')
>>> plt.show()
```

In practice, it often will suffice to remove some of the noise, even if it is not possible to remove all of it. For example, if the purpose of cleaning up an image was to retrieve some information from it that was not easily distinguishable before, then it would be sufficient to only remove enough noise to accurately extract that information. To further improve the image, a similar process to the one described above must be done to cover the remaining spikes.

Problem 6. One potential application of removing noise from an image is cleaning up an image of a license plate to retrieve its information. The file `license_plate.png` contains a blurred image of a license plate. In the bottom right corner of this image, there is a sticker that has information about the month and year that the license plate was renewed. However, in its current state the year is not clearly legible. Use the two-dimensional Fourier transform to clean up the image enough that the year in the bottom right corner is legible.

8 Introduction to Wavelets

Lab Objective: *Wavelets are useful in a variety of applications because of their ability to represent some types of information in a very sparse manner. We will explore both the one- and two-dimensional discrete wavelet transforms using various types of wavelets. We will then use a Python package called PyWavelets for further wavelet analysis including image cleaning and image compression.*

The Discrete Wavelet Transform

In wavelet analysis, information (such as a mathematical function or image) can be stored and analyzed by considering its *wavelet decomposition*. The wavelet decomposition is a way of expressing information as a linear combination of a particular set of wavelet functions. Once a wavelet is chosen, information can be represented by a sequence of coefficients (called *wavelet coefficients*) that define the linear combination. 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. A function called the *wavelet* or *mother wavelet*, usually denoted ψ, and another function called the *scaling* or *father scaling function*, typically represented by ϕ, are the basis of a wavelet family. A countably infinite set of wavelet functions (commonly known as *daughter wavelets*) can be generated using dilations and shifts of the first two functions:

$$\psi_{m,k}(x) = \psi(2^m x - k)$$
$$\phi_{m,k}(x) = \phi(2^m x - k),$$

where $m, k \in \mathbb{Z}$.

Given a wavelet family, a function f can be approximated as a combination of father and daughter wavelets as follows:

$$f(x) = \sum_{k=-\infty}^{\infty} a_k \phi_{m,k}(x) + \sum_{k=-\infty}^{\infty} b_{m,k}\psi_{m,k}(x) + \cdots + \sum_{k=-\infty}^{\infty} b_{n,k}\psi_{n,k}(x)$$

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. Depending on the properties of the wavelet and the function (or signal), f can be approximated to an arbitrary level of accuracy.

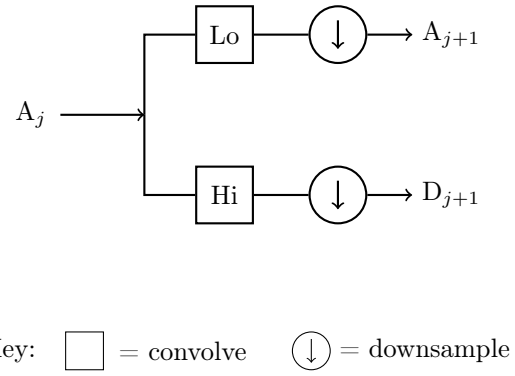


Figure 8.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

In the case of finitely-sampled signals and images, there exists an efficient algorithm for computing the wavelet coefficients. Most 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 approximation highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details.The high frequency details are extracted by the high-pass filter. These detail coefficients highlight the small changes found in the signal. 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. Then the resulting arrays are downsampled to remove redundant information. In the context of this lab, a *filter bank* is the combined process of convolving with a filter then downsampling. This process can be repeated on the new approximation to obtain another layer of approximation and detail coefficients.

See Algorithm 8.1 and Figure 8.1 for the specifications.

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 ← X                                  ▷ Initialization.
3:     for i = 0 ... n − 1 do
4:         D_{i+1} ← DS(A_i * H)                ▷ High-pass filter and downsample.
5:         A_{i+1} ← DS(A_i * L)                ▷ Low-pass filter and downsample.
6:     return A_n, D_n, D_{n−1}, ..., D_1.
```

The *Haar Wavelet* is one of the most widely used wavelets in wavelet analysis. Its wavelet function is defined

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

The associated scaling function is given by

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

For the Haar Wavelet, the low-pass and high-pass filters are given by

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$H = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}.$$

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. In the case of the Haar wavelet, the array will be downsampled by a factor of two, which means keeping only every other entry:

```
>>> # Downsample an array X.
>>> sampled = X[1::2]
```

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.

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: $[A_n, D_n, \ldots, D_1]$.

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.

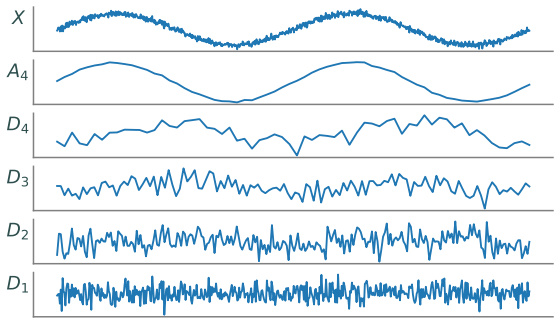


Figure 8.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

The process of the discrete wavelet transform is reversible. Using modified filters, a set of detail coefficients and a set of approximation coefficients can be manipulated and added together to recreate a signal. The Haar wavelet filters for the inverse transformation are

$$L = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

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 L and H, 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 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 the function written in 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 $D_n, D_{n-1}, \ldots, D_1$), 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()`.

Note

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 some 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.

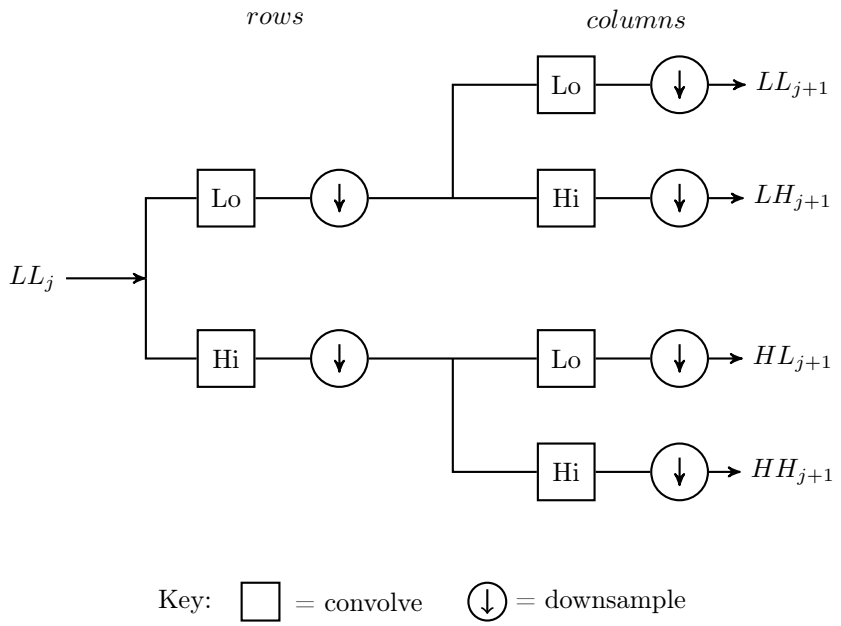


Figure 8.3: The two-dimensional discrete wavelet transform implemented as a filter bank.

The Two-dimensional Wavelet Transform

The generalization of the wavelet transform to two dimensions is fairly straightforward. Once again, the two primary operations 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. 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 for an illustration of this concept.

When implemented as an iterative filter bank, each pass through the filter bank yields an approximation plus three sets of detail coefficients rather than just one. 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 wavelet coefficients capturing 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 different 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 given in Figure 8.5.

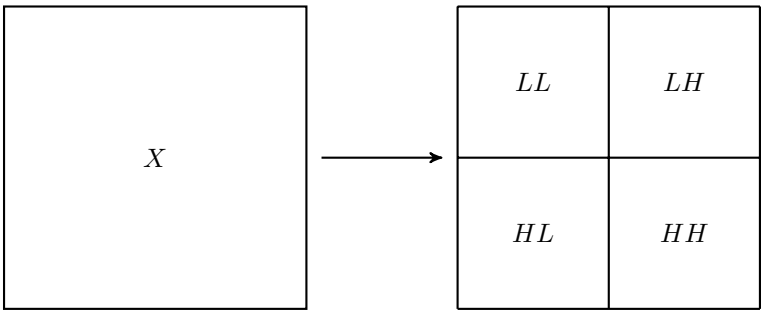


Figure 8.4: The subband pattern for one step in the 2-dimensional wavelet transform.

The wavelet coefficients that are obtained from a two-dimensional wavelet transform are very useful in a variety of image processing tasks. They allow images to be analyzed and manipulated in terms of both their frequency and spatial properties, and at differing levels of resolution. Furthermore, images are often represented in a very sparse manner by wavelets; that is, most of the image information is captured by a small subset of the wavelet coefficients. This is the 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 use in 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 and plot them in a manner similar to Figure 8.5.

```
>>> from scipy.misc 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')
```

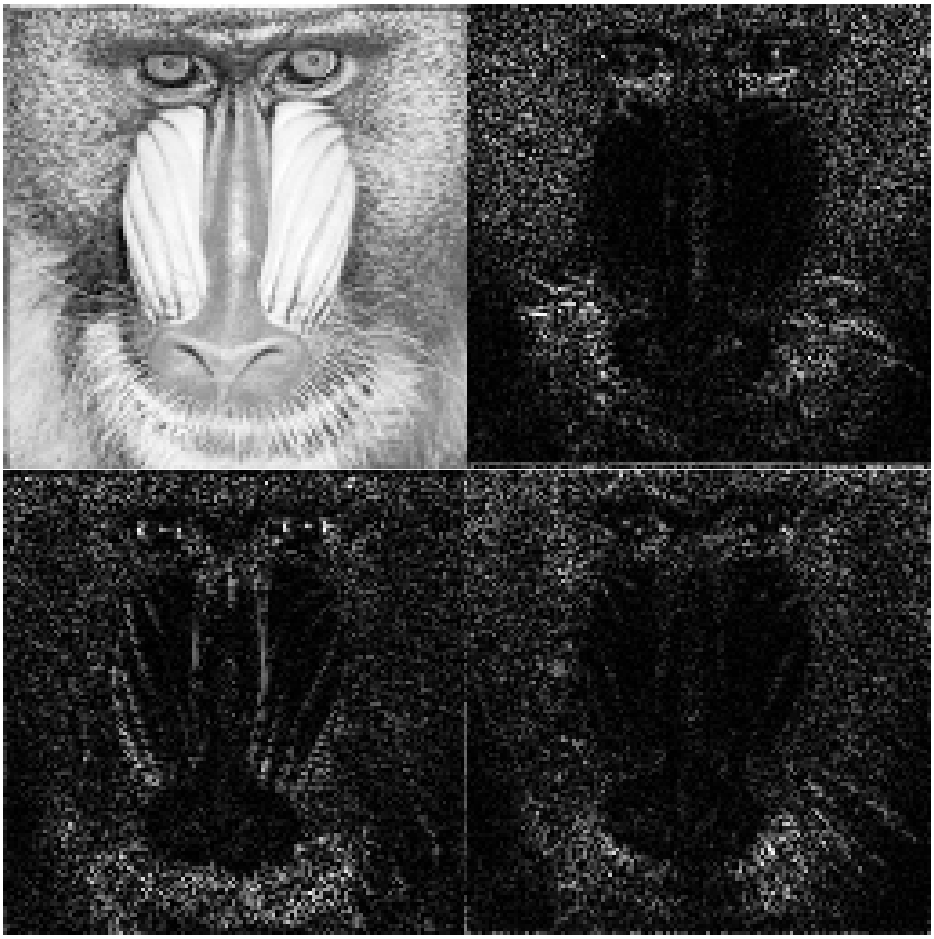


Figure 8.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband (LL) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.
Original image source: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

The function `pywt.dwt2()` calculates the subbands resulting from one pass through the filter bank. 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 second positional argument specifies the type of wavelet to be used in the transform. 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). These subbands can be plotted as follows:

```
>>> plt.subplot(221)
>>> plt.imshow(lw[0], cmap='gray')
>>> plt.axis('off')
>>> plt.subplot(222)
# The absolute value of the detail subbands is plotted to highlight contrast.
>>> plt.imshow(np.abs(lw[1][0]), cmap='gray')
>>> plt.axis('off')
```

```
>>> plt.subplot(223)
>>> plt.imshow(np.abs(lw[1][1]), cmap='gray')
>>> plt.axis('off')
>>> plt.subplot(224)
>>> plt.imshow(np.abs(lw[1][2]), cmap='gray')
>>> plt.axis('off')
>>> plt.subplots_adjust(wspace=0, hspace=0)      # Remove space between plots.
```

As noted, the second positional argument is a string that gives the name of the wavelet to be used. 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. The best wavelet to use in a particular application is rarely known beforehand. A choice about which wavelet to use can be 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 until some type of testing is done.

Problem 3. Explore the two-dimensional wavelet transform by completing the following:

1. Plot the subbands of the file `lenna.jpg` 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.

2. Compare the subband patterns of different wavelets by plotting the LH subband pattern for the Haar wavelet and two other wavelets of your choice using `lenna.jpg`. Note that not all wavelets included in PyWavelets are compatible with every function. For example, only the first seven families listed by `pywt.families()` are compatible with `dwt2()`.

The function `pywt.wavedec2()` is similar to `pywt.dwt2()`; however 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 determined at runtime. 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 `http://pywavelets.readthedocs.io/en/latest/contents.html`.

Applications

Noise Reduction

Noise in an image can be defined as unwanted visual artifacts that obscure the true image. Images can acquire noise from a variety of sources, including the camera, data transfer, and image processing algorithms. This section will focus on reducing a particular type of random noise in images called *Gaussian white noise*.

An image that is distorted by Gaussian white noise is one in which every pixel has been perturbed by a small amount. Many types of noise, including Gaussian white noise, are very high-frequency. Since many images are relatively sparse in the high-frequency domains, noise in an image can be safely removed from the high frequency subbands without distorting the true image very much. A basic but effective approach to reducing Gaussian white noise in an image is thresholding.

Given a positive threshold value τ, hard thresholding sets every wavelet coefficient whose magnitude is less than τ to zero, while leaving the remaining coefficients untouched. Soft thresholding also zeros out all coefficients of magnitude less than τ, but in addition maps the remaining positive coefficients β to $\beta - \tau$ and the remaining negative coefficients α 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, one of which implements the hard thresholding technique and one of which implements the soft. While writing these two functions, remember the following:

- The functions should accept a list of wavelet coefficients in the usual form, as well as a threshold value.
- The functions should return the thresholded wavelet coefficients (also in the usual form).
- Since only the detail coefficients are thresholded, the first entry of the input coefficient list should remain unchanged.

To test your functions, perform hard and soft thresholding on `noisy_lenna.jpg` 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 $\tau = 30$, and for hard thresholding use $\tau = 60$.

Image Compression

Numerous image compression techniques have been developed over the years to reduce the cost of storing large quantities of images. Transform methods based on Fourier and wavelet analysis have long played an important role in these techniques; 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 fairly straightforward. 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 then 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.

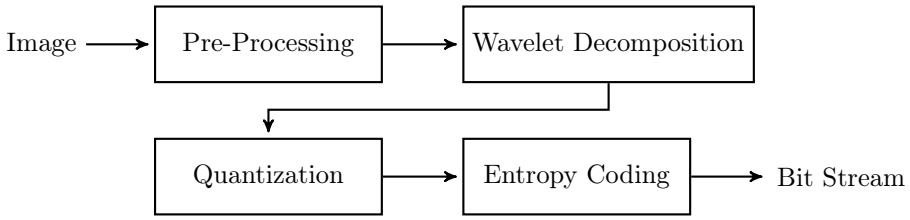
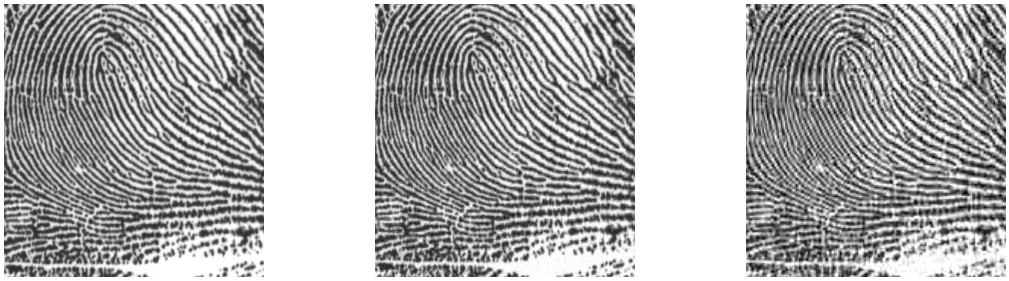


Figure 8.6: Wavelet Image Compression Schematic

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. Some of the difference between this simplified algorithm and the complete algorithm are found in the Additional Material section at the end of this lab. Also included in Additional Materials are the methods of the WSQ class that are needed to complete the algorithm.



(a) Uncompressed (b) 12:1 compressed (c) 26:1 compressed

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

WSQ: Preprocessing

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). This preprocessing ensures that roughly half of the new pixel values are negative, while the other half are positive, and all fall in the range $[-128, 128]$.

Problem 5. Implement the preprocessing step as well as its inverse by implementing the class methods `pre_process()` and `post_process()`. These methods should accept a NumPy array (the image) and return the processed image as a NumPy array. In the `pre_process()` method, calculate the values of m and s given above. These values are needed later on for decompression, so 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 $[LL_5, LH_5, HL_5, HH_5, LH_4, HL_4, \ldots, HH_1]$).

Problem 6. Implement the subband decomposition as described above by implementing 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.

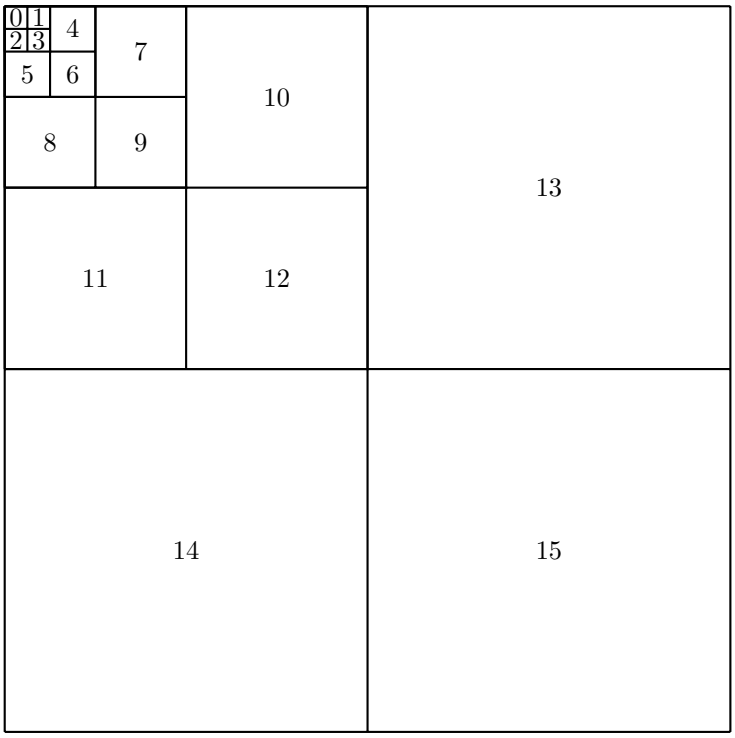


Figure 8.8: Subband Pattern for simplified WSQ algorithm.

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()`.

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

$$p = \begin{cases} \left\lfloor \frac{a - Z_k/2}{Q_k} \right\rfloor + 1, & a > Z_k/2 \\ 0, & -Z_k/2 \leq a \leq Z_k/2 \\ \left\lceil \frac{a + Z_k/2}{Q_k} \right\rceil - 1, & a < -Z_k/2. \end{cases}$$

The values Z_k and Q_k are dependent on the subband, and 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. Therefore, 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 the following formula. This process is called *dequantization*.

$$\hat{a}_k = \begin{cases} (p - C)Q_k + Z_k/2, & p > 0 \\ 0, & p = 0 \\ (p + C)Q_k - Z_k/2, & p < 0 \end{cases}$$

Note the inclusion of a new dequantization parameter C. 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.

Masking and array slicing will help keep your code short and fast when implementing both of these methods. Remember the case for $Q_k = 0$. You can check that your functions are working correctly 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 image is just eight times the number of pixels (recall that 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:

```
# 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

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 performace.

<table>
<tr><td>0
2</td><td>1
3</td><td>4</td><td>7</td><td>8</td><td>19</td><td>20</td><td>23</td><td>24</td><td colspan="2" rowspan="4">52</td><td colspan="2" rowspan="4">53</td></tr>
<tr><td>5</td><td>6</td><td>9</td><td>10</td><td>21</td><td>22</td><td>25</td><td>26</td></tr>
<tr><td>11</td><td>12</td><td>15</td><td>16</td><td>27</td><td>28</td><td>31</td><td>32</td></tr>
<tr><td>13</td><td>14</td><td>17</td><td>18</td><td>29</td><td>30</td><td>33</td><td>34</td></tr>
<tr><td>35</td><td>36</td><td>39</td><td>40</td><td colspan="4" rowspan="4">51</td><td colspan="2" rowspan="4">54</td><td colspan="2" rowspan="4">55</td></tr>
<tr><td>37</td><td>38</td><td>41</td><td>42</td></tr>
<tr><td>43</td><td>44</td><td>47</td><td>48</td></tr>
<tr><td>45</td><td>46</td><td>49</td><td>50</td></tr>
<tr><td colspan="4" rowspan="2">56</td><td colspan="4" rowspan="2">57</td><td colspan="2" rowspan="2">60</td><td colspan="2" rowspan="2">61</td></tr>
<tr></tr>
<tr><td colspan="4" rowspan="2">58</td><td colspan="4" rowspan="2">59</td><td colspan="2" rowspan="2">62</td><td colspan="2" rowspan="2">63</td></tr>
<tr></tr>
</table>

Figure 8.9: True subband pattern for WSQ algorithm.

The following code box contains the partially implemented WSQ class that is needed to complete the problems in this lab.

```
class WSQ:
    """Perform image compression using the Wavelet Scalar Quantization
    algorithm. This class is a structure for performing the algorithm, to
```

```
    actually perform the compression and decompression, use the _compress
    and _decompress methods respectively. Note that all class attributes
    are set to None in __init__, but their values are initialized in the
    compress method.

    Attributes:
        _pixels (int): Number of pixels in source image.
        _s (float): Scale parameter for image preprocessing.
        _m (float): Shift parameter for image preprocessing.
        _Q ((16, ), ndarray): Quantization parameters q for each subband.
        _Z ((16, ), ndarray): Quantization parameters z for each subband.
        _bitstrings (list): List of 3 BitArrays, giving bit encodings for
            each group.
        _tvals (tuple): Tuple of 3 lists of bools, indicating which
            subbands in each groups were encoded.
        _shapes (tuple): Tuple of 3 lists of tuples, giving shapes of each
            subband in each group.
        _huff_maps (list): List of 3 dictionaries, mapping huffman index to
            bit pattern.
    """

    def __init__(self):
        self._pixels = None
        self._s = None
        self._m = None
        self._Q = None
        self._Z = None
        self._bitstrings = None
        self._tvals = None
        self._shapes= None
        self._huff_maps = None
        self._infoloss = None

    def compress(self, img, r, gamma=2.5):
        """The main compression routine. It computes and stores a bitstring
        representation of a compressed image, along with other values
        needed for decompression.

        Parameters:
            img ((m,n), ndarray): Numpy array containing 8-bit integer
                pixel values.
            r (float): Defines compression ratio. Between 0 and 1, smaller
                numbers mean greater levels of compression.
            gamma (float): A parameter used in quantization.
        """
        self._pixels = img.size   # Store image size.
        # Process then decompose image into subbands.
        mprime = self.pre_process(img)
        subbands = self.decompose(img)
```

```
        # Calculate quantization parameters, quantize the image then group.
        self._Q, self._Z = self.get_bins(subbands, r, gamma)
        q_subbands = [self.quantize(subbands[i],self._Q[i],self._Z[i])
                      for i in range(16)]
        groups, self._shapes, self._tvals = self.group(q_subbands)

        # Complete the Huffman encoding and transfer to bitstring.
        huff_maps = []
        bitstrings = []
        for i in range(3):
            inds, freqs, extra = self.huffman_indices(groups[i])
            huff_map = huffman(freqs)
            huff_maps.append(huff_map)
            bitstrings.append(self.encode(inds, extra, huff_map))

        # Store the bitstrings and the huffman maps.
        self._bitstrings = bitstrings
        self._huff_maps = huff_maps

    def pre_process(self, img):
        """Preprocessing routine that takes an image and shifts it so that
        roughly half of the values are on either side of zero and fall
        between -128 and 128.

        Parameters:
            img ((m,n), ndarray): Numpy array containing 8-bit integer
                pixel values.

        Returns:
            ((m,n), ndarray): Processed numpy array containing 8-bit
                integer pixel values.
        """
        pass

    def post_process(self, img):
        """Postprocess routine that reverses pre_process().

        Parameters:
            img ((m,n), ndarray): Numpy array containing 8-bit integer
                pixel values.

        Returns:
            ((m,n), ndarray): Unprocessed numpy array containing 8-bit
                integer pixel values.
        """
        pass

    def decompose(self, img):
        """Decompose an image into the WSQ subband pattern using the
```

```
        Coiflet1 wavelet.

        Parameters:
            img ((m,n) ndarray): Numpy array holding the image to be
                decomposed.

        Returns:
            subbands (list): List of 16 numpy arrays containing the WSQ
                subbands in order.
        """
        pass

    def recreate(self, subbands):
        """Recreate an image from the 16 WSQ subbands.

        Parameters:
            subbands (list): List of 16 numpy arrays containing the WSQ
                subbands in order.

        Returns:
            img ((m,n) ndarray): Numpy array, the image recreated from the
                WSQ subbands.
        """
        pass

    def get_bins(self, subbands, r, gamma):
        """Calculate quantization bin widths for each subband. These will
        be used to quantize the wavelet coefficients.

        Parameters:
            subbands (list): List of 16 WSQ subbands.
            r (float): Compression parameter, determines the degree of
                compression.
            gamma(float): Parameter used in compression algorithm.

        Returns:
            Q ((16, ) ndarray): Array of quantization step sizes.
            Z ((16, ) ndarray): Array of quantization coefficients.
        """
        subband_vars = np.zeros(16)
        fracs = np.zeros(16)

        for i in range(len(subbands)): # Compute subband variances.
            X,Y = subbands[i].shape
            fracs[i]=(X*Y)/(np.float(finger.shape[0]*finger.shape[1]))
            x = np.floor(X/8.).astype(int)
            y = np.floor(9*Y/32.).astype(int)
            Xp = np.floor(3*X/4.).astype(int)
            Yp = np.floor(7*Y/16.).astype(int)
```

```
            mu = subbands[i].mean()
            sigsq = (Xp*Yp-1.)**(-1)*((subbands[i][x:x+Xp, y:y+Yp]-mu)**2).sum↩
                ()
            subband_vars[i] = sigsq

        A = np.ones(16)
        A[13], A[14] = [1.32]*2

        Qprime = np.zeros(16)
        mask = subband_vars >= 1.01
        Qprime[mask] = 10./(A[mask]*np.log(subband_vars[mask]))
        Qprime[:4] = 1
        Qprime[15] = 0

        K = []
        for i in range(15):
            if subband_vars[i] >= 1.01:
                K.append(i)

        while True:
            S = fracs[K].sum()
            P = ((np.sqrt(subband_vars[K])/Qprime[K])**fracs[K]).prod()
            q = (gamma**(-1))*(2**(r/S-1))*(P**(-1./S))
            E = []
            for i in K:
                if Qprime[i]/q >= 2*gamma*np.sqrt(subband_vars[i]):
                    E.append(i)
            if len(E) > 0:
                for i in E:
                    K.remove(i)
                continue
            break

        Q = np.zeros(16) # Final bin widths.
        for i in K:
            Q[i] = Qprime[i]/q
        Z = 1.2*Q

        return Q, Z

    def quantize(self, coeffs, Q, Z):
        """Implementation of a uniform quantizer which maps wavelet
        coefficients to integer values using the quantization parameters
        Q and Z.

        Parameters:
            coeffs ((m,n) ndarray): Contains the floating-point values to
                be quantized.
            Q (float): The step size of the quantization.
```

```
            Z (float): The null-zone width (of the center quantization bin).

        Returns
            out ((m,n) ndarray): Numpy array of the quantized values.
        """
        pass

    def dequantize(self, coeffs, Q, Z, C=0.44):
        """Given quantization parameters, approximately reverses the
        quantization effect carried out in quantize().

        Parameters:
            coeffs ((m,n) ndarray): Array of quantized coefficients.
            Q (float): The step size of the quantization.
            Z (float): The null-zone width (of the center quantization bin).
            C (float): Centering parameter, defaults to .44.

        Returns:
            out ((m,n) ndarray): Array of dequantized coefficients.
        """
        pass

    def group(self, subbands):
        """Split the quantized subbands into 3 groups.

        Parameters:
            subbands (list): Contains 16 numpy arrays which hold the
                quantized coefficients.

        Returns:
            gs (tuple): (g1,g2,g3) Each gi is a list of quantized coeffs
                for group i.
            ss (tuple): (s1,s2,s3) Each si is a list of tuples which
                contain the shapes for group i.
            ts (tuple): (s1,s2,s3) Each ti is a list of bools indicating
                which subbands were included.
        """
        g1 = [] # This will hold the group 1 coefficients.
        s1 = [] # Keep track of the subband dimensions in group 1.
        t1 = [] # Keep track of which subbands were included.
        for i in range(10):
            s1.append(subbands[i].shape)
            if subbands[i].any(): # True if there is any nonzero entry.
                g1.extend(subbands[i].ravel())
                t1.append(True)
            else: # The subband was not transmitted.
                t1.append(False)

        g2 = [] # This will hold the group 2 coefficients.
```

```
        s2 = [] # Keep track of the subband dimensions in group 2.
        t2 = [] # Keep track of which subbands were included.
        for i in range(10, 13):
            s2.append(subbands[i].shape)
            if subbands[i].any(): # True if there is any nonzero entry.
                g2.extend(subbands[i].ravel())
                t2.append(True)
            else: # The subband was not transmitted.
                t2.append(False)

        g3 = [] # This will hold the group 3 coefficients.
        s3 = [] # Keep track of the subband dimensions in group 3.
        t3 = [] # Keep track of which subbands were included.
        for i in range(13,16):
            s3.append(subbands[i].shape)
            if subbands[i].any(): # True if there is any nonzero entry.
                g3.extend(subbands[i].ravel())
                t3.append(True)
            else: # The subband was not transmitted.
                t3.append(False)

        return (g1,g2,g3), (s1,s2,s3), (t1,t2,t3)

    def ungroup(self, gs, ss, ts):
        """Re-create the subband list structure from the information stored
        in gs, ss and ts.

        Parameters:
            gs (tuple): (g1,g2,g3) Each gi is a list of quantized coeffs
                for group i.
            ss (tuple): (s1,s2,s3) Each si is a list of tuples which
                contain the shapes for group i.
            ts (tuple): (s1,s2,s3) Each ti is a list of bools indicating
                which subbands were included.

        Returns:
            subbands (list): Contains 16 numpy arrays holding quantized
                coefficients.
        """
        subbands1 = [] # The reconstructed subbands in group 1.
        i = 0
        for j, shape in enumerate(ss[0]):
            if ts[0][j]: # True if the j-th subband was included.
                l = shape[0]*shape[1] # Number of entries in the subband.
                subbands1.append(np.array(gs[0][i:i+l]).reshape(shape))
                i += l
            else: # The j-th subband wasn't included, so all zeros.
                subbands1.append(np.zeros(shape))
```

```
        subbands2 = [] # The reconstructed subbands in group 2.
        i = 0
        for j, shape in enumerate(ss[1]):
            if ts[1][j]: # True if the j-th subband was included.
                l = shape[0]*shape[1] # Number of entries in the subband.
                subbands2.append(np.array(gs[1][i:i+l]).reshape(shape))
                i += l
            else: # The j-th subband wasn't included, so all zeros.
                subbands2.append(np.zeros(shape))

        subbands3 = [] # the reconstructed subbands in group 3
        i = 0
        for j, shape in enumerate(ss[2]):
            if ts[2][j]: # True if the j-th subband was included.
                l = shape[0]*shape[1] # Number of entries in the subband.
                subbands3.append(np.array(gs[2][i:i+l]).reshape(shape))
                i += l
            else: # The j-th subband wasn't included, so all zeros.
                subbands3.append(np.zeros(shape))

        subbands1.extend(subbands2)
        subbands1.extend(subbands3)
        return subbands1

    def huffman_indices(self, coeffs):
        """Calculate the Huffman indices from the quantized coefficients.

        Parameters:
            coeffs (list): Integer values that represent quantized
                coefficients.

        Returns:
            inds (list): The Huffman indices.
            freqs (ndarray): Array whose i-th entry gives the frequency of
                index i.
            extra (list): Contains zero run lengths and coefficient
                magnitudes for exceptional cases.
        """
        N = len(coeffs)
        i = 0
        inds = []
        extra = []
        freqs = np.zeros(254)

        # Sweep through the quantized coefficients.
        while i < N:

            # First handle zero runs.
            zero_count = 0
```

```
            while coeffs[i] == 0:
                zero_count += 1
                i += 1
                if i >= N:
                    break

            if zero_count > 0 and zero_count < 101:
                inds.append(zero_count - 1)
                freqs[zero_count - 1] += 1
            elif zero_count >= 101 and zero_count < 256: # 8 bit zero run.
                inds.append(104)
                freqs[104] += 1
                extra.append(zero_count)
            elif zero_count >= 256: # 16 bit zero run.
                inds.append(105)
                freqs[105] += 1
                extra.append(zero_count)
            if i >= N:
                break

            # now handle nonzero coefficients
            if coeffs[i] > 74 and coeffs[i] < 256: # 8 bit pos coeff.
                inds.append(100)
                freqs[100] += 1
                extra.append(coeffs[i])
            elif coeffs[i] >= 256: # 16 bit pos coeff.
                inds.append(102)
                freqs[102] += 1
                extra.append(coeffs[i])
            elif coeffs[i] < -73 and coeffs[i] > -256: # 8 bit neg coeff.
                inds.append(101)
                freqs[101] += 1
                extra.append(abs(coeffs[i]))
            elif coeffs[i] <= -256: # 16 bit neg coeff.
                inds.append(103)
                freqs[103] += 1
                extra.append(abs(coeffs[i]))
            else: # Current value is a nonzero coefficient in the range [-73, ↩
                74].
                inds.append(179 + coeffs[i])
                freqs[179 + coeffs[i].astype(int)] += 1
            i += 1

        return list(map(int,inds)), list(map(int,freqs)), list(map(int,extra))

    def indices_to_coeffs(self, indices, extra):
        """Calculate the coefficients from the Huffman indices plus extra
        values.
```

```
        Parameters:
            indices (list): List of Huffman indices.
            extra (list): Indices corresponding to exceptional values.

        Returns:
            coeffs (list): Quantized coefficients recovered from the indices.
        """
        coeffs = []
        j = 0 # Index for extra array.

        for s in indices:
            if s < 100: # Zero count of 100 or less.
                coeffs.extend(np.zeros(s+1))
            elif s == 104 or s == 105: # Zero count of 8 or 16 bits.
                coeffs.extend(np.zeros(extra[j]))
                j += 1
            elif s in [100, 102]: # 8 or 16 bit pos coefficient.
                coeffs.append(extra[j]) # Get the coefficient from the extra ↩
                    list.
                j += 1
            elif s in [101, 103]: # 8 or 16 bit neg coefficient.
                coeffs.append(-extra[j]) # Get the coefficient from the extra ↩
                    list.
                j += 1
            else: # Coefficient from -73 to +74.
                coeffs.append(s-179)
        return coeffs

    def encode(self, indices, extra, huff_map):
        """Encodes the indices using the Huffman map, then returns
        the resulting bitstring.

        Parameters:
            indices (list): Huffman Indices.
            extra (list): Indices corresponding to exceptional values.
            huff_map (dict): Dictionary that maps Huffman index to bit
                pattern.

        Returns:
            bits (BitArray object): Contains bit representation of the
                Huffman indices.
        """
        bits = bs.BitArray()
        j = 0 # Index for extra array.
        for s in indices: # Encode each huffman index.
            bits.append('0b' + huff_map[s])

            # Encode extra values for exceptional cases.
            if s in [104, 100, 101]: # Encode as 8-bit ints.
```

```
                bits.append('uint:8={}'.format(int(extra[j])))
                j += 1
            elif s in [102, 103, 105]: # Encode as 16-bit ints.
                bits.append('uint:16={}'.format(int(extra[j])))
                j += 1
        return bits

    def decode(self, bits, huff_map):
        """Decodes the bits using the given huffman map, and returns
        the resulting indices.

        Parameters:
            bits (BitArray object): Contains bit-encoded Huffman indices.
            huff_map (dict): Maps huffman indices to bit pattern.

        Returns:
            indices (list): Decoded huffman indices.
            extra (list): Decoded values corresponding to exceptional indices.
        """
        indices = []
        extra = []

        # Reverse the huffman map to get the decoding map.
        dec_map = {v:k for k, v in huff_map.items()}

        # Wrap the bits in an object better suited to reading.
        bits = bs.ConstBitStream(bits)

        # Read each bit at a time, decoding as we go.
        i = 0 # The index of the current bit.
        pattern = '' # The current bit pattern.
        while i < bits.length:
            pattern += bits.read('bin:1') # Read in another bit.
            i += 1

            # Check if current pattern is in the decoding map.
            if pattern in dec_map:
                indices.append(dec_map[pattern]) # Insert huffman index.

                # If an exceptional index, read next bits for extra value.
                if dec_map[pattern] in (100, 101, 104): # 8-bit int or 8-bit ↩
                    zero run length.
                    extra.append(bits.read('uint:8'))
                    i += 8
                elif dec_map[pattern] in (102, 103, 105): # 16-bit int or 16-↩
                    bit zero run length.
                    extra.append(bits.read('uint:16'))
                    i += 16
                pattern = '' # Reset the bit pattern.
```

```
        return indices, extra

    def decompress(self):
        """Return the uncompressed image recovered from the compressed
            bistring representation.

        Returns:
            img ((m,n) ndaray): The recovered, uncompressed image.
        """
        # For each group, decode the bits, map from indices to coefficients.
        groups = []
        for i in range(3):
            indices, extras = self.decode(self._bitstrings[i],
                                          self._huff_maps[i])
            groups.append(self.indices_to_coeffs(indices, extras))

        # Recover the subbands from the groups of coefficients.
        q_subbands = self.ungroup(groups, self._shapes, self._tvals)

        # Dequantize the subbands.
        subbands = [self.dequantize(q_subbands[i], self._Q[i], self._Z[i])
                    for i in range(16)]

        # Recreate the image.
        img = self.recreate(subbands)

        # Post-process, return the image.
        return self.post_process(img)

    def get_ratio(self):
        """Calculate the compression ratio achieved.

        Returns:
            ratio (float): Ratio of number of bytes in the original image
                to the number of bytes contained in the bitstrings.
        """
        pass
```

The following code includes the methodes used in the WSQ class to perform the Huffman encoding.

```
# Helper functions and classes for the Huffman encoding portions of WSQ ↩
    algorithm.

import queue
class huffmanLeaf():
    """Leaf node for Huffman tree."""
    def __init__(self, symbol):
        self.symbol = symbol
```

```
    def makeMap(self, huff_map, path):
        huff_map[self.symbol] = path

    def __str__(self):
        return str(self.symbol)

    def __lt__(self,other):
        return False

class huffmanNode():
    """Internal node for Huffman tree."""
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def makeMap(self, huff_map, path):
        """Traverse the huffman tree to build the encoding map."""
        self.left.makeMap(huff_map, path + '0')
        self.right.makeMap(huff_map, path + '1')

    def __lt__(self,other):
        return False

def huffman(freqs):
    """
    Generate the huffman tree for the given symbol frequencies.
    Return the map from symbol to bit pattern.
    """
    q = queue.PriorityQueue()
    for i in range(len(freqs)):
        leaf = huffmanLeaf(i)
        q.put((freqs[i], leaf))
    while q.qsize() > 1:
        l1 = q.get()
        l2 = q.get()
        weight = l1[0] + l2[0]
        node = huffmanNode(l1[1], l2[1])
        q.put((weight,node))
    root = q.get()[1]
    huff_map = dict()
    root.makeMap(huff_map, '')
    return huff_map
```

9 Gaussian Quadrature

Lab Objective: *Numerical quadrature is an important numerical integration technique. The popular Newton-Cotes quadrature uses uniformly spaced points to approximate the integral, but Runge's phenomenon prevents Newton-Cotes from being effective for many functions. The Gaussian Quadrature method, on the other hand, uses carefully chosen points and weights to mitigate this problem.*

Shifting the Interval of Integration

As with all quadrature methods, we begin by choosing a set of points x_i and weights w_i to approximate an integral.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i).$$

When we do Gaussian quadrature, we are required to choose a weight function $W(x)$. This function determines both the x_i's and the w_i's. Theoretically, the weight function determines a set of orthogonal polynomials to approximate the function f.

The weight function also determines the interval over which the integration will occur. For example, we may choose the weight function as $W(x) = 1$ over $[-1, 1]$ to integrate functions on $[-1, 1]$. To calculate the definite integrate over a the arbitrary interval $[a, b]$, we perform a u-substitution to shift the interval of integration from $[a, b]$ to $[-1, 1]$. Let

$$u = \frac{2x - b - a}{b - a}$$

so that $u = -1$ when $x = a$ and $u = 1$ when $x = b$. Then we have

$$x = \frac{b-a}{2}u + \frac{a+b}{2} \qquad \text{and} \qquad dx = \frac{b-a}{2}du,$$

so the transformed integral is given by

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}u + \frac{a+b}{2}\right)du. \tag{9.1}$$

The general quadrature formula is then given by the following equation.

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_i w_i f\left(\frac{(b-a)}{2}x_i + \frac{(a+b)}{2}\right)$$

Problem 1. Write a function that accepts a function f, interval endpoints a and b, and a keyword argument `plot` that defaults to `False`. Use (9.1) to construct a function g such that

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} g(u)du.$$

(Hint: this can be done in a single line using the `lambda` keyword.)

If `plot` is `True`, plot f over $[a, b]$ and g over $[-1, 1]$ in separate subplots. The functions probably have similar shapes, but note the difference in the scaling of the y-axis. Try $f(x) = x^2$ and $f(x) = (x-3)^3$ over $1, 4]$ as examples.

Finally, return the new function g.

Integrating with Given Weights and Points

With the new shifted function g from Problem 1, we can write the quadrature formula as follows.

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_i w_i g(x_i) \tag{9.2}$$

Suppose for a given weight function $W(x)$ and a given number of points n, we have the sample points $\mathbf{x} = [x_1, \dots, x_n]^\mathsf{T}$ and weights $\mathbf{w} = [w_1, \dots, w_n]^\mathsf{T}$ stored as 1-d NumPy arrays. The summation in (9.2) can then be calculated with the vector multiplication $\mathbf{w}^\mathsf{T} g(\mathbf{x})$, where $g(\mathbf{x}) = [g(x_1), \dots, g(x_n)]^\mathsf{T}$.

Problem 2. Write a function that accepts a function f, interval endpoints a and b, an array of points $\mathbf{x}$, and an array of weights $\mathbf{w}$. Use (9.2) and your function from Problem 1 to calculate the integral of f over $[a, b]$. Return the value of the integral.

To test this function, use the following 5 points and weights that accompany the constant weight function $W(x) = 1$ (this weight function corresponds to the *Legendre polynomials*). See the next page for their definitions using NumPy.

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	$\frac{322-13\sqrt{70}}{900}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{128}{225}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{322-13\sqrt{70}}{900}$

```
import numpy as np
from math import sqrt

s1 = 2 * sqrt(10. / 7.)
points = np.array([-sqrt(5 + s1) / 3.,
                   -sqrt(5 - s1) / 3.,
                                  0.,
                    sqrt(5 - s1) / 3.,
                    sqrt(5 + s1) / 3.])
```

```
s2 = 13 * sqrt(70)
weights = np.array([(322 - s2) / 900.,
                    (322 + s2) / 900.,
                           128 / 225.,
                    (322 + s2) / 900.,
                    (322 - s2) / 900.])
```

Using these points and weights should yield the approximations

$$\int_{-\pi}^{\pi} \sin(x)dx \approx 0 \quad \text{and} \quad \int_{-\pi}^{\pi} \cos(x)dx \approx 0.000196.$$

Calculating Weights and Points

Calculating an integral when the weights and points are given is straightforward. But how are these weights and points found? There are many publications that will give tables of points for various weight functions. We will demonstrate how to find such a list using the Golub-Welsch algorithm.

The Golub-Welsch Algorithm

The Golub-Welsch algorithm builds a tri-diagonal matrix and finds its eigenvalues. These eigenvalues are the points at which a function is evaluated for Guassian quadrature. The weights are the length of the shifted interval of integration times the first coordinate of each eigenvector squared. We note that finding eigenvalues for a tridiagonal matrix is a well conditioned, relatively painless problem. Using a good eigenvalue solver gives the Golub-Welsch algorithm a complexity of $O(n^2)$. A full treatment of the Golub-Welsch algorithm may be found at `http://gubner.ece.wisc.edu/gaussquad.pdf`.

We mentioned that the choice of weight function corresponds to a class of orthogonal polynomials. An important fact about orthogonal polynomials is that any set of orthogonal polynomials $\{u_i\}_{i=1}^N$ satisfies a three term recurrence relation

$$u_i(x) = (\gamma_{i-1}x - \alpha_i)u_{i-1}(x) - \beta_i u_{i-2}(x)$$

where $u_{-1}(x) = 0$ and $u_0(x) = 1$. The coefficients $\{\gamma_i, \alpha_i, \beta_i\}$ have been calculated for several classes of orthogonal polynomials, and may be determined for an arbitrary class using the procedure found in "Calculation of Gauss Quadrature Rules" by Golub and Welsch. Using these coefficients we may create a tri-diagonal matrix

$$J = \begin{bmatrix} a_1 & b_1 & 0 & 0 & \dots & 0 \\ b_1 & a_2 & b_2 & 0 & \dots & 0 \\ 0 & b_2 & a_3 & b_3 & \dots & 0 \\ \vdots & & & & & \vdots \\ \vdots & & & & & \vdots \\ 0 & \dots & & & & b_{N-1} \\ 0 & \dots & & & b_{N-1} & a_N \end{bmatrix}$$

Where $a_i = \frac{-\beta_i}{\alpha_i}$ and $b_i = (\frac{\gamma_{i+1}}{\alpha_i \alpha_{i+1}})^{\frac{1}{2}}$. This matrix is called the Jacobi matrix.

Problem 3. Write a function that will accept three arrays representing the coefficients $\{\gamma_i, \alpha_i, \beta_i\}$ from the recurrence relation above and return the Jacobi matrix.

The eigenvalues of the Jacobi matrix are the sample points x_i. The length of the shifted interval of integration (in this case 2) times the squares of the first entries of the corresponding eigenvectors give the weights.

Problem 4. The coefficients of the Legendre polynomials (which correspond to the weight function $W(x) = 1$ on $[-1, 1]$) are given by

$$\gamma_k = \frac{k-1}{k} \qquad \alpha_k = \frac{2k-1}{k} \qquad \beta_k = 0$$

Write a function that accepts an integer n representing the number of points to use in the quadrature. Calculate α, β, and γ as above, form the Jacobi matrix, then use it to find the points x_i and weights w_i that correspond to this weight function. Verify that when $n = 5$ the points and weights match the ones given in the first part of this lab.

Problem 5. Write a new function that accepts a function f, bounds a and b, and n for the number of points to use. Use the previously defined functions to estimate $\int_a^b f(x)dx$ using the coefficients of the Legendre polynomials.

This completes our implementation of the Gaussian Quadrature for a particular set orthogonal polynomials.

Numerical Integration with SciPy

There are many other techniques for finding the weights and points for a given weighting function. SciPy's `integrate` module provides general-purpose integration tools. For example, `scipy.integrate.quadrature()` offers a reasonably fast Gaussian quadrature implementation.

Problem 6. The standard normal distribution is an important object of study in probability and statistic. It is defined by the probability density function $p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$ (here we are assuming a mean of 0 and a variance of 1). This is a function that cannot be integrated symbolically.

The probability that a normally distributed random variable X will take on a value less than (or equal to) a given value x is

$$P(X \leq x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

This function is essentially zero for values of x that lie reasonably far from the mean, so we can estimate this probability by integrating from -5 to x instead of from $-\infty$ to x.

Write a function that uses `scipy.integrate.quad()` to estimate the probability that this normally distributed random variable will take a value less than a given number x that lies relatively close to the mean. You can test your result at $x = 1$ by comparing it with the following code:

```
from scipy.stats import norm
N = norm()                          # Make a standard normal random variable.
N.cdf(1)                            # Integrate the pdf from -infinity to 1.
```

10 CVXOPT

Lab Objective: *Introduce some of the basic optimization functions available in the CVXOPT package*

Notebox: CVXOPT is not part of the standard library, nor is it included in the Anaconda distribution. To install CVXOPT, use the following commands:

for Windows: `conda install -c http://conda.anaconda.org/omnia cvxopt`

for Unix: `pip install cvxopt` End notebox.

Linear Programs

CVXOPT is a package of Python functions and classes designed for the purpose of convex optimization. In this lab we will focus on linear and quadratic programming. A *linear program* is a linear constrained optimization problem. Such a problem can be stated in several different forms, one of which is

$$\begin{aligned} \text{minimize} \quad & c^\mathsf{T} x \\ \text{subject to} \quad & Gx + s = h \\ & Ax = b \\ & s \geq 0. \end{aligned}$$

This is the formulation used by CVXOPT. In this formulation, we require that the matrix A has full row rank, and that the block matrix $[G \quad A]^\mathsf{T}$ has full column rank.

Note that the constraint $Gx + s = h$ includes the term s, which is not part of the objective function, and is known as the *slack variable*. Since $s \geq 0$, the constraint $Gx + s = h$ is equivalent to $Gx \leq h$.

Consider the following example:

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

The final two constraints, $x_1, x_2 \geq 0$, 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

$$G = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \text{and} \qquad h = \begin{bmatrix} 3 \\ 3 \\ 0 \\ 0 \end{bmatrix},$$

then we can express the constraints compactly as

$$Gx \leq h, \qquad \text{where} \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

By adding a slack variable s, we can write our constraints as

$$Gx + s = h,$$

which matches the form discussed above. In the case of this particular example, we ignore the extra constraint

$$Ax = b,$$

since we were given no equality constraints.

Now we proceed to solve the problem using CVXOPT. We need to initialize the arrays c, G, and h, and then pass them to the appropriate function. CVXOPT uses its own data type for an array or matrix, and while similar to the NumPy array, it does have a few differences, especially when it comes to initialization. Below, we initialize CVXOPT matrices for c, G, and h.

```
>>> from cvxopt import matrix
>>> c = matrix([-4., -5.])
>>> G = matrix([[1., 2., -1., 0.],[2., 1., 0., -1.]])
>>> h = matrix([ 3., 3., 0., 0.])
```

Achtung!

Observe that 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.],[2., 1.],[-1., 0.],[0., -1]])
>>> h = np.array([3., 3., 0., 0.])

>>> #Now convert to CVXOPT matrix type
>>> c = matrix(c)
>>> G = matrix(G)
>>> h = matrix(h)
```

Use whichever method is most convenient. In this lab, we will initialize non-trivial matrices first as ndarrays for consistency.

Finally, be sure the entries in the matrices are floats!

Having initialized the necessary objects, we are now ready to solve the problem. We will use the CVXOPT function for linear programming `solvers.lp`, and we simply need to pass in c, G, and h as arguments.

```
>>> from cvxopt import solvers
>>> sol = solvers.lp(c, G, h)
     pcost       dcost       gap    pres   dres   k/t
 0: -8.1000e+00 -1.8300e+01  4e+00  0e+00  8e-01  1e+00
 1: -8.8055e+00 -9.4357e+00  2e-01  1e-16  4e-02  3e-02
 2: -8.9981e+00 -9.0049e+00  2e-03  1e-16  5e-04  4e-04
 3: -9.0000e+00 -9.0000e+00  2e-05  1e-16  5e-06  4e-06
 4: -9.0000e+00 -9.0000e+00  2e-07  1e-16  5e-08  4e-08
Optimal solution found.
>>> print sol['x']
[ 1.00e+00]
[ 1.00e+00]
>>> print sol['primal objective']
-8.99999981141
>>> print type(sol['x'])
<type 'cvxopt.base.matrix'>
```

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 the time being, we will only focus on the values of x and the primal objective value (i.e. the minimum value achieved by the objective function).

Achtung!

Note that the minimizer of the `solvers.lp()` function returns a `cvxopt.base.matrix` object. In order to use the minimzer again in other algebraic expressions, you need to convert it first to a flattened numpy array, which can be done quickly with `np.ravel()`. Please return all minimizers in this lab as flattened numpy arrays.

Problem 1. Solve the following convex optimization problem:

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

Report the values for x and the objective value that you obtain. Remember to make the necessary adjustments so that all inequality constraints are $\leq$ rather than $\geq$.

The l_1 minimization problem is to

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

This problem can be converted into a linear program by introducing an additional vector u of length n, and then solving:

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

Of course, solving this gives values for the optimal u and the optimal x, but we only care about the optimal x.

Problem 2. Write a function called `l1Min()` that takes a matrix A and vector b as inputs, and solves the l_1 optimization problem. Report the values for x and the objective value. Remember to first discard the unneccesary u values from the minimizer.

Supply Center	Number of pianos available
1	7
2	2
3	4

Table 10.1: Number of pianos available at each supply center

Demand Center	Number of pianos needed
4	5
5	8

Table 10.2: Number of pianos needed at each demand center

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 10.3. The company wants to minimize shipping costs for the pianos while meeting the demand. How many pianos should each supply center send to each demand center?

The variables p, q, r, s, t, and u must be nonnegative and satisfy the following three supply constraints and two demand constraints:

$$\begin{aligned} p + q &= 7 \\ r + s &= 2 \\ t + u &= 4 \\ p + r + t &= 5 \\ q + s + u &= 8 \end{aligned}$$

The objective function is the number of pianos shipped from each location multiplied by the respective cost:

$$4p + 7q + 6r + 8s + 8t + 9u.$$

There a several ways to solve this linear program. We want our answers to be integers, and this added constraint in general 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.

Here, G and h constrain the variables to be non-negative. Because CVXOPT uses the format $Gx \leq h$, we see that G must be a 6×6 identity matrix multiplied by -1, and h is just a column vector of zeros. The matrices A and b represent the supply and demand constraints, since these are equality constraints. Try initializing these arrays and solving the linear program by entering the code below. (Notice that we pass more arguments to `solvers.lp` since we have equality constraints.)

```
>>> c = matrix([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.],
```

Supply Center	Demand Center	Cost of transportation	Number of pianos
1	4	4	p
1	5	7	q
2	4	6	r
2	5	8	s
3	4	8	t
3	5	9	u

Table 10.3: Cost of transporting one piano from a supply center to a demand center

```
                        [0.,0.,0.,0.,1.,1.],
                        [1.,0.,1.,0.,1.,0.],
                        [0.,1.,0.,1.,0.,1.]]))
>>> b = matrix([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  4e-17  2e-01  1e+00
Terminated (singular KKT matrix).
>>> print sol['x']
[ 3.00e+00]
[ 4.00e+00]
[ 5.00e-01]
[ 1.50e+00]
[ 1.50e+00]
[ 2.50e+00]
>>> print sol['primal objective']
89.5
```

Notice that some problems occurred. First, CVXOPT alerted us to the fact that the algorithm terminated prematurely (due to a singular matrix). Further, the solution that was obtained does 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 *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

$$x + 2y - 3z = 4.$$

This is equivalent to the pair of inequality constraints

$$x + 2y - 3z \leq 4,$$
$$x + 2y - 3z \geq 4.$$

Of course, we require only $\leq$ constraints, so we obtain the pair of constraints

$$x + 2y - 3z \leq 4,$$
$$-x - 2y + 3z \leq -4.$$

Apply this process to the last equality constraint. You will obtain a new matrix G with several additional rows (to account for the new inequality constraints); a new vector h, also with more entries; a smaller matrix A; a smaller vector b.

Problem 3. Solve the problem by converting the last equality constraint into an inequality constraint. Report the optimal values for x and the objective function.

Quadratic Programming

Quadratic programming is similar to linear programming, one exception being that the objective function is quadratic rather than linear. The constraints, if there are any, are still of the same form. Thus G, h, A, and b are optional. The formulation that we will use is

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^\mathsf{T}Px + q^\mathsf{T}x \\ \text{subject to} \quad & Gx \leq h \\ & Ax = b, \end{aligned}$$

where P is a positive semidefinite symmetric matrix. In this formulation, we require again that A has full row rank, and that the block matrix $[P \quad G \quad A]^\mathsf{T}$ has full column rank.

As an example, let us minimize the quadratic function

$$f(y, z) = 2y^2 + 2yz + z^2 + y - z.$$

Note that there are no constraints, so we only need to initialize the matrix P and the vector q. To find these, we first rewrite our function to match the formulation given above. Note that if we let

$$P = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \qquad q = \begin{bmatrix} d \\ e \end{bmatrix}, \qquad \text{and} \qquad x = \begin{bmatrix} y \\ z \end{bmatrix},$$

then

$$\begin{aligned} \frac{1}{2}x^\mathsf{T}Px + q^\mathsf{T}x &= \frac{1}{2}\begin{bmatrix} y \\ z \end{bmatrix}^\mathsf{T}\begin{bmatrix} a & b \\ b & c \end{bmatrix}\begin{bmatrix} y \\ z \end{bmatrix} + \begin{bmatrix} d \\ e \end{bmatrix}^\mathsf{T}\begin{bmatrix} y \\ z \end{bmatrix} \\ &= \frac{1}{2}ay^2 + byz + \frac{1}{2}cz^2 + dy + ez \end{aligned}$$

Thus, we see that the proper values to initizalize our matrix P and vector q are:

$$\begin{aligned} a &= 4 & d &= 1 \\ b &= 2 & e &= -1 \\ c &= 2 \end{aligned}$$

Now that we have the matrix P and vector q, we are ready to use the CVXOPT function for quadratic programming `solvers.qp`.

```
>>> P = matrix(np.array([[4., 2.], [2., 2.]]))
>>> q = matrix([1., -1.])
>>> sol=solvers.qp(P, q)
```

```
>>> print(sol['x'])
[-1.00e+00]
[ 1.50e+00]
>>> print sol['primal objective']
-1.25
```

Problem 4. Find the minimizer and minimum of

$$g(x,y,z) = \frac{3}{2}x^2 + 2xy + xz + 2y^2 + 2yz + \frac{3}{2}z^2 + 3x + z$$

Problem 5. The l_2 minimization problem is to

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

This problem is equivalent to a quadratic program, since $\|x\|_2 = x^\mathsf{T}x$. Write a function called `l2Min()` that takes a matrix A and vector b as inputs and solves the l_2 minimization problem. Report the values for x and the objective value.

Allocation Models

Allocation models lead to simple linear programs. An allocation model seeks to allocate a valuable resource among competing needs. Consider the following example is taken from "Optimization in Operations Research" by Ronald L. Rardin.

The U.S. Forest service has used an allocation model to deal with the task of managing national forests. The model begins by dividing the land into a set of analysis areas. Several land management policies (also called prescriptions) are then proposed and evaluated for each area. An *allocation* is how much land (in acreage) in each unique analysis area will be assigned to each of the possible prescriptions. We seek to find the best possible allocation, subject to forest-wide restrictions on land use.

The file `ForestData.npy` contains data for a fictional national forest (you can also find the data in Table **??**). There are 7 areas of analysis and 3 prescriptions for each of them.

Column 1: i, area of analysis

Column 2: s_i, size of the analysis area (in thousands of acres)

Column 3: j, prescription number

Column 4: $p_{i,j}$, net present value (NPV) per acre of in area i under prescription j

Column 5: $t_{i,j}$, protected timber yield per acre in area i under prescription j

Column 6: $g_{i,j}$, protected grazing capability per acre for area i under prescription j

Column 7: $w_{i,j}$, wilderness index rating (0 to 100) for area i under prescription j

Forest Data

Analysis Area, i	Acres (1000)'s s_i	Prescription j	NPV, (per acre) $p_{i,j}$	Timber, (per acre) $t_{i,j}$	Grazing, (per acre) $g_{i,j}$	Wilderness Index, $w_{i,j}$
1	75	1	503	310	0.01	40
		2	140	50	0.04	80
		3	203	0	0	95
2	90	1	675	198	0.03	55
		2	100	46	0.06	60
		3	45	0	0	65
3	140	1	630	210	0.04	45
		2	105	57	0.07	55
		3	40	0	0	60
4	60	1	330	112	0.01	30
		2	40	30	0.02	35
		3	295	0	0	90
5	212	1	105	40	0.05	60
		2	460	32	0.08	60
		3	120	0	0	70
6	98	1	490	105	0.02	35
		2	55	25	0.03	50
		3	180	0	0	75
7	113	1	705	213	0.02	40
		2	60	40	0.04	45
		3	400	0	0	95

Let $x_{i,j}$ be the amount of land in area i allocated to prescription j. Under this notation, an allocation is just a vector consisting of the $x_{i,j}$'s. For this particular example, the allocation vector is of size $7 \cdot 3 = 21$. Our goal is to find the allocation vector that maximizes net present value, while producing at least 40 million board-feet of timber, at least 5 thousand animal-unit months of grazing, and keeping the average wilderness index at least 70.

Of course, the allocation vector is also constrained to be nonnegative, and all of the land must be allocated precisely.

Note that since acres are in thousands, we will divide the constraints of timber and animal months of grazing by 1000 in our problem setup, and compensate for this after we obtain a solution. We can summarize our problem as follows:

$$\begin{aligned}
\text{maximize } & \sum_{i=1}^{7}\sum_{j=1}^{3} p_{i,j}x_{i,j} \\
\text{subject to } & \sum_{j=1}^{3} x_{i,j} = s_i \text{ for } i = 1,..,7 \\
& \sum_{i=1}^{7}\sum_{j=1}^{3} t_{i,j}x_{i,j} \geq 40,000 \\
& \sum_{i=1}^{7}\sum_{j=1}^{3} g_{i,j}x_{i,j} \geq 5 \\
& \frac{1}{788}\sum_{i=1}^{7}\sum_{j=1}^{3} w_{i,j}x_{i,j} \geq 70 \\
& x_{i,j} \geq 0 \text{ for } i = 1,...,7 \text{ and } j = 1,2,3
\end{aligned}$$

Problem 6. Solve the problem above. Return the minimizer x of $x_{i,j}$'s. Also return the maximum total net present value, which will be equal to the primal objective of the appropriately minimized linear function, multiplied by -1000. (This final multiplication after we have obtained a solution changes our answer to be a maximum, and compensates for the data being in thousands of acres).

You can learn more about CVXOPT at `http://abel.ee.ucla.edu/cvxopt/documentation/`.

11 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.*

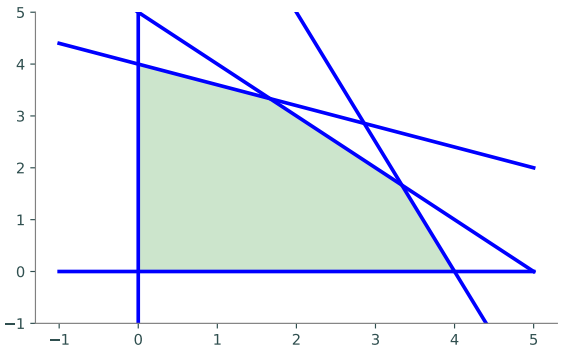
Standard Form

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

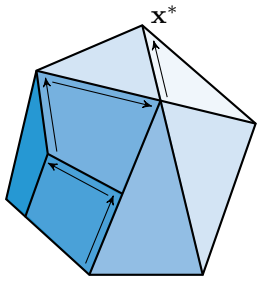
$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \preceq \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0} \end{aligned}$$

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 11.1). 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 11.1: 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.

$$\begin{aligned} \text{maximize} \quad & 3x_0 + 2x_1 \\ \text{subject to} \quad & x_0 - x_1 \leq 2 \\ & 3x_0 + x_1 \leq 5 \\ & 4x_0 + 3x_1 \leq 7 \\ & x_0, x_1 \geq 0. \end{aligned}$$

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.

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[a]. 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.

[a]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.

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 ith row of the constraint matrix is represented by the index $n+i-1$).

We also need some way to track which variables are *basic* (non-zero) and which variables are *nonbasic* (those that have value 0). A useful representation for the variables is a Python list (or NumPy array), where the elements of the list are integers. Since we know how many basic variables we have (m), we can partition the list so that all the basic variables are kept in the first m locations, and all the non-basic variables are stored at the end of the list. The ordering of this list is important. In particular, if $i \leq m$, the ith element of the list represents the basic variable corresponding to the ith row of A. Henceforth we will refer to this list as the *index list*.

Initially, the basic variables are simply the slack variables, and their values correspond to the values of the vector $\mathbf{b}$. In our example, we have 2 primal variables x_0 and x_1, and we must add 3 slack variables. Thus, we instantiate the following index list:

```
>>> L = [2, 3, 4, 0, 1]
```

Notice how the first 3 entries of the index list are $2, 3, 4$, the indices representing the slack variables. This reflects the fact that the basic variables at this point are exactly the slack variables.

As the Simplex Algorithm progresses, however, the basic variables change, and it will be necessary to swap elements in our index list. For example, suppose the variable represented by the index 4 becomes nonbasic, while the variable represented by index 0 becomes basic. In this case we swap these two entries in the index list.

```
>>> L[2], L[3] = L[3], L[2]
>>> L
[2, 3, 0, 4, 1]
```

Now our index list tells us that the current basic variables have indices $2, 3, 0$.

Problem 2. Design and implement a way to store and track all of the basic and non-basic variables.

Hint: Using integers that represent the index of each variable is useful for Problem 4.

Creating a Tableau

After we have determined that our program is feasible, we need to create the *tableau* (sometimes called the *dictionary*), a data structure to track the state of the algorithm. You may structure the tableau to suit your specific implementation. Remember that your tableau will need to include in some way the slack variables that you created in Problem 2.

There are many different ways to build your tableau. One way is to mimic the tableau that is often used when performing the Simplex Algorithm by hand. Define

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

where I_m is the $m \times m$ identity matrix, 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 tableau has the form

$$T = \begin{bmatrix} 0 & -\bar{\mathbf{c}}^\mathsf{T} & 1 \\ \mathbf{b} & \bar{A} & \mathbf{0} \end{bmatrix} \tag{11.1}$$

The columns of the tableau correspond to each of the variables (both primal and slack), and the rows of the tableau correspond to the basic variables. Using the convention introduced above of representing the variables by indices in the index list, we have the following correspondence:

$$\text{column } i \Leftrightarrow \text{index } i-2, \qquad i = 2, 3, \ldots, n+m+1,$$

and

$$\text{row } j \Leftrightarrow L_{j-1}, \qquad j = 2, 3, \ldots, m+1,$$

where L_{j-1} refers to the $(j-1)$th entry of the index list.

For our example problem, the initial index list is

$$L = (2, 3, 4, 0, 1),$$

and the initial tableau is

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

The third column corresponds to index 1, and the fourth row corresponds to index 4, since this is the third entry of the index list.

The advantage of using this kind of tableau is that it is easy to check the progress of your algorithm by hand. The disadvantage is that pivot operations require careful bookkeeping to track the variables and constraints.

Problem 3. Add a method to your Simplex solver that will create the initial tableau as a NumPy array.

Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping basic and nonbasic variables, and transforming the tableau 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 tableau 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 tableau so that the pivot column becomes an 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 tableau (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 basic variable.

Using our initial tableau T in the example, we stop at the second column:

$$T = \left[\begin{array}{c|c|cccc} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5 & 3 & 1 & 0 & 1 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{array}\right]$$

We now know that our pivot element will be found in the second column. The entering index is thus 0.

Next, we select the pivot element from among the positive entries in the pivot column (ignoring the entry in the first row). *If all entries in the pivot column are non-positive, the problem is unbounded and has no solution.* In this case, the algorithm should terminate. Otherwise, assuming our pivot column is the jth column of the tableau and that the positive entries of this column are $T_{i_1,j}, T_{i_2,j}, \ldots, T_{i_k,j}$, we calculate the ratios

$$\frac{T_{i_1,1}}{T_{i_1,j}}, \frac{T_{i_2,1}}{T_{i_2,j}}, \ldots, \frac{T_{i_k,1}}{T_{i_k,j}},$$

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 nonbasic variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are positive, 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, third row is our designated pivot element, and our leaving index is $L_2 = 3$:

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

Problem 4. Write a method that will determine the pivot row and pivot column according to Bland's Rule.

Definition 11.1 (Bland's Rule). *Choose the nonbasic variable with the smallest index that has a positive coefficient in the objective function as the leaving variable. Choose the basic variable with the smallest index among all the binding basic 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.

The next step is to swap the entering and leaving indices in our index list. In the example, we determined above that these indices are 0 and 3. We swap these two elements in our index list, and the updated index list is now

$$L = (2, 0, 4, 3, 1),$$

so the basic variables are now given by the indices $2, 0, 4$.

Finally, we perform row operations on our tableau in the following way: divide the pivot row by the 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:

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

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

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

The result of these row operations is our updated Tableau, and the pivot operation is complete.

Problem 5. 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 Tableau

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial tableau. 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 tableau. More specifically, before each pivoting operation, check whether all of the entries in the top row of the tableau (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 tableau and index list tell us everything we need to know. The maximum value attained by the objective function is found in the upper leftmost entry of the tableau. The nonbasic variables, whose indices are located in the last n entries of the index list, all have the value 0. The basic variables, whose indices are located in the first m entries of the index list, have values given by the first column of the tableau. Specifically, the basic variable whose index is located at the ith entry of the index list has the value $T_{i+1,1}$.

In our example, suppose that our algorithm terminates with the tableau and index list in the following state:

$$T = \begin{bmatrix} 5.2 & 0 & 0 & 0 & .2 & .6 & 1 \\ .6 & 0 & 0 & 1 & -1.4 & .8 & 0 \\ 1.6 & 1 & 0 & 0 & .6 & -.2 & 0 \\ .2 & 0 & 1 & 0 & -.8 & .6 & 0 \end{bmatrix}$$

$$L = (2, 0, 1, 3, 4).$$

Then the maximum value of the objective function is 5.2. The nonbasic variables have indices $3, 4$ and have the value 0. The basic variables have indices $2, 0$, and 1, and have values $.6, 1.6$, and $.2$, respectively. In the notation of the original problem statement, the solution is given by

$$x_0 = 1.6$$
$$x_1 = .2.$$

Problem 6. Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the maximum value, the basic variables, and the nonbasic variables. The basic and nonbasic 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 simple 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)
(5.200,
 {0: 1.600, 1: 0.200, 2: 0.600},
 {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 tableau 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 basic 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 ith product is sold at a unit price p_i, and there are at most m_j units of the jth resource available. Additionally, each unit of the ith 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 $x_1, x_2, \ldots, x_n$ 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

$$\sum_{i=1}^{n} a_{j,i} x_i \leq m_j \text{ for } j = 1, 2, \ldots, m.$$

Finally, we have the demand constraints which tell us not to exceed the demand for the products:

$$x_i \leq d_i \text{ for } i = 1, 2, \ldots, n$$

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 7. 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.

Beyond Simplex

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [**Nash2000**]. 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 [**Klee1972**]. 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.

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

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 $(0, 0, \ldots, 0, 5^n)$. Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.

12 Optimization with Scipy

Lab Objective: *The Optimize package in Scipy provides highly optimized and versatile methods for solving fundamental optimization problems. In this lab we introduce the syntax and variety of* `scipy.optimize` *as a foundation for unconstrained numerical optimization.*

Numerical optimization is one of the most common modern applications of mathematics. Many mathematical problems can be rewritten in terms of optimization, and unless the problem is both simple and small, usually cannot be solved analytically, and must be approximated by numerical computation.

To assist with this, the `scipy.optimize` package has several functions for minimizing, root finding, and curve fitting, of which we will introduce the most essential.

You can learn about all of the functions at `http://docs.scipy.org/doc/scipy/reference/optimize.html`.

Local Minimization

First we will test out a few of the minimization algorithms on the Rosenbrock function, which is defined as

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2.$$

The Rosenbrock function is commonly used when evaluating the performance of an optimization algorithm. Reasons for this include the fact that its minimizer `x = np.array([1., 1.])` is found in curved valley, and so minimizing the function is non-trivial. See Figure 12.1. The Rosenbrock function is included in the optimize package (as `rosen`), as well as its gradient (`rosen_der`) and its hessian (`rosen_hess`).

We will use the `minimize()` function and test some of its algorithms (specified by the keyword argument "method" – see the documentation page for `minimize()`). For each algorithm, you need to pass in a callable function object for the Rosenbrock function, as well as a NumPy array giving the initial guess. For some algorithms, you will additionally need to pass in the jacobian or hessian. You may recognize some of these algorithms, and several of them will be discussed in greater detail later. For this lab, you do not need to understand how they work, just how to use them.

As an example, we'll minimize the Rosenbrock with the Newton-CG method. This method often performs better by including the optional hessian as an argument, which we will do here.

```
>>> import numpy as np
```

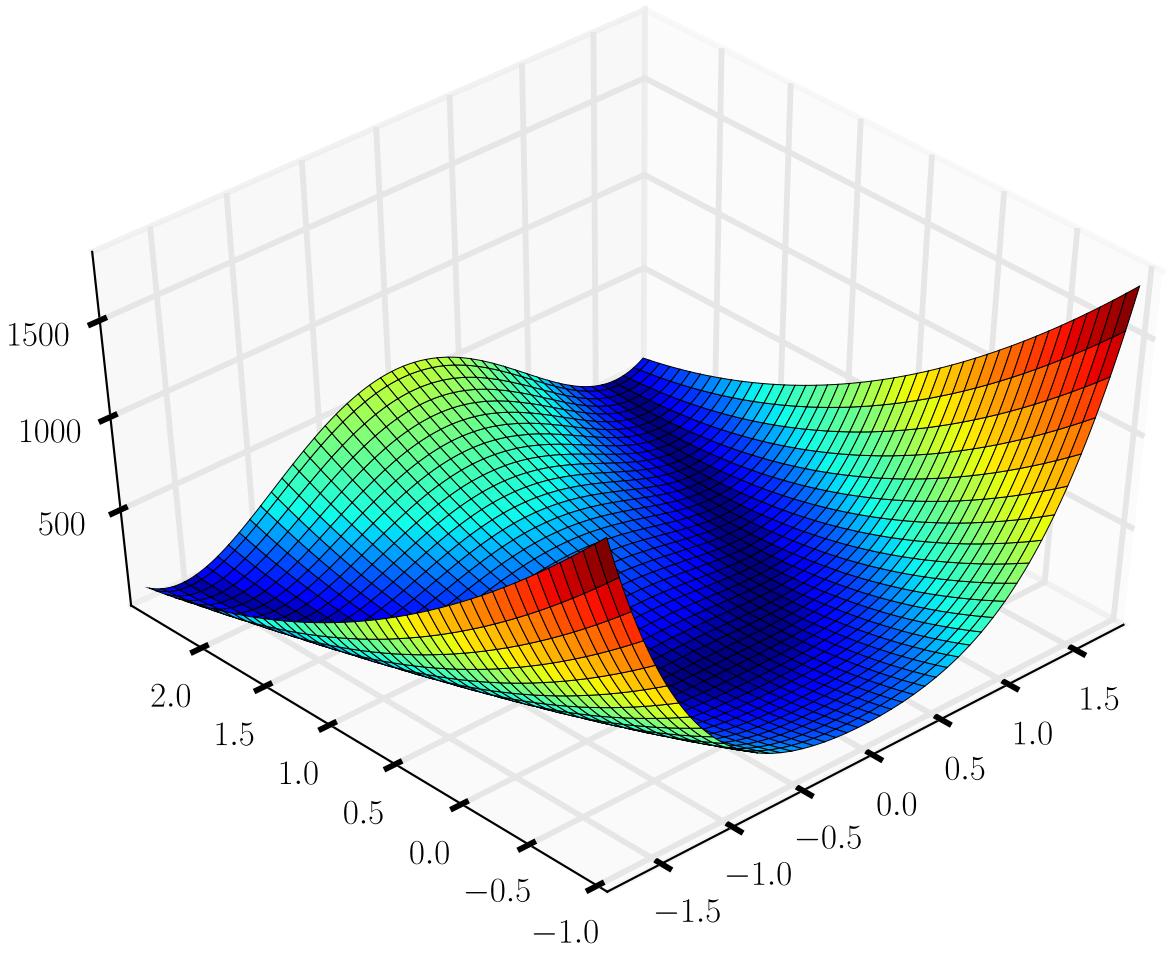


Figure 12.1: $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$

```
>>> from scipy import optimize as opt
>>> x0 = np.array([4., -2.5])
>>> opt.minimize(opt.rosen, x0, method='Newton-CG', hess=opt.rosen_hess,
                                                    jac=opt.rosen_der)
     fun: 1.1496545381999877e-15
     jac: array([  1.12295570e-05,  -5.63744647e-06])
 message: 'Optimization terminated successfully.'
    nfev: 45
    nhev: 34
     nit: 34
    njev: 78
  status: 0
 success: True
       x: array([ 0.99999997,  0.99999993])
```

As the online documentation indicates, `opt.minimize()` returns an object of type `opt.optimize.OptimizeResult`.

The printed output gives you information on the performance of the algorithm. The most relevant output for this lab include

`fun: 1.1496545381999877e-15`, the obtained minimum;

`nit: 96`, the number of iterations the algorithm took to complete;

`success: True`, whether the algorithm converged or not;

`x: array([ 0.99999997, 0.99999993])`, the obtained minimizer.

Each of these outputs can be accessed either by indexing `OptimizeResult` object like a dictionary (`result['nit']`), or as attributes of a class (`result.nit`). We recommend access by indexing, as this is consistent with other optimization packages in Python.

The online documenation for `scipy.optimize.minimize()` includes other optional parameters available to users, for example, to set a tolerance of convergence. In some methods, the derivative may be optional, while it may be necessary in others. While we do not cover all possible parameters in this lab, they should be explored as needed for specific applications.

Problem 1. Use the `opt.minimize()` function to find the minimum of the Rosenbrock function. Test Nelder-Mead, CG, and BFGS, starting each with the initial guess `x_0 = np.array([4., -2.5])`. For each method, print whether it converged, and if so, print how many iterations it took.

Note that the hessian argument is not needed for these paticular methods.

Each of these three algorithms will be explored in great detail later in Volume 2: Nelder-Mead is a variation of the Simplex algorithm, CG is a variant of the Conjugate Gradient algorithm, and BFGS is a quasi-Newton method developed by Broyden, Fletcher, Goldfarb, and Shanno.

The `minimize()` function can use various algorithms, each of which is best for certain problems. Which algorithm one uses depends on the specific nature of one's problem.

It is also important to note that in many optimization applications, very little is known about the function to optimize. These functions are often called *blackbox functions*. For example, one may be asked in the airline industry to analyze and optimize certain properties involving a segment of airplane wing. Perhaps expert engineers have designed extremely robust and complicated software to model this wing segment given certain inputs, but this function is so complicated that nobody except the experts dares to try parcing it. Briefly said, you simply don't want to understand it; nobody wants to understand it.

Fortunately, one can still optimize effectively in such situations by wisely selecting a correct algorithm. However, because so little is known about the blackbox function, one must wisely select an appropriate minimization method and follow the specifications of the problem exactly.

Problem 2. Minimize the `blackbox()` function in the `blackbox_function` module. You don't need to know the source code or how it works in order to minimize it. Simply select the appropriate method of `scipy.optimize.minimize()` for this problem, without passing your method a derivative. You may need to test several methods and determine which is most appropriate.

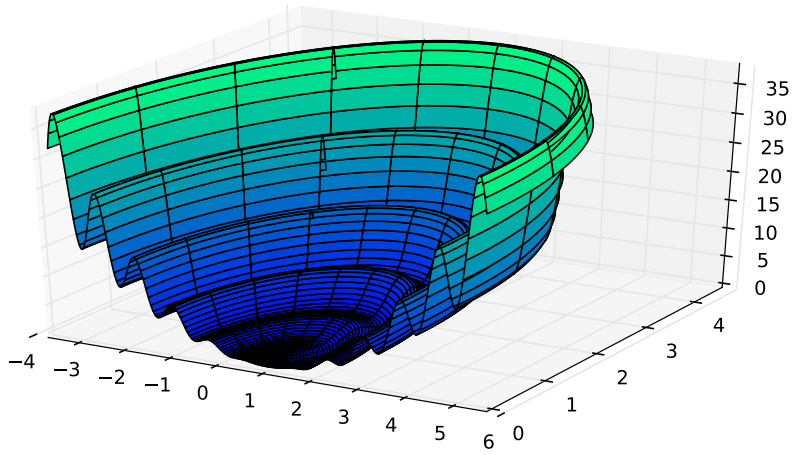


Figure 12.2: $z = r^2(1 + 2\sin^2(4r))$

The function `blackbox()` returns a certain measure of a piecewise-linear curve between two fixed points: the origin, and the point `(40,30)`. This function accepts a one-dimensional `ndarray` of length `m` of y-values, where `m` is the number of points of the piecewise curve excluding endpoints. These points are spaced evenly along the x-axis, so only the y-values of each point are passed into `blackbox()`.

Once you have selected a method, select an initial point with the following code:

```
y_initial = 30*np.random.random_sample(18)
```

Then plot your initial curve and minimizing curve together on the same plot, including endpoints. Note that this will require padding your array of internal y-values with the y-values of the endpoints, so that you plot a total of 20 points for each curve.

Global Minimization via Basin Hopping

In the realm of optimization, convex functions are the most well-behaved, as any local minimum is a global minimum. However, in practice one must frequently deal with non-convex functions, and sometimes we need pick the global minimum out of many local minima.

For example, consider the function

$$z = r^2(1 + \sin^2(4r)),$$

where

$$r = \sqrt{(x+1)^2 + y^2}.$$

Essentially, this is a wavy crater offset from the origin by 1 along the x axis (see Figure 12.2). The presence of many local minima proves to be a difficulty for the minimization algorithms.

For example, if we try using the Nelder-Mead method as previously, with an initial point of `x0 = np.array([-2, -2])`, the algorithm fails to find the global minimum, and instead comes to rest on a local minimum.

```
>>> def multimin(x):
>>>     r = np.sqrt((x[0]+1)**2 + x[1]**2)
>>>     return r**2 *(1+ np.sin(4*r)**2)
>>>
>>> x0 = np.array([-2, -2])
>>> res = opt.minimize(multimin, x0, method='Nelder-Mead')
 final_simplex: (array([[-2.11758025, -2.04313668], [-2.11748198, -2.04319043],
       [-2.11751491, -2.04317242]]), array([ 5.48816866,
                                                5.48816866,  5.48816866]))
           fun: 5.488168656962328
       message: 'Optimization terminated successfully.'
          nfev: 84
           nit: 44
        status: 0
       success: True
             x: array([-2.11758025, -2.04313668])
>>> print res['x']
[-2.11758025, -2.04313668]
>>> print res['fun']
5.488168656962328
>>> print multimin([-1,0])
0.0
```

However, SciPy does have some tools to help us with these problems. Specifically, we can use the `opt.basinhopping()` function.

The `opt.basinhopping()` function uses the same minimizing algorithms (in fact, you can tell it whatever minimizing algorithm you can pass to `opt.minimize()`). However, once it settles on a minimum, it hops randomly to a new point in the domain (depending on how we set the "hopping" distance) that hopefully lies outside of the valley or basin belonging to the current local minimum. It then searches for the minimum from this new starting point, and if it finds a better minimizer, it repeats the hopping process from this new minimizer. Thus, the `opt.basinhopping()` function has multiple chances to escape a local basin and find the correct global minimum.

Only Scipy Version 0.12+ has ***`opt.basinhopping`****. In earlier versions, such as 0.11, you won't find it.*

Problem 3. Explore the documentation for the `opt.basinhopping()` function online or via IPython, and use it to find the global minimum of our `multimin()` function with `x0 = np.array([-2, -2])`. Call it using the same `Nelder-Mead` algorithm with `opt.basinhopping (multimin, x0, stepsize=0.5, minimizer_kwargs={'method':'nelder-mead'})`. Try it first with `stepsize=0.5` and then with `stepsize=0.2`.

Plot the multimin function using the following code:

```
xdomain = np.linspace(-3.5,1.5,70)
```

```
ydomain = np.linspace(-2.5,2.5,60)
X,Y = np.meshgrid(xdomain,ydomain)
Z = multimin((X,Y))
fig = plt.figure()
ax1 = fig.add_subplot(111, projection='3d')
ax1.plot_wireframe(X, Y, Z, linewidth=.5, color='c')
```

Plot the initial point and minima by adapting the following line:

```
ax1.scatter(x_value, y_value, z_value)
```

Why doesn't the alogrithm find the global minimum with `stepsize=0.2`? Print your answer to this question, and return the true global minimum.

Root Finding

The `optimize` package also has functions useful in root-finding. The next example, taken from the online documentation, solves the following nonlinear system of equations using `opt.root`.

$$\begin{bmatrix} x_0 + 1/2(x_0 - x_1)^3 - 1 \\ 1/2(x_1 - x_0)^3 + x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

```
>>> def func(x):
>>>     return [x[0] + 0.5 * (x[0] - x[1])**3 -1.0,
>>>             0.5 * ( x[1] - x[0])**3 + x[1]]
>>> def jac(x):
>>>     return np.array([[1 + 1.5 * (x[0] - x[1])**2,
>>>                       -1.5 * (x[0] - x[1])**2],
>>>                      [-1.5 * (x[1] - x[0])**2,
>>>                       1 + 1.5 * (x[1] - x[0])**2]])
>>> sol = opt.root(func, [0, 0], jac=jac, method='hybr')
>>> print sol.x
[ 0.8411639  0.1588361]
>>> print func(sol.x)
[-1.1102230246251565e-16, 0.0]
```

Problem 4. Find the roots of the system

$$\begin{bmatrix} -x + y + z \\ 1 + x^3 - y^2 + z^3 \\ -2 - x^2 + y^2 + z^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Return the values of x, y, z as an array.

As with `opt.minimize()`, `opt.root()` has more than one algorithm for root finding. Here we have used the `hybr` method. There are also several algorithms for scalar root finding. See the online documentation for more.

Curve Fitting

SciPy also has methods for curve fitting wrapped by the `opt.curve_fit()` function. Just pass it data and a function to be fit. The function should take in the independent variable as its first argument and values for the fitting parameters as subsequent arguments. Examine the following example from the online documentation.

```
>>> import numpy as np
>>> import scipy.optimize as opt

>>> #the function with which to create the data and later fit it
>>> def func(x,a,b,c):
>>>     return a*np.exp(-b*x) + c

>>> #create perturbed data
>>> x = np.linspace(0,4,50)
>>> y = func(x,2.5,1.3,0.5)
>>> yn = y + 0.2*np.random.normal(size=len(x));

>>> #perform the fit
>>> popt, pcov = opt.curve_fit(func,x,yn)
```

The variable `popt` now contains the fitted parameters and `pcov` gives the covariance of the fit. See Figure 12.3 for a plot of the data and the fitted curve.

One of the most fundamental phenomena in the physical and engineering sciences is turbulent convection, wherein an unstable density gradient induces a fluid to move chaotically (basically, hot air rises). This problem is so important that experiments and numerical simulations have been pushed to their limits in the past several decades to determine the qualitative nature of the fluid's motion under an extreme forcing (think of boiling a pot of water, but instead at temperatures akin to the interior of the sun). The strength of the forcing (amount of the enforced temperature gradient) is measured by the non-dimensional Rayleigh number R. Of paticular interest is to determine how well the chaotic turbulent flow transports the heat from the hot bottom to the cold top, as measured by the Nusselt number ν. One of the primary goals of experiments, simulations, and analysis is to determine how the Nusselt number ν depends on the Rayleigh number R, i.e. if the bottom of the pot of water is heated more strongly, how much faster does the boiling water transport heat to the top?

It is often generically believed that the Nusselt number obeys a power law of the form $\nu = cR^{\beta}$, where $\beta \leq 1/2$. Through some mild assumptions on the temperature, we can construct an eigenvalue problem that we solve numerically for a variety of Rayleigh numbers, thus obtaining an upper bound on the Nusselt number as $\nu \leq cR^{\beta}$. With our physical specifications of the problem, we may predict $\beta < 1/2$.

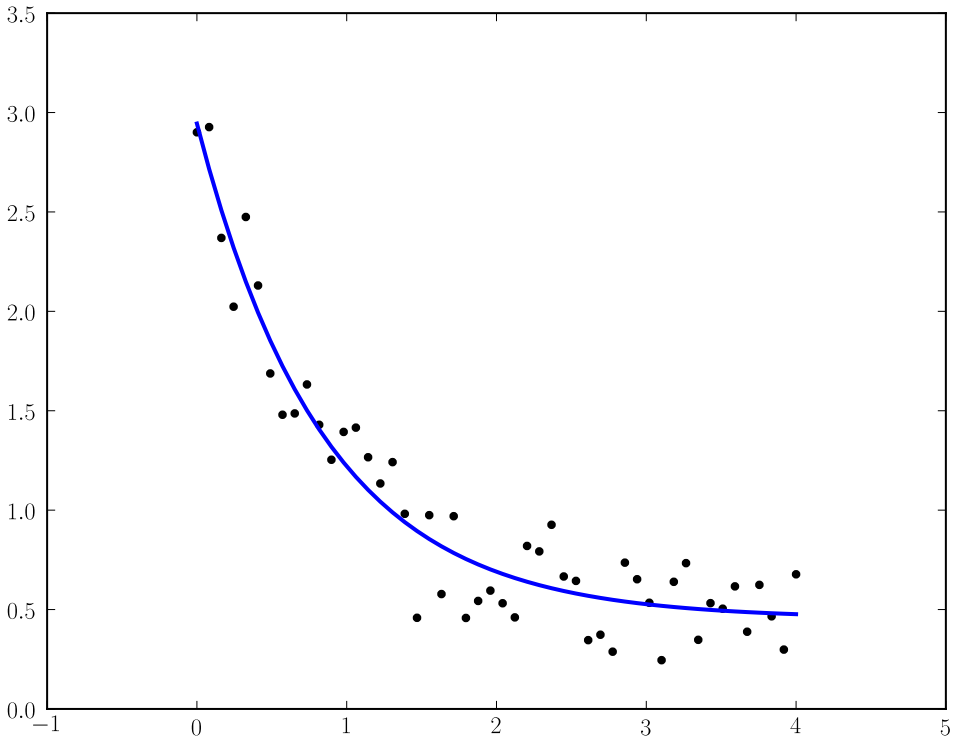


Figure 12.3: Example of perturbed data graphed with the resulting curve using the fitted parameters: $a = 2.72$, $b = 1.31$, and $c = 0.45$.

Problem 5. Use `opt.curve_fit()` to fit a curve to data obtained from numerical simulations of convection. The data are in the file `convection.npy`. The first column is R, the Rayleigh number, and the second column is ν, the Nusselt number. With the convection equation

$$\nu = cR^{\beta},$$

use `opt.curve_fit()` to find a fit to the data using c and β as the fitting parameters. See Figure 12.4 for a plot of the data along with a fitted curve. Though it may be difficult to see in the figure, the first four points skew the data, and do not help us determine the appropriate long-term values of c and β. Thus, do not use the first four points when fitting a curve to the data, but include them in the plot.

Just so that you know that you are getting realistic values, c should be around .1, and β should be less than 1/2. Return your values for c and β in a NumPy array of length 2.

The `scipy.optimize` package has many other useful functions, and is a good first resource when confronting a numerical optimization problem. See the online documentation for further details.

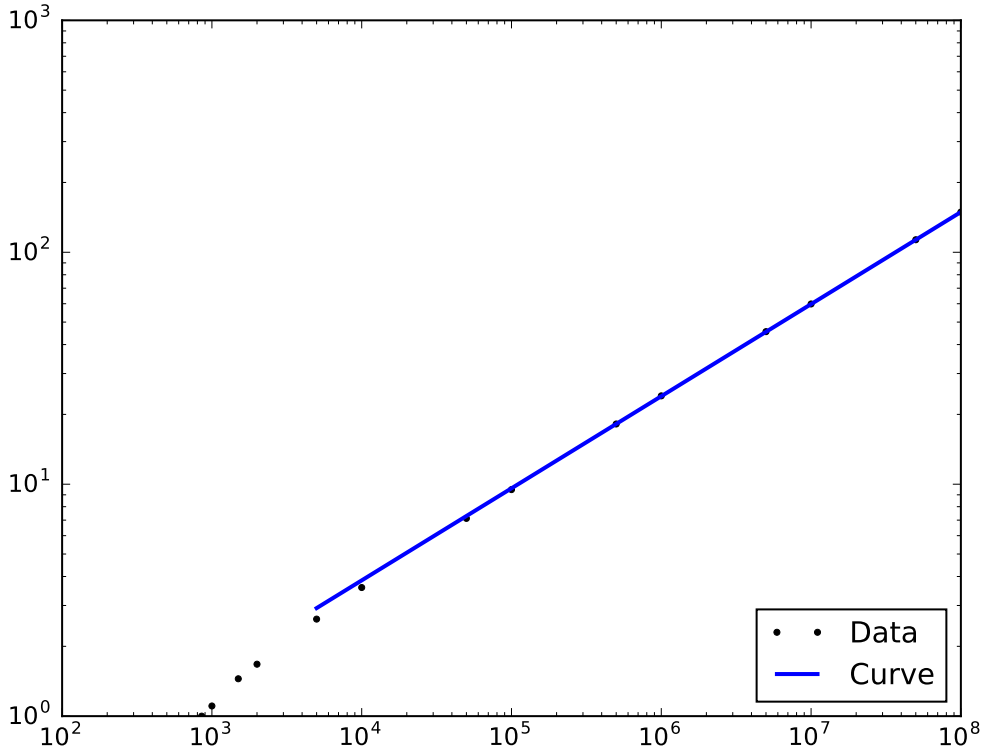


Figure 12.4: The black points are the data from `convection.npy` plotted as a scatter plot. The blue line is a fitted curve.

13 Line Search Methods

Lab Objective: *Many high-dimensional optimization algorithms rely on one-dimensional optimization methods. In this lab, we implement four* line search *algorithms for optimizing scalar-valued functions defined on* $\mathbb{R}$*. We will generalize some of these approaches to high-dimensional optimization problems in subsequent labs.*

Overview of Line Search Algorithms

Imagine you are hiking a steep mountain. When it is time to head home, thick fog gathers around, reducing visibility to just a couple of feet. How can you find your way back with such limited visibility? One strategy is to pick a downhill direction and follow that direction as far as you can, or until it starts leading upward again. Then choose another downhill direction, and take that as far as you can, repeating the process. By always choosing a downhill direction, you hope to eventually make it back to the base of the mountain.

This is the basic approach of *line search algorithms* for numerical optimization. Let f be a scalar-valued function. The goal is to find the *global minimizer* $\mathbf{x}^*$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x}$ in the domain of f. After choosing an initial guess $\mathbf{x}_0$, we produce a sequence of approximations $\mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{x}_3,\ \ldots$ using the rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \tag{13.1}$$

The point $\mathbf{p}_k$ is the *search direction* (which way to go) and the scalar α_k is called the *step size* (how far to go). The choice of search direction and step size at each step is what defines different line search algorithms.

Derivative versus Derivative-Free Methods

A function's derivative provides information about how the value of the function changes at each point, and can be used to determine local optima. Unfortunately, not all objective functions are differentiable, and many are difficult or costly to differentiate. Thus some line search methods utilize the derivative of the objective function, but others do not.

Interval Approximation Methods for Unimodal Functions

A function $f : [a, b] \to \mathbb{R}$ satisfies the *unimodal property* if it has just **one** local (hence global) minimum and is monotonic to the left and right of the minimum. The following line search methods optimize a unimodal function by partitioning the function's domain into progressively smaller intervals that contain the minimizer.

Golden Section Search

Let $f : [a, b] \to \mathbb{R}$ be unimodal. Starting with the closed interval $[a, b]$, the *golden section search* finds a smaller interval that contains x^* as follows.

Choose two points a' and b' with $a' < b'$. If $f(a') \geq f(b')$, the unimodal property guarantees that the minimizer must be in the interval $[a', b]$, for otherwise the function f would have a local minimum in both $[a, a']$ and $[a', b]$. In the next step, we repeat the process over the interval $[a', b]$. If instead $f(a') \leq f(b')$, then we choose the interval $[a, b']$ for the next step. Finally, if $f(a') = f(b')$, then it does not matter which interval is chosen.

It turns out that there is an optimal choice for the two test points a' and b'. Given the current interval $[a, b]$, choose a' and b' satisfying

$$a' = a + \rho(b - a)$$
$$b' = a + (1 - \rho)(b - a),$$

where $\rho = \frac{1}{2}(3 - \sqrt{5}) \approx 0.382$ (this constant is related to the famous *golden ratio*, hence the name of the algorithm).

At each step, the interval is reduced by a factor of $1 - \rho$, which means that after n steps, the minimizer is pinned down to within an interval approximately $(0.618)^n$ times the length of the original interval. Note that this rate of convergence is independent of the objective function.

Problem 1. Write a function that accepts a callable function f, interval limits a and b, and a number of iterations `niter` to calculate. Implement the golden section search as described above, returning the midpoint of the final interval.

Use your function to minimize $f(x) = e^x - 4x$ on the interval $[0, 3]$. How many steps do you need to take to get within .001 of the true minimizer?

Bisection Algorithm

This method is similar to the Golden Ratio method, but instead of cutting our interval into two overlapping sections, we divide the interval evenly in half, and then use the derivative $f'(x)$ to determine where the minimizer lies. This method takes advantage of the fact that a unimodal function's derivative is strictly less than 0 to the left of the minimizer, and strictly greater than 0 to the right.

For a unimodal function $f : [a, b] \to \mathbb{R}$, take the midpoint, $x_1 = \frac{b+a}{2}$. If $f'(x_1) > 0$, the critical point must lie in the interval $[a, x_1]$; otherwise, the critical point lies in the other half of the interval, $[x_1, b]$. We repeat this process on the correct interval until the desired accuracy is achieved.

At each step, the interval is reduced by a factor of two. This is slightly quicker than the factor of $1 - \rho$ in the Golden Section search method. Like the golden section search, the convergence of this method is independent of the objective function.

Problem 2. Write a function that accepts a callable function f, interval limits a and b, and a number of iterations `niter` to calculate. Implement the bisection algorithm as described above, returning the midpoint of the final interval.

Use your function to minimize the same objective function as in Problem 1. How many steps do you need to take to get within .001 of the true minimizer? Time both algorithms and report which one is faster to converge.

Newton's Method and Quasi-Newton Methods

Newton's Method, is a basic line search algorithm that uses the derivatives of the function to select a direction and step size.

To use this method, we need a function that is twice differentiable. The idea is to approximate the function with a quadratic polynomial and then solve the trivial problem of minimizing the polynomial. Doing so in an iterative manner can lead us to the actual minimizer. Let f be a function satisfying the appropriate conditions, and let us make an initial guess, x_0. The relevant quadratic approximation to f at x_0 is

$$q(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2,$$

which is the second-degree Taylor polynomial for f centered at x_0. The minimum for this quadratic function is easily found by solving $q'(x) = 0$, and we take the obtained x-value as our new approximation. The formula for the $(k+1)$-th approximation, which the reader can verify, is

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

In the one dimensional case, there are only two search directions: to the right $(+)$ or to the left $(-)$. Newton's method chooses the search direction $\mathbf{p}_k = \text{sign}(-f'(x_k)/f''(x_k))$ and the step size $\alpha_k = |f'(x_k)/f''(x_k)|$.

The convergence properties of this sequence depend heavily on the initial guess x_0 and the function f. Roughly speaking, if x_0 is sufficiently close to the actual minimizer, and if f is well-approximated by parabolas, then one can expect the sequence to converge quickly. However, there are cases when the sequence converges slowly or not at all. See Figure 13.1.

Problem 3. Implement Newton's Method. You will write a function that takes a function and its two derivatives, as well as a starting point. It will return the minimizer.

Use this function to minimize $f(x) = x^2 + \sin(5x)$ with an initial guess of $x_0 = 0$. Now try other initial guesses farther away from the true minimizer, and note when the method fails to obtain the correct answer.

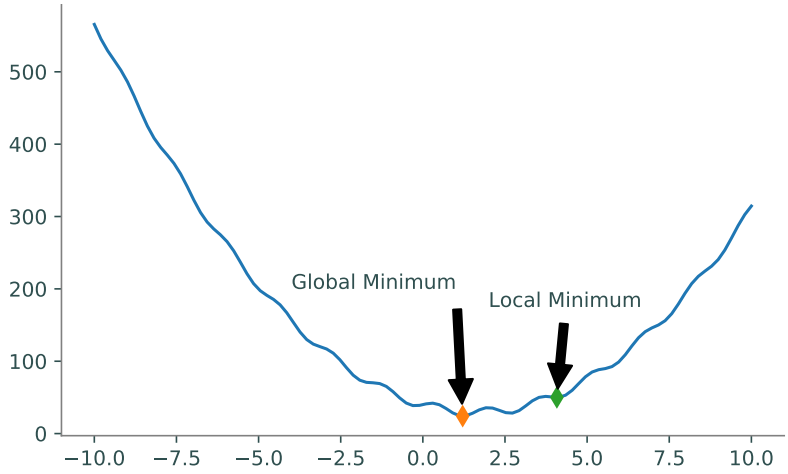


Figure 13.1: The results of Newton's Method using two different initial guesses. The global minimizer was correctly found with initial guess of 1. However, an initial guess of 4 led to only a local minimum.

One-Dimensional Secant Method

Sometimes we would like to use Newton's Method, but for whatever reason we don't have a second derivative. Maybe calculating it is too costly or the function is not twice differentiable. In this situation we can approximate the second derivative using the first derivative. The approximation using a secant calculation, hence the name. We use the same basic algorithm as in Newton's Method, but with an approximation for the second derivative of the objective function.

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''_{approx}(x_n)},$$

where we approximate the second derivative as follows:

$$f''_{approx}(x_n) = \frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}.$$

This gives us the final equation:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f'(x_n) - f'(x_{n-1})} f'(x_n),$$

Notice that this equation requires two initial points (both x_n and x_{n-1}) to calculate the next estimate.

Problem 4. Write a function that accepts a scalar-valued function f, its first derivative Df, and two starting points. Use the secant method to find and return the minimizer of f.

Use your function to minimize $f(x) = x^2 + \sin(x) + \sin(10x)$ with initial guesses of $x_0 = 0$ and $x_1 = -1$. Now try other initial guesses farther away from the true minimizer, and note when the method fails to obtain the correct answer.

(Hint: You may want to plot this function to understand why this problem is so sensitive to the starting point.)

General Line Search Methods

Step Size Calculation

Given a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ that we wish to minimize, and assuming that we already have a current point $\mathbf{x}_k$ and direction $\mathbf{p}_k$ in which to search, how do we choose our step size α_k? If our step size is too small, we will not make good progress toward the minimizer, and convergence will be slow. If the step size is too large, however, we may overshoot and produce points that are far away from the solution. A common approach to pick an appropriate step size involves the *Wolfe conditions*:

$$\begin{aligned} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) &\leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k, && (0 < c_1 < 1), \\ \nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\mathsf{T} \mathbf{p}_k &\geq c_2 \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k, && (c_1 < c_2 < 1). \end{aligned}$$

The search direction p_k is often required to satisfy $p_k^\mathsf{T} \nabla f(\mathbf{x}_k) < 0$, in which case it is called a *descent direction*, since the function is guaranteed to decrease in this direction. Generally speaking, choosing a step size α_k satisfying these conditions ensures that we achieve sufficient decrease in the function and also that we do not terminate the search at a point of steep decrease (since then we could achieve even better results by choosing a slightly larger step size). The first condition is known as the *Armijo* condition.

We will discuss methods of finding search directions in future labs. For now, we will discuss one simple algorithm for finding an appropriate step size which satisfies the Armijo conditions.

Backtracking

Finding such a step size satisfying these conditions is not always an easy task, however. One simple approach, known as *backtracking*, starts with an initial step size α, and repeatedly scales it down until the Armijo condition is satisfied. That is, choose $\alpha > 0, \rho \in (0, 1), c \in (0, 1)$, and while

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) > f(\mathbf{x}_k) + c\alpha \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k,$$

re-scale $\alpha = \rho\alpha$. Once the loop terminates, set $\alpha_k = \alpha$. Note that the value $\nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k$ remains fixed for the duration of the backtracking algorithm, and hence need only be calculated once at the beginning.

Problem 5. Implement this backtracking algorithm using the function definition in the spec file. Your function will accept a function, its derivative, a starting point, and the direction p.

Line Search in SciPy

SciPy's `optimize` module contains implementations of various optimization algorithms, including several line search methods. In particular, the module provides a useful routine for calculating a step size satisfying the Wolfe Conditions described above, which is more robust and efficient than our simple backtracking approach. We recommend its use for the remainder of this lab. The function is called `line_search()`, and accepts several arguments. We can typically leave the keyword arguments at their default values, but we do need to pass in the objective function, its gradient, the current point, and the search direction. The following code gives an example of its usage, using the objective function $f(x, y) = x^2 + 4y^2$.

```
>>> import numpy as np
>>> from scipy.optimize import line_search
>>>
>>> def objective(x):
>>>     return x[0]**2 + 4*x[1]**2
>>>
>>> def grad(x):
>>>     return np.array([2*x[0], 8*x[1]])
>>>
>>> x = np.array([1., 3.]) # current point
>>> p = -grad(x)           # current search direction
>>> a = line_search(objective, grad, x, p)[0]
>>> print a
0.125649913345
```

Note that the function returns a tuple of values, the first of which is the step size.

14 Newton and Quasi-Newton Methods

Lab Objective: *Newton's method is generally useful because of its fast convergence properties. However, Newton's method requires the explicit calculation of the second derivative (i.e., the Hessian matrix) at each step, which is computationally costly. Quasi-Newton methods modify Newton's method so that the Hessian does not have to be computed at each step, thus making computations faster. This generally comes at the cost of slower convergence speed, but the increased computation speed can make these methods more effective in many cases.*

Newton's Method

At this point, we have discussed Newton's Method several times. In the n dimensional version, the next step is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - D^2 f(\mathbf{x}_k)^{-1} Df(\mathbf{x}_k)^T \tag{14.1}$$

Problem 1. Write code implementing Newton's method in n Dimensions. Ensure that it takes its Jacobian, and Hessian as arguments and returns the minimizer. Test it on the Rosenbrock function:

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

using two different starting points $(-2, 2)$ and $(10, -10)$.

Broyden's Method

One quasi-Newton method is known as Broyden's method. Broyden's Method is a multidimensional version of the secant method we have discussed previously. Just like the secant method approximates the second derivative of a function by using the first derivatives at nearby points, Broyden's Method uses the Jacobian to update an initial Hessian matrix.

Broyden's Method in n Dimensions

If we have the point $\mathbf{x}_k$ and the Hessian $D^2f(\mathbf{x}_k)$ at that point, we can use the following equation to select our guess for the zero of the function:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - D^2f(\mathbf{x}_k)^{-1}Df(\mathbf{x}_k)^T. \tag{14.2}$$

This is precisely Newton's method. However, since calculating the Hessian at each step is costly, we instead estimate the Hessian (just as we used a secant line to approximate the derivative in the one-dimensional case).

The idea is to construct the best rank-one approximation of the Hessian at each iteration. This approximation, denoted A_{k+1}, must satisfy

$$Df(\mathbf{x}_{k+1}) - Df(\mathbf{x}_k) = (\mathbf{x}_{k+1} - \mathbf{x}_k)^T A_{k+1}. \tag{14.3}$$

In multiple dimensions, this equation will be underdetermined (i.e., many A_{k+1}'s will satisfy the equation). Suppose that we have a previous estimate of the Hessian A_k at the point $\mathbf{x}_k$ (note that for the first iteration, we plug in the starting point to the Hessian as our first approximation). We can then find the best approximation of A_{k+1} that minimizes $\|A_{k+1} - A_k\|$. If we define $\mathbf{y}_k = Df(\mathbf{x}_{k+1})^T - Df(\mathbf{x}_k)^T$ and $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, this requirement is uniquely fulfilled by the following:

$$A_{k+1} = A_k + \frac{\mathbf{y}_k - A_k\mathbf{s}_k}{\|\mathbf{s}_k\|^2}\mathbf{s}_k^T. \tag{14.4}$$

After we have obtained the approximation of the Hessian (Equation 14.4), we can apply Equation 14.2 to find $\mathbf{x}_{k+1}$. We can then repeat this process until we have (presumably) converged to a zero of the function.

Problem 2. Write code implementing Broyden's method. Test it on the function:

$$f(x, y) = e^{x-1} + e^{1-y} + (x - y)^2$$

using starting points $(2, 3)$ and $(3, 2)$.

Note

We can often make Broyden's method faster using the Sherman-Morrison-Woodbury Formula. This formula allows us to efficiently calculate the inverse of a matrix when we add a low rank update to that matrix. After manipulation of the Sherman-Morrison-Woodbury Formula, we obtain the following:

$$A_{k+1}^{-1} = A_k^{-1} + \frac{\mathbf{s}_k - A_k^{-1}\mathbf{y}_k}{\mathbf{s}_k^T A_k^{-1}\mathbf{y}_k}(\mathbf{s}_k^T A_k^{-1})$$

Thus, we can calculate the inverse of the Hessian in the first step of our algorithm and then calculate an update to the inverse at each step using the above formula.

BFGS

To be a descent method, we need a monotonically decreasing sequence of functions. In other words, $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. However, if the Hessian or the approximated Hessian is not positive definite, we cannot expect that $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. A drawback of Broyden's method is that the Hessian approximations are not always positive definite, even if $D^2 f(\mathbf{x}_k) > 0$. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is an adjustment to Broyden's method that maintains a positive-definite Hessian approximation. To do this, it makes a rank-two approximation instead of a rank-one approximation. It uses the same update of $\mathbf{x}_k$ as Broyden's method, but with a different update of A_k:

$$A_{k+1} = A_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{A_k \mathbf{s}_k \mathbf{s}_k^T A_k}{\mathbf{s}_k^T A_k \mathbf{s}_k} \tag{14.5}$$

where $\mathbf{y}_k$ and $\mathbf{s}_k$ have the same definitions as given above.

Problem 3. Implement the BFGS method in a new function. Your function should take in the Jacobian, the Hessian, a starting point, a number of iterations, and a stopping tolerance, and return the approximated root along with the number of iterations it took. Test your implementation with the same function given in Problem 2. Experiment with different starting points. Is BFGS faster than Broyden's method? Are there cases (i.e., different starting points or varying number of iterations) where one implementation is better than the other?

Comparing Newton and Quasi-Newton Methods

Different optimization algorithms are more efficient in different situations. If the Jacobian and Hessian are readily available and the Hessian is easily inverted, the standard Newton's Method is probably the best option. If the Hessian is not available or difficult to compute, then using Broyden's Method may be a better option. In circumstances where computing the inverse of the Hessian is difficult, BFGS will allow us to update the inverted Hessian at each step without repeatedly inverting a matrix.

Problem 4. Compare the performance of Newton, Broyden, and BFGS on the following functions:

$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$$

$$f(x, y) = sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

Output the number of iterations of each algorithm as well as the total time each algorithm takes to run. Calculate the time per iteration for each algorithm and compare. You should see that Newton's Method takes more time per iteration, but that is takes fewer steps than the other algorithms.

The Gauss-Newton Method

Non-linear Least Squares Problems

We now discuss a very important class of problems known as Least Squares problems. These are unconstrained optimization problems that seek to minimize an objective function of the form

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\mathbf{x}),$$

where each $r_i : \mathbb{R}^n \to \mathbb{R}$ is smooth, and $m \geq n$. Such problems arise in many scientific fields, including economics, physics, and statistics. Linear Least Squares problems form an important subclass, and can be solved directly without the need for an iterative method. At present we will focus on the non-linear case, which can be solved with a line search method.

To motivate the problem further, suppose you are given a set of data points, and you have some kind of model for the data. You need to choose particular values for the parameters in your model, and you wish to do so in a way that "best fits" the observed data. What do we mean by "best fit"? We need some way to measure the error between our model and the data set, and then minimize this error. The best fit will correspond to the choice of parameters that minimize the error function.

More formally, suppose we are given the data points $(t_1, y_1), (t_2, y_2), \ldots, (t_m, y_m)$, where $y_i, t_i \in \mathbb{R}$ for $i = 1, \ldots, m$. Let $\phi(\mathbf{x}, \mathbf{t})$ be our model for this data set, where $\mathbf{x}$ is a vector of parameters of the model, and $\mathbf{t} \in \mathbb{R}^n$. We can measure the error at the i-th data point by the value

$$r_i(\mathbf{x}) := \phi(x_i, t_i) - y_i,$$

and by summing the squares of these errors, we obtain our non-linear least squares objective function:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\mathbf{x}).$$

The individual functions r_i that measure the error between the model and the data point are known as *residuals*, and we can aggregate these functions into a *residual vector*

$$\mathbf{r}(\mathbf{x}) := (r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T.$$

The Jacobian of $\mathbf{r}(\mathbf{x})$ can be expressed in terms of the gradients of each r_i as follows:

$$J(\mathbf{x}) = \begin{bmatrix} \nabla r_1(\mathbf{x})^T \\ \nabla r_2(\mathbf{x})^T \\ \vdots \\ \nabla r_m(\mathbf{x})^T \end{bmatrix}$$

You can further verify that

$$\nabla f(\mathbf{x}) = J(\mathbf{x})^T r(\mathbf{x}),$$
$$\nabla^2 f(\mathbf{x}) = J(\mathbf{x})^T J(\mathbf{x}) + \sum_{j=1}^{m} r_j(\mathbf{x}) \nabla^2 r_j(\mathbf{x}).$$

That second term in the formula for $\nabla^2 f$ involves second derivatives and can be problematic to compute. Often in practice, this term is small, either because the residuals themselves are small, or are nearly affine in a neighborhood of the solution and hence the second derivatives are small. The simplest method for solving the nonlinear least squares problem, known as the *Gauss-Newton Method*, exploits this observation, simply ignoring the second term and making the approximation

$$\nabla^2 f(\mathbf{x}) \approx J(\mathbf{x})^T J(\mathbf{x}).$$

The method then proceeds in a manner similar to Newton's Method. In particular, at the k-th iteration, we choose a search direction p_k that solves the linear system

$$J_k^T J_k p_k = -J_k^T r_k.$$

Thus, at each iteration, we find $\mathbf{x}_{k+1}$ as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (J(\mathbf{x}_k)^T J(\mathbf{x}_k))^{-1} J(\mathbf{x}_k)^T \mathbf{r}(\mathbf{x}_k). \tag{14.6}$$

Problem 5. Implement the Gauss-Newton Method. Your function should take the gradient of the objective function Df, the residual vector r, and the Jacobian of the residual vector J. Return the minimizer.

Test your function by using `optimize.leastsq` and comparing your minimizer with Scipy's minimizer. Use the Jacobian function, residual function, and starting point given in the example below.

Let us work through an example of a nonlinear least squares problem. Suppose we have data points generated from a sine function and slightly perturbed by gaussian noise. In Python we can generate such data as follows:

```
>>> t = np.arange(10)
>>> y = 3*np.sin(0.5*t)+ 0.5*np.random.randn(10)
```

Now we write Python functions for our model, the residual vector, the Jacobian, the objective function, and the gradient. The calculations for all of these are straight forward.

```
>>> def model(x, t):
>>>     return x[0]*np.sin(x[1]*t)
>>> def residual(x):
>>>     return model(x, t) - y
>>> def jac(x):
>>>     ans = np.empty((10,2))
>>>     ans[:,0] = np.sin(x[1]*t)
>>>     ans[:,1] = x[0]*t*np.cos(x[1]*t)
>>>     return ans
>>> def objective(x):
>>>     return .5*(residual(x)**2).sum()
>>> def grad(x):
>>>     return jac(x).T.dot(residual(x))
```

By inspecting our data, we might make an initial guess for the parameters $x_0 = (2.5, 0.6)$. We are now ready to use our `gaussNewton` function to find the least squares solution.

```
>>> x0 = np.array([2.5,.6])
>>> x = gaussNewton(jac, residual, x0, niter=10)
```

We can plot everything together to compare our fitted model with the data and the original sine curve from which the data were generated.

```
dom = np.linspace(0,10,100)
plt.plot(t, y, '*')
plt.plot(dom, 3*np.sin(.5*dom), '--')
plt.plot(dom, x[0]*np.sin(x[1]*dom))
plt.show()
```

Non-linear Least Squares in Python

The module `scipy.optimize` also has a method to solve non-linear least squares problem, and it is quite convenient. The function is called `leastsq`, and in its most basic use, you only need to pass in the residual function and starting point as arguments. In the example above, we simply need to execute the following code:

```
>>> from scipy.optimize import leastsq
>>> x2 = leastsq(residual, x0)[0]
```

This should give us the same answer, but much faster.

Problem 6. We have census data giving the population of the United States every ten years since 1790. For convenience, we have entered the data in Python below, so that you may simply copy and paste.

```
>>> #Start with the first 8 decades of data
>>> years1 = np.arange(8)
>>> pop1 = np.array([3.929, 5.308, 7.240, 9.638, 12.866,
>>>                  17.069, 23.192, 31.443])
>>>
>>> #Now consider the first 16 decades
>>> years2 = np.arange(16)
>>> pop2 = np.array([3.929, 5.308, 7.240, 9.638, 12.866,
>>>                  17.069, 23.192, 31.443, 38.558, 50.156,
>>>                  62.948, 75.996, 91.972, 105.711, 122.775,
>>>                  131.669])
```

Consider just the first 8 decades of population data. By plotting the data and having an inclination that population growth tends to be exponential, it is reasonable to hypothesize an exponential model for the population, that is,

$$\phi(x_1, x_2, x_3, t) = x_1 \exp(x_2(t + x_3)).$$

By inspection, find a reasonable initial guess for the parameters (x_1, x_2, x_3) (i.e. $(150, .4, 2.5)$). Write a function for this model in Python, along with the corresponding residual vector, and fit the model using the `leastsq` function. Plot the data against the fitted curve, to see how close you are.

Now consider all 16 decades of data. If you plot your curve from above with this more complete data, you will see that the model is no longer a good fit. Instead, the data suggest a logistic model, which also arises from a differential equations treatment of population growth. Thus, your new model is

$$\phi(x_1, x_2, x_3, t) = \frac{x_1}{1 + \exp(-x_2(t + x_3))}.$$

By inspection, find a reasonable initial guess for the parameters (x_1, x_2, x_3) (i.e. $(150, .4, -15)$). Again, write Python functions for the model and the corresponding residual vector, and fit the model. Plot the data against the fitted curve. It should be a good fit.

15 Gradient Descent Methods

Lab Objective: *Many optimization methods fall under the umbrella of descent algorithms. The idea is to choose an initial guess, identify a direction from this point along with the objective function decreases, and preform a line search to find a new point where the objective function is smaller than at the initial guess.*

In this lab, we implement two major descent algorithms: the method of steepest descent, which uses the gradient of the objective function as the descent direction, and the conjugate gradient algorithm, which uses a conjugate basis as the descent directions. We then apply these techniques to linear and logistic regression problems.

The Method of Steepest Descent

Recall the basic idea of a line search algorithm to optimize: at the kth iteration, choose a search direction $\mathbf{p}_k$ and a step size α_k, then compute $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$. If $f : \mathbb{R}^n \to \mathbb{R}$ is a scalar-valued function, its gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ points in the direction of steepest increase of f. It follows that the opposite direction is the direction of steepest decrease, or *descent*.

The *method of steepest descent* is a line search method that uses $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$ as the search direction. A simple version of the method chooses the step size as follows:

1. At the kth iteration, start with a step size of $\alpha = 1$.
2. Check that $f(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)) < f(\mathbf{x}_k)$ (i.e. check that the function decreases in the search direction), and if it does, set $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)$.
3. If the function does *not* decrease, set $\alpha = \frac{\alpha}{2}$ and check the search direction again. Repeat this step until the function decreases in the search direction.
4. If α is scaled below a given tolerance, terminate the iteration.

Problem 1. Write a function that accepts a convex objective function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an initial guess $x_0 \in \mathbb{R}^n$, a convergence tolerance, and a maximum number of iterations. Use the Method of Steepest Descent to compute the minimizer of f. Continue the algorithm until α is less than the convergence tolerance, or until iterating the maximum number of times. Return the minimizer.

The Conjugate-Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems: depending on the geometry of the objective function, the sequence of points can "zig-zag" back and forth without making significant progress toward the true minimizer. In contrast, the *Conjugate-Gradient algorithm* ensures that the true global minimizer is reached in at most n steps. See Figure 15.1 for an illustration of this contrast.

There are many methods for solving the linear system of equations $A\mathbf{x} = \mathbf{b}$, each with its own pros and cons. The Conjugate-Gradient algorithm is one such method for solving large systems of equations where other methods, such as Cholesky factorization and simple Gaussian elimination, are unsuitable. However, the algorithm works equally well for optimizing convex quadratic functions, and it can even be extended to more general classes of optimization problems.

The type of linear system that Conjugate-Gradient can solve involves a matrix with special structure. Given a symmetric positive-definite $n \times n$ matrix Q and an n-vector b, we wish to find the n-vector x satisfying

$$Qx = b.$$

A unique solution exists because positive-definiteness implies invertibility. For our purposes here, it is useful to recast this problem as an equivalent optimization problem:

$$\min_{\mathbf{x}} f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^\mathsf{T} Q\mathbf{x} - \mathbf{b}^\mathsf{T}\mathbf{x} + \mathbf{c}.$$

Note that $\nabla f(\mathbf{x}) = Q\mathbf{x} - \mathbf{b}$, so that minimizing f is the same as solving the equation

$$0 = \nabla f(\mathbf{x}) = Q\mathbf{x} - \mathbf{b},$$

which is the original linear system.

Conjugacy

Unlike the method of steepest descent, the conjugate gradient algorithm chooses a search direction based off of the matrix Q that is guaranteed to be a direction of descent, though not the direction of greatest descent.

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are said to be *conjugate* with respect to Q if $\mathbf{x}^\mathsf{T} Q\mathbf{y} = 0$. A set of vectors $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m\}$ is said to be conjugate if each pair of vectors are conjugate to each other. Note that if $Q = I$, then conjugacy is the same as orthogonality. Thus, the notion of conjugacy is in some ways a generalization of orthogonality. It turns out that a conjugate set of vectors is linearly independent, and a conjugate basis–which can be constructed in a manner analogous to the Gram-Schmidt orthogonalization process–can be used to diagonalize the matrix Q. These are some of the theoretical reasons behind the effectiveness of the Conjugate-Gradient algorithm.

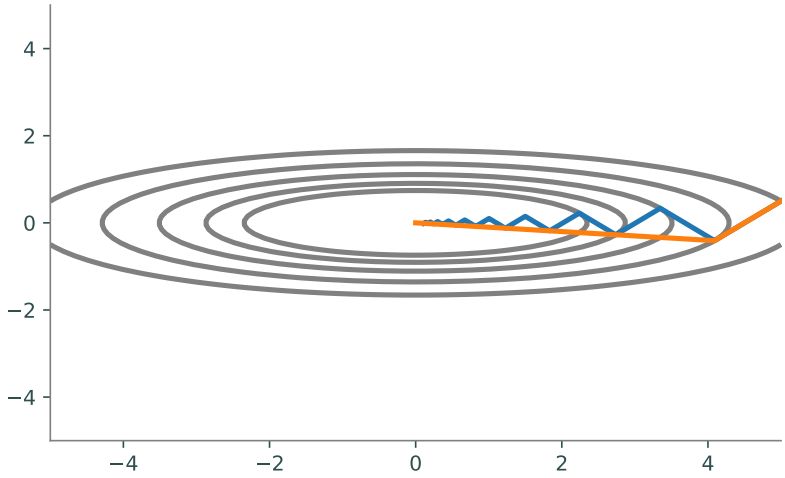


Figure 15.1: Paths traced by Steepest Descent (blue) and Conjugate-Gradient (green). Notice the zig-zagging nature of the Steepest Descent path, as opposed to the direct Conjugate-Gradient path, which finds the minimizer in 2 steps.

The Algorithm

If we are given a set of n Q-conjugate vectors, we can simply choose these as our direction vectors and follow the basic descent algorithm. Convergence to the minimizer in at most n steps is guaranteed because each iteration in the algorithm minimizes the objective function over an expanding affine subspace of dimension equal to the iteration number. Thus, at the n-th iteration, we have minimized the function over all of $\mathbb{R}^n$.

Unfortunately, we are not often given a set of conjugate vectors in advance, so how do we produce such a set? As mentioned earlier, a Gram-Schmidt process could be used, and the set of eigenvectors also works, but both of these options are computationally expensive. Built into the algorithm is a way to determine a new conjugate direction based only on the previous direction, which means less memory usage and faster computation. We have stated the details of Conjugate-Gradient in Algorithm 15.1.

Algorithm 15.1 Conjugate-Gradient Algorithm

procedure CONJUGATE-GRADIENT ALGORITHM
  Choose initial point x_0.
  $r_0 \leftarrow Qx_0 - b, d_0 \leftarrow -r_0, k \leftarrow 0$.
  while $r_k \neq 0$ **do**
    $\alpha_k \leftarrow \frac{r_k^T r_k}{d_k^T Q d_k}$.
    $x_{k+1} \leftarrow x_k + \alpha_k d_k$.
    $r_{k+1} \leftarrow r_k + \alpha_k Q d_k$.
    $\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$.
    $d_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} d_k$.
    $k \leftarrow k + 1$.

Note that the points x_i are the successive approximations to the minimizer, the vectors d_i are the conjugate descent directions, and the vectors r_i, which actually correspond to the steepest descent directions, are used in determining the conjugate directions. The constants α_i and β_i are used, respectively, in the line search, and in ensuring the Q-conjugacy of the descent directions.

The most numerically expensive computation in the algorithm is matrix-vector multiplication. Notice, however, that each iteration of the algorithm only requires one distinct matrix-vector multiplication, Qd_k. The rest of the operations are simply vector-vector multiplication, addition, and scalar multiplication. This makes for a very fast algorithm. As noted earlier, Conjugate-Gradient is especially preferred when Q is large and sparse. In this case, it may be possible to design a specialized sub-routine that performs matrix-vector multiplication by Q, by taking advantage of its sparseness. Doing so may lead to further speed-ups in the overall algorithm.

We now have an algorithm that can solve certain $n \times n$ linear systems and minimize quadratic functions on $\mathbb{R}^n$ in at most n steps, and sometimes fewer, depending on the spectrum of the matrix Q. Further improvements on convergence may be obtained by preconditioning the matrix, but we do not go into detail here.

Problem 2. Implement the basic Conjugate-Gradient algorithm presented above. Write a function `conjugateGradient()` that accepts a vector b, an initial guess x_0, a symmetric positive-definite matrix Q, and a default tolerance of .0001 as inputs. Continue the algorithm until $\|r_k\|$ is less than the tolerance. Return the solution x^* to the linear system $Qx = b$.

Example

We now work through an example that demonstrates the usage of the Conjugate-Gradient algorithm. We assume that we have already written the specified function in the above problem.

We must first generate a symmetric positive-definite matrix Q. This can be done by generating a random matrix A and setting $Q = A^T A$. So long as A is of full column rank, the matrix Q will be symmetric positive-definite.

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

>>> # initialize the desired dimension of the space
>>> n = 10

>>> # generate Q, b
>>> A = np.random.random((n,n))
>>> Q = A.T.dot(A)
>>> b = np.random.random(n)
```

At this point, check to make sure that Q is nonsingular by examining its determinant (use `scipy.linalg.det()`). Provided that the determinant is nonzero, we proceed by writing a function that performs matrix-vector multiplication by Q (we will not take advantage of sparseness just now), randomly selecting a starting point (Conjugate-Gradient is not sensitive to the location of the starting point), obtaining the answer using our function, and checking it with the answer obtained by `scipy.linalg.solve()`.

```
>>> # generate random starting point
>>> x0 = np.random.random(n)

>>> # find the solution
>>> x = conjugateGradient(b, x0, mult)

>>> # compare to the answer obtained by SciPy
>>> print np.allclose(x, la.solve(Q,b))
```

The output of the print statement should be `True`.

Time the performance of your algorithm and of `scipy.linalg.solve()` on inputs of size 100.

Application: Least Squares and Linear Regression

The Conjugate-Gradient method can be used to solve linear least squares problems, which are ubiquitous in applied science. Recall that a least squares problem can be formulated as an optimization problem:

$$\min_x \|Ax - b\|_2,$$

where A is an $m \times n$ matrix with full column rank, $x \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$. The solution can be calculated analytically, and is given by

$$x^* = (A^T A)^{-1} A^T b,$$

or in other words, the minimizer solves the linear system

$$A^T Ax = A^T b.$$

Since A has full column rank, we know that A^TA is an $n \times n$ matrix of rank n, which means it is invertible. We can therefore conclude that A^TA is symmetric positive-definite, so we may use Conjugate-Gradient to solve the linear system and obtain the least squares solution.

Linear least squares is the mathematical underpinning of linear regression, which is a very common technique in many scientific fields. In a typical linear regression problem, we have 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

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_n x_{i,n} + \epsilon_i$$

for $i = 1, 2, \dots, m$. The real numbers $\beta_0, \dots, \beta_n$ are known as the parameters of the model, and the ϵ_i are independent normally-distributed error terms. Our task 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

$$b = [y_1, \dots, y_m]^T,$$

$$A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix},$$

and

$$x = [\beta_0, \beta_1, \dots, \beta_n]^T.$$

Now use Conjugate-Gradient to solve the system

$$A^TAx = A^Tb.$$

The solution $x^* = [\beta_0^*, \beta_1^*, \dots, \beta_n^*]^T$ gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data. See Figure 15.2.

Problem 3. Using your Conjugate-Gradient function, solve the linear regression problem specified by the data contained in the file `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 the function `numpy.loadtxt()` to load in the data. Report your solution.

Non-linear Conjugate-Gradient Algorithms

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. There are multiple ways to modify the algorithm, and they all involve getting rid of Q, since there is no such Q for non-quadratic functions. Generally speaking, we need to find new formulas for α_k, r_k, and β_k.

The scalar α_k is simply the result of performing a line-search in the given direction d_k, so we may define

$$\alpha_k = \arg\min_x f(x_k + \alpha d_k).$$

The vector r_k in the original algorithm was really just the gradient of the objective function, and so we may define

$$r_k = \nabla f(x_k).$$

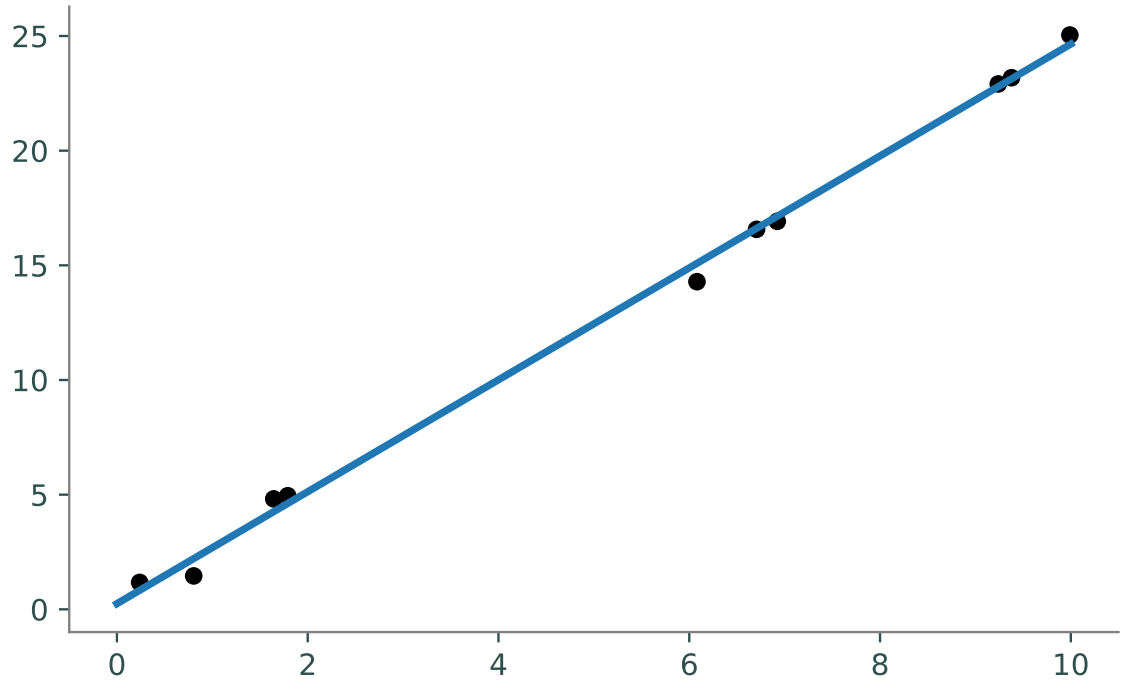


Figure 15.2: Solving the Linear Regression problem results in a best-fit hyperplane.

There are various ways to define the constants β_k in this more general setting, and the right choice will depend on the nature of the objective function. A well-known formula, due to Fletcher and Reeves, is

$$\beta_{k+1} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}.$$

Making these adjustments is not difficult, but we will opt instead to use built-in functions in Python. In particular, the SciPy module `scipy.optimize` provides a function `fmin_cg()`, which uses a non-linear Conjugate-Gradient method to minimize general functions. Using this function is easy – we only need to pass to it the objective function and an initial guess.

Application: Logistic Regression

Logistic regression is an important technique in statistical analysis and classification. The core problem in logistic regression involves an optimization that we can tackle using nonlinear Conjugate-Gradient.

As in linear regression, we have a set of data points y_i together with predictor variables $x_{i,1}, x_{i,2}, \ldots, x_{i,n}$ for $i = 1, \ldots, m$. However, the y_i are binary data points – that is, they are either 0 or 1. Furthermore, instead of having a linear relationship between the data points and the response variables, we assume the following probabilistic relationship:

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

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers $\beta_0, \beta_1, \ldots, \beta_n$. Observe that we have $p_i \in (0,1)$ regardless of the values of the predictor variables and parameters.

The probability of observing the data points y_i under this model, assuming they are independent, is given by the expression

$$\prod_{i=1}^{m} p_i^{y_i}(1-p_i)^{1-y_i}.$$

We seek to choose the parameters $\beta_0, \ldots, \beta_n$ that maximize this probability. To this end, define the *likelihood function* $L : \mathbb{R}^{n+1} \to \mathbb{R}$ by

$$L(\beta_0, \ldots, \beta_n) = \prod_{i=1}^{m} p_i^{y_i}(1-p_i)^{1-y_i}.$$

We can now state our core problem as follows:

$$\max_{(\beta_0,\ldots,\beta_n)} L(\beta_0, \ldots, \beta_n).$$

Maximizing this function can be problematic for numerical reasons. By taking the logarithm of the likelihood, we have a more suitable objective function whose maximizer agrees with that of the original likelihood function, since the logarithm is strictly monotone increasing. Thus, we define the *log-likelihood function* $l : \mathbb{R}^{n+1} \to \mathbb{R}$ by $l = \log \circ L$.

Finally, we multiply by -1 to turn our problem into minimization. The final statement of the problem is:

$$\min_{(\beta_0,\ldots,\beta_n)} -l(\beta_0, \ldots, \beta_n).$$

A few lines of calculation reveal that

$$l(\beta_0, \ldots, \beta_n) = -\sum_{i=1}^{m} \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))) + \sum_{i=1}^{m} y_i(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}).$$

The values for the parameters that we obtain are known collectively as the *maximum likelihood estimate*.

Let's work through a simple example. We will deal with just one predictor variable, and therefore two parameters. The data is given in Table 15.1. This is obviously just toy data with no meaning, but one can think of the y_i data points as indicating, for example, the presence of absence of a particular disease in subject i, with x_i being the subject's weight, or age, or something of the sort.

In the code below we initialize our data.

```
>>> y = np.array([0, 0, 0, 0, 1, 0, 1, 0, 1, 1])
>>> x = np.ones((10, 2))
>>> x[:,1] = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Although we have just one predictor variable, we initialized `x` with two columns, the first of which consists entirely of ones, and the second of which contains the values of the predictor variable. This extra column of ones corresponds to the parameter β_0, which, as you will note, is not multiplied by any of the predictor variables in the log-likelihood function.

We next need to write a Python function that returns the value of our objective function for any value of the parameters, (β_0, β_1).

Table 15.1: Data for Logistic Regression Example

y	x
0	1
0	2
0	3
0	4
1	5
0	6
1	7
0	8
1	9
1	10

```
>>> def objective(b):
...     #Return -1*l(b[0], b[1]), where l is the log likelihood.
...    return (np.log(1+np.exp(x.dot(b))) - y*(x.dot(b))).sum()
```

Finally, we minimize the objective function using `fmin_cg()`.

```
>>> guess = np.array([1., 1.])
>>> b = fmin_cg(objective, guess)
Optimization terminated successfully.
         Current function value: 4.310122
         Iterations: 13
         Function evaluations: 128
         Gradient evaluations: 32
>>> print b
[-4.35776886  0.66220658]
```

We can visualize our answer by plotting the data together with the function

$$\phi(x) = \frac{1}{1 + \exp(-\beta_0 - \beta_1 x)},$$

using the values β_0, β_1 that we obtained from the minimization.

```
>>> dom = np.linspace(0, 11, 100)
>>> plt.plot(x, y, 'o')
>>> plt.plot(dom, 1./(1+np.exp(-b[0]-b[1]*dom)))
>>> plt.show()
```

Using this procedure, we obtain the plot in Figure 15.3. Note that the graph of ϕ, known as a *sigmoidal curve*, gives the probability of y taking the value 1 at a particular value of x. Observe that as x increases, this probability approaches 1. This is reflected in the data.

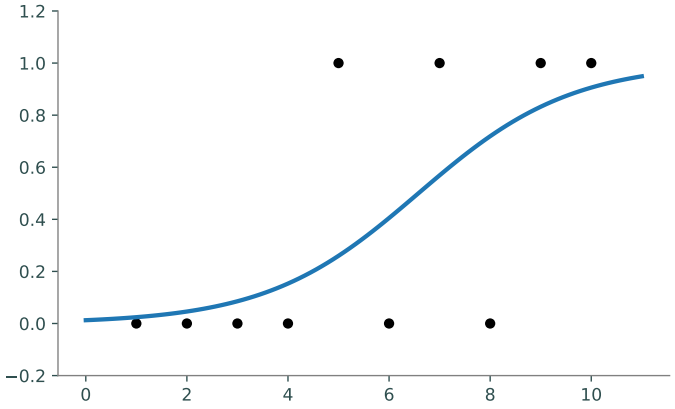


Figure 15.3: Data from the logistic regression example together with the calculated sigmoidal curve.

Problem 4. Following the example given above, find the maximum likelihood estimate of the parameters for the logistic regression data in the file `logregression.txt`. This is a whitespace-delimited text file formatted so that the i-th row consists of $y_i, x_{i,1}, x_{i,2}, x_{i,3}$. Since there are three predictor variables, there are four parameters in the model. Report the calculated values.

You should be able to use much of the code above unchanged. In particular, the function `objective()` does not need any changes. You simply need to set your variables `y` and `x` appropriately, and choose a new initial guess (an array of length four). Note that `x` should be an $m \times 4$ array whose first column consists entirely of ones, whose second column contains the values in the second column of the data file, and so forth.

Logistic regression can become a bit more tricky when some of the predictor variables take on binary or categorical values. In such situations, the data requires a bit of pre-processing before running the minimization.

The values of the parameters that we obtain can be useful in analyzing relationships between the predictor variables and the y_i data points. They can also be used to classify or predict values of new data points.

16 SQL and Relational Databases

Lab Objective: *Understand concepts of a relational database and the fundamentals of the SQL language via SQLite.*

When working with large amounts of data, it is important to be able to quickly find and retrieve interesting information. Fortunately, there is a way to handle such massive amounts of data in a reasonably efficient way: a database. A database is simply a structured repository of data, and it allows us to store and retrieve information very quickly. It is managed by a *database management system*, or DBMS. The DBMS is software that allows users to interact directly with the database.

Relational Databases

A *relational database* is a paradigm for organizing data inside of a database. In this paradigm, the data is broken down into tuples of information. These tuples are then grouped into tables, or *relations*, each of which is simply a set of tuples. Each table has a *schema* that defines the attributes of the tuples within the table. If we fix an order to the attributes in the schema, we can think of each attribute as a column of the table, and each tuple as a row of the table. See Figure 16.1 for an illustration of these ideas.

As an example, suppose we have demographic data for a large number of individuals. If we are interested in the gender and age of the individuals, we might make a table with schema (Name, Gender, Age). This table would consist of several 3-tuples, such as (Jane Doe, F, 20). Alternatively, we can view this table as having three columns and as many rows as there are individuals within our data set. We might also create a table with schema (Name, Employment Status, Income, Education).

In the relational paradigm, there must be at least one attribute in each schema that can act as a *primary key*. This can uniquely identify each tuple of the table. It is common to use an ID number or other such unique information for the primary key. In our example above, the “Name" attribute acted as a primary key. However, this attribute only works as a primary key provided no two individuals within the data set have the same name.

One important feature of a database is the *transaction*, which is a conceptual protocol for interacting with the database. Most relational databases are transactional databases. The best way to conceptualize this is imagine that your database is like a bank. Your connection to the database is analogous to the bank teller. When you make one or more deposits and withdrawals, you are making a transaction. A database transaction should have certain properties to protect the integrity of the data. These properties are described in detail in `en.wikipedia.org/wiki/ACID`

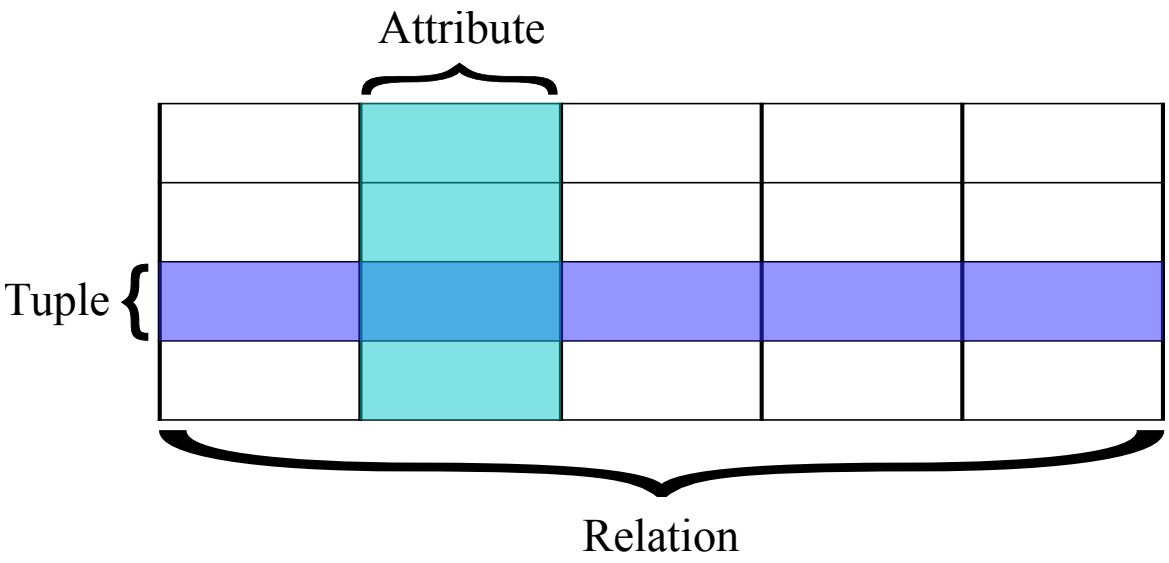


Figure 16.1: Elements of a relation.

Introduction to SQL

Most common DBMSs use a variant of the SQL language to interact with the database. SQL is an acronym for *Structured Query Language*, and may be pronounced like the word "sequel" or by saying the letters "s", "q", and "l" separately. While SQL is not generally portable across different DBMSs, we will focus on the parts of SQL that are relatively common. In particular, we will base our discussion on the SQLite database management system, a very popular DBMS.

SQL consists of blocks of code called statements. Each statement is made up of clauses which may or may not require predicates. Predicates specify conditions that can limit the effect of a clause.

> **Note**
>
> SQL commands are often written in all caps to help distinguish them from the other parts of the query. It is only a matter of style. SQLite, along with most other database managers, is case insensitive. In Python's SQL interface, the semicolon is also not needed. However, most other database systems will require it, so it's a good idea to conform in Python.

Let's look at an example SQL statement:

```
SELECT * FROM table WHERE id=3+1 AND name='Bob';
```

This statement includes a SELECT clause and a WHERE clause. The WHERE clause contains two predicates: `id=3+1` and `name='Bob'`. These two predicates limit the effect of the SELECT clause because any resulting tuples in the table must satisfy both conditions. This entire statement is classified as a query since it does not modify the database in any way.

SQL has several classes of statements. The two main classes we will cover in this lab are schema (Table 16.1) and data manipulation (Table 16.2). We will give you a simplified description of each command and its syntax. You are encouraged to look up the full syntax outside of this lab.

Keyword	Syntax
CREATE TABLE	CREATE TABLE <table> (<col1> <type>, <col2> <type>, ...);
DROP TABLE	DROP TABLE <table>;
CREATE INDEX	CREATE INDEX <name> ON <table> (<col>);
DROP INDEX	DROP INDEX <name>;

Table 16.1: The SQL Schema commands

Keyword	Syntax
INSERT INTO	INSERT INTO <table> <attributes> VALUES (<value1>, <value2>, ...);
UPDATE	UPDATE <table> SET (<col1>=<val1>, <col2>=<val2>, ...) WHERE <condition>;
DELETE	DELETE FROM <table> WHERE <condition>;
SELECT	SELECT <attributes> FROM <table> WHERE <condition>;

Table 16.2: The SQL Data Manipulation commands

SQL in Python

Python has built-in support for SQLite databases using the standard library. Let's open a database called `test1`.

```
import sqlite3 as sql
db = sql.connect("test1")
```

The `connect()` function is used to connect to a database. If it does not already exist, then a new database will be created using the string passed as the argument for the name. The new database was created as a file in the current working directory.

Ending the SQL Session

Once we are finished performing SQL statements and interacting with the database, we need to commit our changes and safely close the connection to the database. This can be done by calling methods on the database connection object.

```
db.commit()     #save changes made in the transaction
db.close()      #safely close the database
```

A database connection is automatically closed in Python when the connection object is garbage-collected. However, it is nice to be safe and explicit in closing a database connection using the `close()` method.

Cursor

To execute SQL commands, we need to get a cursor object from the database.

```
cur = db.cursor()
```

Method	Description
`execute`	Execute a single SQL statement
`executemany`	Execute a single SQL statement over a sequence
`executescript`	Execute a SQL script (multiple SQL commands)
`close`	Closes the cursor object

Table 16.3: Cursor object methods

Python Type	SQLite Type
`None`	`NULL`
`int`	`INTEGER`
`long`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`buffer`	`BLOB`

Table 16.4: Python and SQLite types mapping

The cursor object has several useful methods (Table 16.3). Through the cursor, we will execute all of our SQL commands.

Before creating a table, we need to understand how SQLite stores information in a database. SQLite uses five native data types (a simplified system from other SQL database managers). Table 16.4, gives a mapping between Python and SQLite native types.

Creating and Dropping Tables

Let's create a table.

```
cur.execute('CREATE TABLE StudentInformation (StudentID INTEGER NOT NULL, Name ↩
    TEXT, MajorCode INTEGER);')
```

This will create the empty table in Table 16.5.

The arguments in parentheses are the column names followed by the data type that entries in that column will be, and together these form the schema of the table. The `INTEGER` data type in SQLite is a 1, 2, 3, 4, 6, or 8 byte integer depending on the value. The `NOT NULL` command is a *constraint* on the StudentID column. It requires that all records in the table have a student ID.

> **Note**
>
> SQLite does not enforce types on columns. Just like Python, SQLite is dynamically typed. However, most other database systems strictly enforce column types. It is a good idea to conform to the column types specified in the schema.

StudentID	Name	MajorCode

Table 16.5: StudentInformation

Note that each command we execute returns the same cursor object. This object is equipped with a method that allows us to look at any results of the previous command. The result is formally known as the *result set*. If you use `cur.fetchall()`, you will see an empty list. That is because the create table command does not return a result set.

Now we want to build a relation between students, the courses they've had, and their grades in those courses.

```
cur.execute('CREATE TABLE StudentGrades (StudentID INT NOT NULL, CourseID INT, ↩
    Grade TEXT);')
```

Problem 1. In this problem, you will create two new tables in the database "sql1". The first table will be called MajorInfo and have a column called MajorID and MajorName. MajorID is an integer and MajorName is a string.

The second table will be called CourseInfo and have columns called CourseID and CourseName, also integers and strings, respectively.

Hint: In order to view information about the columns of the table, run the following command:

```
cur.execute("PRAGMA table_info('table_name')")
for info in cur:
    print info
```

For each column, this command will output (ID, name, type, notnull, default value, primary key). Also, don't forget to commit and close your database.

We can also destroy tables using the `DROP TABLE` command.

```
cur.execute("CREATE TABLE test_table (id INT, name TEXT);")
```

We can delete the table by dropping it.

```
cur.execute("DROP TABLE test_table;")
```

If a table doesn't exist, an exception will be raised. We can tell the database to drop the table only if it really exists by using `DROP TABLE IF EXISTS test_table;`.

Inserting and Removing Data

Let's insert some data into our new tables. We can add rows to tables using the `INSERT INTO` command.

```
cur.execute("INSERT INTO StudentInformation VALUES(55, 'John Smith', 2);")
```

After running this statement, we will have the table in Table 16.6.

Note that SQLite will assume that values match sequentially with the schema of the table. We can also specify the schema of the table to use in the mapping of the values.

StudentID	Name	MajorCode
55	John Smith	2

Table 16.6: StudentInformation

```
cur.execute("INSERT INTO StudentInformation(MajorCode, Name, StudentID)  VALUES↩
    (55, 'John Smith', 2);")
```

This will map the value 55 to MajorCode and the value 2 to StudentID. This may be useful sometimes.

It can quickly become tedious to insert large amounts of data into a table, one row at a time. We can automate the process somewhat by using the `executemany` method of the cursor object. To insert several rows into a table using a single command, we can do the following:

```
cur.executemany("INSERT INTO StudentInformation VALUES (?, ?, ?, ?);", rows)
```

In the code above, we assume that `rows` is a Python list of tuples, each tuple containing the data for one row.

We may remove rows from a table using the `DELETE FROM` command.

```
cur.execute("DELETE FROM StudentInformation WHERE MajorCode=55;")
```

Achtung!

Never use Python's string operations to construct a SQL query. It is extremely insecure and is an easy target for a well known type of database called a SQL injection attack.

Parameter substitution can be used to construct dynamic queries. In the simplest way, it involves using a '?' character whenever you want to use a value and providing a sequence of values as a second argument to `execute()`.

```
statement = "INSERT INTO StudentInformation VALUES(?, ?, ?, ?);"
values = (55, 'John Smith', 372897382, 2)
cur.execute(statement, values)
```

Problem 2. The ICD is a large collection of codes used to classify any diagnosis that a doctor would make. When someone goes to the hospital or doctors office, their visit will be recorded using these codes. Insurance companies, the government, and researchers find this data useful. The data file provided to you, `icd9.csv`, has simulated health histories for one million persons. Each line has columns for identification number, gender, and age, followed by ICD-9 codes of various quantities. Note that the codes for each individual are written in a single string, each code separated by semicolons. Create a new database with a single table to store all the simulated data. Call the database "sql2" and the table "ICD." Your table should have four columns, one each for id number, gender, age, and codes.

Because of the volume of data, it is highly recommended you use the `executemany()` method of the cursor. It will be about twice as fast as using an `execute()` for each line of the CSV file. Recall the `csv` package in Python. To read a CSV file into a list of tuples, where each tuple consists of the delimited values of a particular line in the file, one can use the following code as a guideline:

```
import csv
with open('filename', 'rb') as csvfile:
    rows = [row for row in csv.reader(csvfile, delimiter=',')]
```

Hint: Don't forget to commit and close your database.

Problem 3. Create the following tables in the same database you created in Problem 1 ("sql1"). You may do so however you think is best.

StudentID	Name	MajorCode
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	1
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	3
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	3
622665098	Sammy Burke	2

Table 16.7: StudentInformation

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 16.8: MajorInfo

StudentID	ClassID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

Table 16.9: StudentGrades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 16.10: CourseInfo

Updating Rows of a Table

We can modify records in a table by using the `UPDATE` command.

```
cur.execute("UPDATE StudentInformation SET MajorCode=2, StudentID=55, Name='↩
    Jonathan Smith' WHERE StudentID=2;")
```

NOTE

When updating a table, having a sufficient `WHERE` clause is essential. Any record that matches the criteria will be modified. If we omitted the `WHERE` clause, every record in the table would be set to the values given in the example.

Adding Columns to a Table

After you have created a table, you can add more columns to the table by using the `ALTER TABLE` command. For example, if we wanted to add the column "age" into our students table. We run this command.

```
cur.execute("ALTER TABLE StudentInformation ADD COLUMN age INTEGER;")
```

Selecting Data From Tables

The process of retrieving data from a table in a database is accomplished by the `SELECT` statement. The `SELECT` statement can be thought of as a very high level set description. For example, to view the contents of an entire table, we simply need to unconditionally select its contents.

```
SELECT * FROM students;
```

This is equivalent to the following set (where x is a row).

$$\{x : x \in \text{classes}\}$$

We can also select specific columns.

```
SELECT StudentID, Name FROM students;
```

Or we can impose conditions on the selected rows.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1;
```

This query results in the following table (Table 16.11) where the contents are all the students that are math majors.

You can also further refine which rows you select by using the `AND` or `OR` commands. These commands will connect expressions. For example, to get the math and science majors. One could run the command.

```
SELECT StudentID, Name FROM students WHERE MajorCode=1 OR MajorCode=2;
```

Select statements return a *result set*. This is an iterable object. Each row in the object is represented as a tuple of values.

```
cur.execute('SELECT StudentID, Name FROM students WHERE MajorCode=1;')
for student in cur:
    print student
```

StudentID	Name
401767594	Michelle Fernandez
678665086	Gilbert Chapman
341324754	Cassandra Holland

Table 16.11: Selected students who are math majors.

Method	Description
`fetchone()`	Return a single row from the result set
`fetchmany(n)`	Return the next n rows from the result set
`fetchall()`	Return the entire result set

Table 16.12: Fetch methods of a cursor.

We can also use the fetch methods of the returned cursor to extract rows from the result set (Table 16.12).

Problem 4. From the ICD9 table you created in Problem 2, how many men between the ages of 25 and 35 are there? How many women between those same ages? Return your answers as a tuple.

When an Error Occurs

It is important to be able to recover from errors gracefully, especially when working a database. Data integrity in a database is often a critical need. When an error occurs, we need to undo the changes that triggered the error. Fortunately, `sqlite3` reports a variety of errors. These errors and when they are raised is explained in PEP249 (`http://legacy.python.org/dev/peps/pep-0249/`).

Error The base class for errors thrown by `sqlite3`. All other errors inherit from this class. Catching this error will catch any error raised.

InterfaceError Raised when there is a problem with the interface to the database rather than the database itself.

DatabaseError Raised when there is an error with the database itself.

DataError Subclass of DatabaseError. Raised when there are errors in the processed data (division by zero, value out of range, etc.).

OperationalError Subclass of DatabaseError. Raised for errors related to the database that are not the fault of the programmer. For example, an unexpected disconnect, failure to process a transaction, a memory allocation error during a transaction, etc.

IntegrityError Subclass of DatabaseError. Raised when the relational integrity of the database is compromised.

InternalError Subclass of DatabaseError. Raised when there is an internal error such as an invalid cursor, out-of-sync transaction, etc.

ProgrammingError Subclass of DatabaseError. Raised for programming errors.

NotSupportedError Subclass of DatabaseError. Raised when a method is called that is not supported by the database.

The way to gracefully recover from errors is to catch them and handle them accordingly. For example, if any error occurs, with the interface or the database, we immediately rollback the transaction. If no error occurs, commit. We could use if-statements or we could use a try-except block.

```
try:
    <code>
    db.commit()
except sql.Error:
    db.rollback()
```

Note that rolling back is not needed if we are just performing queries. If we don't change any of the data in the database, there is no need to roll anything back. However, even with queries, there is the potential for errors. You must design your code to handle these errors gracefully.

17 SQL 2 (The SQL Sequel)

Lab Objective: *Learn more of the advanced and specialized features of SQL.*

Database Normalization

Normalizing a database is the process of organizing tables and columns to minimize the amount of redundant information in the database. For example, a non-normalized database might have a table that stores customer contact information and a table that contains all of the products a company has sold. However, they might want to track who buys what products in case they need to contact them later. To do so, they store all the contact information of a particular buyer along with every item they purchased. Now, two tables store the customer contact information. If we needed to update a customer's phone number, we have to update two tables. While that may not be bad for small databases, larger databases would be near impossible to update correctly. The idea of normalizing a database allows us to store all customer contact information in one place in the database. All other tables that might need a customer's name, phone number, or address would reference the contact information table. When an update needs to be performed, we only need to update the contact information table. Then any table that references this information is also automatically up to date.

To properly normalize a database, we need to discuss the types of relations tables might have.

One to One

This is the simplest relation to model. A single table can be used to express this relation. The relation is between one record and at most one other record. An example of this relationship is an employee and their organization. One employee works at one organization. Another example would be that a driver's license belongs to only one person.

One to Many

This relationship and its inverse must be modeled with at least two tables. The general approach is to use a unique ID. Note that a relationship that appears one to one may actually be a one to many relationship. Many people will, therefore, use the same unique ID approach on one to one relationships too in the case it turns out to be a one to many relationship. An example of a one to many relationship would be between an department and its employees. The department would receive a unique ID and then each employee in that department would be tagged with that ID.

StudentID	Name	MajorCode	MinorCode
401767594	Michelle Fernandez	1	NULL
678665086	Gilbert Chapman	NULL	NULL
553725811	Roberta Cook	2	1
886308195	Rene Cross	3	1
103066521	Cameron Kim	4	2
821568627	Mercedes Hall	NULL	3
206208438	Kristopher Tran	2	4
341324754	Cassandra Holland	1	NULL
262019426	Alfonso Phelps	NULL	NULL
622665098	Sammy Burke	2	3

Table 17.1: students

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 17.2: fields

Many to Many

This relationship requires at least three tables. A many to many relationship can be visualized as two, separate one to many relationships. The records in each of the two tables receive a unique ID. A third table then serves as a map between IDs of table to IDs of the other table. An example of a many to many relationship is doctors and patients. One doctor can have several patients and one patient can have several doctors.

For the rest of the lab, we will be using the following tables: 17.1, 17.2, 17.3, and 17.4.

Problem 1. Classify the relations between the various records in these tables: 17.1, 17.2, 17.3, and 17.4.

Classify each relation as either one to one, one to many, or many to many. Identify the tables used in each relationship.

Note

There are instances where you would not want a completely normalized database. Whether to normalize your database depends on your specific needs. Usually, though, the decision to denormalize a database is a last-resort attempt to improve performance.

StudentID	ClassID	Grade
401767594	4	C
401767594	3	B-
678665086	4	NULL
678665086	3	A+
553725811	2	C
678665086	1	NULL
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	NULL
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	NULL
103066521	4	A
262019426	2	B
262019426	3	NULL
622665098	1	A
622665098	2	A-

Table 17.3: grades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 17.4: classes

Joining Tables

We can use SQL to join two or more tables together for a query. This is a very powerful tool. SQLite supports three types of standard table joins.

Joining tables is a common practice to collect data from different parts of the database into a single table. Joins are absolutely essential in a normalized database since data is split between multiple tables.

Inner Join

This is often the default join operation in SQL. An inner join can be depicted as an intersection of two or more tables. When performing an inner join on tables, the result will only be those records that match across all tables. For example, this join will select all the students' names along with their major as long as the students.majorcode matches the majors.id. If the students.majorcode is missing or null, then they won't be selected.

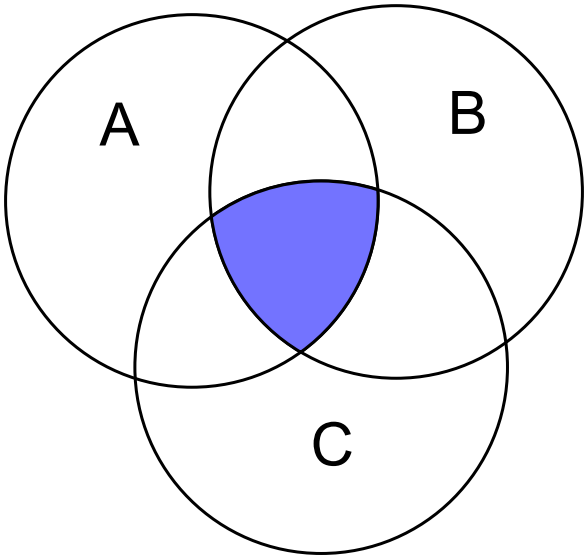


Figure 17.1: An inner joining of tables A, B, and C.

students.name	majors.name
Michelle Fernandez	Math
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Kristopher Tran	Science
Cassandra Holland	Math
Sammy Burke	Science

Table 17.5: An inner join of students and majors

```
SELECT students.name, majors.name FROM students JOIN majors ON students.↩
    majorcode=majors.id;
```

An inner join is equivalent to the following pseudo-loop in Python

```
for row_s in students:
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
```

Left Outer Join

A left outer join will return all relations from the left table even if they don't match any relation on the joined tables. An illustration of a left outer join is given in figure 17.2.

A Pythonesque loop that illustrates how to perform a left outer join is

```
for row_s in students:
```

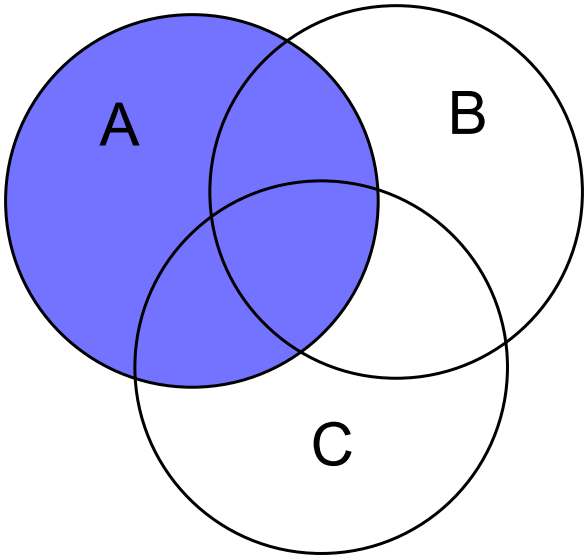


Figure 17.2: A left outer table join of A with tables B and C.

students.name	majors.name
Michelle Fernandez	Math
Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Mercedes Hall	None
Kristopher Tran	Science
Cassandra Holland	Math
Alfonso Phelps	None
Sammy Burke	Science

Table 17.6: A left outer join of students and majors

```
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
        else:
            yield columns(row_s)
```

The following left outer join will result in the table shown in table 17.6.

```
SELECT students.name, majors.name FROM students LEFT OUTER JOIN majors ON ↩
    students.majorcode=majors.id;
```

Function	Description
MIN()	Retrieve the smallest numeric value of a column
MAX()	Retrieve the largest numeric value of a column
SUM()	Sum the numeric values of a column
AVG()	Retrieve the average numeric value of the column
COUNT()	Retrieve the total number of matching records in a column

Table 17.7: SQL aggregation functions

Cross Join

Essentially a Cartesian product of tables. Care must be taken when using cross join because of the size of the joined table. A cross join should only be used on small tables. It matches each relation in one table with every other possible combination of relations in the joined tables. For example, if you were to run a cross join on the above join. You would get every student matched with every major. This doesn't really make much sense but can be useful for other applications. Use with caution.

Advanced Selections

Aggregate functions are useful for summarizing the data in a column. The functions can be found in 17.7.

We can count the number of students by executing the following SQL statement.

```
SELECT COUNT(*) FROM students;
```

Ordering and Grouping Relations

The `ORDER BY` keyword can be used to sort the result set by columns. We can sort in ascending order or descending order.

```
SELECT name FROM students ORDER BY name ASC;
SELECT name FROM students ORDER BY name DESC;
```

Another useful SQL keyword is the `GROUP BY` keyword. It is used along with an aggregating function to group the result set by columns.

```
SELECT grade, COUNT(studentid) FROM grades GROUP BY grade;
```

The result set is given in table 17.8.

Problem 2. Create a database containing tables 17.1, 17.2, 17.3, and 17.4. Write a SQL query to count how many students belong to each major, including students that don't have a major. Sort your results in ascending order by name. Your result set should be table 17.9

grade	COUNT(studentid)
None	5
A	4
A+	1
A-	1
B	2
B-	1
C	3
C+	1
C-	1
D	1
D-	1

Table 17.8: Grouping of students by grade.

None	3
Art	1
Math	2
Science	3
Writing	1

Table 17.9: Result set

Another important keyword is the `HAVING` keyword. This is necessary because the `WHERE` clause does not support aggregate functions. A `HAVING` clause requires a `GROUP BY` clause. The following will not work.

```
SELECT grade FROM grades GROUP BY grade WHERE COUNT(*)=1;
```

Since `COUNT` is an aggregating function, the following is required.

```
SELECT grade FROM grades GROUP BY grade HAVING COUNT(*)=1;
```

This SQL query returns all the grades that occur only once in the table. A simple way to remember the difference is *`WHERE` operates on individual records and `HAVING` operates on groups of records.*

Problem 3. Select all the students who have received grades (non-null grades) in more than two classes. How many grades did he receive?

Case Expression

A case expression allows you to temporarily modify records from a select operation. There are two forms of the case expression; simple and searched. The simple form of the expression is a match and replace on a specified column. A simple case expression is demonstrated below. This code will return the name of the student along with their majorcode. But instead of integers, the names of the majors will be returned.

```
SELECT name,
CASE majorcode
    WHEN 1 THEN 'Math'
    WHEN 2 THEN 'Science'
    WHEN 3 THEN 'Writing'
    WHEN 4 THEN 'Art'
    ELSE 'Undeclared'
    END AS major
FROM students;
```

A searched case expression using a boolean expression for the `WHEN` clauses.

```
SELECT name,
CASE
    WHEN majorcode IS NULL THEN 'Undeclared'
    ELSE majorcode
    END AS major,
CASE
    WHEN minorcode IS NULL THEN 'Undeclared'
    ELSE minorcode
    END AS minor
FROM students;
```

Problem 4. Find the overall GPA of all the students in the school. Use a regular 4.0 scale (A=4.0, B=3.0, C=2.0, D=1.0). Any pluses or minuses are dropped so an A- becomes an A.

Your result set should be one column and one row with average of all GPAs of all the students taking classes. Your solution should return a single floating point number.

Use the `ROUND()` function in SQL to round your result to the nearest hundredth.

Like Command

The SQL keyword, `LIKE`, allows us to match patterns in a column. For example,

```
SELECT name, studentid FROM students WHERE studentid LIKE '\%4';
```

will return all the students that have a student ID that ends with the digit 4. Use '\%' before the '4' to signify that there can be any number of characters before the '4.' If you only want one character, you can use an underscore. For example, if you were to search a database of words in the english dictionary and you entered the following command:

```
SELECT word FROM englishDictionary WHERE word LIKE 'i_';
```

You would get words like 'is,' 'it,' or 'in.'

Problem 5. Write a SQL statement that will find all students with a last name that begins with the letter 'C' and return their names and majors. Your returned records should be

Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing

18 Interior Point I: 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 18.1. In general, the Simplex Method requires many more iterations (though each iteration is less expensive computationally).

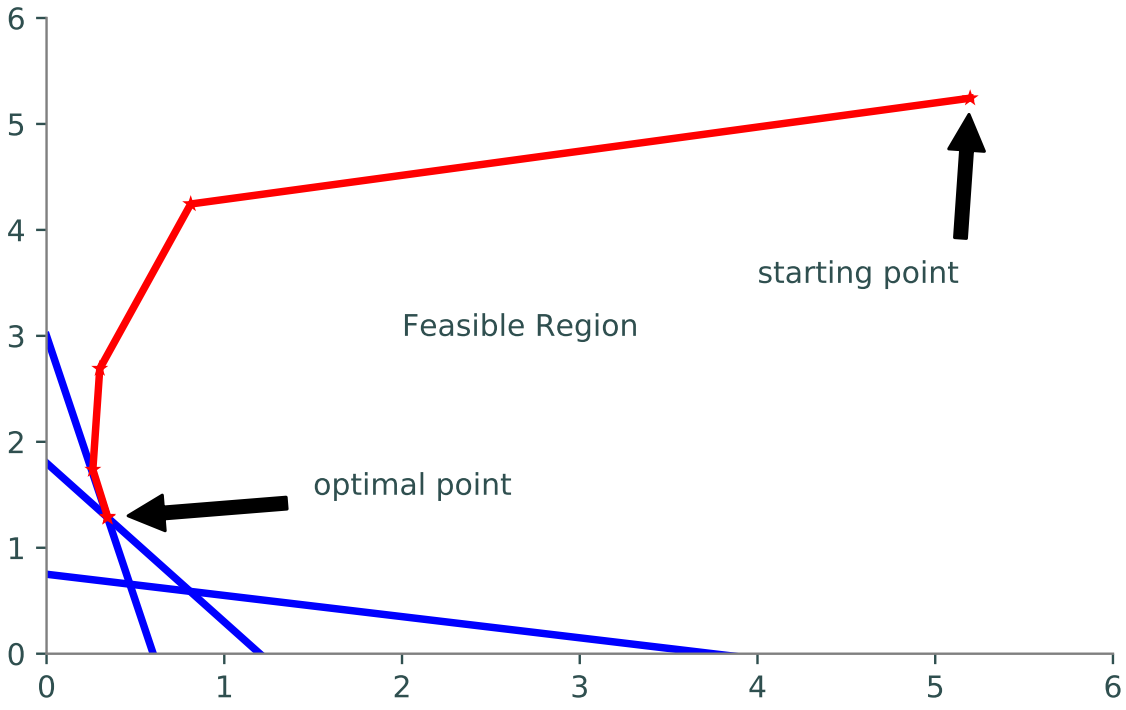


Figure 18.1: A path traced by an Interior Point algorithm.

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:

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

Here, $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and A is an $m \times n$ matrix with full row rank. This is the *primal* problem, and its *dual* takes the form:

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

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

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

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:

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

where $M = \text{diag}(\mu_1, \mu_2, \ldots, \mu_n)$. 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 $(\triangle\mathbf{x}^T, \triangle\boldsymbol{\lambda}^T, \triangle\boldsymbol{\mu}^T)$ in which to step to set the linear approximation equal to $\mathbf{0}$. This equates to solving the linear system:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\boldsymbol{\lambda} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{18.1}$$

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 18.1 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* ν[1] of the problem:

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

The idea is to use Newton's method to identify a direction that strictly decreases ν. Thus instead of solving Equation 18.1, we solve:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\boldsymbol{\lambda} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma\nu\mathbf{e} \end{bmatrix} \tag{18.2}$$

where $\mathbf{e} = (1, 1, \dots, 1)^T$ and $\sigma \in [0, 1)$ is called the *centering parameter*. The closer σ is to 0, the more similar the resulting direction will be to the plain Newton direction. The closer σ is to 1, the more the direction points inward to the interior of the of the feasible region.

Problem 2. Within `interiorPoint()`, write a subroutine to compute the search direction $(\triangle\mathbf{x}^T, \triangle\boldsymbol{\lambda}^T, \triangle\boldsymbol{\mu}^T)$ by solving Equation 18.2. 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). Consider using the functions `lu_factor()` and `lu_solve()` from the `scipy.linalg` module to solving the system of equations efficiently.

Step Length

Now that we have our search direction, it remains to choose our step length. We wish to step nearly as far as possible without violating the problem's constraints, thus remaining in the interior of the feasible region. First, we calculate the maximum allowable step lengths for $\mathbf{x}$ and $\boldsymbol{\mu}$, respectively:

$$\alpha_{\max} = \min\{1, \min\{-\mu_i/\triangle\mu_i \mid \triangle\mu_i < 0\}\}$$
$$\delta_{\max} = \min\{1, \min\{-x_i/\triangle x_i \mid \triangle x_i < 0\}\}$$

Next, we back off from these maximum step lengths slightly:

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

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \delta\triangle\mathbf{x}_k$$
$$(\boldsymbol{\lambda}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\boldsymbol{\lambda}_k, \boldsymbol{\mu}_k) + \alpha(\triangle\boldsymbol{\lambda}_k, \triangle\boldsymbol{\mu}_k).$$

[1] ν 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^T 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.dot(A.T))
    x = A.T.dot(B.dot(b))
    lam = B.dot(A.dot(c))
    mu = c - A.T.dot(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 ν tells us in some sense how close our current point is to the minimizer. The closer ν is to 0, the closer we are to the optimal point. Thus, by printing the value of ν 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.

```
    """Generate a linear program min c^T 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,:].dot(x)
    b[n:] = A[n:,:].dot(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:

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

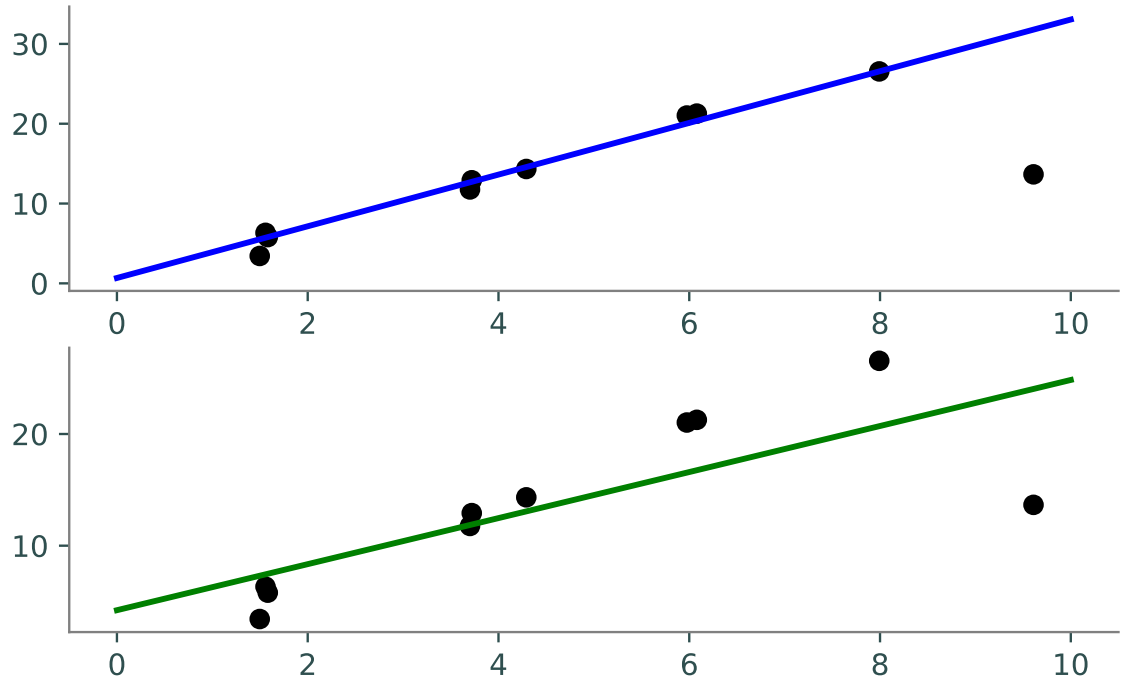


Figure 18.2: Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

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

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

and we seek to choose the parameters $\boldsymbol{\beta}, b$ so as to minimize this error.

Advantages of LAD

The most prominent difference between this approach and least squares is how they respond to outliers in the data. Least absolute deviations is robust in the presence of outliers, meaning that one (or a few) errant data points won't severely affect the fitted line. Indeed, in most cases, the best fit line is guaranteed to pass through at least two of the data points. This is a desirable property when the outliers may be ignored (perhaps because they are due to measurement error or corrupted data). Least squares, on the other hand, is much more sensitive to outliers, and so is the better choice when outliers cannot be dismissed. See Figure 18.2.

While least absolute deviations is robust with respect to outliers, small horizontal perturbations of the data points can lead to very different fitted lines. Hence, the least absolute deviations solution is less stable than the least squares solution. In some cases there are even infinitely many lines that minimize the least absolute deviations error term. However, one can expect a unique solution in most cases.

The least absolute deviations solution arises naturally when we assume that the residual terms $\boldsymbol{\beta}^T\mathbf{x}_i+b-y_i$ 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 $i = 1, 2, \ldots, m$ we introduce the artificial variable u_i to take the place of the error term $|\boldsymbol{\beta}^T\mathbf{x}_i + b - y_i|$, and we require this variable to satisfy $u_i \geq |\boldsymbol{\beta}^T\mathbf{x}_i + b - y_i|$. This constraint is not yet linear, but we can split it into an equivalent set of two linear constraints:

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

The u_i are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

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

Now for each inequality constraint, we bring all variables $(u_i, \boldsymbol{\beta}, b)$ to the left hand side and introduce a nonnegative slack variable to transform the constraint into an equality:

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

Notice that the variables $\boldsymbol{\beta}, b$ 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:

$$\begin{aligned} \boldsymbol{\beta} &= \boldsymbol{\beta}_1 - \boldsymbol{\beta}_2, \\ b &= b_1 - b_2, \\ \boldsymbol{\beta}_1, \boldsymbol{\beta}_2, b_1, b_2 &\geq 0. \end{aligned}$$

Substituting these values into our constraints, we have the following system of constraints:

$$\begin{aligned} u_i - \boldsymbol{\beta}_1^T\mathbf{x}_i + \boldsymbol{\beta}_2^T\mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i, \\ u_i + \boldsymbol{\beta}_1^T\mathbf{x}_i - \boldsymbol{\beta}_2^T\mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i, \\ u_i, \boldsymbol{\beta}_1, \boldsymbol{\beta}_2, b_1, b_2, s_{2i-1}, s_{2i} &\geq 0. \end{aligned}$$

Writing $\mathbf{y} = (-y_1, y_1, -y_2, y_2, \ldots, -y_m, y_m)^T$ and $\boldsymbol{\beta}_i = (\beta_{i,1}, \ldots, \beta_{i,n})^T$ for $i = \{1, 2\}$, we can aggregate all of our variables into one vector as follows:

$$\mathbf{v} = (u_1, \ldots, u_m, \beta_{1,1}, \ldots, \beta_{1,n}, \beta_{2,1}, \ldots, \beta_{2,n}, b_1, b_2, s_1, \ldots, s_{2m})^T.$$

Defining $\mathbf{c} = (1, 1, \ldots, 1, 0, \ldots, 0)^T$ (where only the first m entries are equal to 1), we can write our objective function as

$$\sum_{i=1}^{m} u_i = \mathbf{c}^T \mathbf{v}.$$

Hence, the final form of our linear program is:

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

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

The variable `sol`, however, 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 $\boldsymbol{\beta} = \boldsymbol{\beta}_1 - \boldsymbol{\beta}_2$ 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 ($\mathbf{x}_i$). 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)
```

19 Interior Point II: 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:

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

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

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

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

The Lagrangian function for this problem is:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{c}^T\mathbf{x} - \boldsymbol{\mu}^T(A\mathbf{x} - \mathbf{b}), \tag{19.1}$$

where $\boldsymbol{\mu} \in \mathbb{R}^m$ is the (as of yet unknown) 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

$$A\mathbf{x} - \mathbf{b} - \mathbf{y} = \mathbf{0} \qquad \Longrightarrow \qquad \mathbf{y} = A\mathbf{x} - \mathbf{b} \tag{19.2}$$

Equations 19.1 and 19.2, together with complementary slackness, gives us our complete set of KKT conditions:

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

Quadratic Interior Point Method

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

$$F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q\mathbf{x} - A^T\boldsymbol{\mu} + c \\ A\mathbf{x} - \mathbf{y} - \mathbf{b} \\ YM\mathbf{e} \end{bmatrix} = \mathbf{0}, \qquad (\mathbf{y}, \boldsymbol{\mu}) \succeq \mathbf{0}$$

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

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

and the duality measure ν for this problem is

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

Search Direction

We calculate the search direction for this algorithm the same way that we did in the linear programming case. That is, we solve the system:

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\mathbf{y} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma\nu\mathbf{e} \end{bmatrix}, \tag{19.3}$$

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

Problem 1. Copy your `interiorPoint()` function from the previous lab into your solutions file for this lab, renaming it `qInteriorPoint()`. This new function 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`.

Modify your code to match the F and DF described above, and calculate the search direction $(\triangle\mathbf{x}^T, \triangle\mathbf{y}^T, \triangle\boldsymbol{\mu}^T)$ by solving Equation 19.3. 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. We back off slightly from the maximum allowed step length, however, because an overly greedy step at one iteration may prevent a decent step at the next iteration. Thus, we choose our step size

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

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

This is equivalent to the method of choosing a step direction used in the previous lab. In this case, however, we will use a single step length for all three of the parameters.

$$\begin{aligned}\beta_{\max} &= \min\{1, \min\{-\mu_i/\triangle\mu_i \mid \triangle\mu_i < 0\}\} \\ \delta_{\max} &= \min\{1, \min\{-y_i/\triangle y_i \mid \triangle y_i < 0\}\}\end{aligned}$$

Since $\boldsymbol{\mu}, \mathbf{y} \geq \mathbf{0}$. If all of the entries of $\triangle\mu$ are positive, we let $\beta_{\max} = 1$, and likewise for $\delta_{\max}$. 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 α is our final step length. Thus, the next point in the iteration is given by:

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

This completes one iteration of the algorithm.

Initial Point

As usual, 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^T Gx + x^T c s.t. Ax >= b.
    Inputs:
        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.

---

procedure Interior Point Method for QP
  Choose initial point $(\mathbf{x}_0, \mathbf{y}_0, \boldsymbol{\mu}_0)$.
  while $k <$ `niters` and $\nu <$ `tol`: **do**
    Calculate the duality measure ν.
    Solve 19.3 for the search direction $(\triangle\mathbf{x}_k, \triangle\mathbf{y}_k, \triangle\boldsymbol{\mu}_k)$.
    Calculate the step length α.
    $(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \boldsymbol{\mu}_k) + \alpha(\triangle\mathbf{x}_k, \triangle\mathbf{y}_k, \triangle\boldsymbol{\mu}_k)$.

Problem 2. Complete the implementation of `qInteriorPoint()`. Return the optimal point $\mathbf{x}$ as well as the final objective function value. You may want to print out the duality measure ν to check the progress of the iteration.

Test your algorithm on the simple problem

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

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

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

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

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

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

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 ν.

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 in 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.

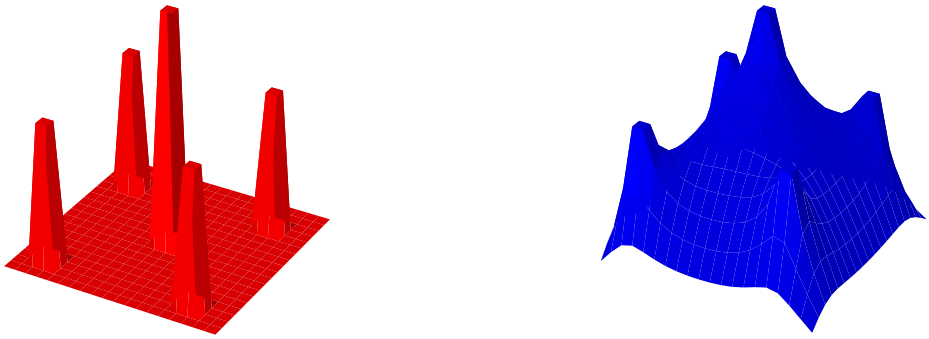


Figure 19.1: Tent pole configuration (left) and optimal elastic tent (right).

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 constraints:

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

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. Then since we have modeled our tent with a discrete grid of values, this energy function has the form

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

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:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{x} \succeq L. \end{aligned}$$

See Figure 19.1 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 $\boldsymbol{\mu}$.

```
# 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 of 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 $\mathbf{x} = (x_1, \dots, x_n)^T$, 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 μ_i and a standard deviation σ_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:

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

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 in general 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:

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

Note that we have slightly altered our objective function for convenience, as minimizing $\frac{1}{2}\mathbf{x}^T Q\mathbf{x}$ is equivalent to minimizing $\sqrt{\mathbf{x}^T Q\mathbf{x}}$. 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:

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

Each return value R can be paired with its corresponding minimal risk σ. 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 19.2.

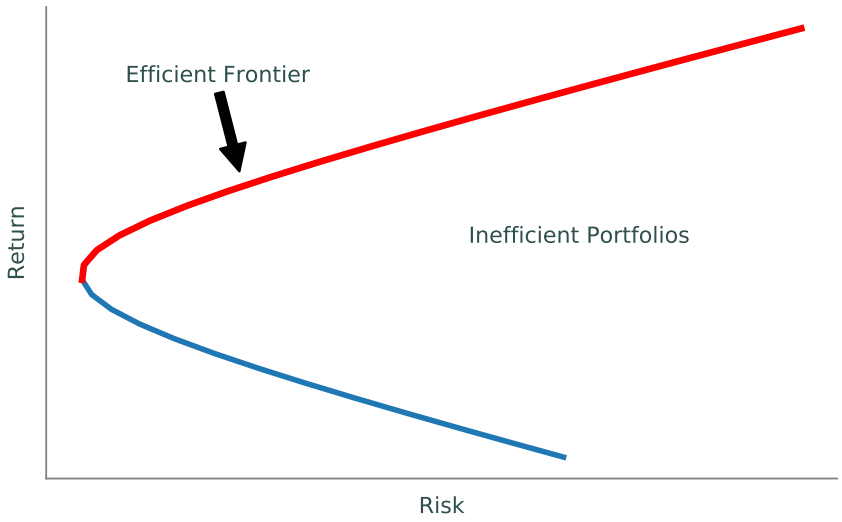


Figure 19.2: Efficient frontier on the risk-return plane.

One difficulty 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

$$y^i = [y_1^i, \ \ldots, \ y_k^i]^T.$$

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 Q and each μ_i.

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 μ_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.

20 Value Function Iteration

Lab Objective: *Many questions have optimal answers that change over time. Sequential decision making problems are among this classification. In this lab you we learn how to solve sequential decision making problems, also known as dynamic optimization problems. We teach these fundamentals by solving the finite-horizon cake eating problem.*

Dynamic optimization answers different questions than optimization techniques we have studied thus far. For example, an oil company might want to know how much oil to excavate in one day in order to maximize profit. If each day is considered in isolation, then the strategy to optimize profit is to maximize excavation in order to maximize profits. However, in reality oil prices change from day to day as supply increases or decreases, and so maximizing excavation may in fact lead to less profit. On the other hand, if the oil company considers how production on one day will affect subsequent decisions, they may be able to maximize their profits. In this lab we explore techniques for solving such a problem.

The Cake Eating Problem

Rather than maximizing oil profits, we focus on solving a general problem that can be applied in many areas called the cake eating problem. Given a cake of a certain size, how do we eat it to maximize our enjoyment (also known as utility) over time? 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. These preferences are expressed with a utility function. Our task is to find an optimal strategy given a smooth, strictly increasing and concave utility function, u. Precisely, given a cake of size W and some amount of consumption $c_0 \in [0, W]$, the utility gained is given by

$$u(c_0).$$

For this lab we restrict our attention to utility functions that have the point $u(0) = 0$. Although any size of W could be used, for simplicity of this lab assume that W has size 1. To further simplify the problem assume that W is cut into N equally-sized pieces. If we want to maximize utility in one time period, we consume the entire cake. How do we maximize utility over several days?

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. In this lab, 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.

$$u(c_0) > \beta u(c_0).$$

The Optimization Problem

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

$$[c_0, c_1, \dots, c_T]^\mathsf{T}$$

where

$$\sum_{i=0}^{T} c_i = W.$$

This vector is called a *policy vector*. The optimization problem is to

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

Problem 1. Write a function called `graph_policy()` that will accept a policy vector $\mathbf{c}$, a utility function $u(x)$, and a discount factor β. Return the total utility gained with the policy input. Also display a plot of the total cumulative utility gained over time. Ensure that the policy that the user passes in sums to 1. Otherwise, raise a `ValueError`. It might seem obvious what sort of policy will yield the most utility, but the truth may surprise you. See Figure 20.1 for some examples.

```
# The policy functions used in the Figure below.
>>> pol1 = np.array([1, 0, 0, 0, 0])
>>> pol2 = np.array([0, 0, 0, 0, 1])
>>> pol3 = np.array([0.2, 0.2, 0.2, 0.2, 0.2])
>>> pol4 = np.array([.4, .3, .2, .1, 0])
```

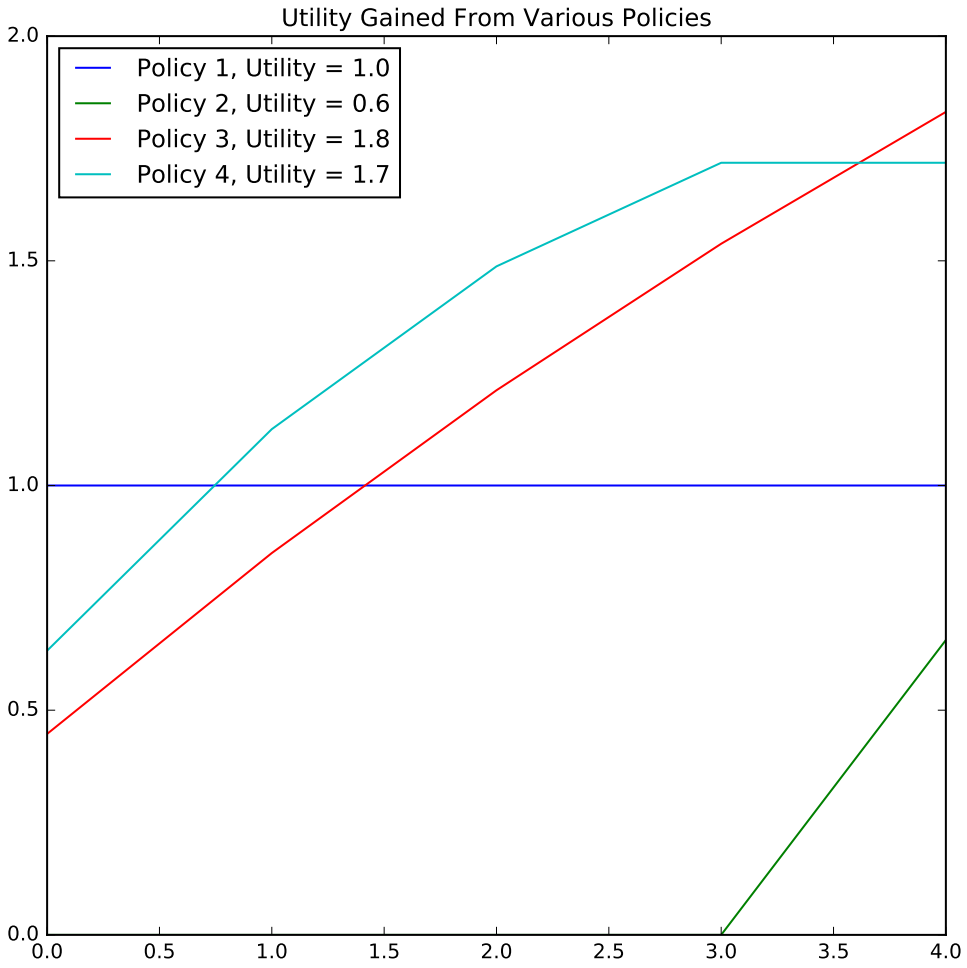


Figure 20.1: Plots for various policies with $u(x) = \sqrt{x}$ and $\beta = 0.9$. Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same amount of cake at each step, while policy 4 begins by eating a lot of the cake but eats less and less as time goes on until the cake runs out.

The Value Function

The cake eating problem is an optimization problem and the value function is used to solve it. The value function $V(a, b, W)$ is the highest utility we can achieve.

$$V(a,b,W) = \max_{c_t} \sum_{t=a}^{b} \beta^t u(c_t)$$
$$\text{subject to } \sum_{t=a}^{b} c_t = W$$
$$c_t \geq 0.$$

The value function gives the utility gained from following an optimal policy from time a to time b. $V(a, b, \frac{W}{2})$ gives how much utility we will gain by proceeding optimally from $t = a$ if half of a cake of size W was eaten before time $t = a$.

By using the optimal value in the future, we can determine the optimal value for the present. In other words, we must iterate backwards to solve the value problem. Let W_i represent the total amount of cake left at time $t = i$. Observe that $W_{i+1} \leq W_i$ for all i, because our problem does not allow for the creation of more cake. The value function can be expressed as

$$V(t,T,W_t) = \max_{W_{t+1}} \left(u(W_t - W_{t+1}) + \beta V(t, T-1, W_{t+1})\right). \tag{20.1}$$

$u(W_t - W_{t+1})$ is the value gained from eating $W_t - W_{t+1}$ cake. $\beta V(t, T-1, W_{t+1})$ is the value of saving W_{t+1} cake until later. Recall that the utility function $u(x)$ and β are known.

In order to solve the problem iteratively, W is split into N equally-sized pieces, meaning that W_t only has $N + 1$ possible values. Programmatically, $V(t, T, W_t)$ can be solved by trying each possible W_{t+1} and choosing the one that gives the highest utility. Knowing the maximum utility in the future allows us to calculate the maximum utility in the present.

Problem 2. Write a helper function to assist in solving the value function. Assume our cake has volume 1 and N equally-sized pieces. Write a method that accepts N and a utility function $u(x)$. Create a partition vector whose entries correspond to possible amounts of cake. For example, if split a cake into 4 pieces, the vector is

$$\mathbf{w} = [0, 0.25, 0.5, 0.75, 1.0]^\mathsf{T}.$$

Construct and return a matrix whose $(ij)^{th}$ entry is the amount of utility gained by starting with i pieces and saving j pieces (where i and j start at 0). In other words, the $(ij)^{th}$ entry should be $u(w_i - w_j)$.

Set impossible situations to 0 (i.e., eating more cake than you have available). The resulting lower triangular matrix is the *consumption matrix*.

For example, the following matrix results with $N = 4$ and $u(x) = \sqrt{x}$.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ u(0.25) & 0 & 0 & 0 & 0 \\ u(0.5) & u(0.25) & 0 & 0 & 0 \\ u(0.75) & u(0.5) & u(0.25) & 0 & 0 \\ u(1) & u(0.75) & u(0.5) & u(0.25) & 0 \end{bmatrix}.$$

Solving the Optimization Problem

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_3 represents having three pieces of cake left and $w_3 = 0.75$.

The $(N+1) \times (T+1)$ matrix, A, that solves the value function is called the *value function matrix*. We will calculate the value function matrix step-by-step. A_{ij} is the value of having w_i cake at time j. Like the consumption matrix, i and j start at 0. It should be noted that $A_{0j} = 0$ because there is never any value in having w_0 cake, $u(w_0) = u(0) = 0$.

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)? Indeed there may be many scenarios to consider. It may not be obvious which option is best and that option may change depending on the discount factor β.

Instead of asking how much cake to eat at some time t, we should ask how valuable w_i cake is at time t? At some time t, there may be numerous decisions, but at the last time period, the only decision to make is how much cake to eat at $t = T$.

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 $u(w_i)$. This utility is A_{iT}. Written in the form of (20.1),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} \left(u(w_i - w_j) + \beta V(0, -1, w_j)\right) = u(w_i). \tag{20.2}$$

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 $t = T$ it is true that the true utility of waiting until time T to eat w_i cake is actually $\beta^T u(W_i)$, and can be verified by inspecting the difference of policies 1 and 2 in Figure 20.1. However, programmatically 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 $V(T, T, W_i)$. After calculating $t = T$, $t = T - 1$ is introduced, and its value is calculated by considering the utility from eating $w_i - w_j$ cake at time $t = T - 1$, plus the utility of β times the value of w_j at time T. We then proceed iteratively backwards, considering $t = T - 2$ and considering its utility plus the utility of β times the value at time $T - 1$.

Problem 3. Write a function called `eat_cake()` that accepts the stopping time T, the number of equal sized pieces that divides the cake N, a discount factor β, and a utility function $u(x)$. Return the value function matrix with all zeros except for the last column. The spec file indicates returning a policy matrix as well, for now return a matrix of zeros.

For example, the following matrix results with $T = 3$, $N = 4$, $\beta = 0.9$, and $u(x) = \sqrt{x}$.

$$\begin{bmatrix} 0 & 0 & 0 & u(0) \\ 0 & 0 & 0 & u(0.25) \\ 0 & 0 & 0 & u(0.5) \\ 0 & 0 & 0 & u(0.75) \\ 0 & 0 & 0 & u(1) \end{bmatrix}.$$

We can evaluate the next column of the value function matrix, $A_{i(T-1)}$, by modifying (20.2) as follows,

$$A_{i(T-1)} = V(0, 1, w_i) = \max_{w_j} \left(u(w_i - w_j) + \beta V(0, 0, w_j)\right) = \max_{w_j} \left(u(w_i - w_j) + \beta A_{jT}\right). \tag{20.3}$$

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 $i - j$ pieces of cake at time t and saving j pieces of cake until the next period. For $t = T - 1$,

$$CV_{ij}^{T-1} = u(w_i - w_j) + \beta A_{jT}. \tag{20.4}$$

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 $T - 1$. Because we know the last column of A, we may iterate backwards to fill in the rest of A.

Achtung!

The notation CV^t does not mean raising the matrix to the t^{th} power, 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.

$$CV^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0.45 & 0 & 0 & 0 \\ 0.707 & 0.95 & 0.636 & 0 & 0 \\ 0.866 & 1.157 & 1.136 & 0.779 & 0 \\ 1 & 1.316 & 1.343 & 1.279 & 0.9 \end{bmatrix}$$

This 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 $t = 2$. See Figure 20.2.

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 (20.3) and (20.4) shows

$$CV_{ij}^t = u(w_i - w_j) + \beta A_{j(t+1)}. \qquad A_{it} = \max_j \left(CV_{ij}^t\right). \tag{20.5}$$

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 0.95 & 0.95 & 0.95 & 0.707 \\ 1.355 & 1.355 & 1.157 & 0.866 \\ 1.7195 & 1.562 & 1.343 & 1 \end{bmatrix}.$$

Figure 20.2: 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 4. Complete the function `eat_cake()` to determine the entire value function matrix. Starting from the next to last column, iterate backwards by

- calculating the current value matrix for time t using (20.5),
- finding the largest value in each row of the current value matrix,
- filling in the corresponding column of A with these values.

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 a straightforward calculation similar 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,

$$P_{it} = w_i - w_j.$$

$$\text{where } j = \{\, min\{j\} \mid CV_{ij}^t \geq CV_{ik}^t \; \forall\, k \in [0, 1, \ldots, N] \,\}$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.5 \\ 0.25 & 0.25 & 0.5 & 0.75 \\ 0.25 & 0.5 & 0.5 & 1. \end{bmatrix}$$

An example of P where $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$. The optimal policy is found by starting at $i = N$, $j = 0$ and eating as much cake as the $(ij)^{th}$ entry indicates, as traced out by the red arrows. The blue arrows trace out the policy that would occur if we only had 2 time intervals. What would be the optimal policy if we had 3 time intervals?

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

The non-simplified version of Figure 20.2. Notice that the value of A_{ij} is equal to following optimal path if you start at P_{ij}. A_{40} has the same values traced by the red arrows in P above and A_{42} has the same values traced by the blue arrows.

Problem 5. Modify `eat_cake()` 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 6. The $(ij)^{th}$ entry of the policy matrix tells us how much cake to eat at time j if we start with i pieces. Use this information to write a function that will find the optimal policy for starting with a cake of size 1 split into N pieces given the stopping time T, the utility function u, and a discount factor β. Use `graph_policy()` to plot the optimal policy. See Figure 20.3 for an example.

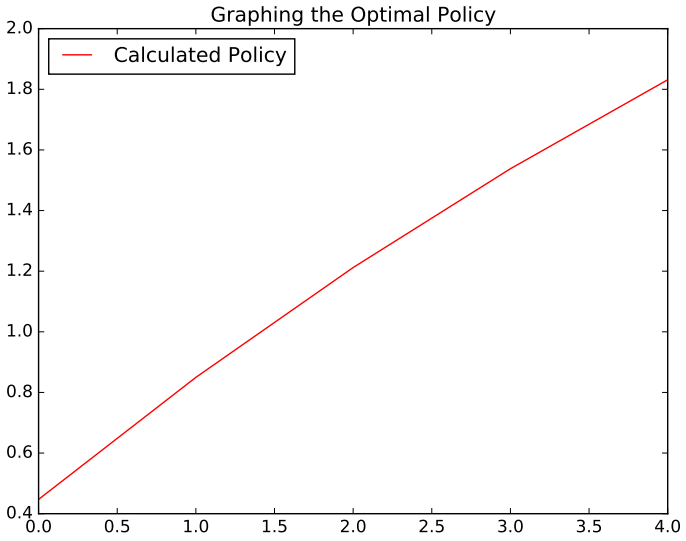


Figure 20.3: The graph of the optimal policy (Policy 3 from Figure 20.1) where $T = 4$, $\beta = .9$, $N = 5$, and $u(x) = \sqrt{x}$. It achieves a value of roughly 1.83.

A summary of the arrays generated in this lab is given below, in the order that they were generated in the lab:

Consumption matrix: Equal to $u(u_i - w_j)$, the utility gained when you start with i pieces and end with j pieces.

Value Function Matrix, A: How valuable it is to have w_i pieces at time j.

Current Value Matrix, CV^t: How valuable each possible decision is at time t.

Policy Matrix, P: The amount of cake you should eat at time t.

21 Policy Function Iteration

Lab Objective: *This section teaches how to improve dynamic programming convergence using policy function iteration.*

Now that we have covered how to solve simple dynamic programming problems by value function iteration, we consider the convergence of the algorithm. We demonstrate two additional methods known as policy function iteration, or Howard's Improvement, and modified policy function iteration.

Policy Function Iteration

For infinite horizon dynamic programming problems, it can be shown that value function iteration converges at the rate β, where β is the discount factor. In practice, β is usually close to one, which means this algorithm often converges slowly.

In order to examine the value function iteration algorithm, it is helpful to see which functions take the most runtime. We can perform this analysis using the profiling tools in IPython.

Problem 1. Import your function `eatCake` from the Value Function Iteration lab into an IPython environment. Profile this function with arguments $\beta = .9$, $N = 100$, `finite=False`, `plot=False`. Now change the arguments $\beta = .95$ and $N = 1000$. How does your runtime change? What parts of the code take up the most time?

Recall that we can access the profiling tool in IPython as follows:

```
>>> %prun function_call
```

where `function_call` is a placeholder for whatever function you wish to profile.

In Problem 1 you should have noticed that runtime was significantly longer to run for larger N or β closer to 1. The profiler gives more detailed information than just the overall runtime, however. The results of Problem 1 should look something like the following (we have removed a few columns and several rows from the full output):

```
     503 function calls in 2.101 seconds
```

```
Ordered by: internal time

ncalls  tottime  filename:lineno(function)
     1    1.759   <ipython-input-4-1487b3d35ebf>:1(eatCake)
    59    0.244   {method 'argmax' of 'numpy.ndarray' objects}
     1    0.033   twodim_base.py:427(triu)
     1    0.018   {numpy.core.multiarray.where}
     1    0.009   method 'repeat' of 'numpy.ndarray' objects}
     1    0.009   {method 'outer' of 'numpy.ufunc' objects}
     1    0.005   twodim_base.py:349(tri)
     1    0.004   twodim_base.py:657(mask_indices)
     1    0.004   {method 'astype' of 'numpy.ndarray' objects}
    59    0.004   {method 'reduce' of 'numpy.ufunc' objects}
```

We notice that the most time was spent in the maximization step (`method 'argmax'`).

Recall that the value function iteration method for the infinite horizon problem approximates the value function V iteratively, producing a sequence of approximations $V_1, V_2, V_3, \ldots$ that converge to V. At each step, the algorithm also determines the policy function ψ_k corresponding to the approximated value function V_k. This process can also be viewed as applying the approximate policy function ψ_k once to obtain the approximate value function V_k. Unfortunately, this gives us a rather crude approximation to the value function, resulting in slow convergence.

What if instead of iteratively approximating the value function in this manner, we instead iteratively approximate the policy function, and calculate the exact value function associated with each new policy function? This is the idea behind the policy function iteration algorithm. In this way we iterate on the policy functions rather than the value functions. The algorithm for the policy function iteration can be summarized as follows:

1. Set an initial policy rule $W' = \psi_0(W)$ and a tolerance δ.

2. Compute the value function assuming this rule is used forever:

$$V_k(W) = \sum_{t=0}^{\infty} \beta^t u(W_t - \psi_k(W_t)).$$

3. Determine a new policy ψ_{k+1} so that

$$\psi_{k+1}(W) = \underset{W'}{\text{argmax}}\{u(W - W') + \beta V_k(W')\}.$$

4. If $\delta_k = ||\psi_{k+1} - \psi_k|| < \delta$, stop, otherwise go back to step 2 with subscript $k + 1$.

In order to compute the value function, V_k corresponding to a given policy ψ_k, we must solve

$$V_k(W) = u(W - W') + \beta V_k(W') \tag{21.1}$$

for V_k.

As always, we take a discrete approximation to W, obtaining a length N vector

$$w = (w_1, w_2, \ldots, w_N)$$

giving the possible cake quantities. The variable W' becomes $w' = \psi_k(w)$. Once we have done this, equation (21.1) is a linear system which we can rewrite as

$$V_k(w) = u(w - w') + \beta Q V_k(w)$$

where Q is the $N \times N$ matrix

$$Q_{ij} = \begin{cases} 1 & \text{if} \quad w'_i = w_j \\ 0 & \text{otherwise.} \end{cases}$$

Solving this system of equations, we have $V_k(w) = (I - \beta Q)^{-1} u(w - w')$. Although Q may be large, we can take advantage of the fact that it is sparse, containing only N nonzero entries out of N^2 total entries.

Problem 2. Solve the infinite horizon cake eating problem from the Value Function Iteration lab again, this time using policy function iteration. Write a function `policyIter` that takes arguments β (discount factor), N (size of discrete approximation), and W_{max} (size of original cake, set to default value of 1). Take the function u to be the square root function, as before. Return the converged value function and policy function, both of which should be arrays of length N.

As in the value function iteration, first create your discrete approximation of the W values:

```
>>> import numpy as np
>>> w = np.linspace(0, Wmax, N)
```

You will still need to pre-compute all values of $u(W - W')$, storing these in an $N \times N$ array. Make sure, as before, that the upper triangular entries are set to large negative values, so that we don't choose to consume more cake than is available. In the code snippets below, we will refer to this array as `U`.

You may also find it convenient to not track the approximated policy function ψ_k directly during the iteration, but rather to have a length N array `psi_ind`, whose i-th entry gives the index of w'_i relative to the discrete approximation w. Thus, rather than initializing a policy function ψ_0 directly, you can initialize it indirectly by setting, for example,

```
>>> psi_ind = np.arange(N)
```

which corresponds to an initial policy $\psi_0(W) = W$. The policy function vector can be obtained from `psi_ind` by simply slicing `w` as follows:

```
>>> psi = w[psi_ind]
```

In order to take advantage of the sparse matrices I and Q, use the following imports from the SciPy `sparse` library.

```
>>> from scipy import sparse
>>> from scipy.sparse import linalg
```

and the following code to initialize I (outside the loop)

```
>>> I = sparse.identity(N, format='csr')
>>> rows = np.arange(0, N)
```

and Q (inside the loop)

```
>>> columns = psi_ind
>>> data = np.ones(N)
>>> Q = sparse.coo_matrix((data, (rows,columns)), shape=(N,N))
>>> Q = Q.tocsr()
```

Rather than compute $(I - \beta Q)^{-1}$ directly, use Scipy's sparse solver

```
V = linalg.spsolve(I-beta*Q, u(w-w[psi_ind]))
```

where `u` is the square root function.

In each iteration, we update `psi_indices` much as we did in the value function iteration algorithm:

```
>>> psi_ind = np.argmax(U + beta*V, axis=1)
```

where we assume `V` has shape `(1,N)`.

Use the 2-norm when computing $\|\psi_{k+1} - \psi_k\|$, and set the tolerance to $\delta = 1e - 9$.

Take $N = 1000$ and $\beta = .95$. Plot the policy function and compare with your policy function from the Value Function Iteration Lab, using the same inputs. You should obtain the same answer, but in less runtime.

Modified Policy Function Iteration

While policy function iteration converges in fewer iterations, solving the linear system can be slow, especially for problems with a large state space. There is an alternative to this called modified policy function iteration.

In modified policy function iteration, we don't compute the exact value function corresponding to a policy. Instead, at step 2 of the policy iteration algorithm we iterate m times on the value function equation (21.1) to get an approximation of the new value function. This is faster than solving for the exact value function for large state spaces. There is no strict rule on the value of m, the number of value function iterations. In practice values such as $m = 10$ or $m = 15$ often work well.

Note that our methods for solving dynamic programs boil down to some combination of two things: iterating on the value function and iterating on the policy function. Modified policy function does a combination of the two, taking advantage of the advantages of both methods. Because modified policy iteration takes only slightly more work to code than value function iteration, it is often preferred in practice. Whether policy or modified policy iteration will perform better may depend on the problem.

Problem 3. Solve the infinite horizon cake eating problem again, this time using the modified policy function iteration method.

Write a function `modPolicyIter` that takes the same arguments as your `policyIter` function, along with an additional keyword argument m (set to default value $m = 15$), which gives the number of iterations used to approximate the value function at each step. Return the converged value function and policy function.

Much of the code in your `policyIter` will be unchanged when implementing this function. The key differences are as follows. First, you do not need to initialize the sparse arrays I and Q in the modified policy function iteration. Secondly, rather than solving a linear system of equations to obtain the value function, you will loop the equation

```
>>> V = u(w - w[psi_ind]) + beta*V[psi_ind]
```

m times in each iteration. Notice how this line of code corresponds with Equation (21.1), where `w` represents W, `w[psi_ind]` represents W', and `V[psi_ind]` represents $V(W')$.

The remainder of the code should be unchanged.

Problem 4. Solve the cake eating problem with each of the three methods and report how many iterations each takes. Use $N = 1000$ as the number of grid points for W and $\beta = 0.95$. It is important that you use the same initial guess in each case in order to make the results comparable. The accuracy of the initial guess greatly affects the number of iterations to convergence. Take your initial guess as $V = 0$, which corresponds to an initial guess of the policy function with indices $[0, 1, 2, \ldots, N-1]$ (meaning $\psi(W) = W$, i.e. we eat no cake, leaving it all for the next period).

In general we should see that value function iteration takes more iterations than modified policy function iteration which in turn takes more iterations than policy function iteration. It is important to note that this does not directly say anything about runtime. Each iteration of policy iteration may take longer than an iteration of value function iteration.

Discrete Choice (Threshold) Problems

One powerful application of dynamic programming is that we can make models that have both continuous and discrete state variables. These models are sometimes referred to as discrete choice problems or optimal stopping problems. Examples include models of employment that involve both the choice of whether to work and how much to work, models of firm entry and exit that involve the choice of both whether to produce and how much to produce, and models of marriage that involve the choice of whether to date (get married or keep dating) and how much to date. This application illustrates the versatility of dynamic programming as a dynamic solution method

In this lab, we follow a simple version of a standard job search model. Assume that workers live infinitely long. We will split a worker's life into discrete time periods, and in each period the worker is either employed or unemployed, and receives a job offer. The worker must make a choice between discrete actions (such as accepting or rejecting a job offer), with the goal of maximizing some utility function (hence, this is a *discrete choice* problem).

We can state this problem in terms of dynamic programming by defining an appropriate value function. Let the value of entering a period with most recent wage w, current job offer wage w', and employment status s be given by the following value function,

$$V(w, w', s) = \begin{cases} V^E(w) & \text{if} \quad s = E \\ V^U(w, w') & \text{if} \quad s = U \end{cases} \tag{21.2}$$

where employment status is a binary variable $s \in \{E, U\}$ (E indicates "employed" and U indicates "unemployed"); a person can be either employed or unemployed.

As in the cake eating problem, the value function is calculated as the sum of some reward (based on the current state) and the discounted value of entering the next period in some particular state. The reward function, denoted (as usual) by u, gives the utility of spending available funds. Assuming that a worker receives some wage x in a given period and spends all available money in the period, the utility of consumption is given by

$$u(x).$$

Calculating the value for the next period depends on the employment status in the current period, so we address this separately for each case.

Let us first consider the case where the individual is unemployed ($s = U$). As is customary, let s' denote the employment status of the worker in the next period. In this unemployed state, the worker receives unemployment benefits equal to a fraction of her most recent wage, i.e. αw, where $\alpha \in (0, 1)$. Hence, the utility of consumption in the current unemployed state is given by

$$u(\alpha w).$$

The worker also receives one wage offer (w') per period, and will obtain this wage in the next period provided that she chooses to accept employment, i.e. provided $s' = E$. The worker must decide whether to accept the current wage offer w' or to remain unemployed in the next period, i.e. she must decide on the value of s'. How does she make this choice? She must weigh the value of entering the next period as an employed worker with wage w' (given by $V^E(w')$) versus the value of entering the next period as an unemployed worker with previous wage w and unknown wage offer w'' (given by $V^U(w, w'')$). Because the worker cannot know what the future wage offer w'' will be, it is treated as a random variable with a particular probability distribution. Hence, the worker must actually compute the *expected* value of entering the next period unemployed. This term is simply

$$\mathbb{E}_{w''} V^U(w, w''),$$

where $\mathbb{E}_{w''}$ denotes the expectation operator with respect to the probability distribution of future wage offers w''. To sum up, the worker chooses to accept the wage offer w' or remain unemployed in the next period based on which option gives the greater expected value, and the value of this decision is given by

$$\max\Big\{V^E(w'),\ \mathbb{E}_{w''} V^U(w, w'')\Big\}.$$

The overall value of the current unemployed state with previous wage w and current wage offer w' is just the utility of consumption plus the discounted value of the next period, i.e.

$$V^U(w, w') = u(\alpha w) + \beta \max\Big\{V^E(w'),\ \mathbb{E}_{w''} V^U(w, w'')\Big\}, \tag{21.3}$$

where β is the discount factor.

Now we turn to the case where the job status is employed ($s = E$). In this case, the worker receives a wage w in the current period, and so the utility of consumption is just

$$u(w).$$

In the next period, the worker will have most recent wage w, she will receive wage offer w'', and will have employment status s'. As in the unemployed case, w'' is unknown and treated as a random variable. Unlike the unemployed case, however, the worker's future employment status s' is not under her control, but rather is also a random variable. The reason for this is that the worker will remain employed until she loses the job, a random event that occurs with some fixed probability in each time period. Hence, we must calculate the expected value of the next period with respect to both w'' and s'. We may write the entire value function for the employed case as

$$V^E(w) = u(w) + \beta \mathbb{E}_{w'',s'} V(w, w'', s'). \tag{21.4}$$

To calculate the expectation term, we need to know the joint probability distribution over w'' and s'. This can be characterized in the following way. We assume that s' and w'' are independent. Hence, we can split the joint expectation operator into the composition of the two individual expectation operators:

$$\mathbb{E}_{w'',s'} = \mathbb{E}_{w''} \mathbb{E}_{s'}.$$

Let γ represent the probability that an employed worker becomes unemployed in the next period, so that $1-\gamma$ is the probability of remaining employed in the next period. If the worker stays employed in the next period ($s' = E$), then next period's wage equals the current period's wage, and the term inside the expectation is

$$V(w, w'', E) = V^E(w).$$

We then have

$$\begin{aligned} \mathbb{E}_{s'} V(w, w'', s') &= (1-\gamma) V(w, w'', E) + \gamma V(w, w'', U) \\ &= (1-\gamma) V^E(w) + \gamma V^U(w, w''). \end{aligned}$$

Notice that the term $(1-\gamma)V^E(w)$ is constant with respect to w''. Then

$$\begin{aligned} \mathbb{E}_{w''} \mathbb{E}_{s'} V(w, w'', s') &= \mathbb{E}_{w''} \left[(1-\gamma) V^E(w) + \gamma V^U(w, w'') \right] \\ &= \mathbb{E}_{w''} (1-\gamma) V^E(w) + \mathbb{E}_{w''} \gamma V^U(w, w'') \\ &= (1-\gamma) V^E(w) + \gamma \mathbb{E}_{w''} V^U(w, w''). \end{aligned}$$

Hence, we can rewrite (21.4) as follows:

$$V^E(w) = u(w) + \beta \Big[(1-\gamma) V^E(w) + \gamma \mathbb{E}_{w''} V^U(w, w'') \Big]. \tag{21.5}$$

We have now completely described the value function. What about the policy function? The policy function for the unemployed worker gives her decision on whether to accept the job $s' = E$ or to reject the job $s' = U$. This will be a function of both the most recent wage w and the current wage offer w'. The employment status s' in the next period is determined by the policy function ψ:

$$s' = \psi(w, w').$$

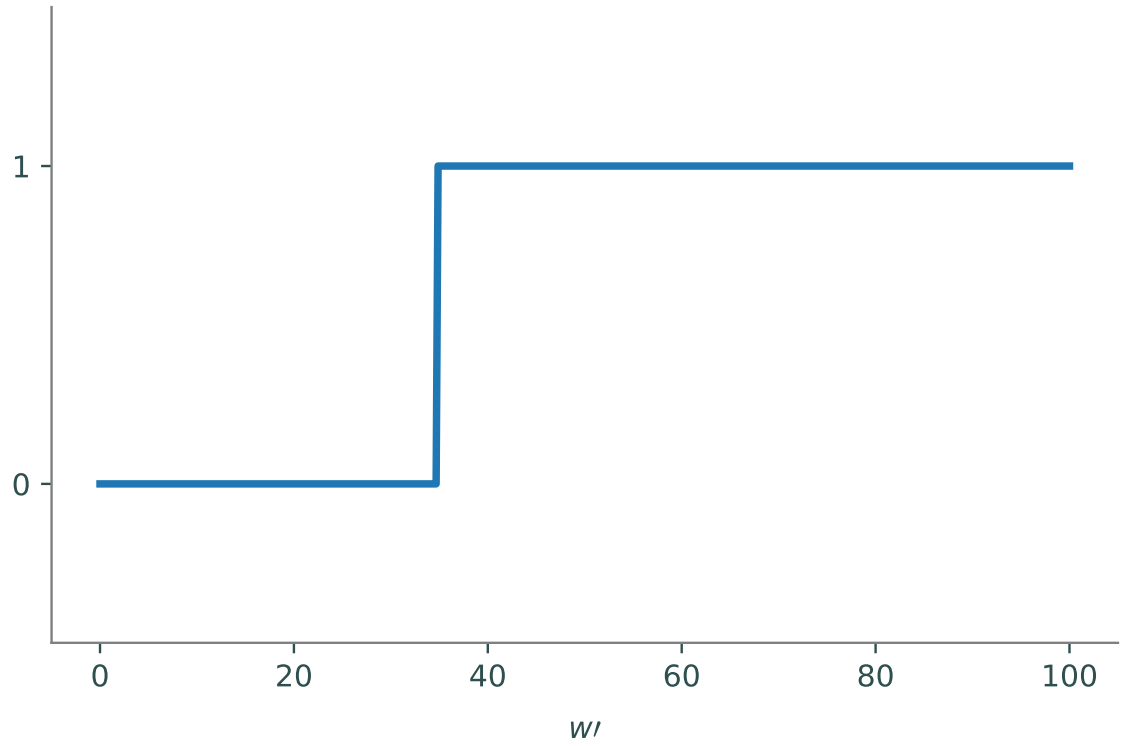


Figure 21.1: Here is the policy function for fixed $w = 50$. Numerically we let 0 represent unemployment, U, and 1 represent employment, E. Thus we see that an individual will choose to take a new job, given their old wage was 50, at a wage of roughly 35. Thus for a previous wage of 50, we say the reservation wage is 35.

These discrete choice problems are often called threshold problems because the policy choice depends on whether the state variable is greater than or less than some threshold level. That is, an unemployed worker will accept a job if and only if the offer wage is above some set amount that depends on the most recent wage w. In the labor search model, the threshold level is called the "reservation wage" w'_R. The reservation wage w'_R is defined as the wage offer such that the worker is indifferent between accepting the job $s' = E$ and staying unemployed $s' = U$. Hence, this reservation wage satisfies the equation

$$V^E(w'_R) = E_{w''}\left[V^U(w, w'')\right]. \tag{21.6}$$

The policy function will then take the form of accepting the job if $w' \geq w'_R$ or rejecting the job offer and remaining unemployed if $w' < w'_R$:

$$s' = \psi(w, w') = \begin{cases} E & \text{if} \quad w' \geq w'_R \\ U & \text{if} \quad w' < w'_R. \end{cases} \tag{21.7}$$

Figure 21.1 shows an example of the discrete policy function.

In summary, the labor search discrete choice problem is characterized by the value functions (21.2), (21.3), and (21.5), the reservation wage (21.6), and the policy function (21.7). Because wage offers are distributed according to some given probability distribution (denote the cdf by $F(w')$), and because the policy function takes the form of (21.7), the probability that the unemployed worker receives a wage offer that she will reject is $F(w'_R)$ and the probability that she receives a wage offer that she will accept is $1 - F(w'_R)$. Just like the continuous-choice cake eating problems, this problem can be solved by value function, policy function, or modified policy function iteration.

The value function iteration solution method for the equilibrium in the labor search problem is analogous to the value function iteration from the previous labs. The only difference is that two value functions (V^E and V^U) must converge to a fixed point in this problem instead of just one value function converging in the previous problems. Although there are two value functions to consider, there is only one policy function, since decisions are only made in the unemployed state. Thus, there is only one policy function on which to iterate in the case of policy or modified policy iteration.

In the following problems, you will solve the job search problem using value function iteration and modified policy function iteration. Assume that the consumption utility function u is given by

$$u(w) = \sqrt{w}.$$

Assume that the probability of becoming unemployed in a given period is $\gamma = 0.10$, the fraction of wages paid in unemployment benefits is $\alpha = 0.5$, and the discount factor is $\beta = 0.9$.

Assume that the log of wage offers are distributed normally. We then say that offers are distributed lognormally and write

$$w' \sim \text{LogN}(\mu, \sigma).$$

This is a convenient choice for the distribution of wage offers. Among other things, it guarantees that wage offers will be positive. A mean of 20 and variance of 200 are typical parameters of such a wage distribution, and we will use these parameters in the following problems.

As usual when dealing with continuous variables, we form a discrete approximation of the possible wage values. In particular, approximate the wage values by a vector of length $N = 500$ of equally-spaced values from $w_{min} = 0$ to $w_{max} = 100$, inclusive. We then form a corresponding discrete approximation of the probability density function $f(w')$ for the lognormal wage offers using code provided in the file `discretelognorm.py`, as follows, where `w` is the length-N vector of wage values, m is the mean, and v is the variance as specified above:

```
>>> from discretelognorm import discretelognorm
>>> f = discretelognorm(w, m, v)
```

The function `discretelognorm` computes the discrete pdf of the specified lognormal distribution in much the same way that you calculate the discrete normal pdf when solving the stochastic cake eating problem.

Problem 5. Solve the job search problem using value function iteration. Return the converged value functions V^E and V^U, as well as the converged policy function ψ. The following steps provide detailed instructions. Note that there are multiple ways to proceed, and the following is simply one (fairly good) possibility.

1. As described above, represent the possible wage values by an array `w` of length N. Denote this array entrywise by
$$w = (w_1, w_2, \ldots, w_N).$$
Calculate the corresponding discrete lognormal pdf `f`, exactly as shown above.

2. Note that $u(w)$ and $u(\alpha w)$ are needed when computing the value functions. Since these quantities do not change from one iteration to another, it is smart to compute them once at the outset. Denote $u(w)$ by `uw` and $u(\alpha w)$ by `uaw`. These are easily calculated as follows:
```
>>> uw = u(w)
>>> uaw = u(alpha*w).reshape((N,1))
```
where `u` is the square root function. We must reshape `uaw` because of array broadcasting issues that arise in the code snippets below.

3. Since V^E is a function of only w, it will be represented by a vector of length N, where the i-th entry gives $V^E(w_i)$. The unemployed value function V^U, however, is a function of both w and w', so it will be represented by an $N \times N$ array, where the (i, j)-th entry gives $V^U(w_i, w_j)$. Initialize the entries of these arrays to 0:
```
>>> VE = np.zeros(N)        #employed value function
>>> VU = np.zeros((N,N))    #unemployed value function
```

4. Note that $\mathbb{E}_{w''} V^U(w, w'')$ is needed to calculate both V^E and V^U. This expectation depends on w, and so can be represented by a length N array, where the i-th entry is $\mathbb{E}_{w''} V^U(w_i, w'')$. It is convenient to assign a variable to this array to keep track of it throughout the iterations. We denote the expectation by `EVU`, and initialize it to zeros:
```
>>> EVU = np.zeros(N)
```

5. For reasons that will soon become apparent, we will need to create an $N \times N$ helper array whose rows are equal to `VE` (call this array `MVE`), and a $N \times N$ helper array whose columns are equal to `EVU` (call this array `MEVU`). At the outset, simply initialize these arrays to zeros.

6. Because job status is a binary variable, the policy function returns one of two possible values. It is convenient to represent "employed" by 1 and "unemployed" by 0. Now the policy function depends on w and w', so it will also be represented by an $N \times N$ array `PSI` of zeros and ones, where the (i, j)-th entry gives $\psi(w_i, w_j)$.

7. Now we are ready to begin the iteration. A single iteration involves computing the updated value functions V^E and V^U from equations (21.5) and (21.3) and then calculating the 2-norm distance between both pairs of old and updated value functions to test for convergence. If both of these 2-norm distances are less than 10^{-9}, terminate the iteration.

   Before calculating the updated value functions, we first update our helper arrays `MVE` and `MEVU`. The rows of `MVE` need to equal `VE`. We can use array broadcasting:

```
>>> MVE[:,:] = VE.reshape((1,N))
```

The columns of `MEVU` need to equal `EVU`, so use a similar technique:

```
>>> MEVU[:,:] = EVU.reshape((N,1))
```

Now let us address how to compute the updated V^U, which we denote by `VU1`. Equation (21.3) shows that it is the sum of two terms. The first, $u(\alpha w)$, we have already computed and stored in the variable `uaw`. The second term involves a maximization between two alternatives. One can imagine writing a double for loop ranging over the values of w' and w to compute each individual $\max\{V^E(w'), \mathbb{E}_{w''}V^U(w, w'')\}$, but we can take advantage of the helper arrays `MVE` and `MEVU` to do this computation in one efficient line of code. Note that the (i, j)-th entry of `MVE` is just $V^E(w_j)$ and the (i, j)-th entry of `MEVU` is $\mathbb{E}_{w''}(w_i, w'')$, and

$$V^U(w_i, w_j) = u(\alpha w_i) + \beta \max\{V^E(w_j), \mathbb{E}_{w''}V^U(w_i, w'')\}.$$

Hence, taking the entrywise maximum of the arrays `MVE` and `MEVU` gives us the appropriate max term for V^U. To get the entrywise maximum of two arrays, stack the arrays along a new axis using `np.dstack`, and maximize along that axis. The computation for `VU`, then, is

```
>>> VU1 = uaw + beta*np.max(np.dstack([MEVU, MVE]), axis=2)
```

Calculating the updated V^E, denoted by `VE1`, is more straightforward. Equation (21.5) shows that it is just a particular linear combination of the arrays `uw`, `VE`, and `EVU`:

```
>>> VE1 = uw + beta*((1-gamma)*VE + gamma*EVU)
```

We can now calculate the 2-norm distances between old and updated value functions. It remains to update `VE`, `VU`, and `EVU`. The first two updates are trivial, and calculating `EVU` is equivalent to the matrix-vector multiplication of `VU` with `f`. This is similar to how we computed expectations in previous labs:

```
>>> EVU = np.dot(VU,f).ravel()
```

We use the `ravel` function to ensure that `EVU` is a flat array.

8. Notice that it is not necessary to iteratively update the policy function, as it is not needed to update the value functions. Thus, we need only compute the policy function once, after convergence of the value functions has been achieved. This is done in a manner similar to calculating $\max\{V^E(w'), \mathbb{E}_{w''}V^U(w, w'')\}$ as described above, except we need to take the *argmax*:

```
>>> PSI = np.argmax(np.dstack([MEVU,MVE]), axis=2)
```

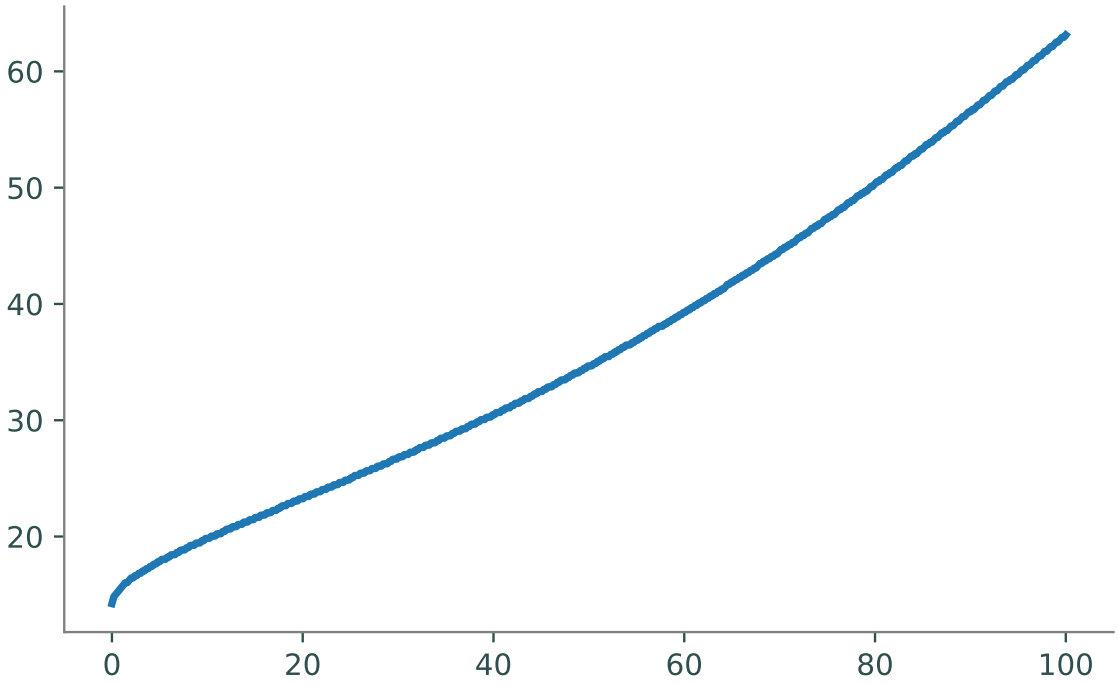


Figure 21.2: The reservation wage as a function of previous wage for the job search problem.

9. Compute the reservation wage w'_R as a function of the current wage w. It will be represented by a length N array called `wr`. The reservation wage is the value of w' where the policy function changes from zeros to ones (the optimal choice changes from remaining unemployed to accepting the job offer). We can calculate this as follows:

```
>>> wr_ind = np.argmax(np.diff(PSI), axis = 1)
>>> wr = w[wr_ind]
```

10. Plot the equilibrium reservation wage w'_R of the converged problem as a function of the current wage w with the current wage on the x-axis and the reservation wage w'_R on the y-axis. This is the most common way to plot discrete choice policy functions. The reservation wage represents the wage that makes the unemployed worker indifferent between taking a job offer and rejecting it. So any wage above the reservation wage line represents $s' = E$ and any wage below the reservation wage line represents $s' = U$. Your plot should resemble that in Figure 21.2.

In the previous problem, it was necessary to iterate on two value functions. Consequently the convergence is relatively slow. We can improve upon this situation by using modified policy function iteration.

Problem 6. Solve the same problem, this time using modified policy function iteration with 15 value function iterations within each policy iteration. You should be able to re-use much of your code from the previous problem.

Start off by initializing all of the same variables. Additionally, initialize your policy function array `PSI`, say

```
>>> PSI = 2*np.ones((N,N))
```

Next comes the iteration. Essentially, the iteration will consist of an outer while-loop (which terminates once the 2-norm distance between successive policy functions passes below 10^{-9}), and an inner for loop (with 15 loops).

The first step in the while-loop is to calculate the new policy function `PSI1`, just as in the previous problem. Next, perform the inner for-loop, which consists simply of the value function iteration, but this time using the current policy function. This means the line of code

```
>>> VU = uaw + beta*np.max(np.dstack([MEVU, MVE]), axis=2)
```

is no longer valid, as it does not use the policy function. We must instead have

```
>>> VU = uaw + beta*(MVE*PSI1 + MEVU*(1 - PSI1))
```

Why is this code correct?

Finally, after exiting the for loop, calculate the 2-norm distance between the old and the new policy function, and then update your old policy function, i.e.

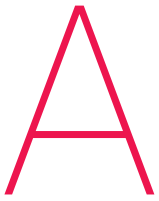
```
>>> PSI = PSI1
```

After convergence is achieved, once again compute the reservation wage array, and plot it as in the previous problem. Then return the converged policy function.

Problem 7. How many iterations did the value function iteration method take? How many iterations did the modified policy function iteration method take? Which was faster?

Part II

Appendices



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.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."

$$\texttt{A[1]} = \texttt{A[1,:]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[:,2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[1:,:2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[1:-1,1:-1]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

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

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

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad y = \begin{bmatrix} * & * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & * & * & * & * & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} \qquad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

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

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

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

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

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

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

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