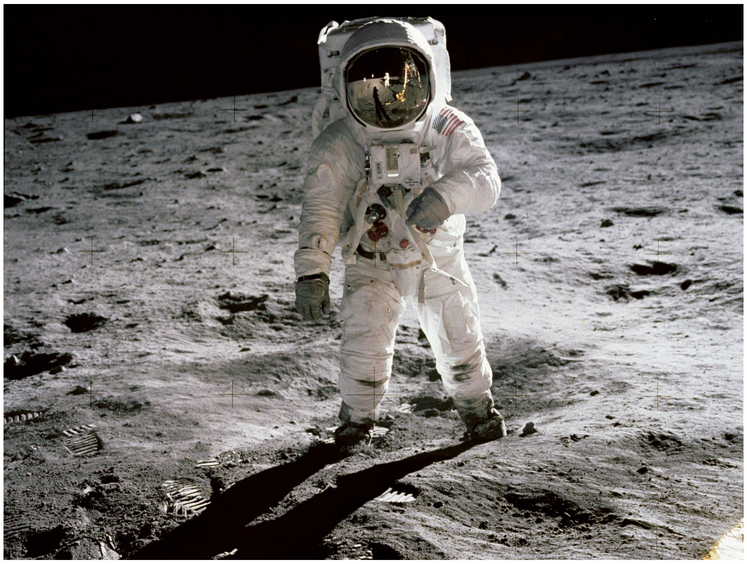
Labs for Foundations of Applied Mathematics

Volume 2
Algorithm Design and Optimization

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker
Brigham Young University

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

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

O. Escobar Rodriguez
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

S. Harding
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

J. Murphey
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

J. Oliphant
Brigham Young University

A. Oveson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

K. Sandall
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

N. Schill
Brigham Young University

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

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

Contents

Part I

Labs

1 Binary Search Trees

Lab Objective: *Linked data structures chain data together in a useful way for many applications. A tree is a linked data structure that can be used for efficient sorting and searching algorithms. In this lab, we overview the basics of linked data structures and recursion. We then implement a recursively structured doubly linked binary search tree (BST). Finally, we compare the standard linked list, our BST, and an AVL tree to illustrate the relative strengths and weaknesses of each data structure.*

Linked Data Structures

A *linked data structure* is a data structure which consists of a set of containers (*nodes*) that are linked together by *references*. Each node in a linked data structure stores a piece of data and at least one reference to another node in the data structure. Linked data structures offer flexibility in organizing data and are the basis of many efficient algorithms.

Linked Lists

A *linked list* is a basic example of linked data structure. Every linked list needs a reference to the first item in the chain, called the `head`. A reference to the last item in the chain, called the `tail`, is also often included. The nodes of a *singly linked list* have a single reference to the next node in the list (see Figure 1.1), while the nodes of a *doubly linked list* have two references: one for the previous node, and one for the next node (see Figure 1.2). This allows for a doubly linked list to be traversed in both directions, whereas a singly linked list can only be traversed in one direction.

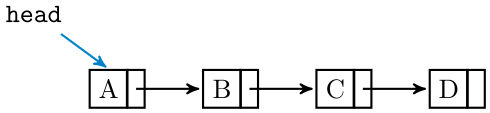


Figure 1.1: A singly linked list with just a reference to the head node.

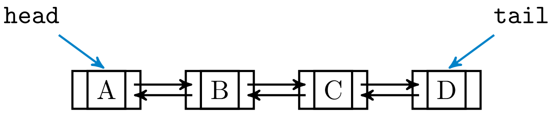


Figure 1.2: A doubly linked list with references to both the head and tail nodes.

Modifying Linked Lists

To insert a new node into a doubly linked list, we assign its `prev` and `next` attributes to reference the nodes that come before and after it. We then adjust the attributes of nodes that come before and after the new node to reference the new node (See Figure 1.3). To remove a node from a doubly linked list, we remove all references to that node. This is done by changing the attributes of the previous and next nodes so that they reference each other (See Figure 1.4).

Achtung!

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

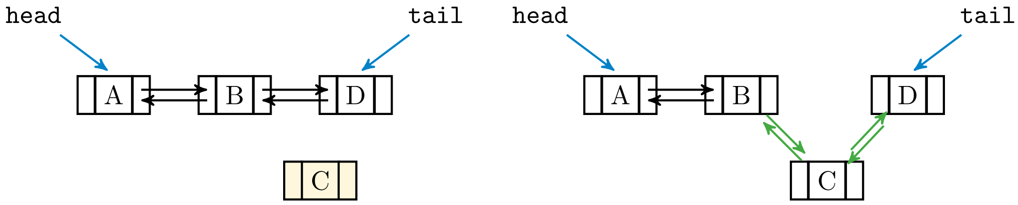


Figure 1.3: Insertion for doubly linked lists.

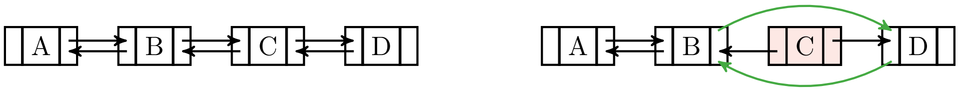


Figure 1.4: Removal for doubly linked Lists. To avoid gaps in the chain, nodes B and D must be linked together. Python will automatically delete node C since there are no references to it.

Problem 1. Consider the following class for doubly linked lists.

```python
class DoublyLinkedListNode:
    """A node with a value and a reference to the previous and next ↩
        nodes."""
    def __init__(self, data):
        self.value = data
        self.prev, self.next = None, None

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

    def __len__(self):
        '''Return the number of nodes in the list.'''
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next
        return count

    def __str__(self):
        '''Format and return the list like a standard Python list.'''
        result = []
        current = self.head
        while current:
            result.append(str(current.value))
            current = current.next
        return '[' + ', '.join(result) + ']'
```

Add an `insert()` method to the `DoublyLinkedList` class that accepts an integer `index` and `data` to add to the list. Insert a new node containing `data` immediately before the node in the list at position `index`. After the insertion, the new node should be at position `index`. For example, Figure 1.3 places a new node containing C at index 2. If `index` is equal to the number of nodes in the list, then add the new node to the end of the list. Carefully account for the special cases of inserting before the first node and after the last node. Make sure to properly reassign the `head` or `tail` attribute to reflect the new structure of the list. If the list is empty, make the new node both the head and tail. If `index` is negative or strictly greater than the number of nodes in the list, raise an `IndexError`.

Hint: The `__len__()` and `__str__()` methods have been adapted for this class. Use them in addition to the provided unit test for writing/debugging purposes. Also, attributes can be chained together for locating specific nodes (`self.head.next`, `self.prev.prev.value`, etc.)

Recursion

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

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

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

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

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

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

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

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

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

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

```python
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    if n <= 0:                  # Special case: g(0) = 0.
        return 0
    g0, g1 = 0, 1               # Initialize g(0) and g(1).
    for i in range(1, n):       # Calculate g(2), g(3), ..., g(n).
        g0, g1 = g1, g0 + g1
    return g1

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

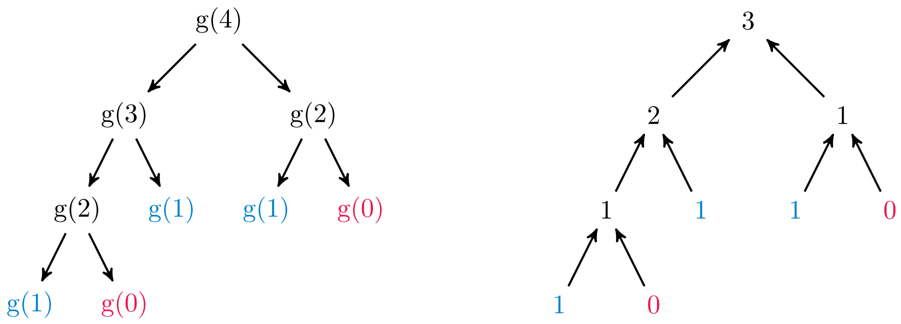


Figure 1.5: To calculate $g(n)$ recursively, call $g(n-1)$ and $g(n-2)$, down to the base cases $g(0)$ and $g(1)$. As the recursion unwinds, the values from the base cases are passed up to previous calls and combined, eventually giving the value for $g(n)$.

Problem 2. Consider the following method for doubly linked lists.

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

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

Note

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

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

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

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

Achtung!

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

Binary Search Trees

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

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

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

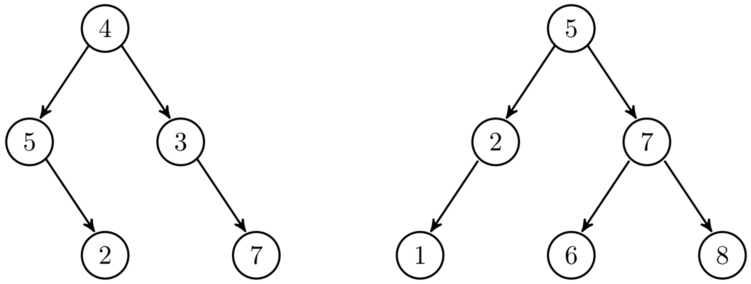


Figure 1.6: Both of these graphs are trees, but the tree on the left is not a binary search tree because 5 is to the left of 4. Swapping 5 and 3 in the graph on the left would result in a BST.

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

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

Note

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

Locating Nodes

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

```python
class BST:
    # ...
    def find(self, data):
        """Return the node containing the data. If there is no such node
        in the tree, including if the tree is empty, raise a ValueError.
        """

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

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

Insertion

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

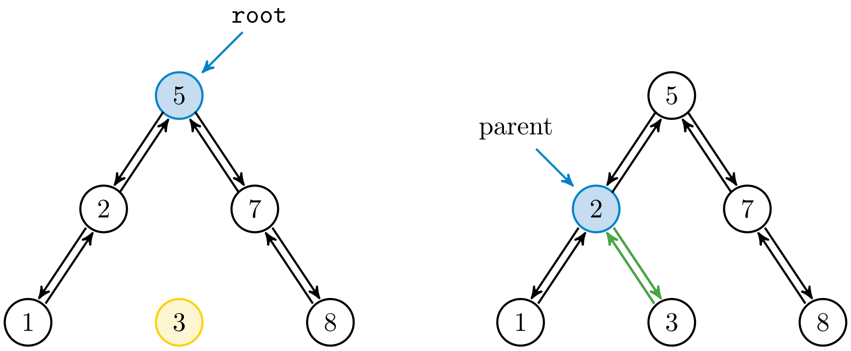


Figure 1.7: To insert 3 to the BST on the left, start at the root and recurse down the tree as if searching for 3: since $3 < 5$, step left to 2; since $2 < 3$, step right. However, 2 has no right child, so 2 becomes the parent of a new node containing 3.

Problem 3. Write an `insert()` method for the `BST` class that accepts some data.

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

To test your method, use the `__str__()` and `draw()` methods provided in the Additional Materials section. Try constructing the binary search trees in Figures 1.6 and 1.7.

Removal

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

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

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

Removing a Leaf Node

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

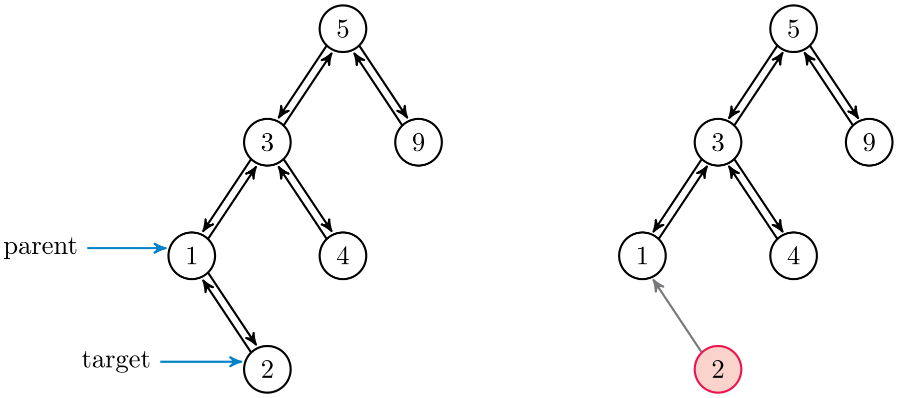


Figure 1.8: To remove 2, get a reference to its parent. Then set the parent's `right` attribute to `None`. Even though 2 still points to 1, 2 is deleted since nothing in the tree points to it.

Removing a Node with One Child

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

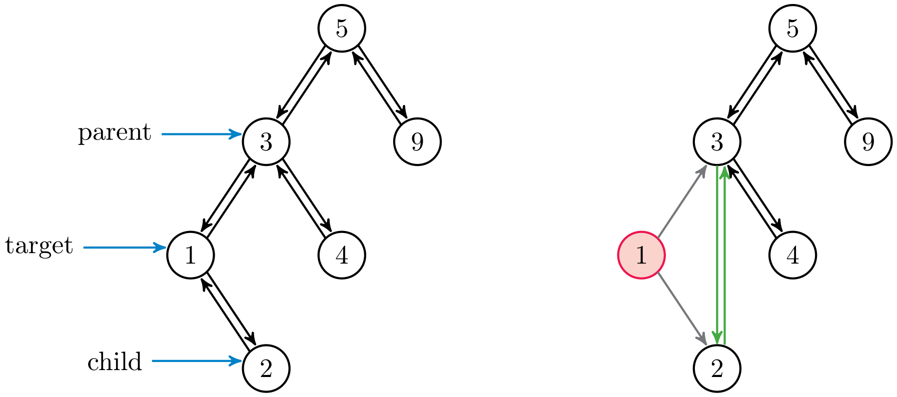


Figure 1.9: To remove 1, locate its parent (3) and its child (2). Set the parent's `left` attribute to the child and the child's `prev` attribute to the parent. Even though 1 still points to other nodes, it is deleted since nothing in the tree points to it.

Removing a Node with Two Children

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

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

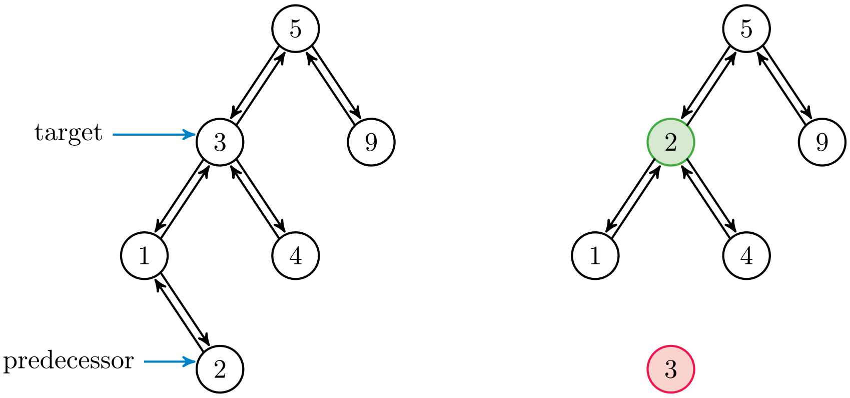


Figure 1.10: To remove 3, locate its immediate predecessor 2 by stepping left to 1, then right as far as possible. Since it is a leaf node, the predecessor can be deleted using the process in Figure 1.8. Delete the predecessor, and replace the value of the target with the predecessor's value. If the predecessor has a left child, it can be deleted with the procedure from Figure 1.9.

Removing the Root Node

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

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

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

Problem 4. Write a `remove()` method for the `BST` class that accepts some data. If the tree is empty, or if there is no node in the tree containing the data, raise a `ValueError`. Otherwise, remove the node containing the specified data using the strategies described in Figures 1.8–1.10. Test your solutions thoroughly.
(Hint: **Before coding anything**, outline the entire method with comments and `if-else` blocks. Consider using the following control flow to account for all possible cases.)

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

Unit Test

Write a unit test for Problem 4: creating a remove method for your BST class. The unit test is found in the file `test_binary_trees.py` and the function is called `test_bst_remove`.

There is an example unit test for Problem 3, your insert method, to help you structure your unit test.

AVL Trees

The advantage of a BST is that it organizes its data so that values can be located, inserted, or removed in $O(\log n)$ time. However, this efficiency is dependent on the *balance* of the tree. In a well-balanced tree, the number of descendants in the left and right subtrees of each node is about the same. An unbalanced tree has some branches with many more nodes than others. Finding a node at the end of a long branch is closer to $O(n)$ than $O(\log n)$. This is a common problem; inserting ordered data, for example, results in a "linear" tree, since new nodes always become the right child of the previously inserted node (see Figure 1.11). The resulting structure is essentially a linked list without a `tail` attribute.

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

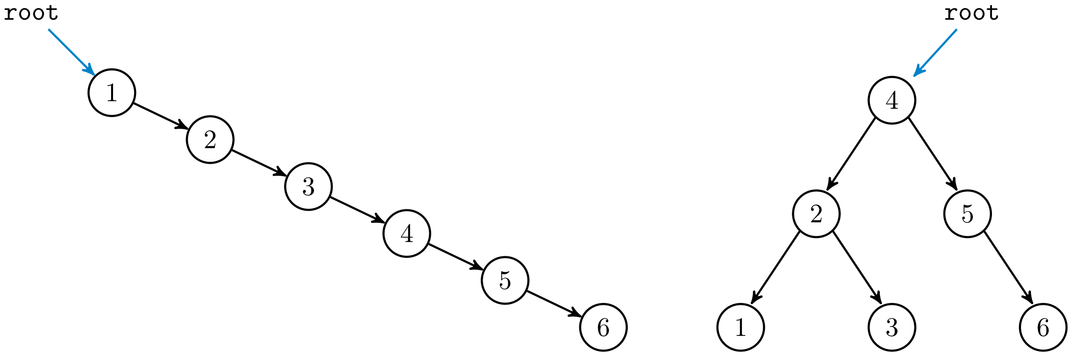


Figure 1.11: On the left, the unbalanced BST resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. On the left, the balanced AVL tree that results from the same insertion. After each insertion, the AVL tree rebalances if necessary.

Problem 5. Write a function to compare the build and search times of the `DoublyLinkedList` from Problem 1, the `BST` from Problems 3 and 4, and the `AVL` provided in the Additional Materials section. Begin by reading the file `english.txt`, storing the contents of each line in a list. For $n = 2^3, 2^4, \ldots, 2^{10}$, repeat the following experiment.

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

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

Additional Material

Possible Improvements to the BST Class

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

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

2. Add an attribute that keeps track of the number of items in the tree. Use this attribute to implement the `__len__()` magic method.

3. Add a method for translating the `BST` into a sorted Python list.
(Hint: examine the provided `__str__()` method carefully.)

4. Add methods `min()` and `max()` that return the smallest or largest value in the tree, respectively. Consider adding `head` and `tail` attributes that point to the minimal and maximal elements; this would make inserting new minima and maxima $O(1)$.

Other Kinds of Binary Trees

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

1. A *B-tree* is a tree whose nodes can contain more than one piece of data and point to more than one other node. See the Volume 2 text for details.

2. The nodes of a *red-black tree* are labeled either red or black. The tree satisfies the following rules to maintain a balanced structure.

   (a) Every leaf node is black.

   (b) Red nodes only have black children.

   (c) Every (directed) path from a node to any of its descendent leaf nodes contains the same number of black nodes.

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

3. A *Splay Tree* includes an additional operation, called splaying, that makes a specified node the root of the tree. Splaying several nodes of interest makes them easier to access because they are placed close to the root.

4. A *heap* is similar to a BST but uses a different binary sorting rule: the value of every parent node is greater than each of the values of its children. This data structure is particularly useful for sorting algorithms; see the Volume 2 text for more details.

Additional Code: Tree Visualization

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

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

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

        Example:  (3)
                  / \     '[3]          The nodes of the BST are printed
                (2) (5)    [2, 5]       by depth levels. Edges and empty
                /   / \    [1, 4, 6]'   nodes are not printed.
              (1) (4) (6)
        """
        if self.root is None:
            return "[]"
        out, current_level = [], [self.root]
        while current_level:
            next_level, values = [], []
            for node in current_level:
                values.append(node.value)
                for child in [node.left, node.right]:
                    if child is not None:
                        next_level.append(child)
            out.append(values)
            current_level = next_level
        return "\n".join([str(x) for x in out])

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

Additional Code: AVL Tree

Use the following class for Problem 5. Note that it inherits from the BST class, so its functionality is dependent on the insert() method from Problem 3. Note that the remove() method is disabled, though it is possible for an AVL tree to rebalance itself after removing a node.

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

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

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

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

    @staticmethod
    def _balance_factor(n):
        return AVL._height(n.right) - AVL._height(n.left)
    def _rotate_left_left(self, n):
        temp = n.left
        n.left = temp.right
        if temp.right:
            temp.right.prev = n
        temp.right = n
        temp.prev = n.prev
        n.prev = temp
        if temp.prev:
            if temp.prev.value > temp.value:
                temp.prev.left = temp
            else:
                temp.prev.right = temp
        if n is self.root:
            self.root = temp
        return temp

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

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

    def _rotate_right_left(self, n):
        temp1 = n.right
        temp2 = temp1.left
        temp1.left = temp2.right
        if temp2.right:
            temp2.right.prev = temp1
        temp2.prev = n
        temp2.right = temp1
        temp1.prev = temp2
        n.right = temp2
        return self._rotate_right_right(n)
```

2 Nearest Neighbor Search

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

The Nearest Neighbor Problem

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

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

Metrics and Distance

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

$$d(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\| = \sqrt{\sum_{i=1}^{k} (x_i - z_i)^2} \tag{2.1}$$

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

$$\mathbf{x}^* = \underset{\mathbf{x} \in X}{\operatorname{argmin}}\, d(\mathbf{x}, \mathbf{z}) \qquad\qquad d^* = \min_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{z}) \tag{2.2}$$

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

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

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

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

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

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

Exhaustive Search

Consider again the post office example. One way to find out which post office is closest is to drive from home to each post office, measuring the distance traveled in each trip. That is, we solve (2.2) by computing $\|\mathbf{x} - \mathbf{z}\|$ for every point $\mathbf{x} \in X$. This strategy is called a *brute force* or *exhaustive search*.

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

Solve (2.2) with an exhaustive search. Return the nearest neighbor $\mathbf{x}^*$ and its Euclidean distance d^* from the target $\mathbf{z}$.
(Hint: use array broadcasting and the `axis` argument to avoid using a loop.)

The complexity of an exhaustive search for $X \subset \mathbb{R}^k$ with m points is $O(km)$, since (2.1) is $O(k)$ and there are m norms to compute. This method works, but it is only feasible for relatively small training sets. Solving the problem with greater efficiency requires the use of a specialized data structure.

K-D Trees

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

A BST creates a partition of $\mathbb{R}$: if a node contains the value x, all of the nodes in its left subtree contain values that are less than x, and the nodes of its right subtree have values that are greater than x. Similarly, a k-d tree partitions $\mathbb{R}^k$. Each node is assigned a *pivot* value $i \in \{0, 1, \ldots, k-1\}$ corresponding to the depth of the node: the root has $i = 0$, its children have $i = 1$, their children have $i = 2$, and so on. If a node has $i = k-1$, its children have $i = 0$, their children have $i = 1$, and so on. The tree is constructed such that for a node containing $\mathbf{x} = [x_0, x_1, \ldots, x_{k-1}]^\mathsf{T} \in \mathbb{R}^k$, if a node in the left subtree contains $\mathbf{y}$, then $y_i < x_i$. Conversely, if a node in the right subtree contains $\mathbf{z}$, then $x_i \leq z_i$. See Figure 2.1 for an example where $k = 3$.

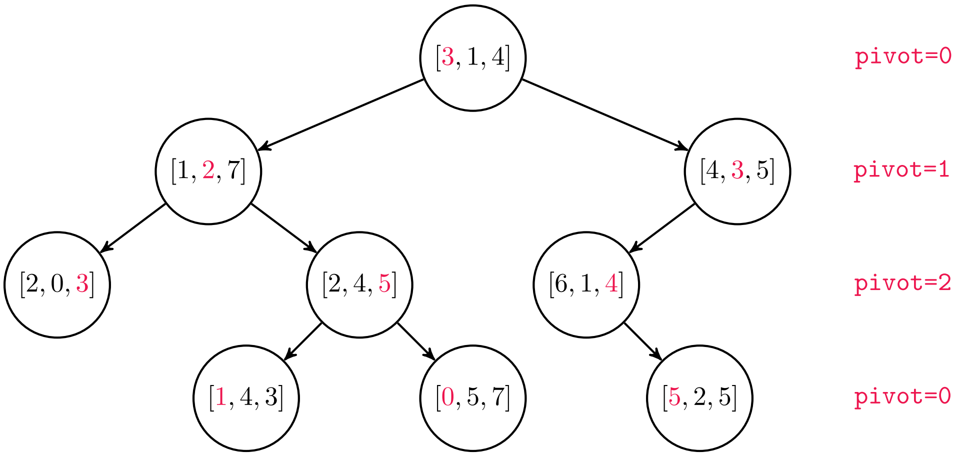


Figure 2.1: A k-d tree with $k = 3$. The root $[3, 1, 4]$ has an `pivot` of 0, so $[1, 2, 7]$ is to the left of the root because $1 < 3$, and $[4, 3, 5]$ is to the right since $3 \leq 4$. Similarly, the node $[2, 4, 5]$ has an `pivot` of 2, so $[1, 4, 3]$ is to its left since $4 < 5$ and $[0, 5, 7]$ is to its right because $5 \leq 7$. The nodes that are furthest from the root have an `pivot` of 0 because their parents have an `pivot` of $2 = k - 1$.

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

Unit Test

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

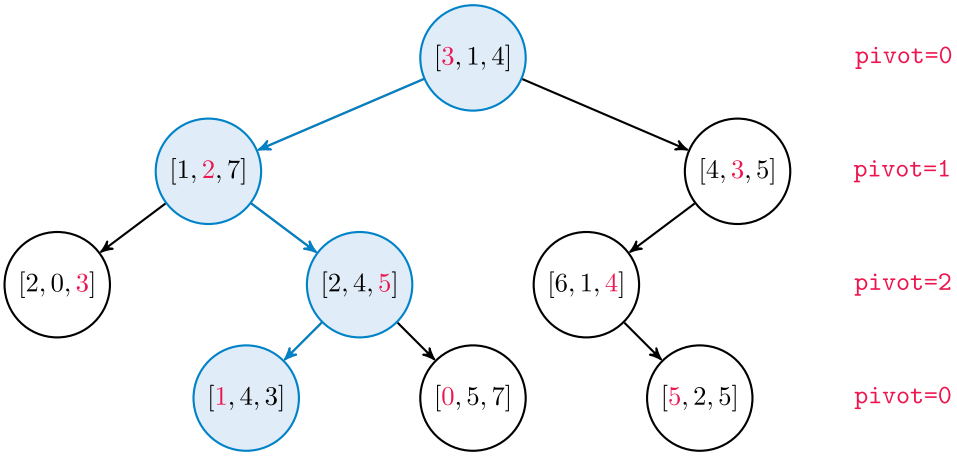


Figure 2.2: To locate the node containing $[1, 4, 3]$, start by comparing $[1, 4, 3]$ to the root $[3, 1, 4]$. The root has an `pivot` of 0, so compare the first component of the data to the first component of the root: since $1 < 3$, step left. Next, $[1, 4, 3]$ must be to the right of $[1, 2, 7]$ because $2 \leq 4$. Similarly, $[1, 4, 3]$ must be to the left of $[2, 4, 5]$ as $3 < 5$.

Constructing the Tree

Locating Nodes

The `find()` methods for k-d trees and binary search trees are very similar. Both recursively compare the values of a target and nodes in the tree, but in a k-d tree, these values must be compared according to their `pivot` attribute. Every comparison in the recursive `_step()` function, implemented below, compares the data of `target` and `current` based on the `pivot` attribute of `current`. See Figure 2.2.

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

    Attributes:
        root (KDTNode): the root node of the tree. Like all other nodes in
            the tree, the root has a NumPy array of shape (k,) as its value.
        k (int): the dimension of the data in the tree.
    """
    def __init__(self):
        """Initialize the root and k attributes."""
        self.root = None
        self.k = None

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

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

Inserting Nodes

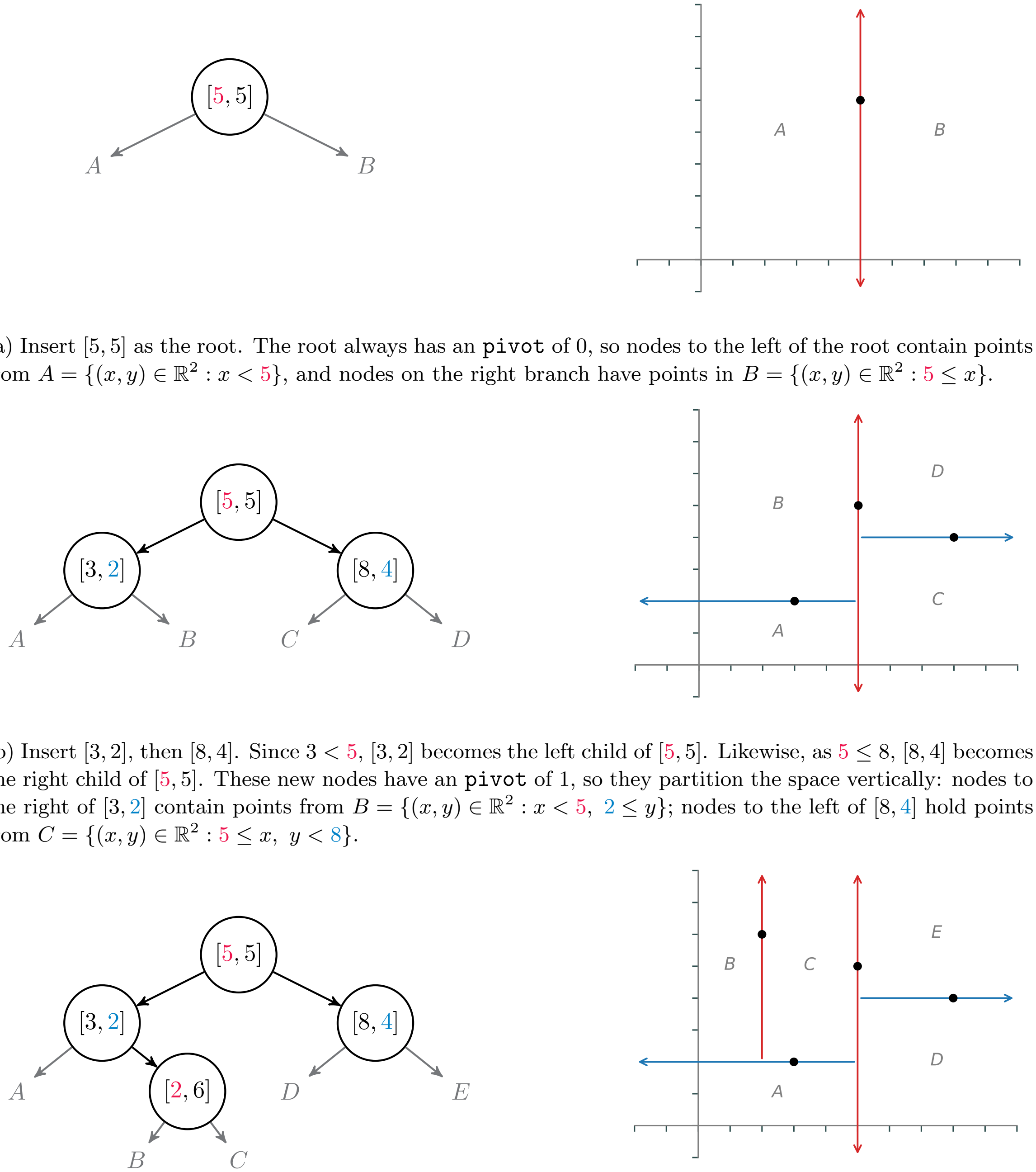
To add a new node to a k-d tree, determine which existing node should be the parent of the new node by recursively stepping down the tree as in the `find()` method. Next, assign the new node as the `left` or `right` child of the parent, and set its `pivot` based on its parent's `pivot`: if the parent's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

Consider again the k-d tree in Figure 2.2. To insert $[2,3,4]$, search the tree for $[2,3,4]$ until hitting an empty slot. In this case, the search steps from the root down to $[1,4,3]$, which has an `pivot` of 0. Then since $1 \leq 2$, the new node should be to the right of $[1,4,3]$. However, $[1,4,3]$ has no right child, so it becomes the parent of $[2,3,4]$. The `pivot` of the new node should therefore be 1. See Figure 2.3 for another example.

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

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

To test your method, use the `__str__()` method provided in the Additional Materials section. Try constructing the trees in Figures 2.1 and 2.3. Also check that the provided `find()` method works as expected.



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

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

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

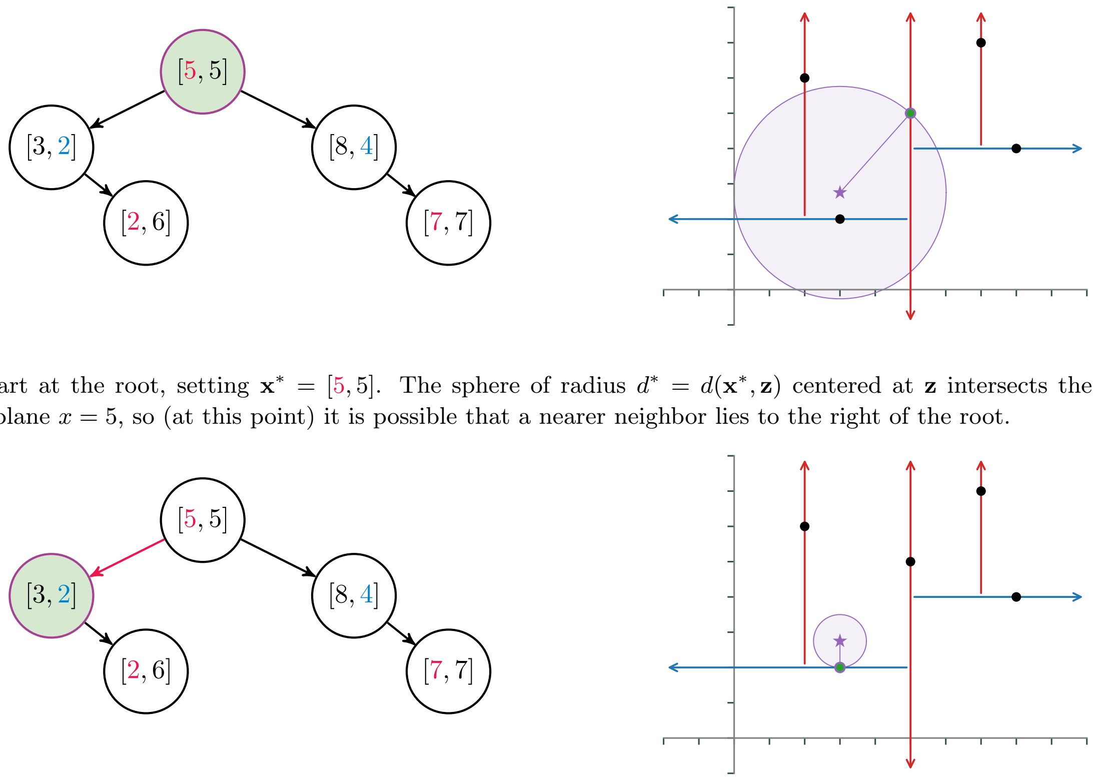
Figure 2.3: As a k-d tree is constructed (left), it creates a partition of $\mathbb{R}^k$ (right) by defining separating hyperplanes that pass through the points. The more points, the finer the partition.

Nearest Neighbor Search with K-D Trees

Given a target $\mathbf{z} \in \mathbb{R}^k$ and a k-d tree containing a set $X \subset \mathbb{R}^k$ of m points, the nearest neighbor problem can be solved by traversing the tree in a manner that is similar to the `find()` or `insert()` methods from the previous section. The advantage of this strategy over an exhaustive search is that not every $\mathbf{x} \in X$ has to be compared to $\mathbf{z}$ via (2.1); the tree structure makes it possible to rule out some elements of X without actually computing their distances to $\mathbf{z}$. The complexity is $O(k \log(m))$, a significant improvement over the $O(km)$ complexity of an exhaustive search.

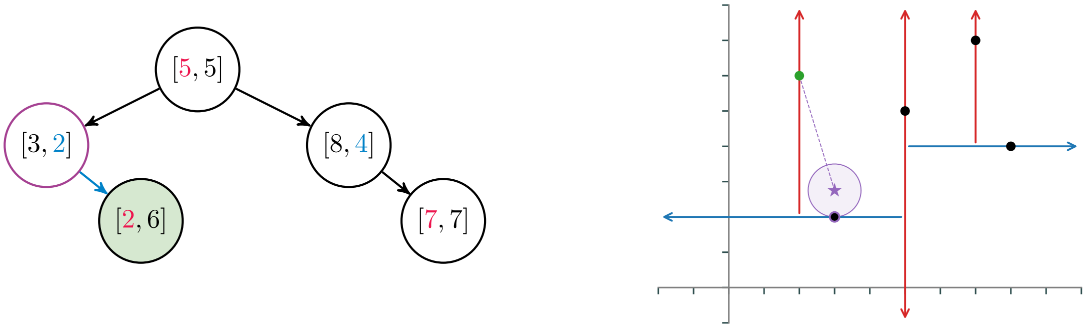
To begin, set $\mathbf{x}^*$ as the value of the root and compute $d^* = d(\mathbf{x}^*, \mathbf{z})$. Starting at the root, step down through the tree as if searching for the target $\mathbf{z}$. At each step, determine if the value $\mathbf{x}$ of the current node is closer to $\mathbf{z}$ than $\mathbf{x}^*$.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

Next, backtrack along the search path and determine if the non-explored branch needs to be searched. To do this, check that the sphere of radius d^* centered at $\mathbf{z}$ does not intersect with the separating hyperplane defined by the current node. That is, if the separating hyperplane is further than d^* from $\mathbf{z}$, then no points on the other side of the hyperplane can possibly be the nearest neighbor. See Figure 2.4 for an example and Algorithm 1 for the details of the procedure.



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

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



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

Figure 2.4: Nearest neighbor search of a k-d tree with $k = 2$. The target is $\mathbf{z} = [3, 2.75]$ and the nearest neighbor is $\mathbf{x}^* = [3, 2]$ with minimal distance $d^* = 0.75$. The tree structure allows the algorithm to eliminate $[8, 4]$ and $[7, 7]$ from consideration without computing their distance from $\mathbf{z}$.

Algorithm 1 k-d tree nearest neighbor search

```
procedure NEAREST NEIGHBOR SEARCH($\mathbf{z}$, root)
    procedure KDSEARCH(current, nearest, $d^*$)
        if current is None then                         ▷ Base case: dead end.
            return nearest, $d^*$
        $\mathbf{x} \leftarrow$ current.value
        $i \leftarrow$ current.pivot
        if $d(\mathbf{x}, \mathbf{z}) < d^*$ then       ▷ Check if current is closer to $\mathbf{z}$ than nearest.
            nearest $\leftarrow$ current
            $d^* \leftarrow d(\mathbf{x}, \mathbf{z})$
        if $z_i < x_i$ then                             ▷ Search to the left.
            nearest, $d^* \leftarrow$ KDSearch(current.left, nearest, $d^*$)
            if $z_i + d^* \geq x_i$ then                ▷ Search to the right if needed.
                nearest, $d^* \leftarrow$ KDSearch(current.right, nearest, $d^*$)
        else                                            ▷ Search to the right.
            nearest, $d^* \leftarrow$ KDSearch(current.right, nearest, $d^*$)
            if $z_i - d^* \leq x_i$ then                ▷ Search to the left if needed.
                nearest, $d^* \leftarrow$ KDSearch(current.left, nearest, $d^*$)
        return nearest, $d^*$
    node, $d^* \leftarrow$ KDSearch(root, root, $d$(root.value, $\mathbf{z}$))
    return node.value, $d^*$
```

Problem 4. Write a method for the `KDT` class that accepts a target point $\mathbf{z} \in \mathbb{R}^k$. Use Algorithm 1 to solve (2.2). Return the nearest neighbor $\mathbf{x}^*$ (the actual NumPy array, not the `KDTNode`) and its distance d^* from $\mathbf{z}$.

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

```
>>> from scipy.spatial import KDTree

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

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

Achtung!

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

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

K-Nearest Neighbors

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

A *k-nearest neighbors* classifier uses a simple strategy to label an arbitrary $\mathbf{z} \in D$: find the k elements of X that are nearest to $\mathbf{z}$ (usually in terms of the Euclidean metric) and choose the most common label from those k elements as the label of $\mathbf{z}$. That is, the points in the k labeled points that are most like $\mathbf{z}$ are allowed to "vote" on how $\mathbf{z}$ should be labeled. See Figure 2.5.

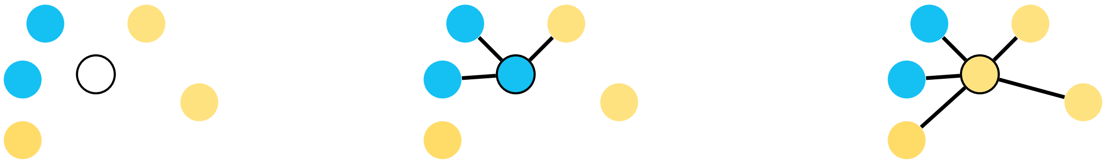


Figure 2.5: To classify the center node, determine its k-nearest neighbors and pick the most common label of the neighbors. If $k = 3$, the k nearest points are two blues and a yellow, so the center node is labeled blue. For $k = 5$, the k nearest points consists of two blues and three yellows, so the center node is labeled yellow.

Achtung!

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

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

1. The constructor should accept an integer `n_neighbors`, the number of neighbors to include in the vote (the k in k-nearest neighbors). Save this value as an attribute.

2. `fit()`: accept an $m \times k$ NumPy array X (the training set) and a 1-dimensional NumPy array $\mathbf{y}$ with m entries (the training labels). As in Problems 1 and 4, each of the m rows of X represents a point in $\mathbb{R}^k$. Here y_i is the label corresponding to row i of X.

   Load a SciPy `KDTree` with the data in X. Save the tree and the labels as attributes.

3. `predict()`: accept a 1-dimensional NumPy array **z** with k entries. Query the `KDTree` for the `n_neighbors` elements of X that are nearest to **z** and return the most common label of those neighbors. If there is a tie for the most common label (such as if $k = 2$ in Figure 2.5), choose the alphanumerically smallest label.
   (Hint: use `scipy.stats.mode()`. The default behavior splits ties correctly.)

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

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

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

Note

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

Handwriting Recognition

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

Problem 6. The file `mnist_subset.npz` contains part of the MNIST dataset,[a] a collection of 28×28 images of handwritten digits and their labels. The data is split into four parts.

- `X_train`: A 3000×728 matrix, the training set. Each of the 3000 rows is a flattened 28×28 image to be used in training the classifier.
- `y_train`: A 1-dimensional NumPy array with 3000 entries. The entries are integers from 0 to 9, the labels corresponding to the images in `X_train`.
- `X_test`: A 500×728 matrix of 500 images to classify.

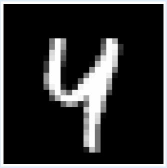
- y_test: A 1-dimensional NumPy array with 500 entries. These are the labels corresponding to X_test, the "right answers" that the classifier will try to guess.

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

```
>>> data = np.load("mnist_subset.npz")
>>> X_train = data["X_train"].astype(np.float64)          # Training data
>>> y_train = data["y_train"]                            # Training labels
>>> X_test = data["X_test"].astype(np.float64)            # Test data
>>> y_test = data["y_test"]                              # Test labels
```

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

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



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

[a] See http://yann.lecun.com/exdb/mnist/.

Note

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

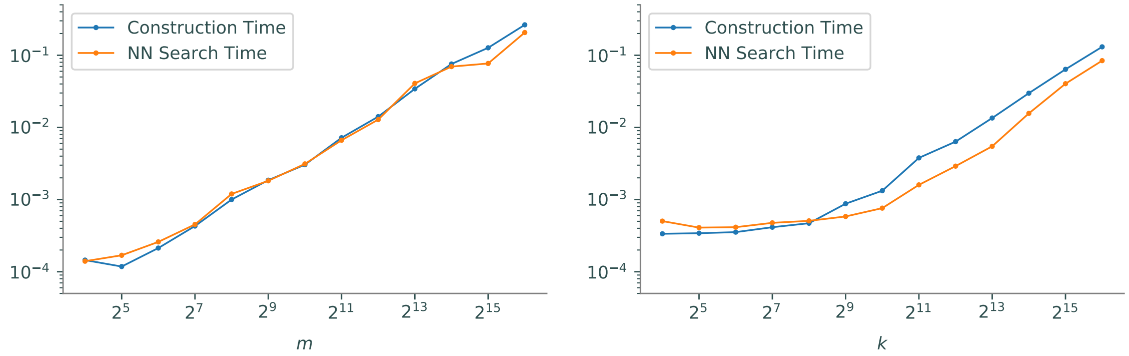
Additional Material

Ball Trees

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

The Curse of Dimensionality

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



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

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

Figure 2.6: Construction and nearest neighbor search times for a k-d tree with a $m \times k$ training set.

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

Tiebreaker Strategies

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

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

Additional Code

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

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

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

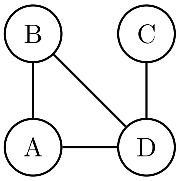
3 Breadth-first Search

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

Adjacency Dictionaries

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

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



$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array}\right] \end{array}$$

$$\begin{aligned} \{A:\ & \{B,\ D\}, \\ B:\ & \{A,\ D\}, \\ C:\ & \{D\}, \\ D:\ & \{A,\ B,\ C\}\} \end{aligned}$$

Figure 3.1: A simple unweighted graph (left), its adjacency matrix (middle), and its adjacency dictionary (right). The graph is undirected, so the adjacency matrix is symmetric. Note that the adjacency dictionary also encodes this behavior: since A and B are connected, B is in the set of values corresponding to the key A, and A is in the set of values corresponding to the key B.

Hash-based Data Structures

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

Method	Description
add()	Add an element to the set. This has no effect if the element is already present.
remove()	Remove an element from the set, raising a KeyError if it is not a member of the set.
discard()	Remove an element from the set without raising an exception if it is not a member of the set.
pop()	Remove and return an arbitrary set element.
union()	Return all elements that are in either set as a new set.
intersection()	Return all elements that are in both sets as a new set.
update()	Add all elements of another set in-place.

Table 3.1: Basic methods of the set class.

```
# Initialize a set. Note that repeats are not added.
>>> animals = {"cow", "cat", "dog", "mouse", "cow"}
>>> print(animals)
{'cow', 'dog', 'mouse', 'cat'}

>>> animals.add("horse")     # Add an object to the set.
>>> "horse" in animals
True
>>> animals.remove("emu")    # Attempt to delete an object from the set,
KeyError: 'emu'              # resulting in an exception.
>>> animals.pop()            # Delete and return a random object from the set.
'mouse'
>>> print(animals)
{'cat', 'horse', 'dog', 'cow'}

# Add all of the elements of another set to this one.
>>> animals.update({"dog", "velociraptor"})
>>> print(animals)
{'velociraptor', 'cat', 'horse', 'dog', 'cow'}

# Intersect this set with another one.
>>> animals.intersection({"cat", "cow", "cheetah"})
{'cat', 'cow'}
```

Sets are extremely fast, but they do not support indexing because the elements are unordered. A Python dict, on the other hand, is a hash-based data structure that stores key-value pairs: the keys of a dictionary act like a set (unique and unordered, with $O(1)$ lookup), but each key corresponds to another object, called its value. The keys index the dictionary and allow $O(1)$ lookup of the values.

Method	Description
`keys()`	Return a set-like iterator for the dictionary's keys.
`values()`	Return a set-like iterator for the dictionary's values.
`items()`	Return an iterator for the dictionary's key-value pairs.
`pop()`	Remove a specified key and return the corresponding value, raising a KeyError if the key is not a member of the dictionary.
`update()`	Add or overwrite key-value pairs in-place with those from another dictionary.

Table 3.2: Basic methods of the dict class.

```
# Initialize a dictionary.
>>> grades = {"business": "A", "math": "A+", "visual arts": "B"}
>>> grades["math"]
'A+'                            # The key "math" maps to the value "A+".

# Add a "science" key with corresponding value "A".
>>> grades["science"] = "A"

# Remove the "business" key.
>>> grades.pop("business")
'A'
>>> print(grades)
{'math': 'A+', 'visual arts': 'B', 'science': 'A'}

# Display the keys, values, and items.
>>> list(grades.keys()), list(grades.values())
(['math', 'visual arts', 'science'], ['A+', 'B', 'A'])
>>> for key, value in grades.items():
...   print(key, "=>", value)
...
math => A+
visual arts => B
science => A

# Add key-value pairs from another dictionary.
>>> grades.update({"cooking":"A+", "math": "C"})
>>> print(grades)
{'math': 'C', 'visual arts': 'B', 'science': 'A', 'cooking': 'A+'}
```

Dictionaries are ideal for storing values that need to be accessed often and for representing one-to-one or one-to-many relationships. Thus, the dict class is a natural choice for implementing adjacency dictionaries. For example, the following code defines the adjacency dictionary for the graph in Figure 3.1. Note that the dictionary values are sets.

```
>>> adjacency = {'A': {'B', 'D'},
                 'B': {'A', 'D'},
                 'C': {'D'},
                 'D': {'A', 'B', 'C'}}

# The nodes of the graph are the dictionary keys.
>>> set(adjacency.keys())
{'B', 'D', 'A', 'C'}

# The values are the nodes that the key node is adjacent to.
>>> adjacency['A']
{'B', 'D'}                  # A is adjacent to B and D.
>>> 'C' in adjacency['B']
False                       # B and C are not adjacent.
>>> 'C' in adjacency['D']
True                        # C and D are adjacent.
```

Achtung!

Elements of a `set` and keys of a `dict` must be *hashable*. Mutable objects—lists, sets and dictionaries—are not hashable, so they are not allowed as set elements or dictionary keys. Thus, in order to represent a graph with an adjacency dictionary, each of the node labels should be a string, a number, or some other hashable type.

Problem 1. Consider the following `Graph` class.

```
class Graph:
    """A graph object, stored as an adjacency dictionary. Each node in the
    graph is a key in the dictionary. The value of each key is a set of
    the corresponding node's neighbors.

    Attributes:
        d (dict): the adjacency dictionary of the graph.
    """
    def __init__(self, adjacency={}):
        """Store the adjacency dictionary as a class attribute"""
        self.d = dict(adjacency)

    def __str__(self):
        """String representation: a view of the adjacency dictionary."""
        return str(self.d)
```

Add the following methods to this class.

1. `add_node()`: Add a node (with no initial edges) if it is not already present. (Hint: use `set()` to create an empty set.)
2. `add_edge()`: Add an edge between two nodes. Add the nodes to the graph if they are not already present.
3. `remove_node()`: Remove a node, including all edges adjacent to it. This method should raise a `KeyError` if the node is not in the graph.
4. `remove_edge()`: Remove the edge between two nodes. This method should raise a `KeyError` if either node is not in the graph, or if there is no edge between the nodes.

Breadth-first Search

Many common problems that arise in graph theory require finding the shortest path between two nodes in a graph. For some highly structured graphs, such as binary search trees, this is a fairly straightforward problem (in the case of a tree, the shortest path is also the only path). Finding a path between nodes in a graph of arbitrary structure, however, requires a careful and methodical approach. The two most common graph search algorithms are *depth-first search* (DFS) and *breadth-first search* (BFS). The breadth-first strategy is almost always better at finding shortest paths than the depth-first strategy,[1] though a DFS can be useful for path problems in certain graphs.

To traverse a graph with a BFS, choose a node to start at, called the *source* node. First, visit each of the source node's neighbors. Next, visit each of the source node's neighbors' neighbors. Then visit each of their neighbors, continuing the process until all nodes have been visited. This strategy explores all of the nodes closest to the source node before incrementally moving "deeper" (further from the source node) into the tree.

The implementation of a BFS requires the following data structures to keep track of which nodes have already been visited and the order in which to visit nodes in future steps.

- A list V: The nodes that have been **visited**, in visitation order.
- A **queue** Q: The nodes to be visited, in the order that they were discovered. Recall that a *queue* is a limited-access list where data is inserted to one end, but removed from the other (first-in, first-out).
- A set M: The nodes that have been visited, or that are **marked** to be visited. This is the union of the nodes in V and Q.

To begin the search, add the source node to Q and M. Then, until Q is empty, repeat the following:

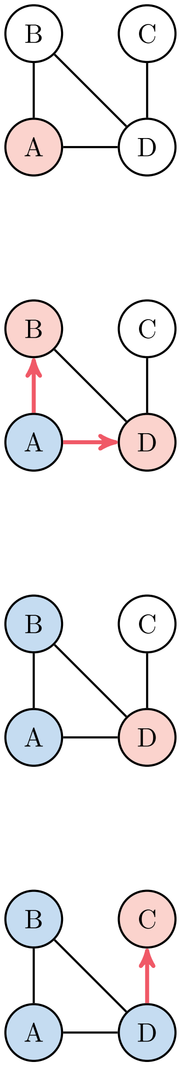
1. Pop a node off of Q; call it the *current* node.
2. "Visit" the current node by appending it to V.
3. Add the neighbors of the current node that are not in M to Q and M.

The "that are not in M" clause of step 3 prevents nodes from being added to Q more than once. Note that step 3 could be replaced with "Add the neighbors of the current node that are not in $V \cup Q$ to Q." However, lookup in M (a set) is much faster than lookup in V and Q (arrays or linked lists), so including M greatly speeds up the algorithm.

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

Note

The first-in, first-out (FIFO) structure of Q enforces the “breadth-first” nature of the BFS: nodes that are marked first are visited first. Using a a last-in, first-out (LIFO) stack for Q changes the search to a DFS: the next node to visit is the one that was marked last.



V	
Q	A
M	A

V	A
Q	B D
M	A B D

V	A B
Q	D
M	A B D

V	A B D
Q	C
M	A B D C

Figure 3.2: To start a BFS from node A to node C, put A in the visit queue Q and mark it by adding it to the set M. Pop A off the queue and “visit” it by adding A to the visited list V and the neighboring nodes B and D to Q. Then visit B, but do not add anything to Q because all of the neighbors of B are already marked. Finally, visit D, at which point the target node C is located because it is adjacent to D.

Problem 2. Write a method for the `Graph` class that accepts a source node. Traverse the graph with a breadth-first search until all nodes have been visited. Return the list of nodes in the order that they were visited. If the source node is not in the graph, raise a `KeyError`.
(Hint: for Q, use a `deque` from the `collections` module, and make sure that nodes are added to one end but popped off of the other.)

Shortest Paths via BFS

Consider the problem of locating a path between two nodes with a BFS. The nodes that are directly connected to the source node are all visited before any other nodes; more generally, the nodes that are n nodes away from the source node are all visited before nodes that are $n+1$ or more nodes from the source point. Therefore, the search path taken to discover to the target with a BFS must be the shortest path from the source node to the target node.

Examine again the graph in Figures 3.1 and 3.2. The shortest path from A to C starts at A, goes to D, and ends at C. During a BFS originating at A, D is placed on the visit queue because it is one of A's neighbors, and C is placed on the queue because it is one of D's neighbors. Given that A was the node that visited D, and that D was the node that visited C, the shortest path from A to C can be constructed by stepping backward along the search path.

To implement this idea, initialize a dictionary before starting the BFS. When a node is marked and added to the visit queue, add a key-value pair mapping the **visited** node to the **visiting** node (for example, B $\mapsto$ A means B was marked while visiting A). When the target node is found, step through the dictionary until arriving at the source node, recording each step.

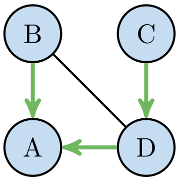


Figure 3.3: In the BFS from Figure 3.2, nodes B and D were marked while visiting node A, and node C was marked while visiting node D (this is same as reversing the red arrows in Figure 3.2). Thus the "visit path" from C to A is C $\rightarrow$ D $\rightarrow$ A, so the shortest path from A to C is [A, D, C].

Problem 3. Add a method to the `Graph` class that accepts source and target nodes. Begin a BFS at the source node and proceed until the target is found. Return a list containing the node values in the shortest path from the source to the target (including the endpoints). If either of the input nodes are not in the graph, raise a `KeyError`.

Shortest Paths via NetworkX

NetworkX is a Python package for creating, manipulating, and exploring graphs. Its `Graph` object represents a graph with an adjacency dictionary, similar to the class from Problems 1–3, and has many methods for interpreting information about the graph and its structure. As before, the nodes must be hashable (a number, string, or another immutable object).

Method	Description
`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.

Table 3.3: Methods of the `nx.Graph` class for adding nodes and edges.

```
>>> import networkx as nx

# Initialize a NetworkX graph from an adjacency dictionary.
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

>>> print(G.nodes())            # Print the nodes.
['A', 'B', 'C', 'D']
>>> print(G.edges())            # Print the edges as tuples.
[('A', 'D'), ('A', 'B'), ('B', 'D'), ('C', 'D')]

>>> G.add_node('E')             # Add a new node.
>>> G.add_edge('A', 'F')        # Add an edge, which also adds a new node 'F'.
>>> G.add_edges_from([('A', 'C'), ('F', 'G')])  # Add several edges at once.

>>> set(G['A'])                 # Get the set of nodes neighboring node 'A'.
{'B', 'C', 'D', 'F'}
```

The Kevin Bacon Problem

The vintage parlor game *Six Degrees of Kevin Bacon* is played by naming an actor, then trying to find the shortest chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Pulp Fiction (1994)* with Frank Whaley, who was in *JFK (1991)* with Kevin Bacon. In other words, the goal of the game is to solve a shortest path problem on a graph that connects actors to the movies that they have been in.

Problem 4. The file `movie_data.txt` contains IMDb data for about 137,000 movies. Each line of the file represents one movie: the title is listed first, then the cast members, with entries separated by a / character. For example, the line for *The Dark Knight (2008)* starts with

```
The Dark Knight (2008)/Christian Bale/Heath Ledger/Aaron Eckhart/...
```

Any / characters in movie titles have been replaced with the vertical pipe character | (for example, `Frost|Nixon (2008)`).

Write a class whose constructor accepts the name of a file to read. Initialize a set for movie titles, a set for actor names, and an empty NetworkX `Graph`, and store them as attributes. Read the file line by line, adding the title to the set of movies and the cast members to the set of actors. Add an edge to the graph between the movie and each cast member.
(Hint: Use the `split()` method for strings to parse each line.)

It should take no more than 20 seconds to construct the entire graph. Check that there are 137,018 movies and 930,717 actors. Compare parts of your graph to Figure 3.4.

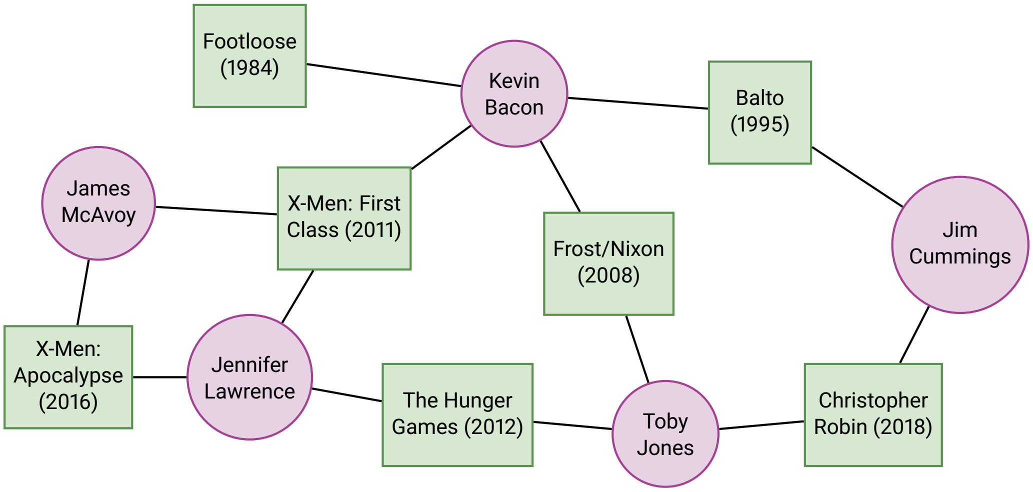


Figure 3.4: A subset of the graph in `movie_data.txt`. Each of these actors have a Bacon number of 1 because they have all been in a movie with Kevin Bacon. Every actor in *The Hunger Games* has a Bacon number of at most 2 because of the paths through Jennifer Lawrence or Toby Jones.

Note

The movie/actor graph of Problem 4 and Figure 3.4 has an interesting property: actors are only directly connected to movies, and movies are only directly connected to actors. This kind of graph is called *bipartite* because there are two types of nodes, and no node has an edge connecting it to another node of its type.

Achtung!

NetworkX `Graph` objects can be visualized with `nx.draw()` (followed by `plt.show()`). However, this visualization tool is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research. Because of the size of the dataset, **do not** attempt to visualize the graph in Problem 4 with `nx.draw()`.

The Six Degrees of Kevin Bacon game poses an interesting question: can any actor be linked to Kevin Bacon, and if so, in how many steps? The game hypothesizes, "Yes, within 6 steps" (hence the title). More precisely, let the *Bacon number* of an actor be the number of steps from that actor to Kevin Bacon, only counting actors. For example, since Samuel L. Jackson was in a film with Frank Whaley, who was in a film with Kevin Bacon, Samuel L. Jackson has a Bacon number of 2. Actors who have been in a movie with Kevin Bacon have a Bacon number of 1, and actors with no path to Kevin Bacon have a Bacon number of ∞. The game asserts that the largest Bacon number is 6.

NetworkX is equipped with a variety of graph analysis tools, including a few for computing paths between nodes (and, therefore, Bacon numbers). To compute a shortest path between nodes u and v, `nx.shortest_path()` starts one BFS from u and another from v, switching off between the two searches until they both discover a common node. This approach is called a *bidirectional BFS* and is typically faster than a regular, one-sided BFS.

Function	Description
`has_path()`	Return `True` if there is a path between two specified nodes.
`shortest_path()`	Return **one** shortest path between nodes.
`shortest_path_length()`	Return the length of the shortest path between nodes.
`all_shortest_paths()`	Yield **all** shortest paths between nodes.

Table 3.4: NetworkX functions for path problems. Each accepts a `Graph`, then a pair of nodes.

```
>>> G = nx.Graph({'A': {'B', 'D'},
                  'B': {'A', 'D'},
                  'C': {'D'},
                  'D': {'A', 'B', 'C'}})

# Compute the shortest path between 'A' and 'D'.
>>> nx.has_path(G, 'A', 'C')
True
>>> nx.shortest_path(G, 'A', 'C')
['A', 'D', 'C']
>>> nx.shortest_path_length(G, 'A', 'C')
2

# Compute all possible shortest paths between two nodes.
>>> G.add_edge('B', 'C')
>>> list(nx.all_shortest_paths(G, 'A', 'C'))
[['A', 'D', 'C'], ['A', 'B', 'C']]

# When the second node is omitted from these functions, the shortest paths
# from the given node to EVERY node are computed and returned as a dictionary.
>>> nx.shortest_path(G, 'A')
{'A': ['A'], 'D': ['A', 'D'], 'B': ['A', 'B'], 'C': ['A', 'D', 'C']}
>>> nx.shortest_path_length(G, 'A')    # Path lengths are defined by the
{'A': 0, 'D': 1, 'B': 1, 'C': 2}       #  number of edges, not nodes.
```

Problem 5. Write a method for your class from Problem 4 that accepts two actors' names. Use NetworkX to compute the shortest path between the actors and the degrees of separation between the two actors (if one of the actors is `"Kevin Bacon"`, this is the Bacon number of the other actor). Note that this number is different than the number of entries in the actual shortest path list, since the movies are just intermediate steps between actors and are not counted when calculating the degress of separation.

The idea of a Bacon number provides a few ways to analyze the connectivity of the Hollywood network. For example, the distribution of all Bacon numbers describes how close Kevin Bacon is to actually knowing all of the actors in Hollywood Someone with a lower average number—for instance, the average *Jackson number*, for Samuel L. Jackson—is, on average, "more connected with Hollywood" than Kevin Bacon. The actor with the lowest average number is sometimes called *the center of the Hollywood universe*.

Problem 6. Write a method for your class from Problem 4 that accepts one actor's name. Calculate the shortest path lengths of every actor in the collection to the specified actor (not including movies). Use `plt.hist()` to plot the distribution of path lengths and return the average path length.
(Hint: Use a NetworkX function to compute all path lengths simultaneously; this is significantly faster than calling your method from Problem 5 repeatedly. Also, use the keyword argument `bins=[i-.5 for i in range(8)]` in `plt.hist()` to get the histogram bins to correspond to integers nicely.)

As an aside, the prolific Paul Erdős is the Kevin Bacon equivalent in the mathematical community. Someone with an *Erdős number* of 2 co-authored a paper with someone who co-authored a paper with Paul Erdős. Having an Erdős number of 1 or 2 is considered quite an achievement (see `https://xkcd.com/599/`).

Additional Material

Other Hash-based Structures

The standard library has a few specialized alternatives to regular sets and dictionaries.

- `frozenset`: an immutable version of the usual set class. Frozen sets cannot be altered after creation and therefore lack methods like `add()`, `pop()`, and `remove()`, but they can be placed in other sets and used as dictionary keys.

- `collections.defaultdict`: a dictionary with default values. For instance, `defaultdict(set)` creates a dictionary that automatically uses an empty set as the value whenever a non-present key is used for indexing. See `https://docs.python.org/3/library/collections.html` for examples.

- `collections.OrderedDict`: a dictionary that remembers insertion order. For example, the `popitem()` method returns the most recently added key-value pair.

Depth-first Search

A *depth-first search* (DFS) takes the opposite approach of a BFS. Instead of checking all neighbors of a single node before moving on, it checks the first neighbor, then their first neighbor, then their first neighbor, and so on until reaching a leaf node. The algorithm then backtracks to the previous node and checks its second neighbor. While a DFS is rarely useful for finding shortest paths, it is a common strategy for solving recursively structured problems, such as mazes or Sudoku puzzles.

Consider adding a keyword argument to your method from Problem 2 that specifies whether to use a BFS (the default) or a DFS. To change from a BFS to a DFS, change the visit queue Q to a stack. You may be able to implement the change in a single line of code.

The Center of the Hollywood Universe

Computing the center of the universe in a graph amounts to solving Problem 6 for every node in the graph. This is computationally expensive, but since each average number is independent of the others, the problem is a good candidate for *parallel programming*, which divides the computational workload between multiple processes. Even with parallelism, however, computing the center of the Hollywood universe may require significant computational time and resources.

Shortest Paths on Weighted Graphs

The graphs presented in this lab are *unweighted*, meaning all edges have the same importance. A *weighted graph* assigns a weight to each edge, which can usually be thought of as the distance between the two adjacent nodes. The shortest path problem becomes much more complicated on weighted graphs, and requires additions to the plain BFS. The standard approach is *Dijkstra's algorithm*, which is implemented as `nx.dijkstra_path()`. Another approach, the *Bellman-Ford algorithm*, is implemented as `nx.bellman_ford_path()`.

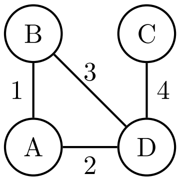
4 Dijkstra's Algorithm

Lab Objective: *We've previously seen the power of shortest path algorithms for solving network problems (e.g. Breadth-First Search). However, many real-world problems incorporate more than just connections between nodes. Networks will often incorporate weights between nodes, which can encode data such as distance, cost, or time. When this is the case, BFS algorithms are often insufficient to find the most effective (least weighted) path between nodes. In this lab, we introduce weighted graphs, building on the Graph class from the BFS lab, discuss how Dijkstra's algorithm can find the shortest path on weighted graphs, and apply Dijkstra's algorithm to a real-world dataset of bathymetry (ocean depth) data.*

Weighted Graphs

Many networks are stored as *weighted graphs*, which not only encode edges between nodes, but weights on those edges. These weights can be thought of as the "cost of travel" between node X and node Y. Weighted graphs work very similarly to unweighted graphs, and can be stored in much the same way by using an *adjacency matrix* or *adjacency dictionary.*

The key differences between unweighted graphs and weighted graphs are that the adjacency matrices of weighted graphs can contain values other than 1 or 0, and adjacency dictionaries encode edges using a pair of values. This pair contains the destination node and the weight of the edge, as shown in 4.1.



$$\begin{array}{c|cccc} & \text{A} & \text{B} & \text{C} & \text{D} \\ \hline \text{A} & 0 & 1 & 0 & 2 \\ \text{B} & 1 & 0 & 0 & 3 \\ \text{C} & 0 & 0 & 0 & 4 \\ \text{D} & 2 & 3 & 4 & 0 \end{array}$$

$$\begin{aligned} \{\text{A}: &\ \{(\text{B},1),\ (\text{D},2)\}, \\ \text{B}: &\ \{(\text{A},1),\ (\text{D},3)\}, \\ \text{C}: &\ \{(\text{D},4)\}, \\ \text{D}: &\ \{(\text{A},2),\ (\text{B},3),\ (\text{C},4)\}\} \end{aligned}$$

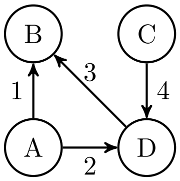
Figure 4.1: A simple weighted graph (left), its adjacency matrix (middle), and its adjacency dictionary (right). The graph is undirected, so the adjacency matrix is symmetric. Note that the adjacency dictionary contains node labels as keys and a set of tuples as values, which encode both edge location and weight.

NOTE

In this lab, all edge weights will be strictly positive. However, it is very possible to have edges with negative or zero weight. In these cases, Dijkstra's algorithm cannot generally find the cheapest path between nodes, but other algorithms exist which can (see the Additional Material section for more).

Directed Graphs

In addition to weighted graphs, some graphs are also *directed graphs*. These graphs can be weighted or unweighted, and edges represent a one-way connection between nodes. As a result, the graph's adjacency matrix may not be symmetric. In these graphs, edges heading away from a node are called *outgoing edges*, and edges coming into a node are called *incoming edges*.



$$\begin{array}{cc} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 3 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{aligned} \{A:\ & \{(B,1),\ (D,2)\}, \\ B:\ & \{\}, \\ C:\ & \{(D,4)\}, \\ D:\ & \{(B,3)\}\} \end{aligned}$$

Figure 4.2: A weighted, directed graph (left), its adjacency matrix (middle), and its adjacency dictionary (right). Note that the adjacency matrix is not symmetric.

Problem 1. Consider the following `Edge` class which wraps node and weight attributes into a single object

```python
class Edge:
    """An edge object, which wraps the node and weight attributes into one
    object, allowing for insertion/deletion from a set using just
    the node attribute

    Attributes:
        node (str): the value for the node the edge is pointing to
        weight (int): the weight of the edge
    """
    def __init__(self, node, weight):
        self.node = node
        self.weight = weight

    def __hash__(self):
        """Use only node attribute for hashing"""
        return hash(self.node)
    def __eq__(self, other):
        """Use only node attribute for equality"""
        if isinstance(other, Edge):
            return self.node == other.node
        return self.node == other

    def __str__(self):
        """String representation: a tuple-like view of the node and weight↩
            """
        return f"({str(self.node)}, {str(self.weight)})"

    def __repr__(self):
        """Repr is used when edges are displayed in a set"""
        return f"Edge({repr(self.node)}, {repr(self.weight)})"
```

This allows edges to be accessed in a set with only their node attribute while still encoding each edge's weight.

```
# Create a set with an Edge object inside
>>> my_set = {Edge('A', 1)}

# Check if the edge is in the set
>>> 'A' in my_set
True

# Remove edge and check again
>>> my_set.remove('A')
>>> 'A' in my_set:
False
```

In the file `dijkstra.py`, you will be filling out the `Graph` class to represent weighted/directed graphs. To do this, you will implement the following methods.

1. The constructor `__init__()` will take arguments `adjacency` and `directed` and store these as attributes (`self.d` and `self.directed`, respectively). `self.d` is a dictionary with node labels pointing to a set of edges originating from that node.

2. The `add_node()` method will take a node label `n` and add a node with that label (and no initial edges) to the graph if it is not already present.

3. The `add_edge()` method will add a weighted edge between two nodes. If an edge is already present, simply update the weight of that edge. Additionally, the method should add both nodes to the graph if they are not already present.

4. The `remove_node()` method will remove a node along with all adjacent edges from the graph. The method should raise a `KeyError` if the node is not in the graph.

5. The remove_edge method should remove the edge between two nodes. This method should raise a KeyError if either node is not in the graph or if there is no edge between the nodes.

Hint: Make sure you add/remove edges only in the direction specified (from the first node to the second) if the graph is directed.

Unit Test

In dijkstra_test.py, you will find a unit test test_graph() where you will write unit tests for your Graph class. Be sure to add and remove nodes and edges and test both directed and undirected graphs.

One good way to test these methods is to simply check for the presence of a node or edge in the self.d attribute of the Graph class rather than checking for a specific ordering of items (as dictionaries and sets are unordered).

Dijkstra's Algorithm

When working with weighted graphs, it is common to look for the path of least weight (or cost) between nodes. This corresponds to finding the minimum sum of weights on a path between the two nodes. For example, when driving you often want to reach your destination as soon as possible, but certain roads may take longer to traverse than others. Therefore, the minimum traversal may not necessarily include the fewest number edges.

Dijkstra's algorithm is a modification of the breadth-first search (BFS) algorithm which allows us to find paths of minimum cost on weighted graphs. While the BFS assumes all edges are the same weight, Dijkstra's algorithm accounts for different edge weights.

Dijkstra's algorithm is structured almost identically to BFS, but it keeps track of total edge weight on the current path, updating the shortest path to any visited nodes as it progresses. Additionally, instead of a normal queue (or deque), Dijkstra's implements a PriorityQueue to determine node traversal order.

Priority Queue

A Priority Queue, which is the subclass of the Queue class, is a special type of data structure that prioritizes the value of each entry when determining the next item to remove. By default, a Priority Queue will return the smallest value next when the .get() method is called. Items are sorted using a special data structure called a *min heap* which allows for item insertion in logarithmic time, and retrieval of the smallest item in constant time.

Method	Description
PriorityQueue()	Constructs an empty PriorityQueue object.
empty()	Returns True if there are no items in the PriorityQueue.
put(item)	Places item into the PriorityQueue.
get_nowait()	Returns the next item immediately (smallest by default).

Table 4.1: Essential methods of the PriorityQueue class.

```
# Import PriorityQueue
>>> from queue import PriorityQueue

# Create a new PriorityQueue object
>>> Q = PriorityQueue()

# Add some items to the PriorityQueue
>>> for item in [5, 4, 2, 1, 3]:
...     Q.put(item)
...

# Remove and print each item
>>> while not Q.empty():
...     print(Q.get_nowait(), end=' ')
...
1 2 3 4 5
```

For more information on `PriorityQueue`, see `https://docs.python.org/3/library/queue.html`.

Dijkstra's Algorithm Flow

We'll now go over the flow of Dijkstra's Algorithm. To traverse a graph with Dijkstra's, choose a node to start at, called the *source* node. Djikstra's keeps track of the shortest discovered path from the source node to each node, so each node starts with a path length of infinity, except the source node which has a path length of 0.

- First, update the path length of each node neighboring the source node to the weight of the edge going to that neighbor plus the weight of the source node (which is 0).
- Add these visited nodes to the priority queue along with the length of the current shortest path to these nodes.
- Next, select the node in the queue with the shortest path length and visit all of its neighboring nodes, updating their shortest path lengths as necessary.
- Add these visited nodes to the priority queue.
- Repeat this process until the destination node is selected from the queue.

The shortest path stored for the destination node is now the shortest path from the source node to the destination node.

The implementation of Dijkstra's requires the following data structures to keep track of which nodes have already been visited, path lengths, and the order in which to visit nodes in future steps.

- A **PriorityQueue** Q: The nodes to be visited, in order of which has the shortest path length. This is accomplished by storing in Q tuples containing first the distance to that node, then the node label.

- A set V: The nodes that have been finished (they are finished when they are popped off the queue). This structure is not necessary for Dijkstra's algorithm to function, but we will be using it to keep track of which nodes we have finished processing.
- A dictionary d: The shortest distance to each node
- A dictionary *pred*: Maps each node to its predecessor when traveling the shortest path to that node.

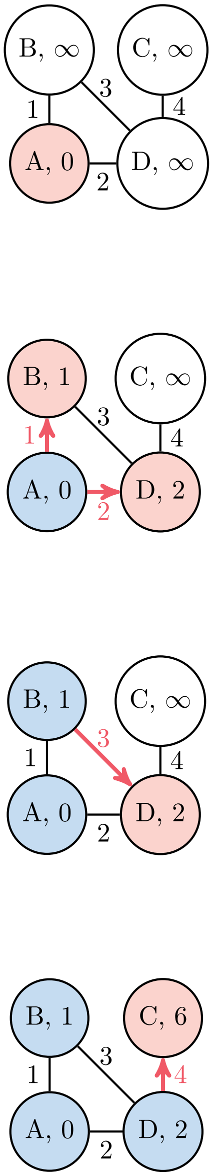
To begin the search, add the source node to Q and initialize d such that each node has value infinity except for the source node, which has value 0. Also, initialize *pred* so that the source node has a predecessor of $None$. Then, until the destination is found, repeat the following:

1. Pop a node off of Q; call it the *current* node.
2. If the current node is the destination, finish the loop.
3. "Visit" the current node by appending it to V.
4. Loop through all the neighbors of current.
5. For each neighbor, if the current path to that neighbors is the new shortest path for that neighbor, perform the following:
    - Update the neighbor's shortest distance value in d.
    - Add the neighbor and the distance to that neighbor to Q.
    - Include the current node as its predecessor by updating its value in *pred*.

Once finished, reconstruct the optimal path by starting at the destination and looking at predecessor nodes in *pred* until the source node is reached (the node with predecessor $None$). Return the shortest path and that path length, which is stored in d.

Note

Dijkstra's is known as a *greedy algorithm*, because the node with the shortest current path is always chosen next. The Bellman Optimality Principle guarantees that any part of the shortest path is itself the shortest path from the source node to that particular point. Because of this, every time a node is removed from the queue for the first time, we know that it's corresponding path value in d is the shortest path from the source to that node.



V	
Q	(0,A)
pred	

V	A
Q	(1,B) (2,D)
pred	B:A D:A

V	A B
Q	(2,D)
pred	B:A D:A

V	A B D
Q	(6,C)
pred	B:A C:D D:A

Figure 4.3: To start Dijkstra's from node A to node C, put A in the priority queue Q. Pop A off the queue and "visit" it by adding A to the visited list V and the neighboring nodes B and D to Q. Then visit B (since it has the shortest path). Do not update Q because the path from A to B to D is longer than the path from A to D, which is already in Q. Finally, visit D, at which point the target node C is located because it is adjacent to D. The values of *pred* give the predecessors, which can be used to reconstruct the shortest path (which, in this case, is A,D,C).

Problem 2. Write the shortest_path() method in the Graph class to implement Dijkstra's Algorithm, which will find the shortest path between the source and target nodes. The method should return the sum of weights along the shortest path, along with the shortest path itself, including endpoints. The method should raise a KeyError if the input nodes are not in the graph. Be sure that your method works on both directed and undirected graphs.

Hint: In order for the PriorityQueue to store both the node and the current path length, consider passing in a tuple where the first value is the current path length and the second value is the node label.

Unit Test

Test your code with the pre-written unit test found in dijkstra_test.py.

Bathymetry Dataset

Bathymetric Data and Tsunami Prediction

Bathymetry is the measurement of water depth to the sea floor. Bathymetric data provides ocean depth measurements in meters at specific latitude and longitude points. Bathymetric data is crucial in predicting the travel paths and speeds of tsunamis. In particular, the speed of a tsunami is directly impacted by the depth of the ocean floor below it. As such, bathymetric data provides a way to predict the speeds and arrival times of tsunamis at particular locations, allowing for sufficient warning to be given to those in the tsunami path. We will be using Dijkstra's Algorithm to predict the travel time of a tsunami between starting and ending locations.

The TsunamiModel Class

The file dijkstra.py contains a class for tsunami travel prediction using bathymetric data called TsunamiModel.[1] This class inherits from the Graph class, and constructs a graph where each depth measurement in the rectangular grid is represented as a node. In this graph, the travel time of the tsunami, which is a function of average depth between two points, serves as the weight between nodes. Once the graph is constructed, we can run Dijkstra's algorithm to determine the shortest time path of a tsunami from its origin to a given location.

The file bathymetry.tt3 contains bathymetric data which covers a rectangular section of the ocean in the Banda Region. Each data point indicates the ocean depth at a particular location. For example, the value in the lower left corner of the dataset (82 meters) corresponds to a latitude of -9.50833 and a longitude of 124.99167. In this context: zero depths represent sea level, positive values indicate elevations above sea level, and negative values denote depths below sea level. The distance between each data point in this particular file represents 1/60 degrees (or one arcminute) in both latitude and longitude.

The file, bathymetry.tt3 is structured as follows:

- The first and second lines contain the number of columns and rows of bathymetry data.

[1] Thanks to Dr. Jared Whitehead and Ashley Spencer, a professor and student at BYU, for the bathymetry data and TsunamiModel class.

- The third and fourth lines contain the longitude and latitude coordinates of the lower-left corner of the bathymetry data.

- The fifth line contains the geographic distance between any two values in the grid of bathymetry data (given in arc-degrees).

- The remaining lines contain the grid of bathymetric data collected at each location.

Problem 3. In the `TsunamiModel` class, complete the following methods:

1. `_read_file()`: reads in the bathymetric data given in filename. This method should store the attributes `ncols`, `nrows`, `long_llcorner`, `lat_llcorner`, `cellsize`, and `depths_grid` (these attributes are described in greater detail in the docstring of the `TsunamiModel` class).
2. `_generate_long_lat_grid()`: generates a grid of longitude and latitude coordinates corresponding to the locations where each bathymetric measurement in `depths_grid` was taken.

Call the `Graph` class constructor (using `super()`) as well as the `_read_file()` method in the `TimeModel` constructor. We will finish the constructor in the next problem. You are welcome to make the graph directed or undirected.

Test your code by comparing the read in attributes to the values found in `bathymetry.tt3`. When using the provided constants in `dijsktra.py`, the value of `long_lat_grid[0][0]` should be about $(124.991667, -2.508333)$.

Note

Several methods of the `TsunamiModel` class begin with an underscore character. These methods are meant to be used internally for use by the class itself and should not be called outside the class declaration.

Achtung!

The attribute variables of the `TsunamiModel` class are provided for you and set to a default value of `None`. Do *not* change these variable names, as they will cause the autograder for this lab to mark your code as incorrect. The original names for each attribute are provided in the docstring of the `TsunamiModel` class in case they've been modified.

In order to accurately model the time and path of a tsunami, we need to know the longitude and latitude coordinates of the tsunami's origin and our target location. Additionally, since tsunamis form over undersea earthquakes which occur over a fault plane with a certain radius, the exact position of the tsunami wave may not be at the epicenter of the earthquake which formed it. To account for this, we shift the starting point to the edge of the fault plane in the direction of our destination point. These values are passed in to the constructor of `TimeModel`, and some example values are given as constants in `dijkstra.py`.

Given the longitude and latitude coordinates of the starting and target points for a tsunami as well as the radius of the fault plane, we must shift the starting point in the direction of the target location by that radius. This will ensure that the calculated time is when the front of the tsunami reaches the target rather than the back which can vary the time calculation by several minutes. Additionally, since bathymetric data is given in only discrete intervals, we must find the closest grid points to our starting and ending coordinates in order to run Dijkstra's algorithm. The `_get_nearest_point()` method of the `TimeModel` class will perform this calculation for us.

Problem 4. In the constructor of the `TsunamiModel` class, store the value given by `fault_plane_radius` as an attribute. Next, complete the following methods:

1. `_shifted_start()`: Shifts the starting point for the tsunami path away from the epicenter and toward the end point according to `fault_plane_radius` and returns this point as a tuple of longitude, latitude coordinates.
2. `_long_and_lat()`: Generates instance attributes associated with longitude and latitude coordinates. Specifically, it should call `_generate_lat_long_grid()` from Problem 3 and store the resulting grid as an attribute as well as find the starting and ending points for the tsunami path prediction using the provides `_get_nearest_point()` method (remember to shift the starting point).

Call `_long_and_lat()` in the `TsunamiModel` constructor. The constructor should now be finished as well.

Test your code using the provided constants in `dijkstra.py`. The value of `start_point` should be $(166, 382)$, and the value of `end_point` should be $(65, 220)$.

Modeling Graphs With a Grid

Many graphs, including this one, are modeled using a 2D grid, where neighboring nodes are adjacent locations in the grid. In the case of our bathymetry data, neighboring nodes are those which are orthogonally adjacent (don't include diagonals). In order to utilize the functionality of the `Graph` class, we store the values of each node as a tuple of their coordinates in the grid (row and column values, zero indexed). Additionally, we do not form edges with nodes which have a non-negative elevation as we assume that tsunamis cannot cross these locations. Because we are modeling the shortest time for a Tsunami to reach out target location, we model weights as the time taken for a tsunami to travel between grid points.

The speed of a tsunami given a depth d is given by

$$s = \sqrt{d * g}$$

where g is the gravitational constant (in the desired units). From this formula we can then derive the travel time between locations of depths d_1 and d_2 of distance $dist$ apart. To do this, we average out the speed of the tsunami between both locations by taking the speed at the average depth of the two points,

$$s_{avg} = \sqrt{g * (d_1 + d_2)/2}.$$

The travel time of the tsunami between these points is then closely estimated by the distance between these points divided by the average speed,

$$t = \frac{dist}{s_{avg}}.$$

This equation is how we determine the tsunami travel time between grid points and thus the weights of our graph.

Problem 5. Implement the following methods of the `TimeModel` class which help generate the graph for Dijkstra's algorithm.

1. `_get_neighbors()`: returns a list of valid neighbors to the given node. Each neighbor should be represented by a tuple of indices which represent its location in `depths_grid`.
2. `_get_time()`: returns the estimated travel time (in seconds) of a tsunami between two adjacent grid points at the given depths. Use 9.8 as the gravitational constant, and be sure to get the distance between the two points in meters (a helpful constant is provided above the `TimeModel` class).
3. `_convert_path_to_long_lat()`: given a list of indices representing grid coordinates, returns a list of corresponding longitude and latitude coordinates.
(Hint: use the `long_lat_grid` attribute.)

Problem 6. Implement the following methods of the `TimeModel` class:

1. `_generate_graph()`: adds nodes and edges to the graph between grid point neighbors with weights corresponding to tsunami travel time along each edge. Edges should only be created between points that are both below sea level. Node labels should be tuples of the indices corresponding to each point in grid (you may ignore points that are at or above sea level).
2. `calculate_tsunami_path()`: calls `_generate_graph()` and uses the `shortest_path()` method of the `Graph` class to find the time (in minutes) and path for a tsunami between `start_point` and `end_point`. The returned path should be a list of longitude and latitude coordinates.

Achtung!

To ensure that your code is graded correctly, be sure that every attribute described in the constructor of the `TimeModel` class is present with the name spelled exactly as given.

Test you algorithm with the constants given above the `TimeModel` class. Your algorithm should return a time of around 48.2310 minutes. Also ensure that the returned path starts and ends at the expected coordinate points (remember the point wills be shifted and approximated to points on the grid of bathymetric readings).

Additional Material

Run Time Complexity of Dijkstra's Algorithm

The main component of the temporal complexity of Dijkstra's Algorithm is priority queue operations. If a graph has E edges and V nodes (or vertices), then there must be at most $E+1$ inserts into the priority queue. Each insert has complexity $O(\log(E))$, so the runtime complexity of the entire algorithm is $O(E\log(E))$. Because the number of edges and vertices are often comparable, especially in connected graphs, this complexity is sometimes written as $O(E\log(V))$. Additionally, priority queues can be implemented with a specialized heap called a *Fibonacci heap* which alters the overall complexity to $O(E+V\log(V))$.

Shortest Path With Negative Weights

Sometimes, graphs can be represented with negative weights. This can represent recouped costs or saved time, among other things. However, Dijkstra's Algorithm often fails when working with negative weights, as the Bellman Optimality Principle no longer applies. This is because negative weights allow for a visit other than the first to a node to obtain an overall lower path sum to that node.

The *Bellman-Ford Algorithm*[2] addresses this issue by scanning through and relaxing each of the edges in the graph. This relaxation is effectively the act of doubling back to previously visited nodes to check for improved paths to that node. If a negative cycle (a cycle in which the sum of weights is negative) exists, then the algorithm will either return negative infinity, or throw an error. Because of the relaxation step, this algorithm has a runtime of $O(VE)$ where V is the number of nodes (vertices) and E is the number of edges in the graph.

[2] See: `https://cp-algorithms.com/graph/bellman_ford.html`

5 Markov Chains

Lab Objective: *A Markov chain is a collection of states with specified probabilities for transitioning from one state to another. They are characterized by the fact that the future behavior of the system depends only on its current state. In this lab we learn to construct, analyze, and interact with Markov chains, then use a Markov-based approach to simulate natural language.*

State Space Models

Many systems can be described by a finite number of *states*. For example, a board game where players move around the board based on dice rolls can be modeled by a Markov chain. Each space represents a state, and a player is said to be in a state if their piece is currently on the corresponding space. In this case, the probability of moving from one space to another only depends on the player's current location; where the player was on a previous turn does not affect their current turn.

Markov chains with a finite number of states have an associated *transition matrix* that stores the information about the possible transitions between the states in the chain. The (i, j)th entry of the matrix gives the probability of moving **from state** j **to state** i. Thus, each of the columns of the transition matrix sum to 1.

Note

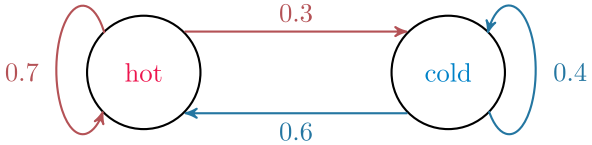
A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the (i, j)th entry of the matrix is the probability of moving from state i to state j. Both representations are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

Consider a very simple weather model in which the weather tomorrow depends only on the weather today. For now, we consider only two possible weather states: hot and cold. Suppose that if today is hot, then the probability that tomorrow is also hot is 0.7, and that if today is cold, the probability that tomorrow is also cold is 0.4. By assigning "hot" to the 0th row and column, and "cold" to the 1st row and column, this Markov chain has the following transition matrix.

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

The 0th column of the matrix says that if it is hot today, there is a 70% chance that tomorrow will be hot (0th row) and a 30% chance that tomorrow will be cold (1st row). The 1st column says if it is cold today, then there is a 60% chance of heat and a 40% chance of cold tomorrow.

Markov chains can be represented by a *state diagram*, a type of directed graph. The nodes in the graph are the states, and the edges indicate the state transition probabilities. The Markov chain described above has the following state diagram.



Problem 1. Define a `MarkovChain` class whose constructor accepts an $n \times n$ transition matrix A and, optionally, a list of state labels. If A is not column stochastic, raise a `ValueError`. Construct a dictionary mapping the state labels to the row/column index that they correspond to in A (given by order of the labels in the list), and save A, the list of labels, and this dictionary as attributes. If there are no state labels given, use the labels $\begin{bmatrix} 0 & 1 & \ldots & n-1 \end{bmatrix}$.

For example, for the weather model described above, the transition matrix is

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix},$$

the list of state labels is `["hot", "cold"]`, and the dictionary mapping labels to indices is `{"hot":0, "cold":1}`. This Markov chain could be also represented by the transition matrix

$$\widetilde{A} = \begin{bmatrix} 0.4 & 0.3 \\ 0.6 & 0.7 \end{bmatrix},$$

the labels `["cold", "hot"]`, and the resulting dictionary `{"cold":0, "hot":1}`.

Simulating State Transitions

Simulating the weather model described above requires a programmatic way of choosing between the outgoing transition probabilities of each state. For example, if it is cold today, we could flip a weighted coin that lands on tails 60% of the time (guess tomorrow is hot) and heads 40% of the time (guess tomorrow is cold) to predict the weather tomorrow. The *Bernoulli distribution* with parameter $p = 0.4$ simulates this behavior: 60% of draws are 0, and 40% of draws are a 1.

A *binomial distribution* is the sum several Bernoulli draws: one binomial draw with parameters n and p indicates the number of successes out of n independent experiments, each with probability p of success. In other words, n is the number of times to flip the coin, and p is the probability that the coin lands on heads. Thus, a binomial draw with $n = 1$ is a Bernoulli draw.

NumPy does not have a function dedicated to drawing from a Bernoulli distribution; instead, use the more general `np.random.binomial()` with $n = 1$ to make a Bernoulli draw.

```
>>> import numpy as np

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

# Draw from the Bernoulli distribution with p = .3 (flip one weighted coin).
>>> np.random.binomial(n=1, p=.3)
0                             # Also tails.
```

For the weather model, if the "cold" state corresponds to row and column 1 in the transition matrix, p should be the probability that tomorrow is cold. So, if today is cold, select $p = 0.4$; if today is hot, set $p = 0.3$. Then draw from the binomial distribution with $n = 1$ and the selected p. If the result is 0, transition to the "hot" state; if the result is 1, stay in the "cold" state.

Using Bernoulli draws to determine state transitions works for Markov chains with two states, but larger Markov chains require draws from a *categorical distribution*, a multivariate generalization of the Bernoulli distribution. A draw from a categorical distribution with parameters $(p_1, p_2, \ldots, p_k)$ satisfying $\sum_{i=1}^{k} p_i = 1$ indicates which of k outcomes occurs. If $k = 2$, a draw simulates a coin flip (a Bernoulli draw); if $k = 6$, a draw simulates rolling a six-sided die. Just as the Bernoulli distribution is a special case of the binomial distribution, the categorical distribution is a special case of the *multinomial distribution* which indicates how many times each of the k outcomes occurs in n repeated experiments. Use `np.random.multinomial()` with $n = 1$ to make a categorical draw.

```
# Draw from the categorical distribution (roll a fair four-sided die).
>>> np.random.multinomial(1, np.array([1./4, 1./4, 1./4, 1./4]))
array([0, 0, 0, 1])     # The roll resulted in a 3.

# Draw from another categorical distribution (roll a weighted four-sided die).
>>> np.random.multinomial(1, np.array([.5, .3, .2, 0]))
array([0, 1, 0, 0])     # The roll resulted in a 1.
```

Consider a four-state weather model with the transition matrix

$$\begin{array}{c c} & \begin{array}{cccc} \text{hot} & \text{mild} & \text{cold} & \text{freezing} \end{array} \\ \begin{array}{c} \text{hot} \\ \text{mild} \\ \text{cold} \\ \text{freezing} \end{array} & \left[\begin{array}{cccc} 0.5 & 0.3 & 0.1 & 0 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0 & 0.1 & 0.2 & 0.2 \end{array}\right] \end{array}.$$

If today is hot, the probabilities of transitioning to each state are given by the "hot" column of the transition matrix. Therefore, to choose a new state, draw from the categorical distribution with parameters $(0.5, 0.3, 0.2, 0)$. The result $\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$ indicates a transition to the state corresponding to the 1st row and column (tomorrow is mild), while the result $\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$ indicates a transition to the state corresponding to the 2nd row and column (tomorrow is cold). In other words, the position of the 1 tells which column of the matrix to use as the parameters for the next categorical draw.

Problem 2. Write a method for the `MarkovChain` class that accepts a single state label. Use the label-to-index dictionary to determine the column of A that corresponds to the provided state label, then draw from the corresponding categorical distribution to choose a state to transition to. Return the corresponding label of the new state (not its index) as a string. (Hint: `np.argmax()` may be useful.)

Problem 3. Add the following methods to the `MarkovChain` class.

- `walk()`: Accept a state label and an integer N. Starting at the specified state, use your method from Problem 2 to transition from state to state $N-1$ times, recording the state label at each step. Return the list of N state labels (as strings), including the initial state.
- `path()`: Accept labels for an initial state and an end state. Beginning at the initial state, transition from state to state until arriving at the specified end state, recording the state label at each step. Return the list of state labels (as strings), including the initial and final states.

Test your methods on the two-state and four-state weather models described previously.

General State Distributions

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

If A is a transition matrix for a Markov chain with n states and $\mathbf{x}$ is a corresponding state distribution vector, then $A\mathbf{x}$ is also a state distribution vector. In fact, if $\mathbf{x}_k$ is the state distribution vector corresponding to a certain time k, then $\mathbf{x}_{k+1} = A\mathbf{x}_k$ contains the probabilities of being in each state after allowing the system to transition again. For the weather model, this means that if there is an 80% chance that it will be hot 5 days from now, written $\mathbf{x}_5 = \begin{bmatrix}0.8 & 0.2\end{bmatrix}^\mathsf{T}$, 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 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, a pattern emerges in A^k for even small values of k:

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0.67 & 0.66 \\ 0.33 & 0.34 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0.667 & 0.666 \\ 0.333 & 0.334 \end{bmatrix}.$$

As $k \to \infty$, the entries of A^k converge, written

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

In addition, for any initial state distribution vector $\mathbf{x}_0 = [a, \ b]^\mathsf{T}$ (meaning $a, b \geq 0$ and $a + b = 1$),

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

Thus, $\mathbf{x}_k \to \mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ as $k \to \infty$, regardless of the initial state distribution $\mathbf{x}_0$. So, according to this model, no matter the weather today, the probability that it is hot a week from now is approximately 66.67%. In fact, approximately 2 out of 3 days in the year should be hot.

Steady State Distributions

The state distribution $\mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ 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 corresponding to the eigenvalue $\lambda = 1$.

Every finite Markov chain has at least one steady state distribution. If some power A^k of A has all positive (nonzero) entries, then the steady state distribution is unique.[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. For instance, consider the transition matrix

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

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

[1] This is a consequence of the *Perron-Frobenius theorem*, which is presented in detail in Volume 1.

Problem 4. Write a method for the `MarkovChain` class that accepts a convergence tolerance `tol` and a maximum number of iterations `maxiter`. Generate a random state distribution vector $\mathbf{x}_0$ and calculate $\mathbf{x}_{k+1} = A\mathbf{x}_k$ until $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_1 <$ `tol`, where A is the transition matrix saved in the constructor. If k exceeds `maxiter`, raise a `ValueError` to indicate that A^k does not converge. Return the approximate steady state distribution $\mathbf{x}$ of A.

To test your function, generate a random transition matrix A. Verify that $A\mathbf{x} = \mathbf{x}$ and that the columns of A^k approach $\mathbf{x}$ as $k \to \infty$. 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 method to validate the results of Problem 3: for the two-state and four-state weather models,

1. Calculate the steady state distribution corresponding to the transition matrix.
2. Run a weather simulation for a large number of days using `walk()` and verify that the results match the steady state distribution (for example, approximately 2/3 of the days should be hot for the two-state model).

Note

Problem 4 is a special case of the *power method*, an algorithm for calculating an eigenvector of a matrix corresponding to the eigenvalue of largest magnitude. The general power method, together with a discussion of its convergence conditions, is discussed in Volume 1.

Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study *natural languages*, meaning spoken or written languages like English or Russian [VHL06]. In the early 20th century, Markov used his chains to model how Russian switched from vowels to consonants. By mid-century, they had been used as an attempt to model English. It turns out that plain Markov chains are, by themselves, insufficient to model or produce very good English. However, they can approach a fairly good model of bad English, with sometimes amusing results.

By nature, a Markov chain is only concerned with its current state, not with previous states. A Markov chain simulating transitions between English words is therefore completely unaware of context or even of previous words in a sentence. For example, if a chain's current state is the word "continuous," the chain may say that the next word in a sentence is more likely to be "function" rather than "raccoon." However the phrase "continuous function" may be gibberish in the context of the rest of the sentence.

Generating Random Sentences

Consider the problem of generating English sentences that are similar to the text contained in a specific file, called the *training set*. The goal is to construct a Markov chain whose states and transition probabilities represent the vocabulary and—hopefully—the style of the source material. There are several ways to approach this problem, but one simple strategy is to assign each unique word in the training set to a state, then construct the transition probabilities between the states based on the ordering of the words in the training set. To indicate the beginning and end of a sentence requires two extra states: a *start state*, $tart, marking the beginning of a sentence; and a *stop state*, $top, marking the end. The start state should only transition to words that appear at the beginning of a sentence in the training set, and only words that appear at the end a sentence in the training set should transition to the stop state.

Consider the following small training set, paraphrased from Dr. Seuss [Gei60].

```
I am Sam Sam I am.
Do you like green eggs and ham?
I do not like them, Sam I am.
I do not like green eggs and ham.
```

There are 15 unique words in this training set, including punctuation (so "ham?" and "ham." are counted as distinct words) and capitalization (so "Do" and "do" are also different):

I am Sam am. Do you like green
eggs and ham? do not them, ham.

With start and stop states, the transition matrix should be 17 × 17. Each state must be assigned a row and column index in the transition matrix, for example,

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

The (i, j)th entry of the transition matrix A should be the probability that word j is followed by word i. For instance, the word "Sam" is followed by the words "Sam" once and "I" twice in the training set, so the state corresponding to "Sam" (index 3) should transition to the state for "Sam" with probability $1/3$, and to the state for "I" (index 1) with probability $2/3$. That is, $A_{3,3} = 1/3$, $A_{1,3} = 2/3$, and $A_{i,3} = 0$ for $i \notin \{1, 3\}$. Similarly, the start state should transition to the state for "I" with probability $3/4$, and to the state for "Do" with probability $1/4$; the states for "am.", "ham?", and "ham." should each transition to the stop state.

To construct the transition matrix, parse the training set and add 1 to $A_{i,j}$ every time word j is followed by word i, in this case arriving at the matrix

$$\begin{array}{c|ccccccc|}
 & \text{\$start} & \text{I} & \text{am} & \text{Sam} & & \text{ham.} & \text{\$top} \\
\text{\$start} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\text{I} & 3 & 0 & 0 & 2 & \dots & 0 & 0 \\
\text{am} & 0 & 1 & 0 & 0 & \dots & 0 & 0 \\
\text{Sam} & 0 & 0 & 1 & 1 & \dots & 0 & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\text{ham.} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\text{\$top} & 0 & 0 & 0 & 0 & \dots & 1 & 0
\end{array}.$$

To avoid a column of zeros, set $A_{j,j} = 1$ where j is the index of the stop state (so the stop state always transitions to itself). Next, divide each column by its sum so that each column sums to 1:

$$\begin{array}{c|ccccccc|}
 & \text{\$start} & \text{I} & \text{am} & \text{Sam} & & \text{ham.} & \text{\$top} \\
\text{\$start} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\text{I} & 3/4 & 0 & 0 & 2/3 & \dots & 0 & 0 \\
\text{am} & 0 & 1/5 & 0 & 0 & \dots & 0 & 0 \\
\text{Sam} & 0 & 0 & 1 & 1/3 & \dots & 0 & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\text{ham.} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\text{\$top} & 0 & 0 & 0 & 0 & \dots & 1 & 1
\end{array}.$$

The 3/4 indicates that 3 out of 4 times, the sentences in the training set start with the word "I". Similarly, the 2/3 and 1/3 says that "Sam" is followed by "I" twice and by "Sam" once in the training set. Note that "am" (without a period) always transitions to "Sam" and that "ham." (with a period) always transitions the stop state.

The entire procedure of creating the transition matrix for the Markov chain with words from a file as states is summarized below.

Algorithm 1 Convert a training set of sentences into a Markov chain.

```
procedure MakeTransitionMatrix(filename)
    Read the training set from the file filename.
    Get the set of unique words in the training set (the state labels).
    Add labels "$tart" and "$top" to the set of states labels.
    Initialize an appropriately sized square array of zeros to be the transition matrix.
    for each sentence in the training set do
        Split the sentence into a list of words.
        Prepend "$tart" and append "$top" to the list of words.
        for each consecutive pair $(x, y)$ of words in the list of words do
            Add 1 to the entry of the transition matrix that corresponds to transitioning from state $x$ to state $y$.
    Make sure the stop state transitions to itself.
    Normalize each column by dividing by the column sums.
```

Problem 5. Write a class called `SentenceGenerator` that inherits from the `MarkovChain` class. The constructor should accept a filename (the training set). Read the file and build a transition matrix from its contents as described in Algorithm 1. Save the same attributes as the constructor of `MarkovChain` does so that inherited methods work correctly. Assume that the training set has one complete sentence written on each line.
(Hint: if the contents of the file are in the string `s`, then `s.split()` is the list of words and `s.split('n')` is the list of sentences.)

Note

The Markov chains that result from the procedure in Problem 5 have a few interesting structural characteristics. The stop state is a *sink*, meaning it only transitions to itself. Because of this, and since every node has a path to the stop state, any traversal of the chain will end up in the stop state forever. The stop state is therefore called an *absorbing state*, and the chain as a whole is called an *absorbing Markov chain*. Furthermore, the steady state is the vector with a 1 in the entry corresponding to the stop state and 0s everywhere else.

Problem 6. Add a method to the `SentenceGenerator` class called `babble()`. Use the `path()` method from Problem 3 to generate a random sentence based on the training document. That is, generate a path from the start state to the stop state, remove the `"$tart"` and `"$top"` labels from the path, and join the resulting list together into a single, space-separated string. Return this string.

For example, your `SentenceGenerator` class should be able to create random sentences that sound somewhat like Yoda speaking.

```
>>> yoda = SentenceGenerator("yoda.txt")
>>> for _ in range(3):
...     print(yoda.babble())
...
Impossible to my size, do not!
For eight hundred years old to enter the dark side of Congress there is.
But beware of the Wookiees, I have.
```

Additional Material

Other Applications of Markov Chains

Markov chains are a useful way to study many probabilistic phenomena, so they have a wide variety of applications. The following are just a few that are covered in other parts of this lab manual series.

- **PageRank**: Google's PageRank algorithm uses a Markov chain-based approach to rank web pages. The main idea is to use the entries of the steady state vector as a measure of importance for the corresponding states. For example, the steady state $\mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ for the two-state weather model means that the hot state is "more important" (occurs more frequently) than the cold state. See the PageRank lab in Volume 1.
- **MCMC Sampling**: A *Monte Carlo Markov Chain* (MCMC) method constructs a Markov chain whose steady state is a probability distribution that is difficult to sample from directly. This provides a way to sample from nontrivial or abstract distributions. Many MCMC methods are used in various fields, from machine learning to physics. See the Volume 3 lab on the Metropolis-Hastings algorithm.
- **Hidden Markov Models**: The Markov chain simulations in this lab use an initial condition (a state distribution vector $\mathbf{x}_0$) and known transition probabilities to make predictions forward in time. Conversely, a *hidden Markov model* (HMM) assumes that a given set of observations are the result of a Markov process, then uses those observations to infer the corresponding transition probabilities. Hidden Markov models are used extensively in modern machine learning, especially for speech and language processing. See the Volume 3 lab on Speech Recognition.

Large Training Sets

The approach in Problems 5 and 6 begins to fail as the training set grows larger. For example, a single Shakespearean play may not be large enough to cause memory problems, but *The Complete Works of William Shakespeare* certainly will.

To accommodate larger data sets, consider use a sparse matrix from `scipy.sparse` for the transition matrix instead of a regular NumPy array. Specifically, construct the transition matrix as a `lil_array` (which is easy to build incrementally), then convert it to the `csc_array` format (which supports fast column operations). Ensure that the process still works on small training sets, then proceed to larger training sets. How are the resulting sentences different if a very large training set is used instead of a small training set?

Variations on the English Model

Choosing a different state space for the English Markov model produces different results. Consider modifying the `SentenceGenerator` class so that it can determine the state space in a few different ways. The following ideas are just a few possibilities.

- Let each punctuation mark have its own state. In the Dr. Seuss training set, instead of having two states for the words "ham?" and "ham.", there would be three states: "ham", "?", and ".", with "ham" transitioning to both punctuation states.
- Model paragraphs instead of sentences. Add a $tartParagraph state that always transitions to $tartSentence and a $topParagraph state that is sometimes transitioned to by $topSentence.

- Let the states be individual letters instead of individual words. Be sure to include a state for the spaces between words.
- Construct the state space so that the next state depends on both the current and previous states. This kind of Markov chain is called a *Markov chain of order 2*. This way, every set of three consecutive words in a randomly generated sentence should be part of the training set, as opposed to only every consecutive pair of words coming from the set.
- Instead of generating random sentences from a single source, simulate a random conversation between n people. Construct a Markov chain M_i, for each person, $i = 1, \ldots, n$, 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 [BL04]. For example, `nltk.sent_tokenize()` reads a single string and splits it up into sentences. This could be useful, for example, in making the `SentenceGenerator` class compatible with files that do not have one sentence per line.

```
>>> from nltk import sent_tokenize
>>> with open("yoda.txt", 'r') as yoda:
...     sentences = sent_tokenize(yoda.read())
...
>>> print(sentences)
['Away with your weapon!',
 'I mean you no harm.',
 'I am wondering - why are you here?',
 ...
```

The `nltk` module is **not** part of the Python standard library. For instructions on downloading, installing, and using `nltk`, visit `http://www.nltk.org/`.

6 Sampling

Lab Objective: *Sampling is an important and fundamental tool in statistical modeling. In this lab we will learn to use PyMC for Bayesian modeling and statistical sampling. This lab will focus on material from chapters 5 and 6 of Volume 2.*

Sampling

In Statistics, a *population* is the collection of all items, groups, or phenomenon we are seeking to study or understand in an experiment. It is all the data we could possibly have or we wish we had (e.g. all Americans, all NCAA football games). However, in practice, we rarely have access to the entire population. As such, we must then obtain a finite data set. *Sampling* is the process where researchers take a predetermined number of draws from a population, called a *sample* (e.g. 750 Americans, 300 NCAA football games). A good sample can tell us a lot, and while we will not examine what makes a good sample in this lab, we will examine how much a good sample can tell us. One goal of Bayesian statistics is to be able to quantify, with degrees of certainty, how much we can learn from a sample, given what we already know about its source. This quantification of certainty allows us to extract more information and nuance from a sample than would otherwise be possible, and in turn allow us to better predict and describe events.

Parameter Estimation

Given a sample $\mathbf{x} = x_1, x_2, \ldots, x_n$, we often think of this data as draws or realizations from an independent and identically distributed (i.i.d) sequence of random variables $X_1, X_2, ..., X_n$ having the same distribution as some unknown random variable X. This random variable has some distribution (i.e. the pdf/pmf) determined by some parameters that allows us to determine certain measurements about the population like the mean or variance.

A *parameter*, in Statistics, is any measurement that describes a distribution. Thus, we normally apply some function or formula to the sample in order to obtain information about the distribution that generated the sample. This function is called a *statistic*. Specifically, a statistic that is used to estimate a parameter is called an *estimator*, and the result that is obtained when replacing each random variable X_i by each datum x_i is called the *estimate*.

We will examine two methods of parameter estimation: the Frequentist/Classical approach and the Bayesian approach.

Frequentist/Classical Approach: Maximum Likelihood Estimation

The Frequentist approach views probability as the long-term frequency of an event occurring. In this approach, we view the parameter as a fixed value that we are trying to estimate. The estimates we make are known as *point estimates* because they are single values. *Maximum Likelihood Estimation* (MLE) is a frequentist approach to parameter estimation, viewing probability as the limit of the frequency of success generated by several repeated trials. The MLE is a method that chooses the parameter for which the sample is most likely to have occurred.

Likelihood

Finding the maximum likelihood involves, unsurprisingly, maximizing the likelihood function, which is defined as follows

$$\mathcal{L}(\theta) = \mathcal{L}(\theta|\mathbf{x}) = f(\mathbf{x}|\theta) = \prod_{i=1}^{n} f(x_i|\theta),$$

where $\mathbf{x} = x_1, x_2, \ldots, x_n$ is a sample, θ is the parameter we are estimating and f is the pdf/pmf. Thus, a MLE of a parameter θ is the value of $\hat{\theta}$ that maximizes the likelihood function $\mathcal{L}$. This value is known as the *maximum likelihood estimate*. Determining the MLE $\hat{\theta}$ is then as simple as finding the argmax of $\mathcal{L}$

$$\hat{\theta} = \text{argmax}_{\theta\in\Theta}\mathcal{L}(\theta|\mathbf{x}).$$

Bayesian Approach: Maximum A Posteriori Estimate

The Bayesian approach views probability as a measure of belief and updates that belief as more evidence is gathered. We treat the unknown parameter as a random variable and try to compute the distribution of it conditioned on the sample and using some initial belief for the parameter. If we examine closely, we can see a similarity between the likelihood function and Bayes' rule. Bayes' rule gives the following relation

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

In some simple terms, Bayes' rule is framework for updating our "beliefs" about the hypothesis A given some evidence B. To apply this rule to the problem of parameter estimation we get the following relation (note f is a pdf/pmf)

$$f(\theta|\mathbf{x}) = \frac{f(\mathbf{x}|\theta)g(\theta)}{\int_\Theta f(\mathbf{x}|\vartheta)g(\vartheta)d\vartheta}. \tag{6.1}$$

We call $f(\theta|\mathbf{x})$ the *posterior distribution*, $f(\mathbf{x}|\theta)$ the *likelihood function*, and $g(\theta)$ the *prior distribution*. The posterior $f(\theta|\mathbf{x})$ represents the updated distribution for θ that takes into account the sample $\mathbf{x}$. The prior $g(\theta)$ is the initial assumed distribution for θ that represents our beliefs about θ before we see the sample, and the likelihood $f(\mathbf{x}|\theta)$ is the function that represents the probability of the sample holding true given the parameter θ. The denominator is normally called the *marginal likelihood* and is a constant for normalization that represents the probability of generating the sample under any possible value of $\theta \in \Theta$. As such, the marginal likelikhood is independent of θ.

This approach is in contrast to the Frequentist approach where the parameter is fixed, the sample is a result of various attempts, and we consider no prior knowledge for the parameter. The Bayesian approach allows us to incorporate prior knowledge into our model, so that we can update our knowledge as we gather more evidence. But, the result we get is just a probability distribution, not a single value, that tells us which values of θ are most likely to have produced our sample.

The MAP estimate is then the argmax of the posterior

$$\theta_{MAP} = \text{argmax}_{\theta \in \Theta} f(\theta|\mathbf{x}). \tag{6.2}$$

When finding the MAP estimate, the exact posterior is often left uncalculated because it is difficult to compute. Instead, we approximate it by finding its value at grid points of θ. Similarly, because of the complexity of the denominator, we often don't find it until the end of the process. After we calculate $f(\mathbf{x}|\theta)g(\theta)$ for each relevant θ we can then approximate the integral in the denominator with a finite sum. First we find $f(\mathbf{x}|\theta_i)g(\theta_i)$ for a grid of θ values. Then, we can approximate the integral in denominator with the following sum $\frac{1}{n}\sum_{i=1}^{n} f(\mathbf{x}|\theta_i)g(\theta_i)$ (i.e. using finite sums like Riemann Sums) or using Monte Carlo Methods like MCMC.

Here we can see that the likelihood function is similar to Bayes' rule as long as we take $g(\theta)$ to be a constant, i.e. $\theta \sim \mathcal{U}(a, b)$. This means that the MAP estimate is the MLE if we assume a uniform prior distribution.

Example 1: Estimating the Lifespan of a Projector Bulb with a Uniform Prior

Assume that the lifespan of a projector bulb can be modeled as a random variable X with an exponential distribution of unknown parameter λ. Suppose that you have a sample of 7 bulbs which lasted 2, 3.3, 4.5, 1.8, 3.1, 2.7, and 2.2 months, respectively. Moreover, assume we have no reason to believe that the lifespan of a bulb is any more likely to be any particular value than any other. That is,the lifespan can take on any value equally. Find the posterior pdf for λ, compute the MAP for λ, and plot the prior and posterior.

> **Achtung!**
>
> Note that just like NumPy, the stats module of SciPy allows for array broadcasting or vectorization. Most, if not all, of the parameters for all of the distributions in the stats module can be ndarrays, and the functions will return an ndarray of the same shape. Thus, keep track of shapes so you do not run into errors. Lastly, it is important to look at the documentation for each statistical module to understand what parameters are used and if they are different from the ones we or books have described.

Since X is distributed as an exponential distribution, the likelihood pdf is given by $\text{Gamma}(1, \lambda)$. Moreover, recall that $\text{Gamma}(a, b)$ distribution describes the waiting time for $a > 0$ events to occur in a Poisson process with rate $b > 0$ (i.e. the time between events). Some books, modules, and equations, use the parameter *scale*, $\vartheta = \frac{1}{b}$, instead of the rate in the Gamma distribution. `scipy.stats` is one such module, so we will work $\vartheta = \frac{1}{\lambda}$ as the scale, and then compute the inverse to obtain λ.

For our example, the event a is the number of bulbs that have failed, and the rate b represents the number of bulb-months at which they fail (i.e. bulb-months/bulb-failure). In our case, we are trying to estimate the rate λ at which 1 bulb fails, so $a = 1$. Then, the scale ϑ represents bulb-failures per bulb-month. Though $a \in (0, \infty]$ (i.e. we can have any number of bulb failures), we will assume that there are no more than 2 bulb failures per month because the inverse of the sample at least shows that there are no more than 2. Hence, $a \in (0, 2]$. Though this is a continuous interval and a can take on any value between 0 and 2, we have to discretize the interval for computational purposes since we do not have infinite computational power or memory. Thus, we assume that there are only 100 possible events. This implies that there are only 100 possible values for ϑ.

Since the likelihood $f(\mathbf{x}|\theta) = \prod_{i=1}^{n} f(x_i|\theta)$, we can use `np.prod()` to calculate the likelihood for each value of ϑ. Lastly, `scipy.stats.uniform` uses the parameter `loc` and `scale` to create a uniform distribution on the interval [`loc`, `loc`+`scale`]. The normal default values are 1 and 0, respectively.

```
from scipy.stats import uniform, gamma
import matplotlib.pyplot as plt
import numpy as np
>>> sample = 1/np.array([2, 3.3, 4.5, 1.8, 3.1, 2.7, 2.2]) # Sample scale ↩
    values
>>> fails = np.linspace(0, 2, 100)[1:] # Scale values, not including 0
>>> prior = uniform.pdf(x=fails, loc=0, scale=2)
# List comprehension to prevent broadcasting error
# Compute the product f(x_i|scale) for i=1 to 7 for each value of the scale
>>> likelihood = np.array([(gamma.pdf(x=sample, a=1, scale=fail)).prod() for ↩
    fail in fails])
>>> integral = (likelihood*prior).sum()*(2/100) # Riemann sums on [0, 2]
>>> posterior = (likelihood*prior)/integral
>>> 1/fails[posterior.argmax()] # MAP estimate
2.6052631578947367

>>> plt.plot(fails, prior, color='orange', label='Prior')
>>> plt.plot(fails, posterior, color='blue', label='Posterior')
>>> plt.ylabel("Probability Density")
>>> plt.xlabel("Scale (bulb-failures/bulb-month)")
>>> plt.legend()
>>> plt.show()
```

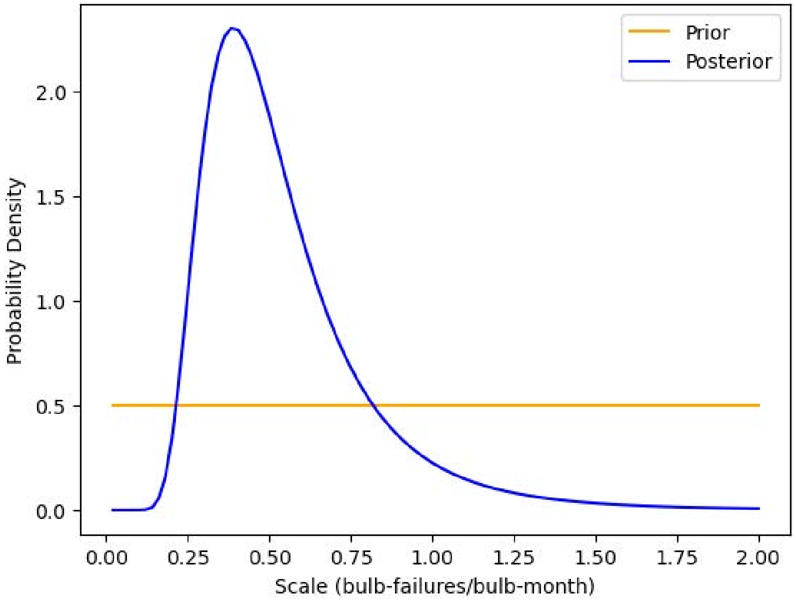


Figure 6.1: The plot of the uniform prior and Gamma posterior distributions for the lifespan of a projector bulb.

Problem 1. Write a function called `bernoulli_sampling()` that takes the following parameters: `p` a float that is the “fairness" of a coin and `n` the number of the Bernoulli trials. In this function simulate `n` tosses of a coin which gives heads with probability `p`. Then use that sample to calculate the posterior distribution on p given a uniform prior using Equation 6.2. Remember that p is the probability parameter, so this is the parameter θ we are trying to estimate.

For `p`=.2 and `n`=100 plot the posterior distribution and return the MAP estimate of p, which is also the MLE in this case. Be sure to give your plot a relevant title and axis labels.

Hint: In this case, f is the Binomial pmf $f(x) = p^{n\bar{x}}(1-p)^{n(1-\bar{x})}$. You do not need to calculate the integral in the denominator exactly; since you are using a finite approximation of the distributions, you may use a finite approximation of the integral. You may simulate the tosses of a coin by using `np.random.binomial()` or `scipy.stats.binom()`. Moreover, you may use `scipy.stats.uniform()` to generate the prior distribution, but for this problem it is not necessary given the definition of $\mathcal{U}(a,b)$. All of these functions accept a `size` parameter, defaulted to 1, that allows you to generate multiple samples at once. We will only be using a single sample in this problem so the returned value should be a single integer. Refer to scipy.stats for more documentation on discrete and continuous distributions info.

Non-Uniform Priors

While we are able to get good estimates, we leave a lot of the power of Bayesian statistics on the table when we only use a uniform prior. While the uniform prior is free from any preconceptions or biases, it also imparts the least amount of information. Using a non-uniform prior allows us to actually incorporate prior knowledge or assumptions into our model. If we have good reason to believe something about a parameter we are exploring before we even draw a sample, we can learn a lot more by accounting for those beliefs.

Example 2: Lifespan of a Projector Bulb with a Non-Uniform Prior

Consider the same initial set up as Example 1 (i.e. $X \sim \text{Gamma}(1, \lambda)$ and the sample of 7 bulb lives). But now, we have reason to believe that the lifespan sample originated from a distribution of $\text{Gamma}(2, 6)$. Using the prior $\text{Gamma}(2, 6)$, find the posterior pdf for λ, compute the MAP for λ, and plot the prior and posterior. The code is very similar with the only difference being the prior distribution.

```
>>> prior = gamma.pdf(x=fails, a=2, scale=1/6) # 2 failures in 6 months
>>> likelihood = np.array([(gamma.pdf(x=sample, a=1, scale=fail)).prod() for ↩
    fail in fails])
>>> integral = (likelihood*prior).sum()*(2/100)
>>> posterior = (likelihood*prior)/integral
>>> 1/fails[posterior.argmax()]
2.9117647058823524

>>> plt.plot(fails, prior, color="orange", label="Prior")
>>> plt.plot(fails, posterior, color="blue", label="Posterior")
>>> plt.ylabel("Probability Density")
>>> plt.xlabel("Scale (bulb-failures/bulb-month)")
>>> plt.legend()
>>> plt.show()
```

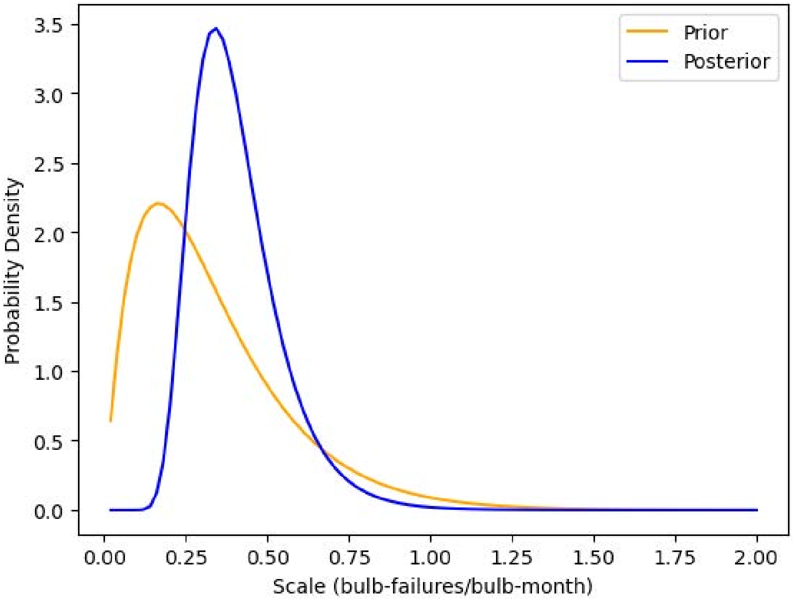


Figure 6.2: The plot of the uniform prior and Gamma posterior distributions for the lifespan of a projector bulb. Note that the initial missing piece of the prior is due to the fact we only selected 100 grid points for the scale. The graph becomes better with more grid points.

Notice how the posterior distribution for the non-uniform prior has a higher probability density and much narrower shape than one given by using a uniform prior. The MAP estimate is also higher than the uniform prior. Furthermore, notice how the prior distribution has a similar shape to the posterior distribution. This is due to *conjugacy*. Conjugacy is a special case where for certain likelihood functions, selecting a specific prior distribution results in the prior and posterior distributions having the same type of distribution. The prior distribution is then said to be a *conjugate prior* for the likelihood function. This is not normally seen in most distributions. But when it does appear, this is a very useful property as it allows us to easily calculate the posterior distribution and prevents us from having to calculate the marginal likelihood or the product of the likelihood and prior.

Overall, remember that the prior distributions represents our initial beliefs about the parameter. It is important to choose a prior that is consistent with the problem at hand as choosing a poor prior could require you to get more samples and lead to slower calculations and processes.

Problem 2. Suppose you choose a coin from a bag that produces coins of many weights. However, the bag seems to be more likely to produce coins that are strongly biased in favor of heads. You're unsure of which kind of coin you've drawn so in order to find out you perform 20 flips.

Write a function called `non_uniform_prior()` that takes the following parameters: `p` a float that is the "fairness" of a coin, `n` the size of the sample to be generated, and `prior` a SciPy distribution object which will act as the prior on p.

Similar to Problem 1, simulate `n` flips and calculate and plot the posterior distribution (with a title and axis labels).

Return the MAP estimate.

Examine the difference in confidence we can have in estimating the bias of the coin if the coin we draw gives heads 90% of the time as opposed to 40% of the time.

Because we think that coins biased in favor of heads are likely, we can choose a prior distribution that matches that assumption. In this case we will choose $\mathrm{Beta}(5, 1.5)$ as the prior distribution because it gives much more weight to parameters larger than .5. This is most easily achieved with `scipy.stats.beta(5,1.5)` and using the `pdf()` method to calculate $g(\theta)$

Sampling from a Markov Chain

A Markov chain is a way to model sequences of states or events. Markov chains make a few assumptions, one of those being that the probability of each state occurring is dependent only on the previous state. The relationship between the states are described by what is called a transition matrix.

Markov chains and sampling are like peanut butter and jelly: neither one really lives up to their full potential without the other. Given the transition matrix of a Markov chain, we can use sampling to better understand what that chain looks and acts like or to get a well-informed idea of what the future may hold.

Sampling from a simple (row stochastic) transition matrix like the one below is as simple as picking a starting state s_0, and then using the corresponding row to sample randomly using the probabilities in the row.

	a	b	c
a	0.7	0.1	0.2
b	0.5	0.4	0.1
c	0.1	0.8	0.1

For example, using the above transition matrix, let $s_0 = a$, so we will randomly sample from the array $[a, b, c]$ using the respective probabilities $[0.7, 0.1, 0.2]$. If the sample gives us c, we can set $s_1 = c$ and can continue the process to find $s_2, ..., s_n$.

Problem 3. Given the transition matrix below and assuming the 0th day is sunny, sample from the markov chain to give a possible forecast of the 10 following days. Return a list of strings, not including the 0th day.

	sun	rain	wind
sun	0.6	0.1	0.3
rain	0.2	0.6	0.2
wind	0.3	0.4	0.3

Hint: `np.random.choice()` may be helpful here.

PyMC

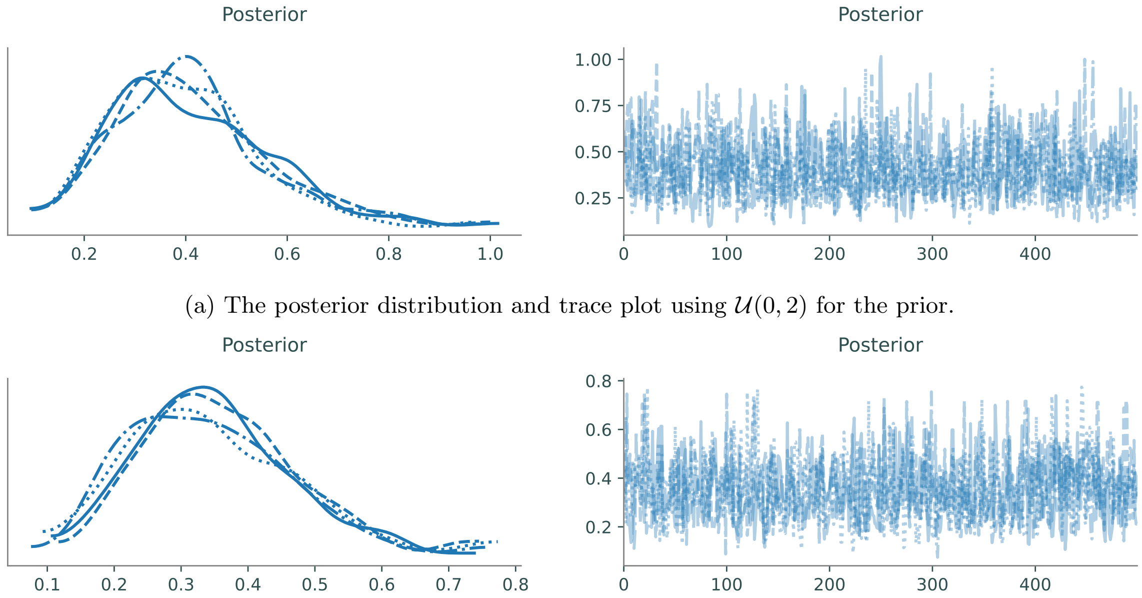
Python has many powerful sampling tools including PyMC, an efficient implementation of a method known as Monte Carlo Markov Chain (MCMC) Sampling. This is a useful technique as it constructs a Markov Chain whose steady state is a probability distribution that is difficult to sample from directly. Unlike our simple Markov Chain from the last problem, certain Markov Chains are abstract. PyMC gives us a way to work with these more complex scenarios.

Single Variable PyMC

Consider the following: owners of a restaurant are trying to decide if they should keep selling nachos. They gather the data for several months about how many people order nachos each day. One of the owners happened to take a class in Bayesian statistics in college, so she decides test her knowledge. She assumes the data are distributed as Poisson(λ) for some unknown value of λ, where λ has a prior of Gamma(2,2). She wishes to solve for λ and sets up a PyMC Model for the situation as follows:

```python
import numpy as np
import pymc as pm
import arviz as az   # visualization package

with pm.Model() as model:
    # define the prior of lambda as a Gamma(2, 2) distribution
    lam = pm.Gamma('lambda', alpha=2, beta=2)

    # define the likelihood of the data (called nacho_data) to be distributed
    # as Poisson where the expected value of the outcome (mu) is lam
    y = pm.Poisson('y', mu=lam, observed=nacho_data)

    # sample from the posterior
    trace = pm.sample(n)   # n is the desired number of samples
    az.plot_trace(trace)   # plot the posterior and trace plot for lambda

    new_lambda = trace.posterior['lambda']  # trace values of lambda as a list
    mean = float(new_lambda.mean())  # expected value of lambda
```

This code generates a model for the prior distribution of λ, and then incorporates that prior into a model for the Poisson likelihood. It then samples from the posterior of λ `n` times, from which we can estimate its expected value. The function `az.plot_trace()` plots both the posterior (on the left) as well as a *trace plot* (on the right), In each panel, you should see different lines with different colors or linestyles. These lines represent the different independent chains that were sample, and if the results are significantly different, it may indicate that there is something wrong with the model. This trace plots indicates how well the sampling converged. The rule of thumb is: the closer the trace plot resembles a fuzzy caterpillar, the better the Markov Chain converged to the posterior.



(a) The posterior distribution and trace plot using $\mathcal{U}(0, 2)$ for the prior.

(b) The posterior distribution and trace plot using Gamma(2,6) for the prior.

Figure 6.3: A comparison of the posterior and trace plots for the lifespan of a projector bulb using a uniform and non-uniform prior, as given in Examples 1 and 2, using PyMC and $n = 500$. Compare to the plots in Figures 1 and 2.

We will now reconsider the initial problem of the coin flip.

Problem 4. Write a function that accepts the coin flip data in array form and an integer n for the desired number of samples. Given data that flips a coin 100 times, assume the data are distributed as Bernoulli(p) for some unknown value of p, where p has a prior of Beta(1,1). Set up a PyMC model for this situation and sample from the posterior n times. Plot the trace plot and return the expected value of the posterior as a float, *not an array*.

Run the function with data generated by the following code

```
from scipy.stats import bernoulli
data = bernoulli.rvs(0.2, size=100)
```

Multivariate PyMC

Unlike the Poisson and Bernoulli distributions, many other distributions (including the Normal, Beta, Gamma, and Binomial distributions) have two or more parameters. These problems can really showcase the usefulness and ease of PyMC. Multivariate PyMC problems are coded up exactly the same way as the single variable example above, except that now there will be multiple priors defined separately, all of which will be called by the likelihood.

Problem 5. Write a function the accepts height data in array form and an integer n for the desired number of samples. Given a dataset of the measured heights of 100 men, assume the data are distributed as Normal(μ, $1/\tau$) where μ has a prior of Normal(m, s), and τ has a prior of Gamma(α, β). Your function should have default values `m=180`, `s=10`, `alpha=2`, and `beta=10`. Set up a PyMC model for this situation and sample from the posterior n times. Plot the trace plots for μ and τ, and return the expected value of the posterior of μ as a float, *not as an array*.

Run the function with data generated by the following code

```
heights = np.random.normal(180, 10, 100)
```

Hint: `pm.Normal()` uses parameters `mu` and either `sigma` or `tau`, where the variance of the distribution is given by `sigma`^{2} or 1/`tau` respectively.

Additional Materials

We will describe some of the most common distributions and what they model. Note X is a random variable and the support is the domain where the pdf/pmf is nonzero.

- Discrete Distributions:
  - Bernoulli: This is normally used to model the probability of success in a single trial when the outcome can be categorized into exactly one of two categories. The support is $\{0, 1\}$. We normally write $X \sim \text{Bernouilli}(p)$, p is the probability of success and is the parameter.
  - Binomial: This distribution is used to model the number of successes in a fixed number of Bernoulli trials. The support is $\{0, 1, \ldots, n\}$ for n trials. We write $X \sim \text{B}(n, p)$ or $X \sim \text{Binomial}(n, p)$. n and p are the parameters.
  - Poisson: We use this distribution to model the number of occurrences that occur in a fixed interval of time or space. The support is $\{1, 2, 3, \ldots\}$. We denote this as $X \sim \text{Poisson}(\lambda)$ where λ is the parameter and is an average rate of occurrence.
- Continuous Distributions:
  - Uniform: This distribution is best used when the every outcome in the sample space ω is equally likely. The support is $[a, b]$, and we denote this as $X \sim \mathcal{U}(a, b)$ where a and b are the parameters.
  - Normal/Gaussian: This distribution is used to analyze and show data near the mean, and how that is more frequent than data far from the mean. The support is $(-\infty, \infty)$. We write $X \sim \mathcal{N}(\mu, \sigma^2)$ where μ is the mean and σ^2 is the variance. Some books use σ as the standard deviation (i.e. the square root of the variance).
  - Gamma: This describes the waiting time for $a > 0$ events to occur in a homogeneous Poisson process with rate $b > 0$ (i.e. the time between events). The support is $[0, \infty)$. We write $X \sim \text{Gamma}(a, b)$ where a is the shape and b, the rate, are the parameters. The scale is $\frac{1}{b}$, and note sometimes the rate is denoted by λ. The Exponential distribution is a special case of the Gamma distribution where $a = 1$.
  - Chi-Squared is another special case of the Gamma distribution where $a = \frac{n}{2}$ and $b = \frac{1}{2}$ where n is the degrees of freedom.
  - Beta: This is best used to describe random variables with a range between 0 and 1 since the support is $[0, 1]$. We write $X \sim \text{Beta}(a, b)$ where a and b are the parameters.

Here is an article that gives a brief overview of what phenomenon some of the common distributions describe (including the ones we have here). This other article gives a good overview of how to choose a prior distribution for a Bayesian model.

7 The Discrete Fourier Transform

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

Achtung!

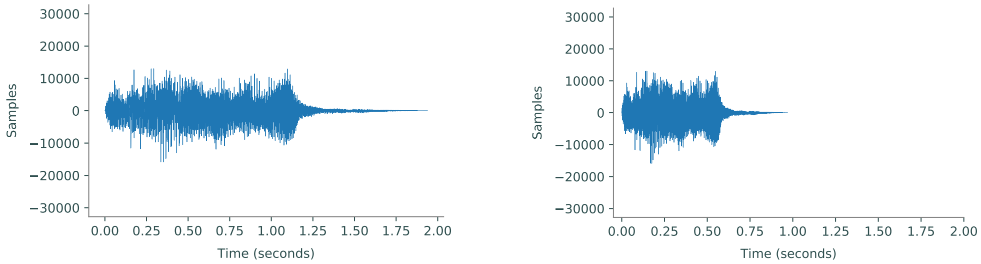
Completing the implementation of the `SoundWave` class in this lab is strongly recommended, as it will also be used in the Convolution and Filtering lab.

Digital Audio Signals

Sound waves have two important characteristics: *frequency*, which determines the pitch of the sound, and *intensity* or *amplitude*, which determines the volume of the sound. Computers use *digital audio signals* to approximate sound waves. These signals have two key components: *sample rate*, which relates to the frequency of sound waves, and *samples*, which measure the amplitude of sound waves at a specific instant in time.

To see why the sample rate is necessary, consider an array with samples from a sound wave. The sound wave can be arbitrarily stretched or compressed to make a variety of sounds. If compressed, the sound becomes shorter and has a higher pitch. Similarly, the same set of samples with a lower sample rate becomes stretched and has a lower pitch.

Given the rate at which a set of samples is taken, the wave can be reconstructed exactly as it was recorded. In most applications, this sample rate is measured in *Hertz* (Hz), the number of samples taken per second. The standard rate for high quality audio is 44100 equally spaced samples per second, or 44.1 kHz.



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

(b) Compressed plot of `tada.wav`.

Figure 7.1: Plots of the same set of samples from a sound wave with varying sample rates. The plot on the left is the plot of the samples with the original sample rate. The sample rate of the plot on the right has been doubled, resulting in a compression of the actual sound when played back.

Wave File Format

One of the most common audio file formats across operating systems is the *wave* format, also called `wav` after its file extension. SciPy has built-in tools to read and create `wav` files. To read a `wav` file, use `scipy.io.wavfile.read()`. This function returns the signal's sample rate and its samples.

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

Sound waves can be visualized by plotting time against the amplitude of the sound, as in Figure 7.1. The amplitude of the sound at a given time is just the value of the sample at that time. Since the sample rate is given in samples per second, the length of the sound wave in seconds is found by dividing the number of samples by the sample rate:

$$\frac{\text{num samples}}{\text{sample rate}} = \frac{\text{num samples}}{\text{num samples/second}} = \text{second}. \tag{7.1}$$

Problem 1. Write a `SoundWave` class for storing digital audio signals.

1. The constructor should accept an integer sample rate and an array of samples. Store each input as an attribute.

2. Write a method that plots the stored sound wave. Use (7.1) to correctly label the x-axis in terms of seconds, and set the y-axis limits to $[-32768, 32767]$ (the reason for this is discussed in the next section).

Use SciPy to read `tada.wav`, then instantiate a corresponding `SoundWave` object and display its plot. Compare your plot to Figure 7.1a.

Scaling

To write to a `wav` file, use `scipy.io.wavfile.write()`. This function accepts the name of the file to write to, the sample rate, and the array of samples as parameters.

```
>>> import numpy as np

# Write a 2-second random sound wave sampled at a rate of 44100 Hz.
>>> samples = np.random.randint(-32768, 32767, 88200, dtype=np.int16)
>>> wavfile.write("white_noise.wav", 44100, samples)
```

For `scipy.io.wavfile.write()` to correctly create a `wav` file, the samples must be one of four numerical datatypes: 32-bit floating point (`np.float32`), 32-bit integers (`np.int32`), 16-bit integers (`np.int16`), or 8-bit unsigned integers (`np.uint8`). If samples of a different type are passed into the function, it may still write a file, but the sound will likely be distorted in some way. In this lab, we only work with 16-bit integer samples, unless otherwise specified.

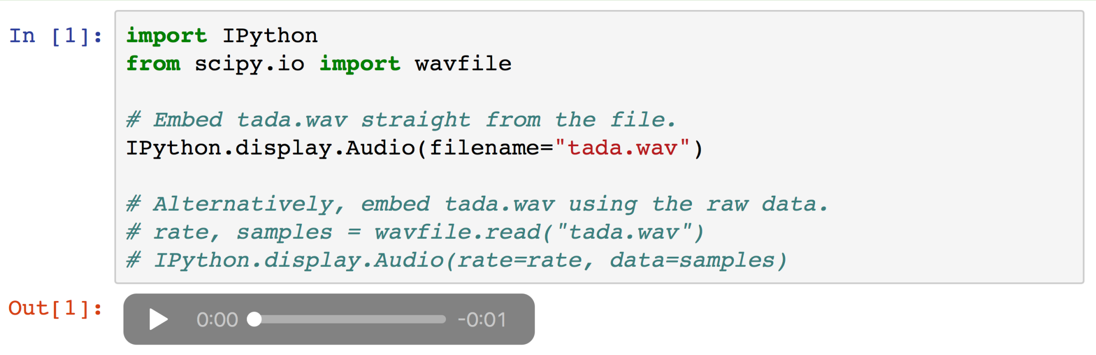
A 16-bit integer is an integer between -32768 and 32767, inclusive. If the elements of an array of samples are not all within this range, the samples must be scaled before writing to a file: multiply the samples by 32767 (the largest number in the 16-bit range) and divide by the largest sample magnitude. This ensures the most accurate representation of the sound and sets it to full volume.

$$\texttt{np.int16}\left(\left(\frac{\text{original samples}}{\max(|\text{original samples}|)}\right) \times 32767\right) = \text{scaled samples} \tag{7.2}$$

Because 16-bit integers can only store numbers within a certain range, it is important to multiply the original samples by the largest number in the 16-bit range *after* dividing by the largest sample magnitude. Otherwise, the results of the multiplication may be outside the range of integers that can be represented, causing overflow errors. Also, samples may sometimes contain complex values, especially after some processing. Make sure to scale and export only the real part (use the `real` attribute of the array).

Note

The IPython API includes a tool for embedding sounds in a Jupyter Notebook. The function `IPython.display.Audio()` accepts either a file name or a sample rate (`rate`) and an array of samples (`data`); calling the function generates an interactive music player in the Notebook.



```
In [1]: import IPython
        from scipy.io import wavfile

        # Embed tada.wav straight from the file.
        IPython.display.Audio(filename="tada.wav")

        # Alternatively, embed tada.wav using the raw data.
        # rate, samples = wavfile.read("tada.wav")
        # IPython.display.Audio(rate=rate, data=samples)

Out[1]:
```

Achtung!

Turn the volume down before listening to any of the sounds in this lab.

Problem 2. Add a method to the `SoundWave` class that accepts a file name and a boolean `force`. Write to the specified file using the stored sample rate and the array of samples. If the array of samples does not have `np.int16` as its data type, or if `force` is `True`, scale the samples as in (7.2) before writing the file.

Use your method to create two new files that contains the same sound as `tada.wav`: one without scaling, and one with scaling (use `force=True`). Use `IPython.display.Audio()` to display `tada.wav` and the new files. All three files should sound identical, except the scaled file should be louder than the other two.

Generating Sounds

Sinusoidal waves correspond to pure frequencies, like a single note on the piano. Recall that the function $\sin(x)$ has a period of 2π. To create a specific tone for 1 second, we sample from the sinusoid with period 1,

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

where k is the desired frequency. According to (7.1), generating a sound that lasts for s seconds at a sample rate r requires rs equally spaced samples in the interval $[0, s]$.

Problem 3. Write a function that accepts floats k and s. Create a `SoundWave` instance containing a tone with frequency k that lasts for s seconds. Use a sample rate of $r = 44100$.

The following table shows some frequencies that correspond to common notes. Octaves of these notes are obtained by doubling or halving these frequencies.

Note	Frequency (Hz)
A	440
B	493.88
C	523.25
D	587.33
E	659.25
F	698.46
G	783.99
A	880

Use your function to generate an A tone lasting for 2 seconds.

Problem 4. Digital audio signals can be combined by addition or concatenation. Adding samples overlays tones so they play simultaneously; concatenated samples plays one set of samples after the other with no overlap.

1. Implement the `__add__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A + B` creates a new `SoundWave` object whose samples are the element-wise sum of the samples from `A` and `B`. Raise a `ValueError` if the sample arrays from `A` and `B` are not the same length.

   Use your method to generate a three-second A minor chord (A, C, and E together).

2. Implement the `__rshift__()` magic method[a] for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A >> B` creates a new `SoundWave` object whose samples are the concatenation of the samples from `A`, then the samples from `B`. Raise a `ValueError` if the sample rates from the two objects are not equal.
   (Hint: `np.concatenate()`, `np.hstack()`, and/or `np.append()` may be useful.)

   Use your method to generate the arpeggio A $\to$ C $\to$ E, where each pitch lasts one second.

Consider using these two methods to produce elementary versions of some simple tunes.

[a] The `>>` operator is a *bitwise shift operator* and is usually reserved for operating on binary numbers.

The Discrete Fourier Transform

As with the chords generated above, all sound waves are sums of varying amounts of different frequencies (pitches). In the case of the discrete samples $\mathbf{f} = \begin{bmatrix} f_0 & f_1 & \cdots & f_{n-1} \end{bmatrix}^\mathsf{T}$ that we have worked with thus far, each f_i gives information about the amplitude of the sound wave at a specific instant in time. However, sometimes it is useful to find out what frequencies are present in the sound wave and in what amount.

We can write the sound wave sample as a sum

$$\mathbf{f} = \sum_{k=0}^{n-1} c_k \mathbf{w}_n{}^{(k)}, \tag{7.3}$$

where $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$, called the *discrete Fourier basis*, represents various frequencies. The coefficients c_k represent the amount of each frequency present in the sound wave.

The *discrete Fourier transform (DFT)* is a linear transformation that takes $\mathbf{f}$ and finds the coefficients $\mathbf{c} = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix}^\mathsf{T}$ needed to write $\mathbf{f}$ in this frequency basis. Later in the lab, we will convert the index k to a value in Hertz to find out what frequency c_k corresponds to.

Because the sample $\mathbf{f}$ was generated by taking n evenly spaced samples of the sound wave, we generate the basis $\{\mathbf{w}_n^{(k)}\}_{k=0}^{n-1}$ by taking n evenly spaced samples of the frequencies represented by the oscillating functions $\{e^{-2\pi ikt/n}\}_{k=0}^{n-1}$. (Note that $i = \sqrt{-1}$, the imaginary unit, is represented as `1j` in Python). This yields

$$\mathbf{w}_n^{(k)} = \begin{bmatrix} \omega_n^0 & \omega_n^{-k} & \cdots & \omega_n^{-(n-1)k} \end{bmatrix}^\mathsf{T}, \tag{7.4}$$

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

The DFT is then represented by the change of basis matrix

$$F_n = \frac{1}{n}\begin{bmatrix}\mathbf{w}_n^0 & \mathbf{w}_n^1 & \mathbf{w}_n^2 & \cdots & \mathbf{w}_n^{n-1}\end{bmatrix} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{bmatrix}, \tag{7.5}$$

and we can take the DFT of f by calculating

$$\mathbf{c} = F_n\mathbf{f}. \tag{7.6}$$

Note that the DFT depends on the number of samples n, since the discrete Fourier basis we use depends on the number of samples taken. The larger n is, the closer the frequencies approximated by the DFT will be to the actual frequencies present in the sound wave.

Achtung!

There are several different conventions for defining the DFT. For example, instead of (7.6), `scipy.fftpack.fft()` uses the formula

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

where F_n is as given (7.5). Denoting this version of the DFT as $\hat{F}_n\mathbf{f} = \hat{\mathbf{c}}$, we have $nF_n = \hat{F}_n$ and $n\mathbf{c} = \hat{\mathbf{c}}$. The conversion is easy, but it is very important to be aware of which convention a particular implementation of the DFT uses.

Problem 5. Write a function that accepts an array $\mathbf{f}$ of samples. Use 7.6 to calculate the coefficients $\mathbf{c}$ of the DFT of $\mathbf{f}$. Include the $1/n$ scaling in front of the sum.

Test your implementation on small, random arrays against `scipy.fftpack.fft()`, scaling your output $\mathbf{c}$ to match SciPy's output $\hat{\mathbf{c}}$. Once your function is working, try to optimize it so that the entire array of coefficients is calculated in the one line.
(Hint: Use array broadcasting.)

The Fast Fourier Transform

Calculating the DFT of a vector of n samples using only (7.6) is at least $O(n^2)$, which is incredibly slow for realistic sound waves. Fortunately, due to its inherent symmetry, the DFT can be implemented as a recursive algorithm by separating the computation into even and odd indices. This method of calculating the DFT is called the *fast Fourier transform* (FFT) and runs in $O(n \log n)$ time.

Algorithm 1 The fast Fourier transform for arrays with 2^a entries for some $a \in \mathbb{N}$.

```
 1: procedure SIMPLE_FFT($\mathbf{f}$, $N$)
 2:     procedure SPLIT($\mathbf{g}$)
 3:         $n \leftarrow \text{size}(\mathbf{g})$
 4:         if $n \leq N$ then
 5:             return $nF_n\mathbf{g}$          ▷ Use the function from Problem 5 for small enough $\mathbf{g}$.
 6:         else
 7:             even ← SPLIT($\mathbf{g}_{::2}$)      ▷ Get the DFT of every other entry of $\mathbf{g}$, starting from 0.
 8:             odd ← SPLIT($\mathbf{g}_{1::2}$)      ▷ Get the DFT of every other entry of $\mathbf{g}$, starting from 1.
 9:             $\mathbf{z} \leftarrow \text{zeros}(n)$
10:             for $k = 0, 1, \ldots, n-1$ do         ▷ Calculate the exponential parts of the sum.
11:                 $z_k \leftarrow e^{-2\pi ik/n}$
12:             $m \leftarrow n\,//\,2$          ▷ Get the middle index for $\mathbf{z}$ (// is integer division).
13:             return [even + $\mathbf{z}_{:m}\odot$odd, even + $\mathbf{z}_{m:}\odot$odd]   ▷ Concatenate two arrays of length $m$.
14:     return SPLIT($\mathbf{f}$) / size($\mathbf{f}$)
```

Note that the base case in lines 4–5 of Algorithm 1 results from setting $n = 1$ in (7.6), yielding the single coefficient $c_0 = g_0$. The $\odot$ in line 13 indicates the component-wise product

$$\mathbf{f} \odot \mathbf{g} = \begin{bmatrix} f_0 g_0 & f_1 g_1 & \cdots & f_{n-1} g_{n-1} \end{bmatrix}^\mathsf{T},$$

which is also called the *Hadamard product* of **f** and **g**.

This algorithm performs significantly better than the naïve implementation of the DFT, but the simple version described in Algorithm 1 only works if the number of original samples is exactly a power of 2. SciPy's FFT routines avoid this problem by padding the sample array with zeros until the size is a power of 2, then executing the remainder of the algorithm from there. Of course, SciPy also uses various other tricks to further speed up the computation.

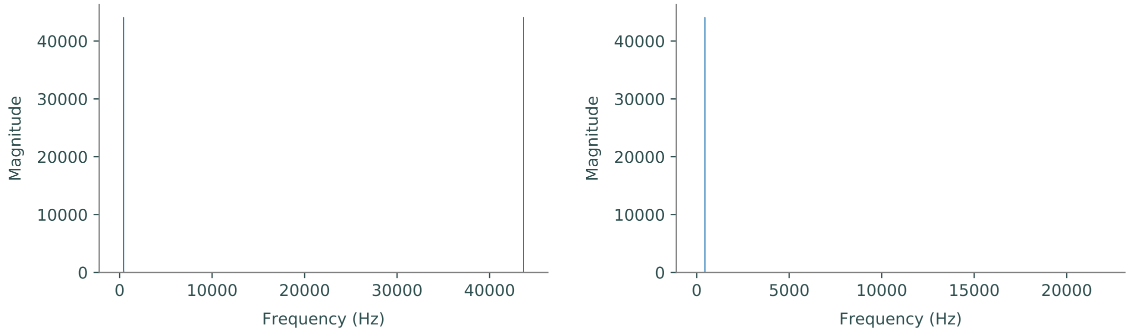
Problem 6. Write a function that accepts an array **f** of n samples where n is a power of 2. Use Algorithm 1 to calculate the DFT of **f**.
(Hint: eliminate the loop in lines 10–11 with `np.arange()` and array broadcasting, and use `np.concatenate()` or `np.hstack()` for the concatenation in line 13.)

Test your implementation on random arrays against `scipy.fftpack.fft()`, scaling your output **c** to match SciPy's output $\hat{\mathbf{c}}$. Time your function from Problem 5, this function, and SciPy's function on an array with 8192 entries.
(Hint: Use `%time` in Jupyter Notebook to time a single line of code.)

Visualizing the DFT

The graph of the DFT of a sound wave is useful in a variety of applications. While the graph of the sound in the time domain gives information about the amplitude (volume) of a sound wave at a given time, the graph of the DFT shows which frequencies (pitches) are present in the sound wave. Plotting a sound's DFT is referred to as plotting in the *frequency domain*.

As a simple example, the single-tone notes generated by the function in Problem 3 contain only one frequency. For instance, Figure 7.2a graphs the DFT of an A tone. However, this plot shows two frequency spikes, despite there being only one frequency present in the actual sound. This is due to symmetries inherent to the DFT; for frequency detection, the second half of the plot can be ignored as in Figure 7.2b.



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

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

Figure 7.2: Plots of the DFT with and without symmetries. Notice that the x-axis of the symmetrical plot on the left goes up to 44100 (the sample rate of the sound wave) while the x-axis of the non-symmetric plot on the right goes up to only 22050 (half the sample rate). Also notice that the spikes occur at 440 Hz and 43660 Hz (which is $44100 - 440$).

The DFT of a more complicated sound wave has many frequencies, each of which corresponds to a different tone present in the sound wave. The magnitude of the coefficients indicates a frequency's influence in the sound wave; a greater magnitude means that the frequency is more influential.

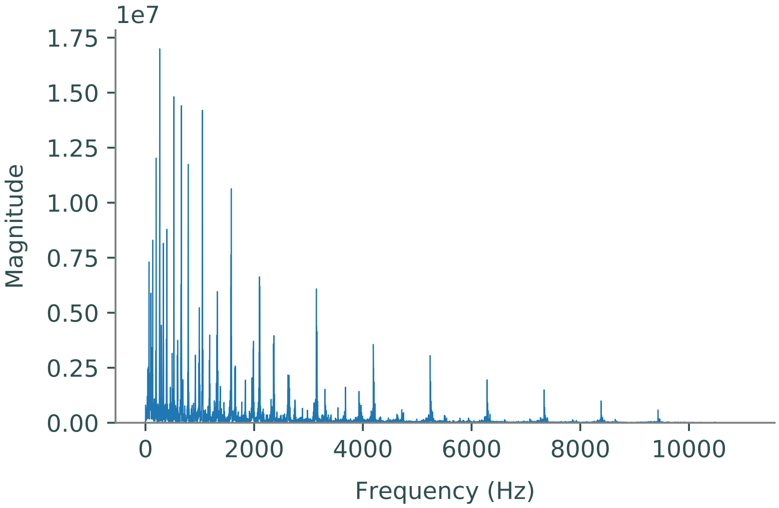


Figure 7.3: The discrete Fourier transform of `tada.wav`. Each spike in the graph corresponds to a frequency present in the sound wave. Since the sample rate of `tada.wav` is 22050 Hz, the plot of its DFT without symmetries only goes up to 11025 Hz, half of its sample rate.

Plotting Frequencies

Since the DFT represents the frequency domain, the x-axis of a plot of the DFT should be in terms of Hertz, which has units $1/s$. In other words, to plot the magnitudes of the Fourier coefficients against the correct frequencies, we must convert the frequency index k of each c_k to Hertz. This can be done by multiplying the index by the sample rate and dividing by the number of samples:

$$\frac{k}{\text{num samples}} \times \frac{\text{num samples}}{\text{second}} = \frac{k}{\text{second}}. \tag{7.7}$$

In other words, $kr/n = v$, where r is the sample rate, n is the number of samples, and v is the resulting frequency.

Problem 7. Modify your `SoundWave` plotting method from Problem 1 so that it accepts a boolean defaulting to `False`. If the boolean is `True`, take the DFT of the stored samples and plot—in a new subplot—the frequencies present on the x-axis and the magnitudes of those frequencies (use `np.abs()` to compute the magnitude) on the y-axis. Only display the first half of the plot (as in Figure 7.2b), and use (7.7) to adjust the x-axis so that it correctly shows the frequencies in Hertz. Use SciPy to calculate the DFT.

Display the DFT plots of the A tone and the A minor chord from Problem 4. Compare your results to Figures 7.2b and 7.4.

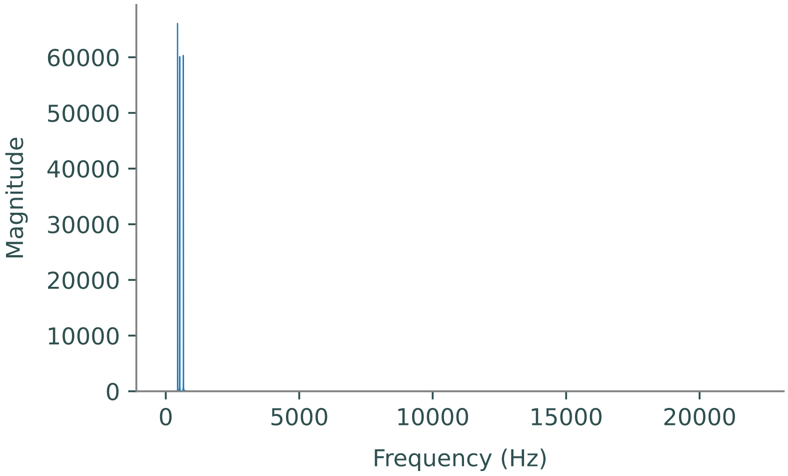


Figure 7.4: The DFT of the A minor chord.

If the frequencies present in a sound are already known before plotting its DFT, the plot may be interesting, but little new information is actually revealed. Thus, the main applications of the DFT involve sounds in which the frequencies present are unknown. One application in particular is sound filtering, which will be explored in greater detail in a subsequent lab. The first step in filtering a sound is determining the frequencies present in that sound by taking its DFT.

Consider the DFT of the A minor chord in Figure 7.4. This graph shows that there are three main frequencies present in the sound. To determine what those frequencies are, find which indices of the array of DFT coefficients have the three largest values, then scale these indices the same way as in (7.7) to translate the indices to frequencies in Hertz.

Problem 8. The file mystery_chord.wav contains an unknown chord. Use the DFT and the frequency table in Problem 3 to determine the individual notes that are present in the sound. (Hint: np.argsort() may be useful.)

8 Convolution and Filtering

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

> ACHTUNG!
>
> This lab is a continuation of The Discrete Fourier Transform lab and will expand the `SoundWave` class that was implemented there. Before beginning this lab, make sure to copy your code for the methods `__init__()`, `plot()`, `export()`, `__add__()`, and `__rshift__()` into the `SoundWave` class in the new Jupyter Notebook.

Convolution

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

$$(\mathbf{f} * \mathbf{g})_k = \sum_{j=0}^{n-1} f_{k-j} g_j, \qquad k = 0, 1, 2, \ldots, n-1. \tag{8.1}$$

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples; even a one-minute signal has $2,646,000$ samples if it is recorded at the standard rate of $44,100$ samples per second ($44,100$ Hz). The naïve method of using the sum in (8.1) n times is $O(n^2)$, which is often too computationally expensive for convolutions of this size.

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

$$F_n(\mathbf{f} * \mathbf{g}) = n(F_n\mathbf{f}) \odot (F_n\mathbf{g}). \tag{8.2}$$

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

$$\mathbf{f} * \mathbf{g} = nF_n^{-1}((F_n\mathbf{f}) \odot (F_n\mathbf{g})), \tag{8.3}$$

where F_n^{-1} is the *inverse discrete Fourier transform* (IDFT). The fast Fourier transform (FFT) puts the cost of (8.3) at $O(n \log n)$, a huge improvement over the naïve method.

Note

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

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

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

Achtung!

SciPy uses a different convention to define the DFT and IDFT than this and the previous lab, resulting in a slightly different form of the convolution theorem. Writing SciPy's DFT as $\hat{F}_n$ and its IDFT as $\hat{F}_n^{-1}$, we have $\hat{F}_n = nF_n$, so (8.3) becomes

$$\mathbf{f} * \mathbf{g} = \hat{F}_n^{-1}((\hat{F}_n\mathbf{f}) \odot (\hat{F}_n\mathbf{g})), \tag{8.4}$$

without a factor of n. Use (8.4), not (8.3), when using `fft()` and `ifft()` from `scipy.fftpack`.

Circular Convolution

The definition (8.1) and the identity (8.3) require $\mathbf{f}$ and $\mathbf{g}$ to be periodic vectors. However, the convolution $\mathbf{f}*\mathbf{g}$ can always be computed by simply treating each vector as periodic. The convolution of two raw sample vectors is therefore called the *periodic* or *circular convolution*. This strategy mixes sounds from the end of each signal with sounds at the beginning of each signal.

Problem 1.

Implement the __mul__() magic method for the SoundWave class so that if A and B are SoundWave instances, A * B creates a new SoundWave object whose samples are the circular convolution of the samples from A and B. If the samples from A and B are not the same length, append zeros to the shorter array to make them the same length before convolving. Use scipy.fftpack and (8.4) to compute the convolution, and raise a ValueError if the sample rates from A and B are not equal.

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

```
>>> rate = 22050         # Create 2 seconds of white noise at a given rate.
>>> white_noise = np.random.randint(-32767, 32767, rate*4, dtype=np.int16)
```

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

Linear Convolution

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

Problem 2.

Implement the __pow__() magic method for the SoundWave class so that if A and B are SoundWave instances, A ** B creates a new SoundWave object whose samples are the linear convolution of the samples from A and B. Raise a ValueError if the sample rates from A and B are not equal.

Because scipy.fftpack performs best when the length of the inputs is a power of 2, start by computing the smallest 2^a such that $2^a \geq n+m-1$, where $a \in \mathbb{N}$ and n and m are the number of samples from A and B, respectively. Append zeros to each sample so that they each have 2^a entries, then compute the convolution of these padded samples using (8.4). Use only the first $n+m-1$ entries of this convolution as the samples of the returned SoundWave object.

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

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

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

Filtering Frequencies with the DFT

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

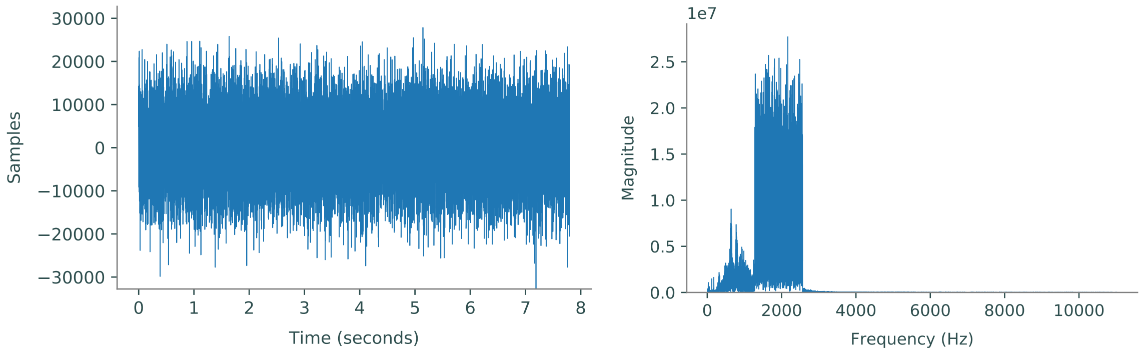


Figure 8.1: The time-domain plot (left) and DFT (right) of `noisy1.wav`.

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

1. Compute the integer indices k_{low} and k_{high} corresponding to v_{low} and v_{high}, respectively.

2. Set the entries of the signal's DFT from k_{low} to k_{high} and from $n - k_{\text{high}}$ to $n - k_{low}$ to zero, effectively removing those frequencies from the signal.

3. Take the IDFT of the modified DFT to obtain the cleaned signal.

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

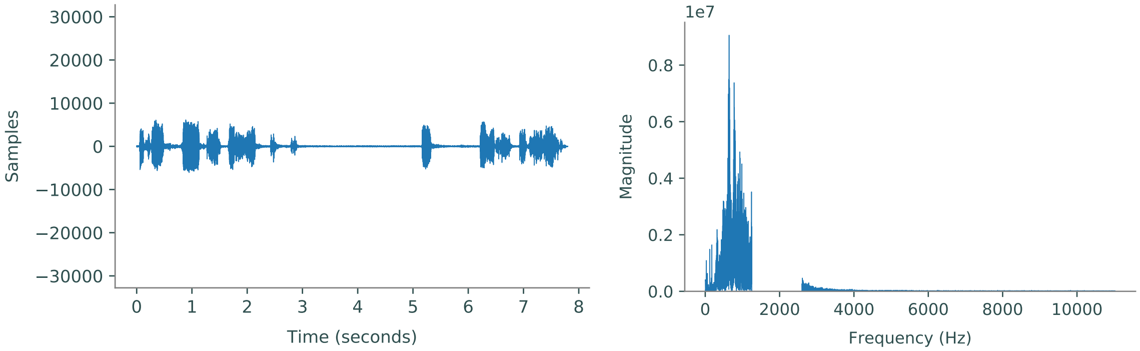


Figure 8.2: The time-domain plot (left) and DFT (right) of `noisy1.wav` after being cleaned.

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

Test your method by cleaning `noisy1.wav`. Then clean `noisy2.wav`, which also has some artificial noise that obscures the intended sound. Embed the original and cleaned versions of both files in the notebook.
(Hint: plot the DFT of `noisy2.wav` to determine which frequencies to eliminate.)

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

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

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

The file `vuvuzela.wav`[a] is a stereo sound with two channels. Use your function from Problem 4 to clean the sound clip by filtering out the vuvuzela frequencies in each channel. Recombine the two cleaned samples. Embed the original and cleaned versions in the notebook.

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

The Two-dimensional Discrete Fourier Transform

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

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

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

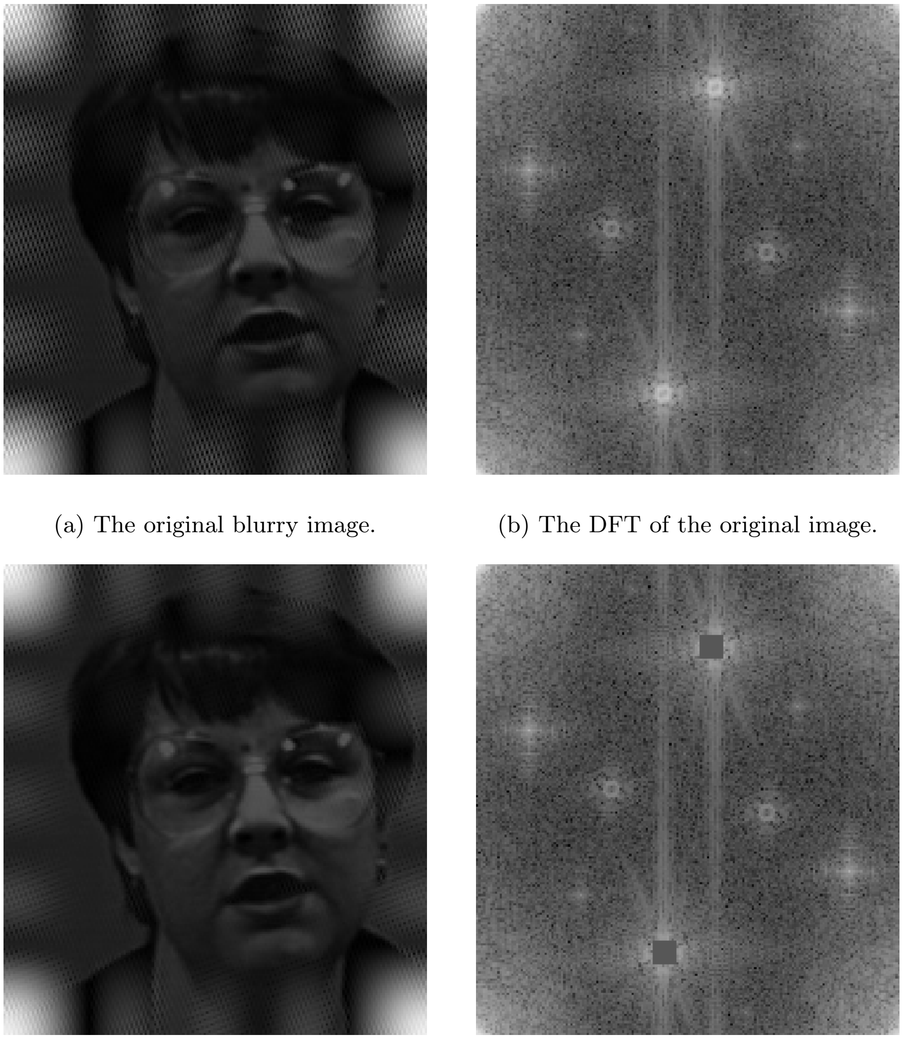
Just as the one-dimensional DFT can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove "noise" in images. The procedure is similar to the filtering technique in Problems 4 and 5: take the two-dimensional DFT of the image matrix, modify certain entries of the DFT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

To begin cleaning an image with the DFT, take the two-dimensional DFT of the image matrix. Identify *spikes*—abnormally high frequency values that may be causing the noise—in the image DFT by plotting the log of the magnitudes of the Fourier coefficients. With `cmap="gray"`, spikes show up as bright spots. See Figures 8.3a–8.3b.

```
# Read the image.
>>> from imageio.v3 import imread
>>> image = imread("noisy_face.png")

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

Instead of setting spike frequencies to zero (as was the case for sounds), replace them with values that are similar to those around them. There are many ways to do this, but one convention is to simply "patch" each spike by setting portions of the DFT matrix to some set value, such as the mean of the DFT array. See Figure 8.3d.

Once the spikes have been covered, take the IDFT of the modified DFT to get a (hopefully cleaner) image. Notice that Figure 8.3c still has noise present, but it is a slight improvement over the original. However, it often suffices to remove some of the noise, even if it is not possible to remove it all with this method.



(a) The original blurry image.

(b) The DFT of the original image.

(c) The improved image.

(d) The DFT of the improved image.

Figure 8.3: To remove noise from an image, take the DFT of the image and replace the abnormalities with values more consistent with the rest of the DFT. Notice that the new image is less noisy, but only slightly. This is because only some of the abnormalities in the DFT were changed; in order to further decrease the noise, we would need to further alter the DFT.

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

Use the two-dimensional DFT to clean up the image enough so that the year in the bottom right corner is legible. This may require a little trial and error. Compare the noisy and cleaned images side by side in subplots and identify the year on the sticker.

9 Introduction to Wavelets

Lab Objective: *Wavelets are used to sparsely represent information. This makes them useful in a variety of applications. We explore both the one- and two-dimensional discrete wavelet transforms using various types of wavelets. We then use a Python package called PyWavelets for further wavelet analysis including image cleaning and image compression.*

Wavelet Functions

Wavelets families are sets of orthogonal functions (wavelets) designed to decompose nonperiodic, piecewise continuous functions. These families have four types of wavelets: mother, daughter, father, and son functions. Father and son wavelets contain information related to the general movement of the function, while mother and daughter wavelets contain information related to the details of the function. The father and mother wavelets are the basis of a family of wavelets. Son and daughter wavelets are just scaled translates of the father and mother wavelets, respectively.

Haar Wavelets

The *Haar Wavelet* family is one of the most widely used wavelet families in wavelet analysis. This set includes the father, mother, son, and daughter wavelets defined below. The Haar father (scaling) function is given by

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

The Haar son wavelets are scaled and translated versions of the father wavelet:

$$\varphi_{jk}(x) = \varphi(2^j x - k) = \begin{cases} 1 & \text{if } \frac{k}{2^j} \leq x < \frac{k+1}{2^j} \\ 0 & \text{otherwise.} \end{cases}$$

The Haar mother wavelet function is defined as

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

The Haar daughter wavelets are scaled and translated versions of the mother wavelet

$$\psi_{jk} = \psi(2^j x - k)$$

Wavelet Decompositions

Information (such as a mathematical function or signal) can be stored and analyzed by considering its *wavelet decomposition*. A *wavelet decomposition* is a linear combination of wavelets. For example, a mathematical function f can be approximated as a combination of Haar son and daughter wavelets as follows:

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

where $m < n$, and all but a finite number of the a_k and $b_{j,k}$ terms are nonzero. The a_k terms are often referred to as *approximation coefficients* while the $b_{j,k}$ terms are known as *detail coefficients*. The approximation coefficients typically capture the broader, more general features of a signal while the detail coefficients capture smaller details and noise.

A wavelet decomposition can be done with any family of wavelet functions. Depending on the properties of the wavelet and the function (or signal) f, f can be approximated to an arbitrary level of accuracy. Each arbitrary wavelet family has a mother wavelet ψ and a father wavelet φ which are the basis of the family. A countably infinite set of wavelet functions (daughter and son wavelets) can be generated using dilations and shifts of the first two functions where $m, k \in \mathbb{Z}$:

$$\begin{aligned} \psi_{m,k}(x) &= \psi(2^m x - k) \\ \varphi_{m,k}(x) &= \varphi(2^m x - k). \end{aligned}$$

The Discrete Wavelet Transform

The mapping from a function to a sequence of wavelet coefficients is called the *discrete wavelet transform*. The discrete wavelet transform is analogous to the discrete Fourier transform. Now, instead of using trigonometric functions, different families of basis functions are used.

In the case of finitely-sampled signals and images, there exists an efficient algorithm for computing the wavelet decomposition. Commonly used wavelets have associated high-pass and low-pass filters which are derived from the wavelet and scaling functions, respectively.

When the low-pass filter is convolved with the sampled signal, low frequency (also known as approximation) information is extracted. This is similar to turning up the bass on a speaker, which extracts the low frequencies of a sound wave. This filter highlights the overall (slower-moving) pattern without paying too much attention to the high frequency details and extracts the approximation coefficients.

When the high-pass filter is convolved with the sampled signal, high frequency information (also known as detail) is extracted. This is similar to turning up the treble on a speaker, which extracts the high frequencies of a sound wave. This filter highlights the small changes found in the signal and extracts the detail coefficients.

The two primary operations of the algorithm are the discrete convolution and downsampling, denoted $*$ and DS, respectively. First, a signal is convolved with both filters. The convolutions fold the signal back on itself so the resulting array is the same size but half the information is duplicated, *downsampling* is then required to eliminate the repeated information. In the context of this lab, a *filter bank* is the combined process of convolving with a filter, and then downsampling. The result will be an array of approximation coefficients A and an array of detail coefficients D. This process can be repeated on the new approximation to obtain another layer of approximation and detail coefficients. See Figure 9.1.

A common lowpass filter is the averaging filter. Given an array $\mathbf{x}$, the averaging filter produces an array $\mathbf{y}$ where y_n is the average of x_n and x_{n-1}. In other words, the averaging filter convolves an array with the array $L = \begin{bmatrix}\frac{1}{2} & \frac{1}{2}\end{bmatrix}$. This filter preserves the main idea of the data. The corresponding highpass filter is the distance filter. Given an array $\mathbf{x}$, the distance filter produces an array $\mathbf{y}$ where y_n is the distance between x_n and x_{n-1} ($|x_n - x_{n-1}|$). In other words, the difference filter convolves an array with the array $H = \begin{bmatrix}-\frac{1}{2} & \frac{1}{2}\end{bmatrix}$. This filter preserves the details of the data.

For the Haar Wavelet, we will use the lowpass and highpass filters mentioned. In order for this filters to have inverses, the filters must be normalized (for more on why this is, see Additional Materials). The resulting lowpass and highpass filters for the Haar Wavelets are the following:

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

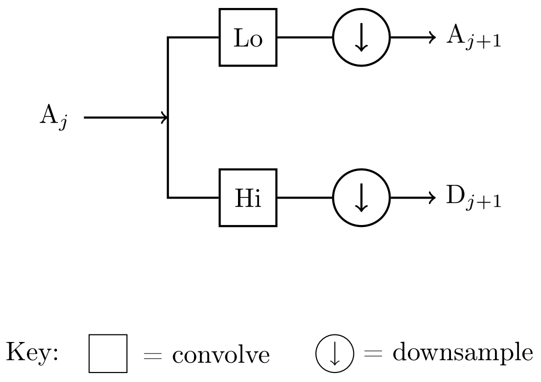


Figure 9.1: The one-dimensional discrete wavelet transform implemented as a filter bank.

As noted earlier, the key mathematical operations of the discrete wavelet transform are convolution and downsampling. Given a filter and a signal, the convolution can be obtained using `scipy.signal.fftconvolve()`.

```
>>> from scipy.signal import fftconvolve
>>> # Initialize a filter.
>>> L = np.ones(2)/np.sqrt(2)
>>> # Initialize a signal X.
>>> X = np.sin(np.linspace(0, 2*np.pi, 16))
>>> # Convolve X with L.
>>> fftconvolve(X, L)
[ -1.84945741e-16   2.87606238e-01   8.13088984e-01   1.19798126e+00
   1.37573169e+00   1.31560561e+00   1.02799937e+00   5.62642704e-01
   7.87132986e-16  -5.62642704e-01  -1.02799937e+00  -1.31560561e+00
  -1.37573169e+00  -1.19798126e+00  -8.13088984e-01  -2.87606238e-01
  -1.84945741e-16]
```

The convolution operation alone gives redundant information, so it is downsampled to keep only what is needed. The array will be downsampled by a factor of 2, which means keeping only every other entry:

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

Both the approximation and detail coefficients are computed in this manner. The approximation uses the low-pass filter while the detail uses the high-pass filter. Implementation of a filter bank is found in Algorithm 1.

Algorithm 1 The one-dimensional discrete wavelet transform. X is the signal to be transformed, L is the low-pass filter, H is the high-pass filter and n is the number of filter bank iterations.

```
procedure DWT($X, L, H, n$)
    $A_0 \leftarrow X$                                   ▷ Initialization.
    for $i = 0 \dots n-1$ do
        $D_{i+1} \leftarrow DS(A_i * H)$                 ▷ High-pass filter and downsample.
        $A_{i+1} \leftarrow DS(A_i * L)$                 ▷ Low-pass filter and downsample.
    return $A_n, D_n, D_{n-1}, \dots, D_1$.
```

Problem 1. Write a function that calculates the discrete wavelet transform using Algorithm 1. The function should return a list of one-dimensional NumPy arrays in the following form: $[A_n, D_n, \dots, D_1]$.

Test your function by calculating the Haar wavelet coefficients of a noisy sine signal with $n = 4$:

```
domain = np.linspace(0, 4*np.pi, 1024)
noise =  np.random.randn(1024)*.1
noisysin = np.sin(domain) + noise
coeffs = dwt(noisysin, L, H, 4)
```

Plot the original signal with the approximation and detail coefficients and verify that they match the plots in Figure 9.2.
(Hint: Use array broadcasting).

Note: the plots in your jupyter notebook *do not* have to be labeled exactly like those in 9.2. As long as the signals are clearly visible that is enough.

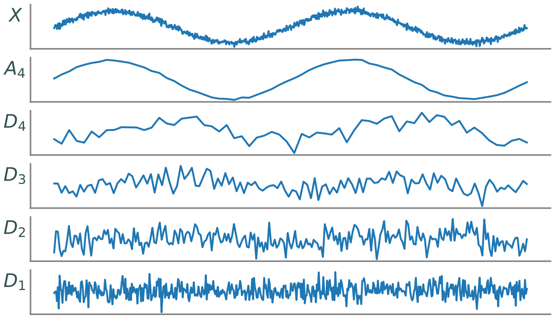


Figure 9.2: A level four wavelet decomposition of a signal. The top panel is the original signal, the next panel down is the approximation, and the remaining panels are the detail coefficients. Notice how the approximation resembles a smoothed version of the original signal, while the details capture the high-frequency oscillations and noise.

Inverse Discrete Wavelet Transform

The process of the discrete wavelet transform is reversible. Using modified filters, a set of detail and approximation coefficients can be manipulated and combined to recreate a signal. The Haar wavelet filters for the inverse transformation are found by reversing the operations for each filter. The Haar inverse filters are given below:

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

The first row refers to the inverse high-pass filter and the second row refers to the inverse low-pass filter.

Suppose the wavelet coefficients A_n and D_n have been computed. A_{n-1} can be recreated by tracing the schematic in Figure 9.1 backwards: A_n and D_n are first upsampled, and then are convolved with the inverse low-pass and high-pass filters, respectively. In the case of the Haar wavelet, *upsampling* involves doubling the length of an array by inserting a 0 at every other position. To complete the operation, the new arrays are convolved and added together to obtain A_{n-1}.

```
>>> # Upsample the coefficient arrays A and D.
>>> up_A = np.zeros(2*A.size)
>>> up_A[::2] = A
>>> up_D = np.zeros(2*D.size)
>>> up_D[::2] = D
>>> # Convolve and add, discarding the last entry.
>>> A = fftconvolve(up_A, L)[:-1] + fftconvolve(up_D, H)[:-1]
```

This process is continued with the newly obtained approximation coefficients and with the next detail coefficients until the original signal is recovered.

Problem 2. Write a function that performs the inverse wavelet transform. The function should accept three things as arguments: a list of arrays (of the same form as the output of Problem 1), a reverse low-pass filter, and a reverse high-pass filter. The function should return a single array, which represents the recovered signal.

Note that the input list of arrays has length $n + 1$ (consisting of A_n together with $D_n, D_{n-1}, \ldots, D_1$). Your code should run once per D_i matrix so it should execute a total of n times.

To test your function, first perform the inverse transform on the noisy sine wave that you created in the first problem. Then, compare the original signal with the signal recovered by your inverse wavelet transform function using `np.allclose()`.

Achtung!

Although Algorithm 1 and the preceding discussion apply in the general case, the code implementations apply only to the Haar wavelet. Because of the nature of the discrete convolution, when convolving with longer filters, the signal to be transformed needs to undergo a different type of lengthening in order to avoid information loss during the convolution. As such, the functions written in Problems 1 and 2 will only work correctly with the Haar filters and would require modifications to be compatible with more wavelets.

The Two-dimensional Wavelet Transform

The generalization of the wavelet transform to two dimensions is similar to one dimensional transforms. Again, the two primary operations used are convolution and downsampling. The main difference in the two-dimensional case is the number of convolutions and downsamples per iteration. First, the convolution and downsampling are performed along the rows of an array. This results in two new arrays, as in the one dimensional case. Then, convolution and downsampling are performed along the columns of the two new arrays. This results in four final arrays that make up the new approximation and detail coefficients. See Figure 9.3.

When implemented as an iterative filter bank, each pass through the filter bank yields one set of approximation coefficients plus three sets of detail coefficients. More specifically, if the two-dimensional array X is the input to the filter bank, the arrays $Approx$, H, V, and D are obtained. $Approx$ is a smoothed approximation of X (similar to A_n in the one-dimensional case), and the other three arrays contain detail coefficients that capture high-frequency oscillations in horizontal (H), vertical (V), and diagonal (D) directions. The arrays A, H, V, and D are known as *subbands*. Any or all of the subbands can be fed into a filter bank to further decompose the signal into additional subbands. This decomposition can be represented by a partition of a rectangle, called a *subband pattern*. The subband pattern for one pass of the filter bank is shown in Figure 9.4, with an example of an image decomposition given in Figure 9.5.

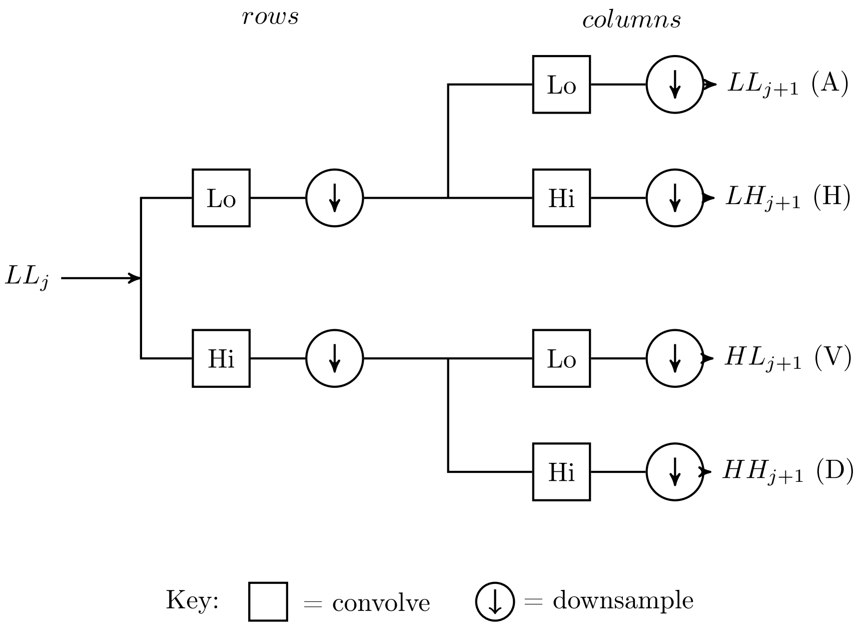


Figure 9.3: The two-dimensional discrete wavelet transform implemented as a filter bank.

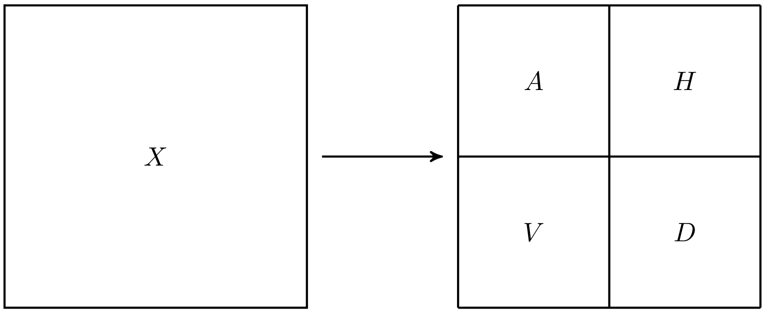


Figure 9.4: The subband pattern for one step in the 2-dimensional wavelet transform.

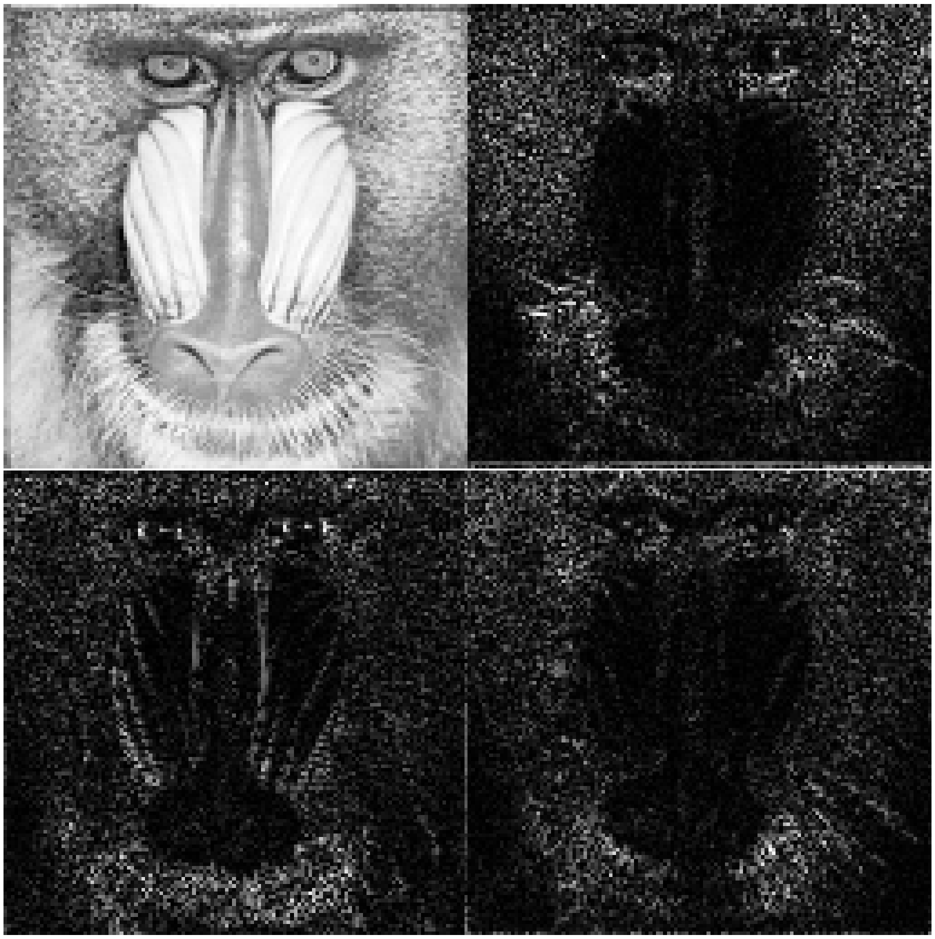


Figure 9.5: Subbands for the mandrill image after one pass through the filter bank. Note how the upper left subband (A) is an approximation of the original Mandrill image, while the other three subbands highlight the stark vertical, horizontal, and diagonal changes in the image.
Original image source: `http://sipi.usc.edu/database/`.

The wavelet coefficients obtained from a two-dimensional wavelet transform are used to analyze and manipulate images at differing levels of resolution. Images are often sparsely represented by wavelets; that is, most of the image information is captured by a small subset of the wavelet coefficients. This is a key fact for wavelet-based image compression and will be discussed in further detail later in the lab.

The PyWavelets Module

PyWavelets is a Python package designed for wavelet analysis. Although it has many other uses, in this lab it will primarily be used for image manipulation. PyWavelets can be installed using the following command:

```
$ pip install PyWavelets
```

PyWavelets provides a simple way to calculate the subbands resulting from one pass through the filter bank. The following code demonstrates how to find the approximation and detail subbands of an image.

```
>>> from imageio.v3 import imread
>>> import pywt                                 # The PyWavelets package.
# The True parameter produces a grayscale image.
>>> mandrill = imread('mandrill1.png', cmap=True)
# Use the Daubechies 4 wavelet with periodic extension.
>>> lw = pywt.dwt2(mandrill, 'db4', mode='per')
```

The function `pywt.dwt2()` calculates the subbands resulting from one pass through the filter bank. The second positional argument specifies the type of wavelet to be used in the transform. The `mode` keyword argument sets the extension mode, which determines the type of padding used in the convolution operation. For the problems in this lab, always use `mode='per'`, which is the periodic extension. The function `dwt2()` returns a tuple. The first entry of the list is the A, or approximation, subband. The second entry of the list is a separate tuple containing the remaining subbands, H, V, and D (in that order).

PyWavelets supports a number of different wavelets which are divided into different classes known as families. The supported families and their wavelet instances can be listed by executing the following code:

```
>>> # List the available wavelet families.
>>> print(pywt.families())
['haar', 'db', 'sym', 'coif', 'bior', 'rbio', 'dmey', 'gaus', 'mexh', 'morl', '↩
    cgau', 'shan', 'fbsp', 'cmor']
>>> # List the available wavelets in a given family.
>>> print(pywt.wavelist('coif'))
['coif1', 'coif2', 'coif3', 'coif4', 'coif5', 'coif6', 'coif7', 'coif8', 'coif9↩
    ', 'coif10', 'coif11', 'coif12', 'coif13', 'coif14', 'coif15', 'coif16', '↩
    coif17']
```

It's important to note that the names from the wavelist are what we use when we call `dwt2`. Sometimes the name of a family will also exist as a wavelet transform in the wave list, but not always. Different wavelets have different properties; the most suitable wavelet is dependent on the specific application. For example, the morlet wavelet is closely related to human hearing and vision. Note that not all of these families work with the function `pywt.dwt2()`, because they are continuous wavelets. Choosing which wavelet is used is partially based on the properties of a wavelet, but since many wavelets share desirable properties, the best wavelet for a particular application is often not known without some type of testing.

Note

The numerical value in a wavelets name refers to the filter length. This value is multiplied by the standard filter length of the given wavelet, resulting in the new filter length. For example, `coif1` has filter length 6 and `coif2` has filter length 12.

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

1. Save a picture of a raccoon with the following code

```
>>> from scipy.misc import face
>>> raccoon = face(True)
```

2. Plot the subbands of raccoon as described above (using the Daubechies 4 wavelet with periodic extension). Compare this with the subbands of the mandrill image shown in Figure 9.5.

3. Compare the subband patterns of the haar, symlet, and coiflet wavelets of the raccoon picture by plotting the subbands after one pass through the filter bank. The haar subband should have more detail than the symlet subband, and the symlet subband should have more detail than the coiflet wavelet.

A few Hints: for plotting, find a function that will plot an image given an array. Also, when plotting the transformations that represent the horizontal, vertical, or diagonal components of the image take `np.abs()` of the array when you plot it. This will radicalize the array and make its detections easier to view. Finally, you will run into an error if you try to use just 'sym' as an argument. Consider why this could be and how you might find the proper argument to use.

The function `pywt.wavedec2()` is similar to `pywt.dwt2()`, but it also includes a keyword argument, `level`, which specifies the number of times to pass an image through the filter bank. It will return a list of subbands, the first of which is the final approximation subband, while the remaining elements are tuples which contain sets of detail subbands (H, V, and D). For example, if I were to call pywt.wavedec2 with level=4, the output would be of the form [Approx, (H4,V4,D4),(H3,V3,D3),(H2,V2,D2),(H1,V1,D1)].

If `level` is not specified, the number of passes through the filter bank will be the maximum level where the decomposition is still useful. The function `pywt.waverec2()` accepts a list of subband patterns (like the output of `pywt.wavedec2()` or `pywt.dwt2()`), a name string denoting the wavelet, and a keyword argument `mode` for the extension mode. It returns a reconstructed image using the reverse filter bank. When using this function, be sure that the wavelet and mode match the deconstruction parameters. PyWavelets has many other useful functions including `dwt()`, `idwt()` and `idwt2()` which can be explored further in the documentation for PyWavelets, `https://pywavelets.readthedocs.io/en/latest/index.html`.

Applications

Noise Reduction

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

Gaussian white noise causes every pixel in an image to be perturbed by a small amount. Many types of noise, including Gaussian white noise, are very high-frequency. Since many images are relatively sparse in high-frequency domains, noise in an image can be safely removed from the high frequency subbands while minimally distorting the true image. A basic, but effective, approach to reducing Gaussian white noise in an image is thresholding. Thresholding can be done in two ways, referred to as hard and soft thresholding.

Given a positive threshold value τ, hard thresholding sets every detail 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 that accept a list of wavelet coefficients in the usual form, as well as a threshold value. Each function returns the thresholded wavelet coefficients (also in the usual form). The first function should implement hard thresholding and the second should implement soft thresholding. While writing these two functions, remember that only the detail coefficients (meaning the elements of H,V, or D arrays) are thresholded, so the first entry of the input coefficient list (The A matrix) should remain unchanged.

To test your functions, perform hard and soft thresholding on `noisy_darkhair.png` and plot the resulting images together. When testing your function, use the Daubechies 4 wavelet and four sets of detail coefficients (`level=4` when using `wavedec2()`). For soft thresholding use $\tau = 20$, and for hard thresholding use $\tau = 40$.

Some notes:

1. Masking will be helpful however it's important to consider the order in which you change values since adjusting certain values by τ initially can skew which conditions that value satisfies. Use your masks in a safe order.

2. In previous problems there was only one tuple of detail coefficients now there may be more. Make sure your code is robust enough to handle any number of tuples of detail coefficients.

3. Remember to recompose your image before attempting to plot the images. Details on how to use `waverec2()` are found in the reading above.

Image Compression

Transform methods based on Fourier and wavelet analysis play an important role in image compression; for example, the popular JPEG image compression standard is based on the discrete cosine transform. The JPEG2000 compression standard and the FBI Fingerprint Image database, along with other systems, take the wavelet approach.

The general framework for compression is as follows. First, the image to be compressed undergoes some form of preprocessing, depending on the particular application. Next, the discrete wavelet transform is used to calculate the wavelet coefficients, and these are then *quantized*, i.e. mapped to a set of discrete values (for example, rounded to the nearest integer). The quantized coefficients are then passed through an entropy encoder (such as Huffman Encoding), which reduces the number of bits required to store the coefficients. What remains is a compact stream of bits that can be saved or transmitted much more efficiently than the original image. The steps above are nearly all invertible (the only exception being quantization), allowing the original image to be almost perfectly reconstructed from the compressed bitstream. See Figure 9.6.

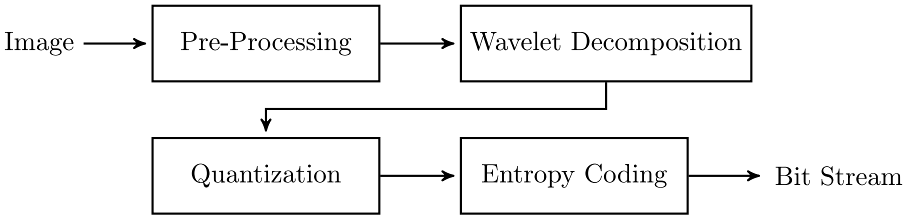
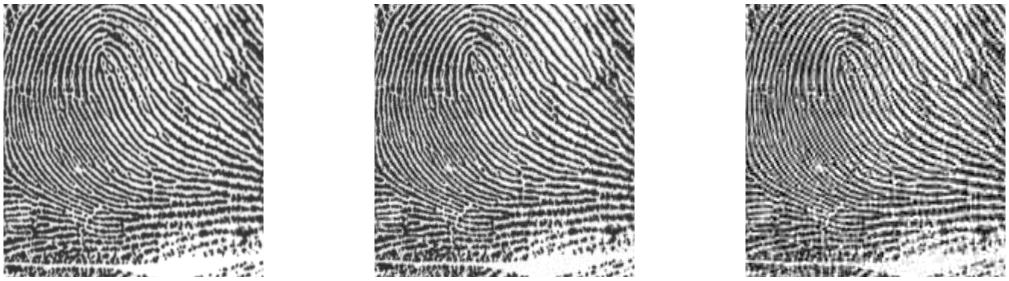


Figure 9.6: Wavelet Image Compression Schematic

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 9.7. This section of the lab steps through a simplified version of this algorithm by writing a Python class that performs both the compression and decompression. Differences between this simplified algorithm and the complete algorithm are found in the Additional Material section at the end of this lab. Most of the methods of the class have already been implemented. The following problems will detail the methods you will need to implement yourself.



(a) Uncompressed

(b) 12:1 compressed

(c) 26:1 compressed

Figure 9.7: Fingerprint scan at different levels of compression. Original image source: `http://www.nist.gov/itl/iad/ig/wsq.cfm`.

WSQ: Preprocessing

Preprocessing in this algorithm ensures that roughly half of the new pixel values are negative, while the other half are positive, and all fall in the range [−128, 128]. The input to the algorithm is a matrix of nonnegative 8-bit integer values giving the grayscale pixel values for the fingerprint image. The image is processed by the following formula:

$$M' = \frac{M - m}{s},$$

where M is the original image matrix, M' is the processed image, m is the mean pixel value, and $s = \max\{\max(M) - m, m - \min(M)\}/128$ (here $\max(M)$ and $\min(M)$ refer to the maximum and minimum pixel values in the matrix). We've provided the code for this part of the algorithm.

WSQ: Calculating the Wavelet Coefficients

The next step in the compression algorithm is decomposing the image into subbands of wavelet coefficients. In this implementation of the WSQ algorithm, the image is decomposed into five sets of detail coefficients (`level=5`) and one approximation subband, as shown in Figure 9.8. Each of these subbands should be placed into a list in the same ordering as in Figure 9.8 (another way to consider this ordering is the approximation subband followed by each level of detail coefficients $[A, H_5, V_5, D_5, H_4, V_4, \ldots, D_1]$).

Problem 5. Implement the class method `decompose()`. This function should accept an image to decompose and should return a list of ordered subbands. Use the function `pywt.wavedec2()` with the `'coif1'` wavelet to obtain the subbands. These subbands should then be ordered in a single list as described above.

Implement the inverse of the decomposition by writing the class method `recreate()`. This function should accept a list of 16 subbands (ordered like the output of `decompose()`) and should return a reconstructed image. Use `pywt.waverec2()` to reconstruct an image from the subbands. Note that you will need to adjust the accepted list in order to adhere to the required input for `waverec2()`.

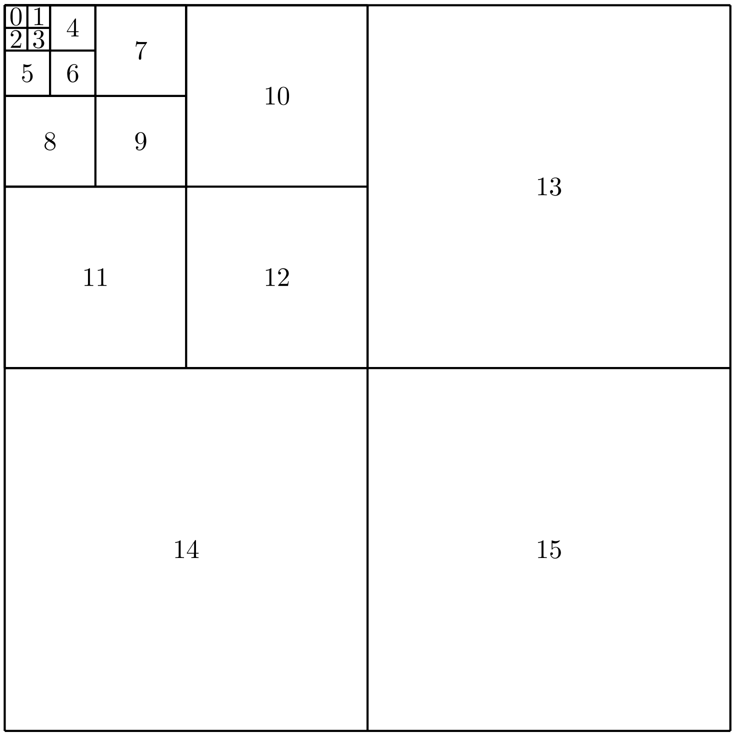


Figure 9.8: Subband Pattern for simplified WSQ algorithm.

WSQ: Quantization

Quantization is the process of mapping each wavelet coefficient to an integer value and is the main source of compression in the algorithm. By mapping the wavelet coefficients to a relatively small set of integer values, the complexity of the data is reduced, which allows for efficient encoding of the information in a bit string. Further, a large portion of the wavelet coefficients will be mapped to 0 and discarded completely. The fact that fingerprint images tend to be very nearly sparse in the wavelet domain means that little information is lost during quantization. Care must be taken, however, to perform this quantization in a manner that achieves good compression without discarding so much information that the image cannot be accurately reconstructed.

Given a wavelet coefficient a in subband k, the corresponding quantized coefficient p is given by

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

where Z_k and Q_k are dependent on the subband. They determine how much compression is achieved. If $Q_k = 0$, all coefficients are mapped to 0.

An array of detail coefficients (such as H) can be quantized by running each value of the array through the piecewise function.

Selecting appropriate values for Z_k and Q_k is a tricky problem in itself, and relies on heuristics based on the statistical properties of the wavelet coefficients. The methods that calculate these values have already been initialized.

Quantization is not a perfectly invertible process. Once the wavelet coefficients have been quantized, some information is permanently lost. However, wavelet coefficients $\hat{a}_k$ in subband k can be roughly reconstructed from the quantized coefficients p using

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

where C is a new dequanitization parameter. This process is called *dequantization*. Again, if $Q_k = 0$, $\hat{a}_k = 0$ should be returned.

Problem 6. Implement the quantization step by writing the `quantize()` method of your class. This method should accept a NumPy array of coefficients and the quantization parameters Q_k and Z_k. The function should return a NumPy array of the quantized coefficients.

Also implement the `dequantize()` method of your class using the formula given above. This function should accept the same parameters as `quantize()` as well as a parameter C which defaults to .44. The function should return a NumPy array of dequantized coefficients.

(Hint: Masking and array slicing will help keep your code short and fast when implementing both of these methods. Remember the case for $Q_k = 0$. Test your functions by comparing the output of your functions to a hand calculation on a small matrix.)

WSQ: The Rest

The remainder of the compression and decompression methods have already been implemented in the WSQ class. The following discussion explains the basics of what happens in those methods. Once all of the subbands have been quantized, they are divided into three groups. The first group contains the smallest ten subbands (positions zero through nine), while the next two groups contain the three subbands of next largest size (positions ten through twelve and thirteen through fifteen, respectively). All of the subbands of each group are then flattened and concatenated with the other subbands in the group. These three arrays of values are then mapped to Huffman indices. Since the wavelet coefficients for fingerprint images are typically very sparse, special indices are assigned to lists of sequential zeros of varying lengths. This allows large chunks of information to be stored as a single index, greatly aiding in compression. The Huffman indices are then assigned a bit string representation through a Huffman map.

Python does not natively include all of the tools necessary to work with bit strings, but the Python package bitstring does have these capabilities. Download bitstring using the following command:

```
$ pip install bitstring
```

You will not use bitstring functions in this lab. but the code provided in the lab does call functions from the bitstring module. So you'll need to import the package with the following line of code:

```
>>> import bitstring as bs
```

WSQ: Calculating the Compression Ratio

The methods of compression and decompression are now fully implemented. The final task is to verify how much compression has taken place. The compression ratio is the ratio of the number of bits in the original image to the number of bits in the encoding. Assuming that each pixel of the input image is an 8-bit integer, the number of bits in the original image is just eight times the number of pixels (the number of pixels in the original source image is stored in the class attribute `_pixels`). The number of bits in the encoding can be calculated by adding up the lengths of each of the three bit strings stored in the class attribute `_bitstrings`.

Problem 7. Implement the method `get_ratio()` by calculating the ratio of compression. The function should not accept any parameters and should return the compression ratio.

Your compression algorithm is now complete! Test your class with the following code. The compression ratio should be approximately 18.

```python
# Try out different values of r between .1 to .9.
r = .5
finger = imread('uncompressed_finger.png', cmap=True)
wsq = WSQ()
wsq.compress(finger, r)
print(wsq.get_ratio())
new_finger = wsq.decompress()
plt.subplot(211)
plt.imshow(finger, cmap=plt.cm.Greys_r)
plt.subplot(212)
plt.imshow(np.abs(new_finger), cmap=plt.cm.Greys_r)
plt.show()
```

Additional Material

Haar Wavelet Transform

The Haar Wavelet Transform is a general matrix transform used to convolve Haar Wavelets. It is found by combining the convolution matrices for a lowpass and highpass filter such that one is directly on top of the other. The lowpass filter is taking the average of every two elements in an array and the highpass filter is taking the difference of every two elements in an array. Redundant information given in the new matrix is then removed via downsampling. However, in order for the transform matrix to have the property $A^{\mathsf{T}} = A^{-1}$, the columns of the matrix must be normalized. Thus, each column is normalized (and subsequently the filters) and the resulting matrix is the Haar Wavelet Transform.

For more on the Haar Wavelet Transform, see *Discrete Wavelet Transformations: An Elementary Approach with Applications* by Patrick J. Van Fleet.

WSQ Algorithm

The official standard for the WSQ algorithm is slightly different from the version implemented in this lab. One of the largest differences is the subband pattern that is used in the official algorithm; this pattern is demonstrated in Figure 9.9. The pattern used may seem complicated and somewhat arbitrary, but it is used because of the relatively good empirical results when used in compression. This pattern can be obtained by performing a single pass of the 2-dimensional filter bank on the image then passing each of the resulting subbands through the filter bank resulting in 16 total subbands. This same process is then repeated with the A, H and V subbands of the original approximation subband creating 46 additional subbands. Finally, the subband corresponding to the top left of Figure 9.9 should be passed through the 2-dimensional filter bank a single time.

As in the implementation given above, the subbands of the official algorithm are divided into three groups. The subbands 0 through 18 are grouped together, as are 19 through 51 and 52 through 63. The official algorithm also uses a wavelet specialized for image compression that is not included in the PyWavelets distribution. There are also some slight modifications made to the implementation of the discrete wavelet transform that do not drastically affect performance.

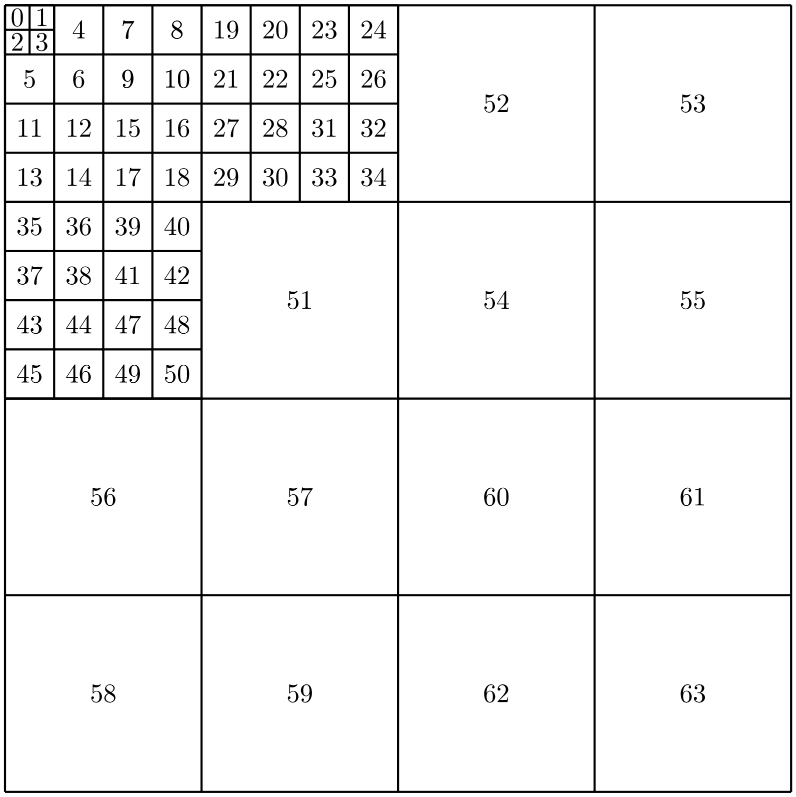


Figure 9.9: True subband pattern for WSQ algorithm.

10 Polynomial Interpolation

Lab Objective: *Learn and compare three methods of polynomial interpolation: standard Lagrange interpolation, Barycentric Lagrange interpolation and Chebyshev interpolation. Explore Runge's phenomenon and how the choice of interpolating points affect the results. Use polynomial interpolation to study air pollution by approximating graphs of particulates in air.*

Polynomial Interpolation

Polynomial interpolation is the method of finding a polynomial that matches a function at specific points in its range. More precisely, if $f(x)$ is a function on the interval $[a, b]$ and $p(x)$ is a polynomial then $p(x)$ interpolates the function $f(x)$ at the points $x_0, x_1, \ldots, x_n$ if $p(x_j) = f(x_j)$ for all $j = 0, 1, \ldots, n$. In this lab most of the discussion is focused on using interpolation as a means of approximating functions or data, however, polynomial interpolation is useful in a much wider array of applications.

Given a function $f(x)$ and a set of unique points $\{x_i\}_{i=0}^n$, it can be shown that there exists a unique interpolating polynomial $p(x)$. That is, there is one and only one polynomial of degree n that interpolates $f(x)$ through those points. This uniqueness property is why, for the remainder of this lab, an interpolating polynomial is referred to as *the* interpolating polynomial. One approach to finding the unique interpolating polynomial of degree n is Lagrange interpolation.

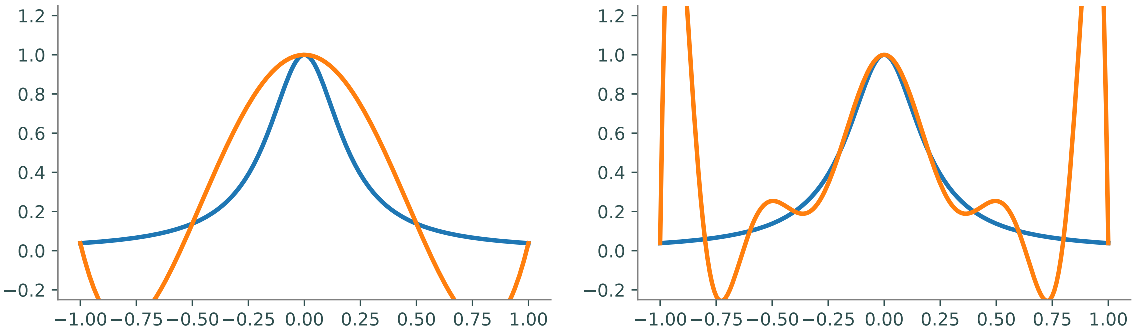
Lagrange interpolation

Given a set $\{x_i\}_{i=1}^n$ of n points to interpolate, a family of n basis functions with the following property is constructed:

$$L_j(x_i) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}.$$

The Lagrange form of this family of basis functions is

$$L_j(x) = \frac{\displaystyle\prod_{k=1, k\neq j}^{n} (x - x_k)}{\displaystyle\prod_{k=1, k\neq j}^{n} (x_j - x_k)} \tag{10.1}$$



(a) Interpolation using 5 equally spaced points.

(b) Interpolation using 11 equally spaced points.

Figure 10.1: Interpolations of Runge's function $f(x) = \frac{1}{1+25x^2}$ with equally spaced interpolating points.

Each of these Lagrange basis functions is a polynomial of degree $n-1$ and has the necessary properties as given above.

Problem 1. Define a function `lagrange()` that will be used to construct and evaluate an interpolating polynomial on a domain of x values. The function should accept two NumPy arrays of length n which contain the x and y values of the interpolating points as well as a NumPy array of values of length m at which the interpolating polynomial will be evaluated.

Within `lagrange()`, write a subroutine that will evaluate each of the n Lagrange basis functions at every point in the domain. It may be helpful to follow these steps:

1. Compute the denominator of each L_j (as in Equation 10.1) .
2. Using the previous step, evaluate L_j at all points in the computational domain (this will give you m values for each L_j.)
3. Combine the results into an $n \times m$ NumPy array, consisting of each of the n L_j evaluated at each of the m points in the domain.

You may find the functions `np.product()` and `np.delete()` to be useful while writing this method.

Lagrange interpolation is completed by combining the Lagrange basis functions with the y-values of the function to be interpolated $y_i = f(x_i)$ in the following manner:

$$p(x) = \sum_{j=1}^{n} y_j L_j(x) \tag{10.2}$$

This will create the unique interpolating polynomial.

Since polynomials are typically represented in their expanded form with coefficients on each of the terms, it may seem like the best option when working with polynomials would be to use Sympy, or NumPy's `poly1d` class to compute the coefficients of the interpolating polynomial individually. This is rarely the best approach, however, since expanding out the large polynomials that are required can quickly lead to instability (especially when using large numbers of interpolating points). Instead, it is usually best just to leave the polynomials in unexpanded form (which is still a polynomial, just not a pretty-looking one), and compute values of the polynomial directly from this unexpanded form.

```
# Evaluate the polynomial (x-2)(x+1) at 10 points without expanding the ↩
    expression.
>>> pts = np.arange(10)
>>> (pts - 2) * (pts + 1)
array([ 2,  0,  0,  2,  6, 12, 20, 30, 42, 56])
```

In the given example, there would have been no instability if the expression had actually been expanded but in the case of a large polynomial, stability issues can dominate the computation. Although the coefficients of the interpolating polynomials will not be explicitly computed in this lab, polynomials are still being used, albeit in a different form.

Problem 2. Complete the implementation of `lagrange()`.

Evaluate the interpolating polynomial at each point in the domain by combining the y values of the interpolation points and the evaluated Lagrange basis functions from Problem 1 as in Equation 10.2. Return the final array of length m that consists of the interpolating polynomial evaluated at each point in the domain.

You can test your function by plotting Runge's function $f(x) = \frac{1}{1+25x^2}$ and your interpolating polynomial on the same plot for different values of n equally spaced interpolating values then comparing your plot to the plots given in Figure 10.1.

The Lagrange form of polynomial interpolation is useful in some theoretical contexts and is easier to understand than other methods, however, it has some serious drawbacks that prevent it from being a useful method of interpolation. First, Lagrange interpolation is $O(n^2)$ where other interpolation methods are $O(n^2)$ (or faster) at startup but only $O(n)$ at run-time, Second, Lagrange interpolation is an unstable algorithm which causes it to return innacurate answers when larger numbers of interpolating points are used. Thus, while useful in some situations, Lagrange interpolation is not desirable in most instances.

Barycentric Lagrange interpolation

Barycentric Lagrange interpolation is simple variant of Lagrange interpolation that performs much better than plain Lagrange interpolation. It is essentially just a rearrangement of the order of operations in Lagrange multiplication which results in vastly improved perfomance, both in speed and stability.

Barycentric Lagrange interpolation relies on the observation that each basis function L_j can be rewritten as

$$L_j(x) = \frac{w(x)}{(x - x_j)} w_j$$

where

$$w(x) = \prod_{j=1}^{n} (x - x_j)$$

and

$$w_j = \frac{1}{\prod_{k=1,k\neq j}^{n} (x_j - x_k)}.$$

The w_j's are known as the *barycentric weights*.

Using the previous equations, the interpolating polynomial can be rewritten

$$p(x) = w(x) \sum_{j=1}^{n} \frac{w_j y_j}{x - x_j}$$

which is the *first barycentric form*. The computation of $w(x)$ can be avoided by first noting that

$$1 = w(x) \sum_{j=1}^{n} \frac{w_j}{x - x_j}$$

which allows the interpolating polynomial to be rewriten as

$$p(x) = \frac{\displaystyle\sum_{j=1}^{n} \frac{w_j y_j}{x - x_j}}{\displaystyle\sum_{j=1}^{n} \frac{w_j}{x - x_j}}$$

This form of the Lagrange interpolant is known as the *second barycentric form* which is the form used in Barycentric Lagrange interpolation. So far, the changes made to Lagrange interpolation have resulted in an algorithm that is $O(n)$ once the barycentric weights (w_j) are known. The following adjustments will improve the algorithm so that it is numerically stable and later discussions will allow for the quick addition of new interpolating points after startup.

The second barycentric form makes it clear that any factors that are common to the w_k can be ignored (since they will show up in both the numerator and denominator). This allows for an important improvement to the formula that will prevent overflow error in the arithmetic. When computing the barycentric weights, each element of the product $\prod_{k=1,k\neq j}^{n} (x_j - x_k)$ should be multiplied by C^{-1}, where $4C$ is the width of the interval being interpolated (C is known as the *capacity* of the interval). In effect, this scales each barycentric weight by C^{1-n} which helps to prevent overflow during computation. Thus, the new barycentric weights are given by

$$w_j = \frac{1}{\prod_{k=1,k\neq j}^{n} [(x_j - x_k)/C]}. \tag{10.3}$$

Once again, this change is possible since the extra factor C^{1-n} is cancelled out in the final product. This process is summed up in the following code:

```
# Given a NumPy array xint of interpolating x-values, calculate the weights.
>>> n = len(xint)                    # Number of interpolating points.
>>> w = np.ones(n)                   # Array for storing barycentric weights.
# Calculate the capacity of the interval.
>>> C = (np.max(xint) - np.min(xint)) / 4
>>> shuffle = np.random.permutation(n-1)
>>> for j in range(n):
>>>     temp = (xint[j] - np.delete(xint, j)) / C
>>>     temp = temp[shuffle]        # Randomize order of product.
>>>     w[j] /= np.product(temp)
```

The order of `temp` was randomized so that the arithmetic does not overflow due to poor ordering (if standard ordering is used, overflow errors can be encountered since all of the points of similar magnitude are multiplied together at once). When these two fixes are combined, the Barycentric Algorithm becomes numerically stable.

Problem 3. Create a class that performs Barycentric Lagrange interpolation. The constructor of your class should accept two NumPy arrays which contain the x and y values of the interpolation points. Store these arrays as attributes. In the constructor, compute the corresponding barycentric weights using Equation 10.3 and store the resulting array as a class attribute. Be sure that the relative ordering of the arrays remains unchanged.

Implement the `__call__()` method so that it accepts a NumPy array of values at which to evaluate the interpolating polynomial and returns an array of the evaluated points. Your class can be tested in the same way as the Lagrange function written in Problem 2

Achtung!

As currently explained and implemented, the Barycentric class from Problem 3 will fail when a point to be evaluated exactly matches one of the x-values of the interpolating points. This happens because a divide by zero error is encountered in the final step of the algorithm. The fix for this, although not required here, is quite easy: keep track of any problem points and replace the final computed value with the corresponding y-value (since this is a point that is exactly interpolated). If you do not implement this fix, just be sure not to pass in any points that exactly match your interpolating values.

Another advantage of the barycentric method is that it allows for the addition of new interpolating points in $O(n)$ time. Given a set of existing barycentric weights $\{w_j\}_{j=1}^{n}$ and a new interpolating point x_i, the new barycentric weight is given by

$$w_i = \frac{1}{\prod_{k=1}^{n}(x_i - x_k)}.$$

In addition to calculating the new barycentric weight, all existing weights should be updated as follows $w_j = \frac{w_j}{x_j - x_i}$.

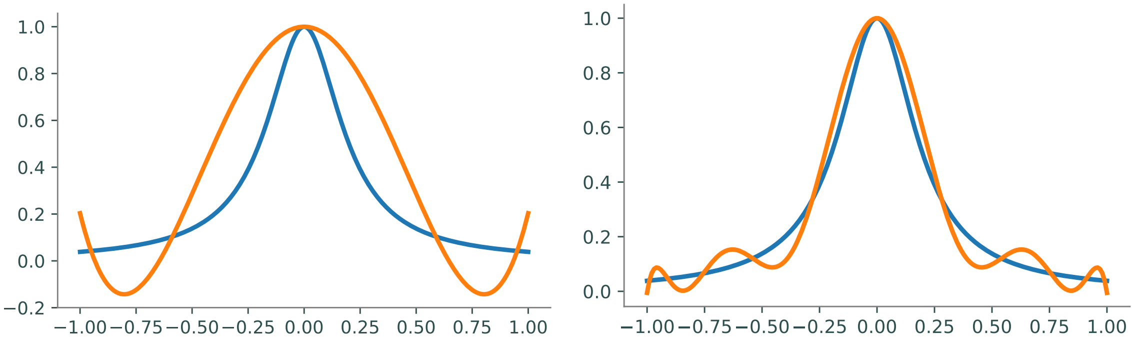
Problem 4. Include a method in the class written in Problem 3 that allows for the addition of new interpolating points by updating the barycentric weights. Your function should accept two NumPy arrays which contain the x and y values of the new interpolation points. Update and store the old weights then extend the class attribute arrays that store the weights, and the x and y values of the interpolation points with the new data. When updating all class attributes, make sure to maintain the same relative order.

The implementation outlined here calls for the y-values of the interpolating points to be known during startup, however, these values are not needed until run-time This allows the y-values to be changed without having to recompute the barycentric weights. This is an additional advantage of Barycentric Lagrange interpolation.

Scipy's Barycentric Lagrange class

Scipy includes a Barycentric interpolator class. This class includes the same functionality as the class described in Problems 3 and 4 in addition to the ability to update the y-values of the interpolation points. The following code will produce a figure similar to Figure 10.1b.

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

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

>>> poly = BarycentricInterpolator(pts[:-1])
>>> poly.add_xi([pts[-1]])        # Oops, forgot one of the points.
>>> poly.set_yi(f(pts))           # Set the y values.

>>> plt.plot(domain, f(domain))
>>> plt.plot(domain, poly(domain))
```



(a) Polynomial using 5 Chebyshev roots.

(b) Polynomial using 11 Chebyshev roots.

Figure 10.2: Example of overcoming Runge's phenomenon by using Chebyshev nodes for interpolating values. Plots made using Runge's function $f(x) = \frac{1}{1+25x^2}$. Compare with Figure 10.1

Chebyshev Interpolation

Chebyshev Nodes

As has been mentioned previously, the Barycentric version of Lagrange interpolation is a stable process that does not accumulate large errors, even with extreme inputs. However, polynomial interpolation itself is, in general, an ill-conditioned problem. Thus, even small changes in the interpolating values can give drastically different interpolating polynomials. In fact, poorly chosen interpolating points can result in a very bad approximation of a function. As more points are added, this approximation can worsen. This increase in error is called *Runge's phenomenon.*

The set of equally spaced points is an example of a set of points that may seem like a reasonable choice for interpolation but in reality produce very poor results. Figure 10.1 gives an example of this using Runge's function. As the number of interpolating points increases, the quality of the approximation deteriorates, especially near the endpoints.

Although polynomial interpolation has a great deal of potential error, a good set of interpolating points can result in fast convergence to the original function as the number of interpolating points is increased. One such set of points is the Chebyshev extremal points which are related to the Chebyshev polynomials (to be discussed shortly). The $n + 1$ Chebyshev extremal points on the interval $[a, b]$ are given by the formula $y_j = \frac{1}{2}(a+b+(b-a)\cos(\frac{j\pi}{n}))$ for $j = 0, 1, \ldots, n$. These points are shown in Figure 10.3. One important feature of these points is that they are clustered near the endpoints of the interval, this is key to preventing Runge's phenomenon.

Problem 5. Write a function that defines a domain $\mathbf{x}$ of 400 equally spaced points on the interval $[-1, 1]$. For $n = 2^2, 2^3, \ldots, 2^8$, repeat the following experiment.

1. Interpolate Runge's function $f(x) = 1/(1+25x^2)$ with n equally spaced points over $[-1, 1]$ using SciPy's `BarycentricInterpolator` class, resulting in an approximating function $\tilde{f}$. Compute the absolute error $\|f(\mathbf{x}) - \tilde{f}(\mathbf{x})\|_\infty$ of the approximation using `la.norm()` with `ord=np.inf`.
2. Interpolate Runge's function with $n + 1$ Chebyshev extremal points, also via SciPy, and compute the absolute error.

Plot the errors of each method against the number of interpolating points n in a log-log plot.

To verify that your figure make sense, try plotting the interpolating polynomials with the original function for a few of the larger values of n.

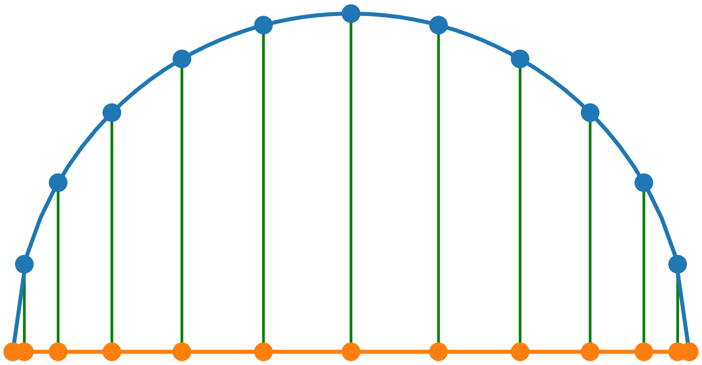


Figure 10.3: The Chebyshev extremal points. The n points where the Chebyshev polynomial of degree n reaches its local extrema. These points are also the projection onto the x-axis of n equally spaced points around the unit circle.

Chebyshev Polynomials

The Chebyshev roots and Chebyshev extremal points are closely related to a set of polynomials known as the Chebyshev polynomials. The first two Chebyshev polynomials are defined as $T_0(x) = 1$ and $T_1(x) = x$. The remaining polynomials are defined by the recursive algorithm $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. The Chebyshev polynomials form a complete basis for the polynomials in $\mathbb{R}$ which means that for any polynomial $p(x)$, there exists a set of unique coefficients $\{a_k\}_{k=0}^n$ such that

$$p(x) = \sum_{k=0}^{n} a_k T_k.$$

Finding the Chebyshev representation of an interpolating polynomial is a slow process (dominated by matrix multiplication or solving a linear system), but when the interpolating values are the Chebyshev extrema, there exists a fast algorithm for computing the Chebyshev coefficients of the interpolating polynomial. This algorithm is based on the Fast Fourier transform which has temporal complexity $O(n \log n)$. Given the $n + 1$ Chebyshev extremal points $y_j = \cos(\frac{j\pi}{n})$ for $j = 0, 1, \ldots, n$ and a function f, the unique n-degree interpolating polynomial $p(x)$ is given by

$$p(x) = \sum_{k=0}^{n} a_k T_k$$

where

$$a_k = \gamma_k \Re \left[DFT(f(y_0), f(y_1), \ldots, f(y_{2n-1})) \right]_k .$$

Note that although this formulation includes y_j for $j > n$, there are really only $n + 1$ distinct values being used since $y_{n-k} = y_{n+k}$. Also, $\Re$ denotes the real part of the Fourier transform and γ_k is defined as

$$\gamma_k = \begin{cases} 1 & k \in \{0, n\} \\ 2 & \text{otherwise.} \end{cases}$$

Problem 6. Write a function that accepts a function f and an integer n. Compute the $n+1$ Chebyshev coefficients for the degree n interpolating polynomial of f using the Fourier transform (`np.real()` and `np.fft.fft()` will be helpful). When using NumPy's `fft()` function, multiply every entry of the resulting array by the scaling factor $\frac{1}{2n}$ to match the derivation given above.

Validate your function with `np.polynomial.chebyshev.poly2cheb()`. The results should be exact for polynomials.

```
# Define f(x) = -3 + 2x^2 - x^3 + x^4 by its (ascending) coefficients.
>>> f = lambda x: -3 + 2*x**2 - x**3 + x**4
>>> pcoeffs = [-3, 0, 2, -1, 1]
>>> ccoeffs = np.polynomial.chebyshev.poly2cheb(pcoeffs)

# The following callable objects are equivalent to f().
>>> fpoly = np.polynomial.Polynomial(pcoeffs)
>>> fcheb = np.polynomial.Chebyshev(ccoeffs)
```

Lagrange vs. Chebyshev

As was previously stated, Barycentric Lagrange interpolation is $O(n^2)$ at startup and $O(n)$ at runtime while Chebyshev interpolation is $O(n \log n)$. This improved speed is one of the greatest advantages of Chebyshev interpolation. Chebyshev interpolation is also more accurate than Barycentric interpolation, even when using the same points. Despite these significant advantages in accuracy and temporal complexity, Barycentric Lagrange interpolation has one very important advantage over Chebyshev interpolation: Barycentric interpolation can be used on any set of interpolating points while Chebyshev is restricted to the Chebyshev nodes. In general, because of their better accuracy, the Chebyshev nodes are more desirable for interpolation, but there are situations when the Chebyshev nodes are not available or when specific points are needed in an interpolation. In these cases, Chebyshev interpolation is not possible and Barycentric Lagrange interpolation must be used.

Utah Air Quality

The Utah Department of Environmental Quality has air quality stations throughout the state of Utah that measure the concentration of particles found in the air. One particulate of particular interest is $PM_{2.5}$ which is a set of extremely fine particles known to cause tissue damage to the lungs. The file `airdata.npy` has the hourly concentration of $PM_{2.5}$ in micrograms per cubic meter for a particular measuring station in Salt Lake County for the year 2016. The given data presents a fairly smooth function which can be reasonably approximated by an interpolating polynomial. Although Chebyshev interpolation would be preferable (because of its superior speed and accuracy), it is not possible in this case because the data is not continous and the information at the Chebyshev nodes is not known. In order to get the best possible interpolation, it is still preferable to use points close to the Chebyshev extrema with Barycentric interpolation. The following code will take the $n+1$ Chebyshev extrema and find the closest match in the non-continuous data found in the variable `data` then calculate the barycentric weights.

```
>>> fx = lambda a, b, n: .5*(a+b + (b-a) * np.cos(np.arange(n+1) * np.pi / n))
>>> a, b = 0, 366 - 1/24
>>> domain = np.linspace(0, b, 8784)
>>> points = fx(a, b, n)
>>> temp = np.abs(points - domain.reshape(8784, 1))
>>> temp2 = np.argmin(temp, axis=0)

>>> poly = barycentric(domain[temp2], data[temp2])
```

Problem 7. Write a function that interpolates the given data along the whole interval at the closest approximations to the $n+1$ Chebyshev extremal nodes. The function should accept n, perform the Barycentric interpolation then plot the original data and the approximating polynomial on the same domain on two separate subplots. Your interpolating polynomial should give a fairly good approximation starting at around 50 points. Note that beyond about 200 points, the given code will break down since it will attempt to return multiple of the same points causing a divide by 0 error. If you did not perform the fix suggested in the `ACHTUNG` box, make sure not to pass in any points that exactly match the interpolating values.

Additional Material

The *Clenshaw Algorithm* is a fast algorithm commonly used to evaluate a polynomial given its representation in Chebyshev coefficients. This algorithm is based on the recursive relation between Chebyshev polynomials and is the algorithm used by NumPy's `polynomial.chebyshev` module.

Algorithm 1 Accepts an array x of points at which to evaluate the polynomial and an array $a = [a_0, a_1, \ldots, a_{n-1}]$ of Chebyshev coefficients.

```
procedure ClenshawRecursion($(x, a)$)
    $u_{n+1} \leftarrow 0$
    $u_n \leftarrow 0$
    $k \leftarrow n - 1$
    while $k \geq 1$ do
        $u_k \leftarrow 2xu_{k+1} - u_{k+2} + a_k$
        $k \leftarrow k - 1$
    return $a_0 + xu_1 - u_2$
```

11 Gaussian Quadrature

Lab Objective: *Learn the basics of Gaussian quadrature and its application to numerical integration. Build a class to perform numerical integration using Legendre and Chebyshev polynomials. Compare the accuracy and speed of both types of Gaussian quadrature with the built-in Scipy package. Perform multivariate Gaussian quadrature.*

Legendre and Chebyshev Gaussian Quadrature

It can be shown that for any class of orthogonal polynomials $p \in \mathbb{R}[x; 2n+1]$ with corresponding weight function $w(x)$, there exists a set of points $\{x_i\}_{i=0}^n$ and weights $\{w_i\}_{i=0}^n$ such that

$$\int_a^b p(x)w(x)dx = \sum_{i=0}^n p(x_i)w_i.$$

Since this relationship is exact, a good approximation for the integral

$$\int_a^b f(x)w(x)dx$$

can be expected as long as the function $f(x)$ can be reasonably interpolated by a polynomial at the points x_i for $i = 0, 1, \ldots, n$. In fact, it can be shown that if $f(x)$ is $2n+1$ times differentiable, the error of the approximation will decrease as n increases.

Gaussian quadrature can be performed using any basis of orthonormal polynomials, but the most commonly used are the Legendre polynomials and the Chebyshev polynomials. Their weight functions are $w_l(x) = 1$ and $w_c(x) = \frac{1}{\sqrt{1-x^2}}$, respectively, both defined on the open interval $(-1, 1)$.

Problem 1. Define a class for performing Gaussian quadrature. The constructor should accept an integer n denoting the number of points and weights to use (this will be explained later) and a label indicating which class of polynomials to use. If the label is not either `"legendre"` or `"chebyshev"`, raise a `ValueError`; otherwise, store it as an attribute.

The weight function $w(x)$ will show up later in the denominator of certain computations. Define the reciprocal function $w(x)^{-1} = 1/w(x)$ as a `lambda` function and save it as an attribute.

Calculating Points and Weights

All sets of orthogonal polynomials $\{u_k\}_{k=0}^n$ satisfy the three-term recurrence relation

$$u_0 = 1, \qquad u_1 = x - \alpha_1, \qquad u_{k+1} = (x - \alpha_k)u_k - \beta_k u_{k-1}$$

for some coefficients $\{\alpha_k\}_{k=1}^n$ and $\{\beta_k\}_{k=1}^n$. For the Legendre polynomials, they are given by

$$\alpha_k = 0, \qquad \beta_k = \frac{k^2}{4k^2 - 1},$$

and for the Chebyshev polynomials, they are

$$\alpha_k = 0, \qquad \beta_k = \begin{cases} \frac{1}{2} & \text{if } k = 1 \\ \frac{1}{4} & \text{otherwise.} \end{cases}$$

Given these values, the corresponding *Jacobi matrix* is defined as follows.

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

According to the *Golub-Welsch algorithm*,[1] the n eigenvalues of J are the points x_i to use in Gaussian quadrature, and the corresponding weights are given by $w_i = \mu_w(\mathbb{R})v_{i,0}^2$ where $v_{i,0}$ is the first entry of the ith eigenvector and $\mu_w(\mathbb{R}) = \int_{-\infty}^{\infty} w(x)dx$ is the *measure* of the weight function. Since the weight functions for Legendre and Chebyshev polynomials have compact support on the interval $(-1, 1)$, their measures are given as follows.

$$\mu_{w_l}(\mathbb{R}) = \int_{-\infty}^{\infty} w_l(x)dx = \int_{-1}^{1} 1dx = 2 \qquad \mu_{w_c}(\mathbb{R}) = \int_{-\infty}^{\infty} w_c(x)dx = \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}}dx = \pi$$

Problem 2. Write a method for your class from Problem 1 that accepts an integer n. Construct the $n \times n$ Jacobi matrix J for the polynomial family indicated in the constructor. Use SciPy to compute the eigenvalues and eigenvectors of J, then compute the points $\{x_i\}_{i=1}^n$ and weights $\{w_i\}_{i=1}^n$ for the quadrature. Return both the array of points and the array weights.

Test your method by checking your points and weights against the following values using the Legendre polynomials with $n = 5$.

x_i	$-\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$-\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	0	$\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$
w_i	$\frac{322-13\sqrt{70}}{900}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{128}{225}$	$\frac{322+13\sqrt{70}}{900}$	$\frac{322-13\sqrt{70}}{900}$

[1] See `http://gubner.ece.wisc.edu/gaussquad.pdf` for a complete treatment of the Golub-Welsch algorithm, including the computation of the recurrence relation coefficients for arbitrary orthogonal polynomials.

Finally, modify the constructor of your class so that it calls your new function and stores the resulting points and weights as attributes.
(Note: The order of the points and weights in the table may differ depending on whether you used `scipy.linalg.eig()` or `scipy.linalg.eigh()`. The order doesn't matter but it is important that each point corresponds to the correct weight.)

Unit Test

In the file `test_gaussian_quadrature.py`, write unit tests for the `points_weights()` method in your `GaussianQuadrature` class. Some unit tests have been provided for Problem 1.

Note: The unit tests provided may have different attribute names than those you have written for your class. You may adjust the provided tests to match your code, or change your code to conform with the given tests.

Integrating with Given Weights and Points

Now that the points and weights have been obtained, they can be used to approximate the integrals of different functions. For a given function $f(x)$ with points x_i and weights w_i,

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

There are two problems with the preceding formula. First, the weight function is part of the integral being approximated, and second, the points obtained are only found on the interval $(-1, 1)$ (in the case of the Legendre and Chebyshev polynomials). To solve the first problem, define a new function $g(x) = f(x)/w(x)$ so that

$$\int_{-1}^{1} f(x)dx = \int_{-1}^{1} g(x)w(x)dx \approx \sum_{i=1}^{n} g(x_i)w_i. \tag{11.1}$$

The integral of $f(x)$ on $[-1, 1]$ can thus be approximated with the inner product $\mathbf{w}^\mathsf{T} g(\mathbf{x})$, where $g(\mathbf{x}) = [g(x_1), \ldots, g(x_n)]^\mathsf{T}$ and $\mathbf{w} = [w_1, \ldots, w_n]^\mathsf{T}$.

Problem 3. Write a method for your class that accepts a callable function f. Use (11.1) and the stored points and weights to approximate of the integral of f on the interval $[-1, 1]$.
(Hint: Use $w(x)^{-1}$ from Problem 1 to compute $g(x)$ without division.)

Test your method with examples that are easy to compute by hand and by comparing your results to `scipy.integrate.quad()` (The answer given below is found by using Chebyshev, even though the default mode is Lagrange.)

```
>>> import numpy as np
>>> from scipy.integrate import quad

# Integrate f(x) = 1 / sqrt(1 - x**2) from -1 to 1.
>>> f = lambda x: 1 / np.sqrt(1 - x**2)
>>> quad(f, -1, 1)[0]
3.141592653589591
```

Note

Since the points and weights for Gaussian quadrature do not depend on f, they only need to be computed once and can then be reused to approximate the integral of any function. The class structure in Problems 1–4 takes advantage of this fact, but `scipy.integrate.quad()` does not. If a larger n is needed for higher accuracy, however, the computations must be repeated to get a new set of points and weights.

Shifting the Interval of Integration

Since the weight functions for the Legendre and Chebyshev polynomials have compact support on the interval $(-1, 1)$, all of the quadrature points are found on that interval as well. To integrate a function on an arbitrary interval $[a, b]$ requires a change of variables. Let

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

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

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

so the transformed integral is given by

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

By defining a new function $h(x)$ as

$$h(x) = f\left(\frac{(b-a)}{2}x + \frac{(a+b)}{2}\right),$$

the integral of f can be approximated by integrating h over $[-1, 1]$ with (11.1). This results in the final quadrature formula

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} h(x)dx = \frac{b-a}{2}\int_{-1}^{1} g(x)w(x)dx \approx \frac{b-a}{2}\sum_{i=1}^{n} g(x_i)w_i, \tag{11.2}$$

where now $g(x) = h(x)/w(x)$.

Problem 4. Write a method for your class that accepts a callable function f and bounds of integration a and b. Use (11.2) to approximate the integral of f from a to b.
(Hint: Define $h(x)$ and use your method from Problem 3.)

Problem 5. The *standard normal distribution* has the following probability density function.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

This function has no symbolic antiderivative, so it can only be integrated numerically. The following code gives an "exact" value of the integral of $f(x)$ from $-\infty$ to a specified value.

```
>>> from scipy.stats import norm

>>> norm.cdf(1)                     # Integrate f from -infty to 1.
0.84134474606854293
>>> norm.cdf(1) - norm.cdf(-1)      # Integrate f from -1 to 1.
0.68268949213708585
```

Write a function that uses `scipy.stats` to calculate the "exact" value

$$F = \int_{-3}^{2} f(x)dx.$$

Then repeat the following experiment for $n = 5, 10, 15, \ldots, 50$.

1. Use your class from Problems 1–4 with the Legendre polynomials to approximate F using n points and weights. Calculate and record the error of the approximation.
2. Use your class with the Chebyshev polynomials to approximate F using n points and weights. Calculate and record the error of the approximation.

Plot the errors against the number of points and weights n, using a log scale for the y-axis. Finally, plot a horizontal line showing the error of `scipy.integrate.quad()` (which doesn't depend on n). Make sure your plot is clearly labeled.

Multivariate Quadrature

The extension of Gaussian quadrature to higher dimensions is fairly straightforward. The same set of points $\{z_i\}_{i=1}^n$ and weights $\{w_i\}_{i=1}^n$ can be used in each direction, so the only difference from 1-D quadrature is how the function is shifted and scaled. To begin, let $h : \mathbb{R}^2 \to \mathbb{R}$ and define $g : \mathbb{R}^2 \to \mathbb{R}$ by $g(x, y) = h(x, y)/(w(x)w(y))$ so that

$$\int_{-1}^{1}\int_{-1}^{1} h(x, y)dx\,dy. = \int_{-1}^{1}\int_{-1}^{1} g(x, y)w(x)w(y)dx\,dy \approx \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j g(z_i, z_j). \tag{11.3}$$

To integrate $f : \mathbb{R}^2 \to \mathbb{R}$ over an arbitrary box $[a_1, b_1] \times [a_2, b_2]$, set

$$h(x, y) = f\left(\frac{b_1 - a_1}{2}x + \frac{a_1 + b_1}{2},\ \frac{b_2 - a_2}{2}y + \frac{a_2 + b_2}{2}\right)$$

so that

$$\int_{a_2}^{b_2}\int_{a_1}^{b_1} f(x)dx\,dy = \frac{(b_1-a_1)(b_2-a_2)}{4}\int_{-1}^{1}\int_{-1}^{1} h(x,y)dx\,dy. \tag{11.4}$$

Combining (11.3) and (11.4) gives the final 2-D Gaussian quadrature formula. Compare it to (11.2).

$$\int_{a_2}^{b_2}\int_{a_1}^{b_1} f(x)dx\,dy \approx \frac{(b_1-a_1)(b_2-a_2)}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j g(z_i,z_j) \tag{11.5}$$

Problem 6. Write a method for your class that accepts a function $f:\mathbb{R}^2\to\mathbb{R}$ (which actually accepts two separate arguments, not one array with two elements) and bounds of integration a_1, a_2, b_1, and b_2. Use (11.5) to compute the double integral

$$\int_{a_2}^{b_2}\int_{a_1}^{b_1} f(x)dx\,dy.$$

Validate your method by comparing it `scipy.integrate.nquad()`. Note carefully that this function has slightly different syntax for the bounds of integration.

```
>>> from scipy.integrate import nquad

# Integrate f(x,y) = sin(x) + cos(y) over [-10,10] in x and [-1,1] in y.
>>> f = lambda x, y: np.sin(x) + np.cos(y)
>>> nquad(f, [[-10, 10], [-1, 1]])[0]
33.658839392315855
```

Note

Although Gaussian quadrature can obtain reasonable approximations in lower dimensions, it quickly becomes intractable in higher dimensions due to the curse of dimensionality. In other words, the number of points and weights required to obtain a good approximation becomes so large that Gaussian quadrature become computationally infeasible. For this reason, high-dimensional integrals are often computed via *Monte Carlo methods*, numerical integration techniques based on random sampling. However, quadrature methods are generally significantly more accurate in lower dimensions than Monte Carlo methods.

12 One-dimensional Optimization

Lab Objective: *Most mathematical optimization problems involve estimating the minimizer(s) of a scalar-valued function. Many algorithms for optimizing functions with a high-dimensional domain depend on routines for optimizing functions of a single variable. There are many techniques for optimization in one dimension, each with varying degrees of precision and speed. In this lab, we explore Newton's method and basins of attraction, and then we implement the secant method and apply it to the backtracking problem.*

Iterative Methods

An *iterative method* is an algorithm that must be applied repeatedly to obtain a result. The general idea behind any iterative method is to make an initial guess at the solution to a problem, apply a few easy computations to better approximate the solution, use that approximation as the new initial guess, and repeat until done. More precisely, let F be some function used to approximate the solution to a problem. Starting with an initial guess of x_0, compute

$$x_{k+1} = F(x_k) \tag{12.1}$$

for successive values of k to generate a sequence $(x_k)_{k=0}^{\infty}$ that hopefully converges to the true solution. If the terms of the sequence are vectors, they are denoted by $\mathbf{x}_k$.

In the best case, the iteration converges to the true solution x, written $\lim_{k\to\infty} x_k = x$ or $x_k \to x$. In the worst case, the iteration continues forever without approaching the solution. In practice, iterative methods require carefully chosen *stopping criteria* to guarantee that the algorithm terminates at some point. The general approach is to continue iterating until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. That is, choose a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$, and update the approximation using (12.1) until either

$$|x_k - x_{k-1}| < \varepsilon \qquad \text{or} \qquad k > N. \tag{12.2}$$

The choices for ε and N are significant: a "large" ε (such as 10^{-6}) produces a less accurate result than a "small" ε (such 10^{-16}), but demands less computations; a small N (10) also potentially lowers accuracy, but detects and halts nonconvergent iterations sooner than a large N (10,000). In code, ε and N are often named `tol` and `maxiter`, respectively (or similar).

While there are many ways to structure the code for an iterative method, probably the cleanest way is to combine a `for` loop with a `break` statement. As a very simple example, let $F(x) = \frac{x}{2}$. This method converges to $x = 0$ independent of starting point.

```
>>> F = lambda x: x / 2
>>> x0, tol, maxiter = 10, 1e-9, 8
>>> for k in range(maxiter):           # Iterate at most N times.
...     print(x0, end='  ')
...     x1 = F(x0)                      # Compute the next iteration.
...     if abs(x1 - x0) < tol:          # Check for convergence.
...         break                       # Upon convergence, stop iterating.
...     x0 = x1                         # Otherwise, continue iterating.
...
10  5.0  2.5  1.25  0.625  0.3125  0.15625  0.078125
```

In this example, the algorithm terminates after $N = 8$ iterations (the maximum number of allowed iterations) because the tolerance condition $|x_k - x_{k-1}| < 10^{-9}$ is not met fast enough. If N had been larger (say 40), the iteration would have quit early due to the tolerance condition.

Newton's Method

Newton's method is an important root-finding algorithm that can also be used for optimization. Given $f : \mathbb{R} \to \mathbb{R}$ and a good initial guess x_0, the sequence $(x_k)_{k=1}^{\infty}$ generated by the recursive rule

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{12.3}$$

converges to a point $\bar{x}$ satisfying $f(\bar{x}) = 0$ as long as three conditions hold:

1. f and f' exist and are continuous,
2. $f'(\bar{x}) \neq 0$, and
3. x_0 is "sufficiently close" to $\bar{x}$.

In applications, the first two conditions usually hold. If $\bar{x}$ and x_0 are not "sufficiently close," Newton's method may converge very slowly, or it may not converge at all. However, when all three conditions hold, Newton's method converges quadratically, meaning that the maximum error is squared at every iteration. This is very quick convergence, making Newton's method as powerful as it is simple.

Problem 1. Write a function that accepts a function f, an initial guess `x0`, the derivative Df, a stopping tolerance `tol` defaulting to 10^{-5}, and a maximum number of iterations `maxiter` defaulting to 15. Use Newton's method as described in (12.3) to compute a zero $\bar{x}$ of f. Terminate the algorithm when $|x_k - x_{k-1}|$ is less than `tol` or after iterating `maxiter` times. Return the last computed approximation to $\bar{x}$, a boolean value indicating whether or not the algorithm converged, and the number of iterations completed.

Test your function against functions like $f(x) = e^x - 2$ (see Figure 12.1) or $f(x) = x^4 - 3$. Check that the computed zero $\bar{x}$ satisfies $f(\bar{x}) \approx 0$. Also consider comparing your function to `scipy.optimize.newton()`, which accepts similar arguments.

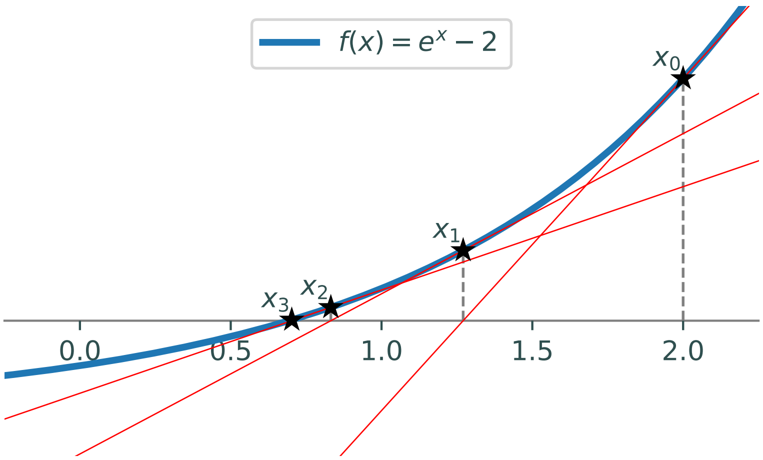


Figure 12.1: Newton's method approximates the zero of a function (blue) by choosing as the next approximation the x-intercept of the tangent line (red) that goes through the point $(x_k, f(x_k))$. In this example, $f(x) = e^x - 2$, which has a zero at $\bar{x} = \log(2)$. Setting $x_0 = 2$ and using (12.3) to iterate, we have $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{e^2-2}{e^2} \approx 1.2707$. Similarly, $x_2 \approx 0.8320$, $x_3 \approx .7024$, and $x_4 \approx 0.6932$. After only a few iterations, the zero $\log(2) \approx 0.6931$ is already computed to several digits of accuracy.

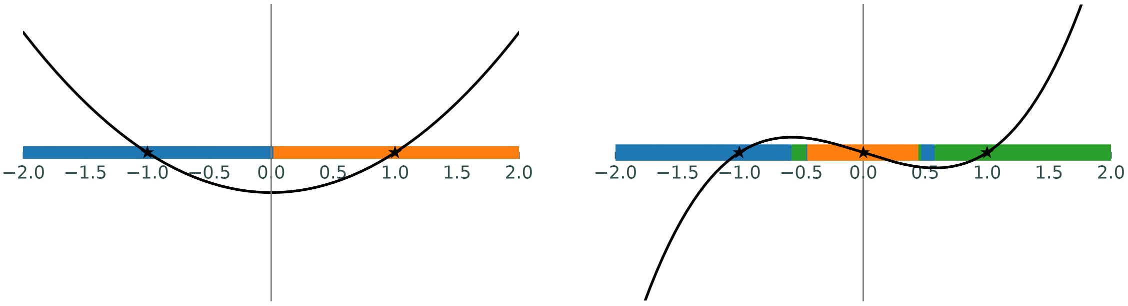
Note

Newton's method can be used to find zeros of functions that are hard to solve for analytically. For example, the function $f(x) = \frac{\sin(x)}{x} - x$ is not continuous on any interval containing 0, but it can be made continuous by defining $f(0) = 1$. Newton's method can then be used to compute the zeros of this function.

Basins of Attraction

When a function f has many zeros, the zero that Newton's method converges to depends on the initial guess x_0. For example, the function $f(x) = x^2 - 1$ has zeros at -1 and 1. If $x_0 < 0$, then Newton's method converges to -1; if $x_0 > 0$ then it converges to 1 (see Figure 12.2a). The regions $(-\infty, 0)$ and $(0, \infty)$ are called the *basins of attraction* of f. Starting in one basin of attraction leads to finding one zero, while starting in another basin yields a different zero.

When f is a polynomial of degree greater than 2, the basins of attraction are much more interesting. For example, the basis of attraction for $f(x) = x^3 - x$ are shown in Figure 12.2b. The basin for the zero at the origin is connected, but the other two basins are disconnected and share a kind of symmetry.



(a) Basins of attraction for $f(x) = x^2 - 1$.

(b) Basins of attraction for $f(x) = x^3 - x$.

Figure 12.2: Basins of attraction with $\alpha = 1$. Since choosing a different value for α can change which zero Newton's method converges to, the basins of attraction may change for other values of α.

It can be shown that Newton's method converges in any Banach space with only slightly stronger hypotheses than those discussed previously. In particular, Newton's method can be performed over the complex plane $\mathbb{C}$ to find imaginary zeros of functions. Plotting the basins of attraction over $\mathbb{C}$ yields some interesting results.

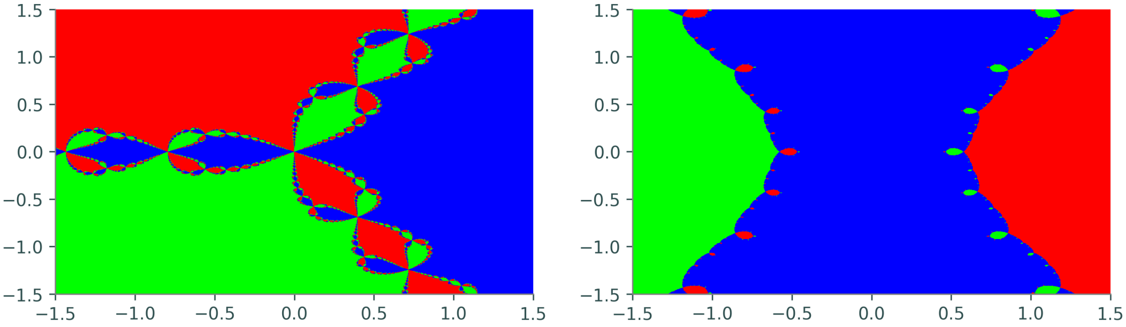
The zeros of $f(x) = x^3 - 1$ are 1, and $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$. To plot the basins of attraction for $f(x) = x^3 - 1$ on the square complex domain $X = \{a+bi \mid a \in [-\frac{3}{2}, \frac{3}{2}], b \in [-\frac{3}{2}, \frac{3}{2}]\}$, create an initial grid of complex points in this domain using `np.meshgrid()`.

```
>>> x_real = np.linspace(-1.5, 1.5, 500)    # Real parts.
>>> x_imag = np.linspace(-1.5, 1.5, 500)    # Imaginary parts.
>>> X_real, X_imag = np.meshgrid(x_real, x_imag)
>>> X_0 = X_real + 1j*X_imag                # Combine real and imaginary parts.
```

The grid X_0 is a 500×500 array of complex values to use as initial points for Newton's method. Array broadcasting makes it easy to compute an iteration of Newton's method at every grid point.

```
>>> f = lambda x: x**3 - 1
>>> Df = lambda x: 3*x**2
>>> X_1 = X_0 - f(X_0)/Df(X_0)
```

After enough iterations, the (i,j)th element of the grid X_k corresponds to the zero of f that results from using the (i,j)th element of X_0 as the initial point. For example, with $f(x) = x^3 - 1$, each entry of X_k should be close to 1, $-\frac{1}{2} + \frac{\sqrt{3}}{2}i$, or $-\frac{1}{2} - \frac{\sqrt{3}}{2}i$. Each entry of X_k can then be assigned a value indicating which zero it corresponds to. Some results of this process are displayed below.



(a) Basins of attraction for $f(x) = x^3 - 1$.

(b) Basins of attraction for $f(x) = x^3 - x$.

Figure 12.3

Note

Notice that in some portions of Figure 12.3a, whenever red and blue try to come together, a patch of green appears in between. This behavior repeats on an infinitely small scale, producing a fractal. Because it arises from Newton's method, this kind of fractal is called a *Newton fractal.*

Newton fractals show that the long-term behavior of Newton's method is **extremely** sensitive to the initial guess x_0. Changing x_0 by a small amount can change the output of Newton's method in a seemingly random way. This phenomenon is called *chaos* in mathematics.

Problem 2. Write a function that accepts a function $f : \mathbb{C} \to \mathbb{C}$, its derivative $Df : \mathbb{C} \to \mathbb{C}$, an array `zeros` of the zeros of f, a list `domain` containing the bounds $[r_{\text{min}}, r_{\text{max}}, i_{\text{min}}, i_{\text{max}}]$ for the domain of the plot, an integer `res` that determines the resolution of the plot, and number of iterations `iters` to run the iteration. Compute and plot the basins of attraction of f in the complex plane over the specified domain in the following steps.

1. Construct a `res`×`res` grid X_0 over the domain $\{a + bi \mid a \in [r_{\text{min}}, r_{\text{max}}], b \in [i_{\text{min}}, i_{\text{max}}]\}$.
2. Run Newton's method on X_0 `iters` times, obtaining the `res`×`res` array x_k. Do not check for convergence at each step.
3. X_k cannot be directly visualized directly because its values are complex. Solve this issue by creating another `res`×`res` array Y. To compute the (i, j)th entry $Y_{i,j}$, determine which zero of f is closest to the (i, j)th entry of X_k. Set $Y_{i,j}$ to the index of this zero in the array `zeros`. If there are R distinct zeros, each $Y_{i,j}$ should be one of $0, 1, \ldots, R - 1$. (Hint: `np.argmin()` may be useful.)
4. Use `plt.pcolormesh()` to visualize the basins. Recall that this function accepts three array arguments: the x-coordinates (in this case, the real components of the initial grid), the y-coordinates (the imaginary components of the grid), and an array indicating color values (Y). Set `cmap="brg"` to get the same color scheme as in Figure 12.3.

Test your function using $f(x) = x^3 - 1$ and $f(x) = x^3 - x$. The resulting plots should resemble Figures 12.3a and 12.3b, respectively (perhaps with the colors permuted).

The Secant Method

The first-order necessary conditions from elementary calculus state that if f is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of f' is a way to identify potential minima or maxima of f. Specifically, starting with an initial guess x_0, set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{12.4}$$

and iterate until $|x_k - x_{k-1}|$ is satisfactorily small. Note that this procedure does not use the actual function f at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

The second derivative of an objective function is not always known or may be prohibitively expensive to evaluate. The *secant method* solves this problem by numerically approximating the second derivative with a difference quotient.

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

Selecting $x = x_k$ and $h = x_{k-1} - x_k$ gives the following approximation.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \tag{12.5}$$

Inserting (12.5) into (12.4) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1} f'(x_k) - x_k f'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \tag{12.6}$$

Notice that this recurrence relation requires two previous points (both x_k and x_{k-1}) to calculate the next estimate. This method converges superlinearly — slower than Newton's method — with convergence criteria similar to Newton's method.

Problem 3. Write a function that accepts a first derivative `df`, starting points `x0` and `x1`, a stopping tolerance `tol`, and a maximum of iterations `maxiter`. Use (12.6) to implement the Secant method. Try to make as few computations as possible by only computing `df` at (x_k) once for each k. Return the minimizer approximation, whether or not the algorithm converged, and the number of iterations computed.

Test your code with the function $f(x) = x^2 + \sin(x) + \sin(10x)$ and with initial guesses of $x_0 = 0$ and $x_1 = -1$. Plot your answer with the graph of the function. Also compare your results to `scipy.optimize.newton()`; without providing the `fprime` argument, this function uses the secant method. However, it still only takes in one initial condition, so it may converge to a different local minimum than your function.

```
>>> df = lambda x: 2*x + np.cos(x) + 10*np.cos(10*x)
>>> optimize.newton(df, x0=0, tol=1e-10, maxiter=500)
2.3155516573790806
```

Descent Methods

Consider now a function $f : \mathbb{R}^n \to \mathbb{R}$. *Descent methods*, also called *line search methods*, are optimization algorithms that create a convergent sequence $(x_k)_{k=1}^{\infty}$ by the following rule.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{12.7}$$

Here $\alpha_k \in \mathbb{R}$ is called the *step size* and $\mathbf{p}_k \in \mathbb{R}^n$ is called the *search direction.* The choice of $\mathbf{p}_k$ is usually what distinguishes an algorithm; in the one-dimensional case ($n = 1$), $p_k = f'(x_k)/f''(x_k)$ results in Newton's method, and using the approximation in (12.5) results in the secant method.

To be effective, a descent method must also use a good step size α_k. If α_k is too large, the method may repeatedly overstep the minimum; if α_k is too small, the method may converge extremely slowly. See Figure 12.4.

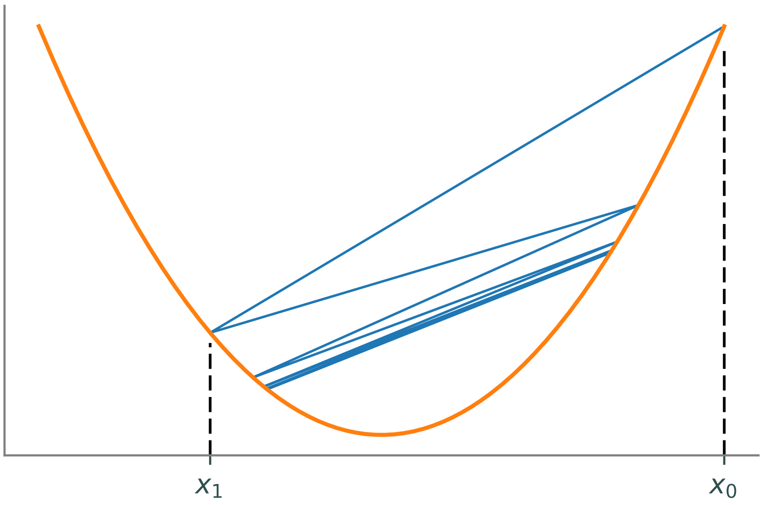


Figure 12.4: If the step size α_k is too large, a descent method may repeatedly overstep the minimizer.

Given a search direction $\mathbf{p}_k$, the best step size α_k minimizes the function $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha\mathbf{p}_k)$. Since f is scalar-valued, $\phi_k : \mathbb{R} \to \mathbb{R}$, so any of the optimization methods discussed previously can be used to minimize ϕ_k. However, computing the best α_k at every iteration is not always practical. Instead, some methods use a cheap routine to compute a step size that may not be optimal, but which is good enough. The most common approach is to find an α_k that satisfies the *Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^{\mathsf{T}} \mathbf{p}_k \tag{12.8}$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^{\mathsf{T}} \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^{\mathsf{T}} \mathbf{p}_k \tag{12.9}$$

where $0 < c1 < c2 < 1$ (for the best results, choose $c1 << c2$). The condition (12.8) is also called the *Armijo rule* and ensures that the step decreases f. However, this condition is not enough on its own. By Taylor's theorem,

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

Thus, a very small α_k will always satisfy (12.8) since $Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k < 0$ (as $\mathbf{p}_k$ is a descent direction). The condition (12.9), called the *curvature condition*, ensures that the α_k is large enough for the algorithm to make significant progress.

It is possible to find an α_k that satisfies the Wolfe conditions, but that is far from the minimizer of $\phi_k(\alpha)$. The *strong Wolfe conditions* modify (12.9) to ensure that α_k is near the minimizer.

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

The *Armijo–Goldstein conditions* provide another alternative to (12.9):

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

where $0 < c < 1$. These conditions are very similar to the Wolfe conditions (the right inequality is (12.8)), but they do not require the calculation of the directional derivative $Df(\mathbf{x}_k + \alpha_k\mathbf{p}_k)^\mathsf{T}\mathbf{p}_k$.

Backtracking

A *backtracking line search* is a simple strategy for choosing an acceptable step size α_k: start with an fairly large initial step size α, then repeatedly scale it down by a factor ρ until the desired conditions are satisfied. The following algorithm only requires α to satisfy (12.8). This is usually sufficient, but if it finds α's that are too small, the algorithm can be modified to satisfy (12.9) or one of its variants.

Algorithm 1 Backtracking using the Armijo Rule

```
procedure BACKTRACKING($f$, $Df$, $\mathbf{x}_k$, $\mathbf{p}_k$, $\alpha$, $\rho$, $c$)
    Dfp ← $Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k$                ▷ Compute these values only once.
    fx ← $f(\mathbf{x}_k)$
    while $\big(f(\mathbf{x}_k + \alpha\mathbf{p}_k) > \texttt{fx} + c\alpha\texttt{Dfp}\big)$ do
        $\alpha \leftarrow \rho\alpha$
    return $\alpha$
```

Problem 4. Write a function that accepts a function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an approximate minimizer $\mathbf{x}_k$, a search direction $\mathbf{p}_k$, an initial step length α, and parameters ρ and c. Implement the backtracking method of Algorithm 1. Return the computed step size.

The functions f and Df should both accept 1-D NumPy arrays of length n. For example, if $f(x,y,z) = x^2 + y^2 + z^2$, then f and Df could be defined as follows.

```
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> Df = lambda x: np.array([2*x[0], 2*x[1], 2*x[2]])
```

SciPy's `scipy.optimize.linesearch.scalar_search_armijo()` finds an acceptable step size using the Armijo rule. It may not give the exact answer as your implementation since it decreases α differently, but the answers should be similar.

```
>>> from scipy.optimize import linesearch
>>> from jax import numpy as jnp
>>> from jax import grad

# Get a step size for f(x,y,z) = x^2 + y^2 + z^2.
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> x = jnp.array([150., .03, 40.])         # Current minimizer guesss.
>>> p = jnp.array([-.5, -100., -4.5])       # Current search direction.
>>> phi = lambda alpha: f(x + alpha*p)      # Define phi(alpha).
>>> dphi = grad(phi)
>>> alpha, _ = linesearch.scalar_search_armijo(phi, phi(0.), dphi(0.))
```

13 Gradient Descent Methods

Lab Objective: *Iterative optimization methods choose a search direction and a step size at each iteration. One simple choice for the search direction is the negative gradient, resulting in the method of steepest descent. While theoretically foundational, in practice this method is often slow to converge. An alternative method, the conjugate gradient algorithm, uses a similar idea that results in much faster convergence in some situations. In this lab we implement a method of steepest descent and two conjugate gradient methods, then apply them to regression problems.*

The Method of Steepest Descent

Let $f : \mathbb{R}^n \to \mathbb{R}$ with first derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$. The following iterative technique is a common template for methods that aim to compute a local minimizer $\mathbf{x}^*$ of f.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{13.1}$$

Here $\mathbf{x}_k$ is the kth approximation to $\mathbf{x}^*$, α_k is the *step size*, and $\mathbf{p}_k$ is the *search direction*. Newton's method and its relatives follow this pattern, but they require the calculation (or approximation) of the inverse Hessian matrix $Df^2(\mathbf{x}_k)^{-1}$ at each step. The following idea is a simpler and less computationally intensive approach than Newton and quasi-Newton methods.

The derivative $Df(\mathbf{x})^\mathsf{T}$ (often called the *gradient* of f at $\mathbf{x}$, sometimes notated $\nabla f(\mathbf{x})$) is a vector that points in the direction of greatest **increase** of f at $\mathbf{x}$. It follows that the negative derivative $-Df(\mathbf{x})^\mathsf{T}$ points in the direction of steepest **decrease** at $\mathbf{x}$. The *method of steepest descent* chooses the search direction $\mathbf{p}_k = -Df(\mathbf{x}_k)^\mathsf{T}$ at each step of (13.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\mathsf{T} \tag{13.2}$$

Setting $\alpha_k = 1$ for each k is often sufficient for Newton and quasi-Newton methods. However, a constant choice for the step size in (13.2) can result in oscillating approximations or even cause the sequence $(\mathbf{x}_k)_{k=1}^\infty$ to travel away from the minimizer $\mathbf{x}^*$. To avoid this problem, the step size α_k can be chosen in a few ways.

- Start with $\alpha_k = 1$, then set $\alpha_k = \frac{\alpha_k}{2}$ until $f(\mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\mathsf{T}) < f(\mathbf{x}_k)$, terminating the iteration if α_k gets too small. This guarantees that the method actually descends at each step and that α_k satisfies the Armijo rule, without endangering convergence.

- At each step, solve the following one-dimensional optimization problem.

$$\alpha_k = \underset{\alpha}{\text{argmin}}\ f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^\mathsf{T})$$

Using this choice is called *exact steepest descent*. This option is more expensive per iteration than the above strategy, but it results in fewer iterations before convergence.

Problem 1. Write a function that accepts an objective function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, a convergence tolerance `tol` defaulting to $1e^{-5}$, and a maximum number of iterations `maxiter` defaulting to 100. Implement the exact method of steepest descent, using a one-dimensional optimization method to choose the step size (use `opt.minimize_scalar()` or your own 1-D minimizer). Iterate until $\|Df(\mathbf{x}_k)\|_\infty <$ `tol` or $k >$ `maxiter`. Return the approximate minimizer $\mathbf{x}^*$, whether or not the algorithm converged (`True` or `False`), and the number of iterations computed.

Test your function on $f(x, y, z) = x^4 + y^4 + z^4$ (easy) and the Rosenbrock function (hard). It should take many iterations to minimize the Rosenbrock function, but it should converge eventually with a large enough choice of `maxiter`.

The Conjugate Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems. Depending on the nature of the objective function, the sequence of points can zig-zag back and forth or get stuck on flat areas without making significant progress toward the true minimizer.

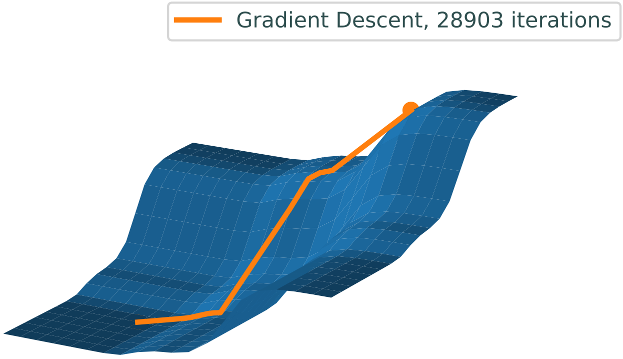


Figure 13.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

Unlike the method of steepest descent, the *conjugate gradient algorithm* chooses a search direction that is guaranteed to be a descent direction, though not the direction of greatest descent. These directions are using a generalized form of orthogonality called *conjugacy*.

Let Q be a square, positive definite matrix. A set of vectors $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_m\}$ is called Q-*conjugate* if each distinct pair of vectors $\mathbf{x}_i, \mathbf{x}_j$ satisfy $\mathbf{x}_i^\mathsf{T} Q \mathbf{x}_j = 0$. A Q-conjugate set of vectors is linearly independent and can form a basis that diagonalizes the matrix Q. This guarantees that an iterative method to solve $Q\mathbf{x} = \mathbf{b}$ only require as many steps as there are basis vectors.

Solve a positive definite system $Q\mathbf{x} = \mathbf{b}$ is valuable in and of itself for certain problems, but it is also equivalent to minimizing certain functions. Specifically, consider the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T} Q \mathbf{x} - \mathbf{b}^\mathsf{T}\mathbf{x} + c.$$

Because $Df(\mathbf{x})^\mathsf{T} = Q\mathbf{x} - \mathbf{b}$, minimizing f is the same as solving the equation

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

which is the original linear system. Note that the constant c does not affect the minimizer, since if $\mathbf{x}^*$ minimizes $f(\mathbf{x})$ it also minimizes $f(\mathbf{x}) + c$.

Using the conjugate directions guarantees an iterative method to converge on the minimizer because each iteration minimizes the objective function over a subspace of dimension equal to the iteration number. Thus, after n steps, where n is the number of conjugate basis vectors, the algorithm has found a minimizer over the entire space. In certain situations, this has a great advantage over gradient descent, which can bounce back and forth. This comparison is illustrated in Figure 13.2. Additionally, because the method utilizes a basis of conjugate vectors, the previous search direction can be used to find a conjugate projection onto the next subspace, saving computational time.

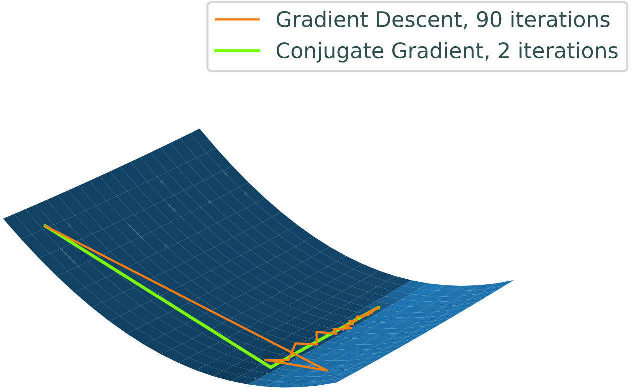

Gradient Descent, 90 iterations
Conjugate Gradient, 2 iterations

Figure 13.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

Algorithm 1

```
procedure Conjugate Gradient($\mathbf{x}_0$, $Q$, $\mathbf{b}$, tol)
    $\mathbf{r}_0 \leftarrow Q\mathbf{x}_0 - \mathbf{b}$
    $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$
    $k \leftarrow 0$
    while $\|\mathbf{r}_k\| \geq$ tol, $k < n$ do
        $\alpha_k \leftarrow \mathbf{r}_k^\mathsf{T}\mathbf{r}_k / \mathbf{d}_k^\mathsf{T} Q \mathbf{d}_k$
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$
        $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k Q \mathbf{d}_k$
        $\beta_{k+1} \leftarrow \mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1} / \mathbf{r}_k^\mathsf{T}\mathbf{r}_k$
        $\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{d}_k$
        $k \leftarrow k + 1.$
    return $\mathbf{x}_{k+1}$
```

The points $\mathbf{x}_k$ are the successive approximations to the minimizer, the vectors $\mathbf{d}_k$ are the conjugate descent directions, and the vectors $\mathbf{r}_k$ (which actually correspond to the steepest descent directions) are used in determining the conjugate directions. The constants α_k and β_k are used, respectively, in the line search, and in ensuring the Q-conjugacy of the descent directions.

Problem 2. Write a function that accepts an $n \times n$ positive definite matrix Q, a vector $\mathbf{b} \in \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, and a stopping tolerance. Use Algorithm 1 to solve the system $Q\mathbf{x} = \mathbf{b}$. Continue the algorithm until $\|\mathbf{r}_k\|$ is less than the tolerance, iterating no more than n times. Return the solution $\mathbf{x}$, whether or not the algorithm converged in n iterations or less, and the number of iterations computed. Test your function on the simple system

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix},$$

which has solution $\mathbf{x}^* = \left[\frac{1}{2}, 2\right]^\mathsf{T}$. This is equivalent to minimizing the quadratic function $f(x, y) = x^2 + 2y^2 - x - 8y$; check that your function from Problem 1 gets the same solution.

More generally, you can generate a random positive definite matrix Q for testing by setting setting $Q = A^\mathsf{T} A$ for any A of full rank. Note, for values of $n \leq 5$ this method is not stable enough to always converge in exaclty n iterations. Try using the code given below to test your function for values of $n < 5$.

There is a file called `test_gradient_methods.py` that contains some prewritten unit tests that you can use to test your function.

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

# Generate Q, b, and the initial guess x0.
>>> n = 4
>>> A = np.random.random((n, n))
>>> Q = A.T @ A
>>> b, x0 = np.random.random((2, n))

>>> x = la.solve(Q, b)      # Use your function here.
>>> np.allclose(Q @ x, b)
True
```

Non-linear Conjugate Gradient

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions. Though the non-linear version does not have guaranteed convergence as the linear formulation does, it can still converge in less iterations than the method of steepest descent. Modifying the algorithm for more general functions requires new formulas for α_k, $\mathbf{r}_k$, and β_k.

- The scalar α_k is simply the result of performing a line-search in the given direction $\mathbf{d}_k$ and is thus defined $\alpha_k = \underset{\alpha}{\text{argmin}}\ f(\mathbf{x}_k + \alpha\mathbf{d}_k)$.
- The vector $\mathbf{r}_k$ in the original algorithm was really just the gradient of the objective function, so now define $\mathbf{r}_k = Df(\mathbf{x}_k)^{\mathsf{T}}$.
- The constants β_k can be defined in various ways, and the most correct choice depends on the nature of the objective function. A well-known formula, attributed to Fletcher and Reeves, is $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^{\mathsf{T}}/Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^{\mathsf{T}}$.

Algorithm 2

```
procedure Non-Linear Conjugate Gradient($f$, $Df$, $\mathbf{x}_0$, tol, maxiter)
    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^{\mathsf{T}}$
    $\mathbf{d}_0 \leftarrow \mathbf{r}_0$
    $\alpha_0 \leftarrow \underset{\alpha}{\text{argmin}} f(\mathbf{x}_0 + \alpha\mathbf{d}_0)$
    $\mathbf{x}_1 \leftarrow \mathbf{x}_0 + \alpha_0\mathbf{d}_0$
    $k \leftarrow 1$
    while $\|\mathbf{r}_k\| \geq$ tol, $k <$ maxiter do
        $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^{\mathsf{T}}$
        $\beta_k = \mathbf{r}_k^{\mathsf{T}}\mathbf{r}_k/\mathbf{r}_{k-1}^{\mathsf{T}}\mathbf{r}_{k-1}$
        $\mathbf{d}_k \leftarrow \mathbf{r}_k + \beta_k\mathbf{d}_{k-1}$.
        $\alpha_k \leftarrow \underset{\alpha}{\text{argmin}}\ f(\mathbf{x}_k + \alpha\mathbf{d}_k)$.
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{d}_k$.
        $k \leftarrow k + 1$.
```

Problem 3. Write a function that accepts a convex objective function f, its derivative Df, an initial guess $\mathbf{x}_0$, a convergence tolerance defaulting to $1e^{-5}$, and a maximum number of iterations defaultin to 100. Use Algorithm 2 to compute the minimizer $\mathbf{x}^*$ of f. Return the approximate minimizer, whether or not the algorithm converged, and the number of iterations computed.

Compare your function to SciPy's `opt.fmin_cg()`.

```
>>> opt.fmin_cg(opt.rosen, np.array([10, 10]), fprime=opt.rosen_der)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 44
         Function evaluations: 102  # Much faster than steepest descent!
         Gradient evaluations: 102
array([ 1.00000007,  1.00000015])
```

Unit Test

There is a file called `test_gradient_methods.py` that contains some prewritten unit tests for Problem 2. There is a place for you to add your own unit tests to test your function from Problem 3 which will be graded.

Regression Problems

A major use of the conjugate gradient method is solving linear least squares problems. Recall that a least squares problem can be formulated as an optimization problem:

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

where A is an $m \times n$ matrix with full column rank, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^m$. The solution can be calculated analytically, and is given by

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

In other words, the minimizer solves the linear system

$$A^\mathsf{T} A\mathbf{x} = A^\mathsf{T}\mathbf{b}. \tag{13.3}$$

Since A has full column rank, it is invertible, $A^\mathsf{T} A$ is positive definite, and for any non-zero vector $\mathbf{z}$, $A\mathbf{z} \neq 0$. Therefore, $\mathbf{z}^\mathsf{T} A^\mathsf{T} A\mathbf{z} = \|A\mathbf{z}\|^2 > 0$. As $A^\mathsf{T} A$ is positive definite, conjugate gradient can be used to solve Equation 13.3.

Linear least squares is the mathematical underpinning of *linear regression*. Linear regression involves a set of real-valued data points $\{y_1, \ldots, y_m\}$, where each y_i is paired with a corresponding set of predictor variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}\}$ with $n < m$. The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_n x_{i,n} + \varepsilon_i$$

for $i = 1, 2, \ldots, m$. The real numbers $\beta_0, \ldots, \beta_n$ are known as the parameters of the model, and the ε_i are independent, normally-distributed error terms. The goal of linear regression is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares. Define

$$\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}.$$

The solution $\mathbf{x}^* = [\beta_0^*, \beta_1^*, \ldots, \beta_n^*]^\mathsf{T}$ to the system $A^\mathsf{T}A\mathbf{x} = A^\mathsf{T}\mathbf{b}$ gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data.

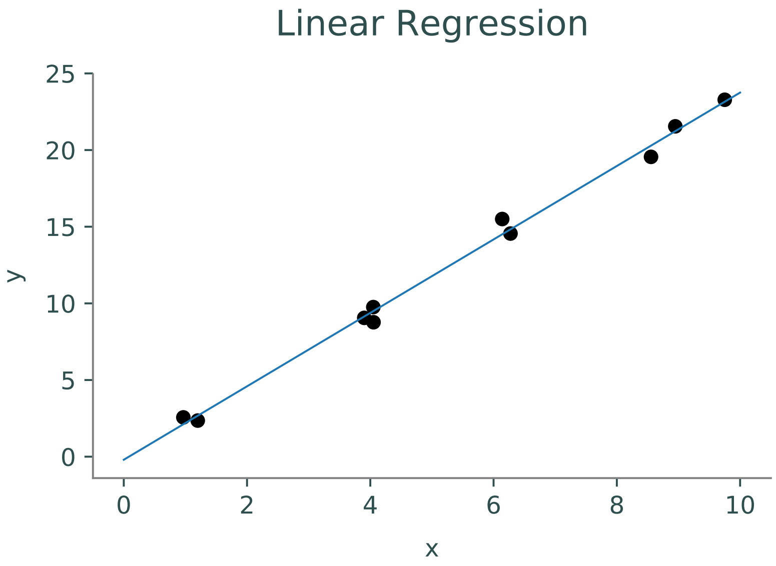


Figure 13.3: Solving the linear regression problem results in a best-fit hyperplane.

Problem 4. Using your function from Problem 2, solve the linear regression problem specified by the data contained in the file[a] `linregression.txt`. This is a whitespace-delimited text file formatted so that the i-th row consists of $y_i, x_{i,1}, \ldots, x_{i,n}$. Use `np.loadtxt()` to load in the data and return the solution to the normal equations.

[a] Source: Statistical Reference Datasets website at `http://www.itl.nist.gov/div898/strd/lls/data/LINKS/v-Longley.shtml`.

Logistic Regression

Logistic regression is another important technique in statistical analysis and machine learning that builds off of the concepts of linear regression. As in linear regression, there is a set of predictor variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}\}_{i=1}^m$ with corresponding outcome variables $\{y_i\}_{i=1}^m$. In logistic regression, the outcome variables y_i are binary and can be modeled by a *sigmoidal* relationship. The value of the predicted y_i can be thought of as the probability that $y_i = 1$. In mathematical terms,

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

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 $\beta_0, \beta_1, \ldots, \beta_n$. Note that $p_i \in (0, 1)$ regardless of the values of the predictor variables and parameters.

The probability of observing the outcome variables y_i under this model, assuming they are independent, is given by the *likelihood function* $\mathcal{L} : \mathbb{R}^{n+1} \to \mathbb{R}$

$$\mathcal{L}(\beta_0, \ldots, \beta_n) = \prod_{i=1}^{m} p_i^{y_i}(1 - p_i)^{1-y_i}.$$

The goal of logistic regression is to find the parameters $\beta_0, \ldots, \beta_k$ that maximize this likelihood function. Thus, the problem can be written as:

$$\max_{(\beta_0,\ldots,\beta_n)} \mathcal{L}(\beta_0, \ldots, \beta_n).$$

Maximizing this function is often a numerically unstable calculation. Thus, to make the objective function more suitable, the logarithm of the objective function may be maximized because the logarithmic function is strictly monotone increasing. Taking the log and turning the problem into a minimization problem, the final problem is formulated as:

$$\min_{(\beta_0,\ldots,\beta_n)} -\log \mathcal{L}(\beta_0, \ldots, \beta_n).$$

A few lines of calculation reveal that this objective function can also be rewritten as

$$\begin{aligned} -\log \mathcal{L}(\beta_0, \ldots, \beta_n) = &\sum_{i=1}^{m} \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}))) + \\ &\sum_{i=1}^{m} (1 - y_i)(\beta_0 + \beta_1 x_{i,1} + \cdots + \beta_n x_{i,n}). \end{aligned}$$

The values for the parameters $\{\beta_i\}_{i=1}^n$ that we obtain are known as the *maximum likelihood estimate* (MLE). To find the MLE, conjugate gradient can be used to minimize the objective function.

For a one-dimensional binary logistic regression problem, we have predictor data $\{x_i\}_{i=1}^m$ with labels $\{y_i\}_{i=1}^m$ where each $y_i \in \{0, 1\}$. The negative log likelihood then becomes the following.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^{m} \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \tag{13.4}$$

Problem 5. Write a class for doing binary logistic regression in one dimension that implement the following methods.

1. `fit()`: accept an array $\mathbf{x} \in \mathbb{R}^n$ of data, an array $\mathbf{y} \in \mathbb{R}^n$ of labels (0s and 1s), and an initial guess $\boldsymbol{\beta}_0 \in \mathbb{R}^2$. Define the negative log likelihood function as given in (13.4), then minimize it (with respect to $\boldsymbol{\beta}$) with your function from Problem 3 or `opt.fmin_cg()`. Store the resulting parameters β_0 and β_1 as attributes.

2. `predict()`: accept a float $x \in \mathbb{R}$ and calculate

$$\sigma(x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x))},$$

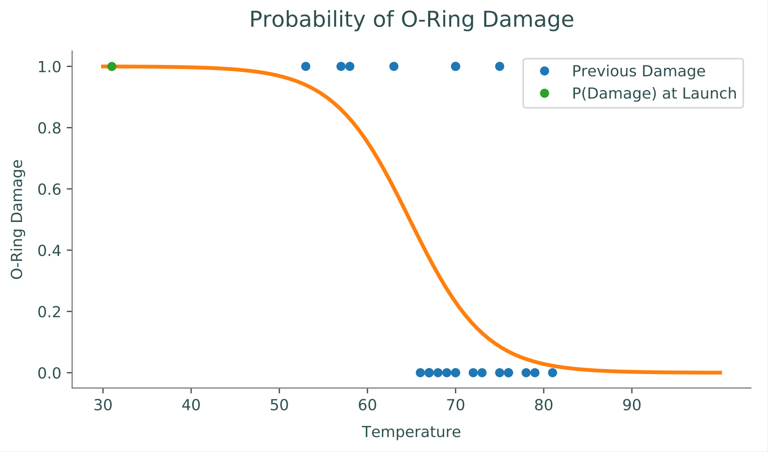
where β_0 and β_1 are the optimal values calculated in `fit()`. The value $\sigma(x)$ is the probability that the observation x should be assigned the label $y = 1$.

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

Problem 6. On January 28, 1986, less than two minutes into the Challenger space shuttle's 10th mission, there was a large explosion that originated from the spacecraft, killing all seven crew members and destroying the shuttle. The investigation that followed concluded that the malfunction was caused by damage to O-rings that are used as seals for parts of the rocket engines. There were 24 space shuttle missions before this disaster, some of which had noted some O-ring damage. Given the data, could this disaster have been predicted?

The file `challenger.npy` contains data for 23 missions (during one of the 24 missions, the engine was lost at sea). The first column ($\mathbf{x}$) contains the ambient temperature, in Fahrenheit, of the shuttle launch. The second column ($\mathbf{y}$) contains a binary indicator of the presence of O-ring damage (1 if O-ring damage was present, 0 otherwise).

Instantiate your class from Problem 5 and fit it to the data, using an initial guess of $\boldsymbol{\beta}_0 = [20, -1]^\mathsf{T}$. Plot the resulting curve $\sigma(x)$ for $x \in [30, 100]$, along with the raw data. Return the predicted probability (according to this model) of O-ring damage on the day the shuttle was launched, given that it was 31°F.



14 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.*

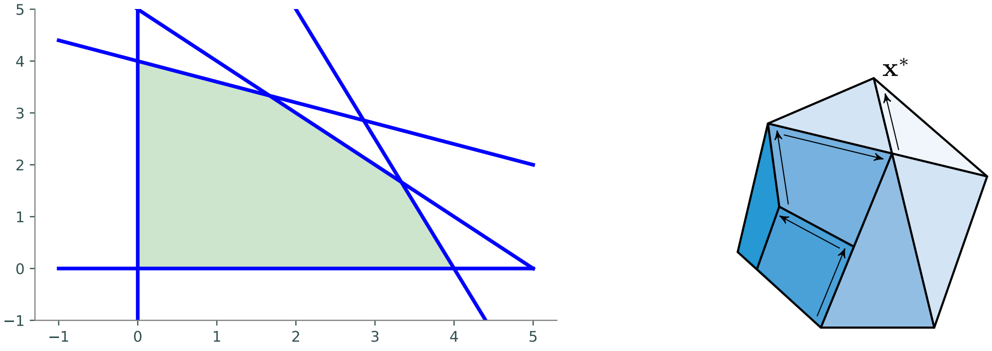
Standard Form

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

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

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 14.1). 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 14.1: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

Implementing the Simplex Algorithm is straightforward, provided one carefully follows the procedure. We will break the algorithm into several small steps, and write a function to perform each one. To become familiar with the execution of the Simplex algorithm, it is helpful to work several examples by hand.

The Simplex Solver

Our program will be more lengthy than many other lab exercises and will consist of a collection of functions working together to produce a final result. It is important to clearly define the task of each function and how all the functions will work together. If this program is written haphazardly, it will be much longer and more difficult to read than it needs to be. We will walk you through the steps of implementing the Simplex Algorithm as a Python class.

For demonstration purposes, we will use the following linear program.

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

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. For now, we only check for feasibility at the origin. A more robust solver sets up the auxiliary problem and solves it to find a starting point if the origin is infeasible.

Problem 1. Write a class that accepts the arrays $\mathbf{c}$, A, and $\mathbf{b}$ of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin. That is, check that $A\mathbf{x} \preceq \mathbf{b}$ when $\mathbf{x} = \mathbf{0}$. Raise a `ValueError` if the problem is not feasible at the origin.

Adding Slack Variables

The next step is to convert the inequality constraints $A\mathbf{x} \preceq \mathbf{b}$ into equality constraints by introducing a slack variable for each constraint equation. If the constraint matrix A is an $m \times n$ matrix, then there are m slack variables, one for each row of A. Grouping all of the slack variables into a vector $\mathbf{w}$ of length m, the constraints now take the form $A\mathbf{x} + \mathbf{w} = \mathbf{b}$. In our example, we have

$$\mathbf{w} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

When adding slack variables, it is useful to represent all of your variables, both the original primal variables and the additional slack variables, in a convenient manner. One effective way is to refer to a variable by its subscript. For example, we can use the integers 0 through $n-1$ to refer to the original (non-slack) variables x_0 through x_{n-1}, and we can use the integers n through $n+m-1$ to track the slack variables (where the slack variable corresponding to the ith row of the constraint matrix is represented by the index $n+i-1$).

We also need some way to track which variables are *independent* (non-zero) and which variables are *dependent* (those that have value 0). This can be done using the objective function. At anytime during the optimization process, the non-zero variables in the objective function are *independent* and all other variables are *dependent*.

Creating a Dictionary

After we have determined that our program is feasible, we need to create the *dictionary* (sometimes called the *tableau*), a matrix to track the state of the algorithm.

There are many different ways to build your dictionary. We will do this by setting the corresponding dependent variable equations to 0. For example, if x_5 were a dependent variable we would expect to see a -1 in the column that represents x_5. Define

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

where I_m is the $m \times m$ identity matrix we will use to represent our slack variables, and define

$$\bar{\mathbf{c}} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}.$$

That is, $\bar{\mathbf{c}} \in \mathbb{R}^{n+m}$ such that the first n entries are $\mathbf{c}$ and the final m entries are zeros. Then the initial dictionary has the form

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^{\mathsf{T}} \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{14.1}$$

The columns of the dictionary correspond to each of the variables (both primal and slack), and the rows of the dictionary correspond to the dependent variables.

For our example the initial dictionary is

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

The advantage of using this kind of dictionary is that it is easy to check the progress of your algorithm by hand.

Problem 2. Add a method to your Simplex solver that takes in arrays c, A, and b to create the initial dictionary as a NumPy array. Make sure to initialize the dictionary in `__init__` by calling the method you just created and name the parameter self.dictionary (failure to do this will result in no points received for the problem).

Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping dependent and independent variables, and transforming the dictionary appropriately. This has the effect of moving from one vertex of the feasible polytope to another vertex in a way that increases the value of the objective function. Depending on how you store your variables, you may need to modify a few different parts of your solver to reflect this swapping.

When initiating a pivot, you need to determine which variables will be swapped. In the dictionary representation, you first find a specific element on which to pivot, and the row and column that contain the pivot element correspond to the variables that need to be swapped. Row operations are then performed on the dictionary so that the pivot column becomes a negative elementary vector.

Let's break it down, starting with the pivot selection. We need to use some care when choosing the pivot element. To find the pivot column, search from left to right along the top row of the dictionary (ignoring the first column), and stop once you encounter the first negative value. The index corresponding to this column will be designated the *entering index*, since after the full pivot operation, it will enter the basis and become a dependent variable.

Using our initial dictionary D in the example, we stop at the second column:

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

We now know that our pivot element will be found in the second column. The entering index is thus 1.

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

$$\frac{-D_{i_1,0}}{D_{i_1,j}}, \frac{-D_{i_2,0}}{D_{i_2,j}}, \ldots, \frac{-D_{i_k,0}}{D_{i_k,j}},$$

and we choose our pivot element to be one that minimizes this ratio. If multiple entries minimize the ratio, then we utilize *Bland's Rule*, which instructs us to choose the entry in the row corresponding to the smallest index (obeying this rule is important, as it prevents the possibility of the algorithm cycling back on itself infinitely). The index corresponding to the pivot row is designated as the *leaving index*, since after the full pivot operation, it will leave the basis and become a independent variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are negative, and hence they are all potential choices for the pivot element. We then calculate the ratios, and obtain

$$\frac{-2}{-1} = 2, \quad \frac{-5}{-3} = 1.66..., \quad \frac{-7}{-4} = 1.75.$$

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column (index 1), third row (index 2) is our designated pivot element.

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

Problem 3. Write a method that will determine the pivot row and pivot column according to Bland's Rule.

Definition 14.1 (Bland's Rule). *Choose the independent variable with the smallest index that has a negative coefficient in the objective function as the leaving variable. Choose the dependent variable with the smallest index among all the binding dependent variables.*

Bland's Rule is important in avoiding cycles when performing pivots. This rule guarantees that a feasible Simplex problem will terminate in a finite number of pivots. *Hint:* Avoid dividing by zero.

Finally, we perform row operations on our dictionary in the following way: divide the pivot row by the negative value of the pivot entry. Then use the pivot row to zero out all entries in the pivot column above and below the pivot entry. In our example, our pivot is -3. So, we must first divide the pivot row by 3, and then zero out the two entries above the pivot element and the single entry below it:

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

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

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

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

Problem 4. Add a method to your solver that checks for unboundedness and performs a single pivot operation from start to completion. If the problem is unbounded, raise a `ValueError`.

Termination and Reading the Dictionary

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial dictionary. After carrying out these initial steps, it then performs the pivoting operation iteratively until the optimal point is found. But how do we determine when the optimal point is found? The answer is to look at the top row of the dictionary, which represents the objective function. More specifically, before each pivoting operation, check whether all of the entries in the top row of the dictionary (ignoring the entry in the first column) are nonnegative. If this is the case, then we have found an optimal solution, and so we terminate the algorithm.

The final step is to report the solution. The ending state of the dictionary and index list tells us everything we need to know. The minimal value attained by the objective function is found in the upper leftmost entry of the dictionary. Variables with nonzero entries in the objective function, or first row of our dictionary array, are independent variables. Variables with an entry of 0 in the objective function are dependent variables, and their values are given by the first column of the dictionary. Specifically, independent variables are given the value of 0 while the dependent variable whose index is located at the ith entry of the index list has the value $T_{i+1,0}$.

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

$$D = \begin{bmatrix} -5.2 & 0 & 0 & 0 & 0.2 & 0.6 \\ 0.6 & 0 & 0 & -1 & 1.4 & -0.8 \\ 1.6 & -1 & 0 & 0 & -0.6 & 0.2 \\ 0.2 & 0 & -1 & 0 & 0.8 & -0.6 \end{bmatrix}$$

Then the minimal value of the objective function is -5.2. The independent variables have indices $4, 5$ and have the value 0. The dependent variables have indices $3, 1$, and 2, and have values $.6, 1.6$, and $.2$, respectively. In the notation of the original problem statement, the solution is given by

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

Problem 5. Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the minimal value, the dependent variables, and the independent variables. The dependent and independent variables should be represented as two dictionaries, where the key:value pairs are `int`:`float`, respectively, that map the index of the variable to its corresponding value.

For our example, we would return the tuple
`(-5.2, {0: 1.6, 1: .2, 2: .6}, {3: 0, 4: 0})`.

Unit Test

There is a file called `test_simplex.py` that contains the following block of code as a unit test. There is a place for you to write your own unit tests for Problem 5, the simplex solver, which will be graded.

At this point, you should have a Simplex solver that is ready to use. The following code demonstrates how your solver is expected to behave:

```
>>> import SimplexSolver

# Initialize objective function and constraints.
>>> c = np.array([-3., -2.])
>>> b = np.array([2., 5, 7])
>>> A = np.array([[1., -1], [3, 1], [4, 3]])

# Instantiate the simplex solver, then solve the problem.
>>> solver = SimplexSolver(c, A, b)
>>> sol = solver.solve()
>>> print(sol)
(-5.2,
 {0: 1.6, 1: 0.2, 2: 0.6},
 {3: 0, 4: 0})
```

If the linear program were infeasible at the origin or unbounded, we would expect the solver to alert the user by raising an error.

Note that this simplex solver is *not* fully operational. It can't handle the case of infeasibility at the origin. This can be fixed by adding methods to your class that solve the *auxiliary problem*, that of finding an initial feasible dictionary when the problem is not feasible at the origin. Solving the auxiliary problem involves pivoting operations identical to those you have already implemented, so adding this functionality is not overly difficult.

The Product Mix Problem

We now use our Simplex implementation to solve the *product mix problem*, which in its dependent form can be expressed as a simple linear program. Suppose that a manufacturer makes n products using m different resources (labor, raw materials, machine time available, etc). The 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 6. Solve the product mix problem for the data contained in the file `productMix.npz`. In this problem, there are 4 products and 3 resources. The archive file, which you can load using the function `np.load`, contains a dictionary of arrays. The array with key `'A'` gives the resource coefficients $a_{i,j}$ (i.e. the (i, j)-th entry of the array give $a_{i,j}$). The array with key `'p'` gives the unit prices p_i. The array with key `'m'` gives the available resource units m_j. The array with key `'d'` gives the demand constraints d_i.

Return a 1-d numpy array of the number of units that should be produced for each product. (For `productMix.npz`, the function should return an array of length four). *Hint:* Because this is a maximization problem and your solver works with minimizations, you will need to change the sign of the array c.

Beyond Simplex

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [Nas00]. However, like any other algorithm, Simplex has its drawbacks.

In 1972, Victor Klee and George Minty Cube published a paper with several examples of worst-case polytopes for the Simplex algorithm [KM72]. In their paper, they give several examples of polytopes that the Simplex algorithm struggles to solve.

Consider the following linear program from Klee and Minty.

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

Klee and Minty show that for this example, the worst case scenario has exponential time complexity. With only n constraints and n variables, the simplex algorithm goes through 2^n iterations. This is because there are 2^n extreme points, and when starting at the point $x = 0$, the simplex algorithm goes through all of the extreme points before reaching the optimal point $(0, 0, \ldots, 0, 5^n)$. Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.

15 Reinforcement Learning 1: Gymnasium

Lab Objective: *Reinforcement learning is a topic found at the intersection between machine learning and control theory. Gymnasium is a module designed to learn and apply reinforcement learning. The purpose of this lab is to learn the variety of functionalities available in Gymnasium and to implement them in various environments to solve the reinforcement learning problem using trial and error and two model-free methods.*

Reinforcement learning, or RL for short, is a problem, a class of solutions that work well on the problem, and a field that studies the problem and solutions to it. As a problem, RL refers to the problem of getting an agent to explore an unknown environment to achieve a given task or goal. That is, we have an agent in some world with a given objective it must accomplish in that world but the agent is not told what to do at any moment (i.e. the agent only knows that it can do some things). You can think of this as baking a cake without a recipe and only being told you can open ingredients, mix them, and put them in the oven. Thus, you know that you can mix, open, or bake eggs and flour, but you don't know if you first need to mix the eggs and flour together, then open them, and bake them or if it needs to be done in some other order.

As a class of solutions, RL refers to the various algorithms or computations whose tasks is to help an agent learn, through sequential decision-making and experience, how to interact with and learn from its environment in order to accomplish the given goal in the most optimal way possible. It is different from other types of machine learning as reinforcement learning is concentrated more on goal-focused learning as a consequence of its interactions with the environment than other types of machine learning. Thus, RL is the umbrella referring to the field of study encompassing all this.

We will introduce a summary of the theory behind reinforcement learning prior to the actual beginning of the lab. Here are a few definitions to help you familiarize yourself with the verbiage used in RL (and by consequence in Gymnasium):

- An *agent*[1] is a learner and decision maker whose goal is to learn a strategy or sequence of actions to accomplish a given task or goal.

- The *environment* is the world in which the agent is located and consists of a set of different states. It is everything outside the control of the agent.

[1] In control theory, the agent is called the *controller*, the environment is the *plant* or *controlled system*, and the action is the *control signal*.

- The *state*, denoted by s, is the set of information that describes the environment completely so as to enable the agent to take an action. Thus, a state is the current representation of the environment. The *state space*, S, is the set of all possible states that the agent can be in, which includes the current state s and all possible future states the agent can reach from s.

- An *observation* is the information the agent gathers about the environment. Thus, the observation is the agent's perception or measurement of the state. In some cases, the observation may be a full measurement[2] of the state, but in other cases, the observation is a partial representation of the state.

- The *reward*, denoted by r[3] or by a function $r(s', a, s)$ or $r(s, a)$ of states and actions, is a real scalar value used to define the goal the agent must accomplish. The reward is used to help the agent know, in the immediate sense, how good or bad the action or sequence of actions that it took was in helping it accomplish the goal. The set of all possible rewards that the agent can receive is denoted $\mathcal{R}$.

- A *timestep* or *time-period*, t, is the smallest discrete unit of time where the agent interacts with the environment once. This interaction typically comprises a cycle of one state, one action, and one reward. An *episode* is a finite sequence of timesteps that starts at some *initial state*, s_0 (i.e. the state at $t = 0$), and ends at some *terminal state*, s_T. The terminal state is the final state representing the end of an episode and can be a maximum number of timesteps or some other desired state. Note that in this latter case, the time of termination, T, can be different for each episode as well as the fact that each episode can have a different terminal state. When, we work with episodes, we call this the *finite horizon*[4] setting and use S to denote the set of all non-terminal or normal states and S^+ to denote the set of all states, including the terminal states. We use $\mathbb{T} = \{0, 1, \ldots, T\}$ to denote the set of all time-steps, including the terminal time T, of one episode.

- A *value function* returns the *value* of a state or state-action pair as the expected future return of rewards. That is, the value function helps us determine the long-term desirability of a state or state-action pair after considering possible future states or state-action pairs that are likely to follow and the rewards that they will bring. Value functions estimate *how good* it is for the agent to be in a given state or to take an action in a given state, after considering the expected future rewards.
  - The *state-action value*, *quality of a state-action pair*, or *action-value function* for a policy π, denoted by $q_\pi(s, a)$ or $q(s, a)$, is a function of state-action pairs that returns the value or *quality* of a given pair as the expected return of taking action a while being in state s and following policy π thereafter. In other words, given a state s, an action a, and a policy π (i.e. a strategy to select an action based on a given state[5]), the value/quality of a state-action pair, $q(s, a)$, is the expected future return that the agent can receive by taking action a while being in state s at some period t and enacting π thereafter.

The RL Interaction-Learning Framework

In this section, we will assume that we are only working within one episode with a set $\mathbb{T} = \{0, 1, \ldots, T\}$ of timesteps.

[2] In this case, the observation space is the state space so that the observation is equal to the state.

[3] We talk more about the function case in the next RL lab.

[4] When there an infinite amount of timesteps, we no longer use the word episode. We call this the *infinite horizon* setting.

[5] We give a formal definition in the next RL lab.

As mentioned earlier, in RL, the goal is to get an agent to learn, through sequential decision-making, how to interact with its environment in order to accomplish some given objective (e.g. teach a robot how to walk, a computer learning how to play chess, or have two chatbots generate dialogue etc.). The agent interacts with the environment at each time-period $t \in \mathbb{T}$ by first receiving some observation of the current state s of the environment at the that timestep t, denoted s_t. Then based on this observation, the agent employs the mapping π to select[6] an action a to perform for that time-period, denoted a_t. One timestep later, $t+1$, the agent receives some feedback, a reward r_t, to help it know how good or bad the action a_t was in helping it accomplish the task. After this, the agent is now in the new state s_{t+1} and the process begins again. This same process of state, observation, action, and reward continues until we reach a terminal state[7] and is one of the main aspects separating RL from other types of machine learning. Note that the reward is received after the action is taken and when the agent is in a new timestep. This is illustrated in Figure 15.1.

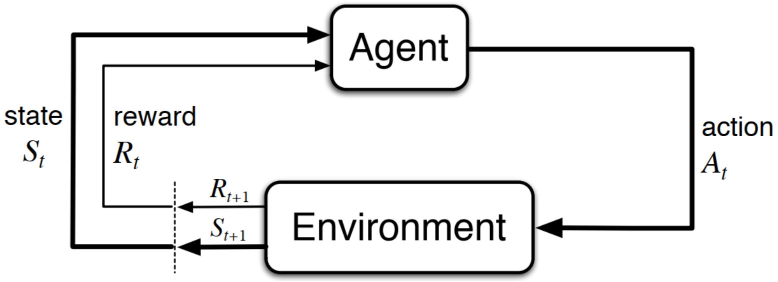


Figure 15.1: The reinforcement learning framework as given by [SB18]. Here, the reward is R and R_{t+1} is used to emphasize that the reward is received one time step later. Notice that the agent had received the reward R_t for taking action A_{t-1}. This was given prior to choosing action A_t during state S_t. The agent chooses the action A_t based on the observation of S_t and, to some extent, based on the reward R_t. The agent then goes to the next state S_{t+1}, receives the reward R_{t+1}, and the process repeats.

Hence, the agent interacts with the environment by taking actions and receiving rewards. These rewards are immediate feedback that the agent can use to determine how good an action is in helping it accomplish the task. In some environments, the reward follows after each action, called *dense rewards*, which can facilitate the problem. In other environments, the reward may be *sparsed* or delayed, perhaps until the end of the final timestep or until we fail/succeed, which makes the problem more challenging.

Reinforcement Learning Techniques

The ultimate goal, that is the optimization problem, of RL is to learn an optimal policy, or optimal strategy to select actions based on given states, that will bring the agent the most future reward, which in turn helps the agent accomplish the given task. There are several ways to solve the RL optimization problem. The major division in RL techniques is between *model-based* and *model-free* methods.

[6] In the stochastic/probabilistic case, the selection makes sense seeing there are multiple actions to choose from. In the deterministic case, the selection is equal to performing the only action available. This will be covered in the next RL lab.

[7] In the infinite horizon case, also known as *continuous*, the process continues indefinitely.

In model-based methods, the agent has a model of the environment, which includes a model for any transitions between states as well as the rewards. The agent makes use of this model to learn the optimal policy and updates the model as it interacts with the environment. One goal can be to learn the dynamics of the environment and then use this to learn the optimal policy. On the other hand, if we have certainty of the model and the dynamics, the agent can use this to evaluate future actions without having to try various actions and measuring the results of such actions as the model will help compute this. In contrast, model-free methods do not have a model of the environment. The agent learns by interacting with the environment and observing the rewards it receives rather than trying to learn the environment's dynamics or using a model to calculate future rewards. Thus, the agent learns by trial and error and creates a policy based on the consequences of its actions.

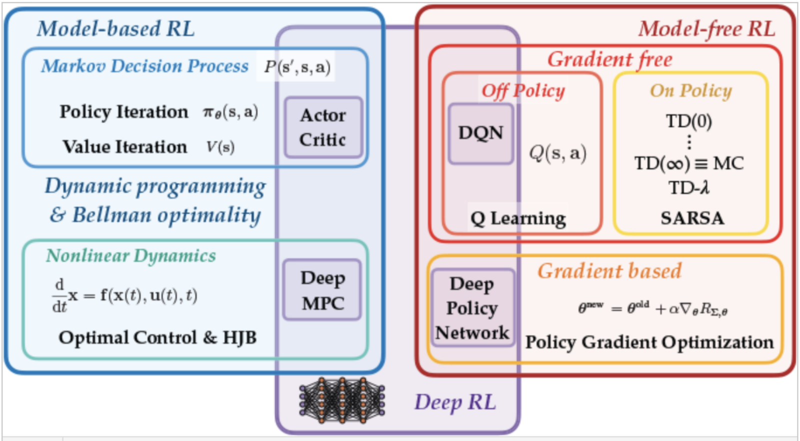


Figure 15.2: A diagram of various techniques used to solve the reinforcement learning optimization problem as given by [BK22]. You can see this video for a more in depth explanation.

The next RL lab, Reinforcement Learning 2: Markov Decision Process, is an example of a model-based method. We will employ an MDP model on the environment and use the Bellman equations to find the optimal policy, assuming we know the dynamics of the environment. In this lab, we will cover two model-free methods called *Q-learning* and *SARSA(0)*. Thus, the main goal of this lab is to find the optimal policy through trial and error, without having a model of the environment.

Gymnasium Module

Gymnasium is a module used to perform reinforcement learning. It contains a collection of environments where reinforcement learning can be used to accomplish various tasks. These environments include performing computer functions such as copy and paste, playing Atari video games, and controlling robots. To install Gymnasium, simply run the following code:

```
>>> pip install gymnasium
>>> # You may also need to install these dependencies
>>> pip install gymnasium[all]
>>> pip install gymnasium[classic-control]
```

Environments

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

```
>>> from gymnasium import envs
>>> print(envs.registry.values())
dict_values([EnvSpec(id='CartPole-v0', entry_point='gymnasium.envs.↩
    classic_control.cartpole:CartPoleEnv', reward_threshold=195.0, ↩
    nondeterministic=False, max_episode_steps=200, order_enforce=True, ↩
    autoreset=False, disable_env_checker=False, apply_api_compatibility=False, ↩
    kwargs={}, namespace=None, name='CartPole', version=0, additional_wrappers↩
    =(), vector_entry_point='gymnasium.envs.classic_control.cartpole:↩
    CartPoleVectorEnv'), ...
```

To learn more about Gymnasium and its environments, visit `gymnasium.farama.org`.

We will demonstrate how to work with Gymnasium environments by walking through the environment `"Blackjack-v1"`. The game Blackjack[8] is a card game where the player receives two cards from a face card deck. The goal of the player is to get cards whose sum is as close to 21 as possible without exceeding 21. In this version of Blackjack, an ace is considered 1 or 11 and any face card is considered 10. On each turn, the player may choose to take another card or stop drawing cards. If their card sum does not exceed 21, they may take another card, but if it does, they lose. After the player stops drawing cards, the computer may play the same game. If the computer gets closer to 21 than the player (without exceeding 21), the player loses.

To begin working in an environment, the environment must be initialized and reset. Resetting the environment puts everything in the correct starting position and is necessary to begin using the environment. For example, in `"Blackjack-v1"`, restarting the environment deals out a new game of Blackjack. Once the environment is complete, it should then be closed. Closing the environment tells the computer to stop running the environment (otherwise it will continue to run in the background).

```
>>> import gymnasium as gym
>>> env = gym.make('Blackjack-v1')  # Initialize Blackjack-v1 environment
>>> env.reset()  # Reset the environment
((16, 6, 1), {})

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

Action Space

An *action*, denoted by a, is the decision that the agent makes, while being in some state s, on what to do next. The set of all possible actions that the agent can enact during some state s is called the *action set* or *set of allowable actions*, denoted by A_s. The overall set of all possible actions that the agent can take is denoted $A = \cup_{s \in S} A_s$ and is called the *action space*. Note each state[9] $s \in S$ has an associated A_s.

[8] For more on how to play Blackjack, see `https://en.wikipedia.org/wiki/Blackjack`.

[9] This does imply that all action spaces of terminal states are empty since the episode has finished. This does not mean you cannot get a reward at any terminal state.

Once the environment is initialized and reset, the player can perform actions from the action space. To perform an action, use the function `step()`, which accepts the action as a parameter and returns an observation (more on those later). Environments may have discrete or continuous action spaces, but the environments presented in this lab all have discrete action spaces. When the action space is discrete, actions are defined as integers 0 through n, where n is the number of actions. The action each integer represents can be found in the documentation of each environment. The action space in `"Blackjack-v1"`[10] has 2 actions, represented by 0 and 1: 0 indicates that the player will stop drawing cards, and 1 indicates that the player will draw another card.

```
>>> env = gym.make('Blackjack-v1')
>>> env.reset() # Returns the initial state
((12,9,0),{})
>>> env.action_space  # Determine the number of actions available
Discrete(2)
# Select a random action and take a step using that action
>>> random_action = env.action_space.sample()
>>> random_action
1
# In this case, the random action was to draw another card
>>> env.step(random_action)
((16, 9, 0), 0.0, False, False, {})
```

Observation Space

The *observation space* of an environment contains all possible observations. For example, in `"Blackjack-v1"`, an observation is a tuple containing the total sum of the player's hand, the first card of the computer's hand, and a boolean indicating whether the player has an ace. The observation from each action can be found in the tuple returned by `step()`, which tells us the following information:

1. `observation`: The current measurement of the current state of the environment.

2. `reward`: The reward given from the observation. In most environments, maximizing the total reward increases performance. For example, the reward in `'Blackjack-v1'` is 1 if the player wins, -1 if the player loses, and 0 if there is a draw.

3. `terminated`: A boolean indicating whether the observation terminates the environment (i.e. a boolean indicating if the agent reaches the terminal state).

4. `truncated`: A boolean indicating whether the episode truncates/finishes for some reason other than having reached the terminal state. Typically, this is for a limit of timesteps.

5. `info`: Various information that may be helpful when debugging.

Consider the code below.

```
>>> env = gym.make('Blackjack-v1')
>>> env.reset()
((12, 1, 0),{})
```

[10] The documentation can be found here

```
>>> random_action = env.action_space.sample()  # Make a random guess
>>> env.step(random_action)
((18, 1, 0), 0.0, False, False {})
```

This tuple can be interpreted as follows:

1. The sum of the player's hand is 18, the computer's first card is 1, and the player has no ace.
2. The reward is currently 0.0 (the game is not over yet).
3. The environment is not terminated.
4. The episode was not truncated
5. Information that may help debugging (which is currently empty).

In practice, this information is usually accessed by setting variables equal to `step()` as in

```
>>> obs, reward, done, trunc, info = env.step(random_action)
```

Problem 1. Write a function `random_blackjack()` that accepts an integer n. Run and initialize `"Blackjack-v1"` a total of n episodes and in each episode take random actions until the game is terminated. Return the percentage of games the player wins. Use your function to print the win percentage after 50,000 episodes (i.e. after 50,000 games).

Understanding Environments

Because each action and observation space is made up of numbers, good documentation is imperative to understanding any given environment. Fortunately, most environments in Gymnasium are very well documented, and most documentation follows the same pattern. There is a docstring which includes a description of the environment, a detailed action space, a detailed observation space, and explanation of rewards. It is always helpful to refer to this documentation when working in a Gymnasium environment.

In addition to documentation, certain environments can be understood better through visualization. For example, the environment `"Acrobot-v1"` displays a double pendulum. Rendering the environment allows the user to see the movement of the double pendulum as forces are applied to it. The best way to render an environment in Gymnasium is by running the following code through a python (`.py`) script, using the argument `render_mode='human'`.

```
>>> import gymnasium as gym

>>> env = gym.make('Acrobot-v1', render_mode='human')
>>> # env.reset() returns the observation space and corresponding info
>>> observation, info = env.reset()

>>> done = False
>>> while not done:  # Until the environment terminates...
>>>     # Take random step
>>>     random_action = env.action_space.sample()
>>>     obs, reward, done, trunc, info = env.step(random_action)

>>> env.close()
```

However, this lab uses a Jupyter (.ipynb) file, and Gymnasium environments do not render well in Jupyter files. The visualization technique shown below is a simple workaround that uses the argument render_mode='rgb_array', but unfortunately it renders slowly.

```
>>> from IPython import display
>>> from matplotlib import pyplot as plt

>>> env = gym.make('Acrobot-v1', render_mode='rgb_array')
>>> observation, info = env.reset()

>>> # Initialize visualization
>>> img = plt.imshow(env.render())

>>> done = False
>>> while not done:
>>>     # Take random step
>>>     random_action = env.action_space.sample()
>>>     obs, reward, done, trunc, info = env.step(random_action)

>>>     # Update visualization
>>>     img.set_data(env.render())
>>>     display.display(plt.gcf())
>>>     display.clear_output(wait=True)

>>> env.close()
```

Figure 15.3: Rendering of "Acrobot-v1"

Solving An Environment

One way to solve an environment is to use information from the current observation to choose our next action. For example, consider `"Blackjack-v1"`. Each observation tells us the player's current card sum. Based on the current card sum, we can decide whether we want to draw another card or stop drawing cards. To take the decided action, simply input the integer representing the action into the function `step()`. In the next three problems, you will try to solve the environment using trial and error. Note that all of these environments are *episodic tasks*[11] since the episode ends after completing the task (or failing as well).

Problem 2. Write a function `blackjack()` which runs a naïve algorithm to win blackjack. The function should receive an integer `n` as input. If the player's hand is less than or equal to `n`, the player should draw another card. If the player's hand is more than `n`, they should stop playing. Within the function, run the algorithm for 10,000 episodes and return the percentage of games the player wins.

For $n = 1, 2, \ldots, 21$, plot the average win rate returned by your function. Identify which value(s) of n wins most often.

Hint: Remember what the actions are in the action space of `"Blackjack-v1"`.

Problem 3. The environment `"CartPole-v1"` presents a cart with a vertical pole. The goal of the environment is to keep the pole vertical as long as possible. The cart moves to the left with action 0, and it moves to the right with action 1. The observation space of this environment is a 4-dimensional array containing: the cart position, the cart velocity, the pole angle, and the pole angular velocity, respectively. More information about this environment can be found at `gymnasium.farama.org/environments/classic_control/cart_pole/`.

Write a function `cartpole()` which initializes the `"CartPole-v1"` environment and keeps the pole vertical as long as possible based on the angular velocity of the tip of the pole. Return the number of timesteps it takes before it terminates (about 200 on average).

Run the game for a single episode and render the environment at each timestep. Then run your function 100 episodes without rendering, and print the average number of timesteps before it terminates.

Problem 4. The environment `"MountainCar-v0"` shows a car in a valley. The goal of the environment is to get the car to the top of the right mountain. The car can be driven forward (toward the goal) with action 2, can be driven backward with action 0, and will be put in neutral with action 1. Note that the car cannot immediately get up the hill because of gravity, so in order to move the car to goal, momentum will need to be gained by going back and forth between both sides of the valley. The observation space of this environment is a 2-dimensional array containing the x position and the velocity of the car, respectively. More information about this environment can be found at `gymnasium.farama.org/environments/classic_control/mountain_car/`.

[11] When working in the continuous or infinite horizon case, we call the task *continuous task*.

Using the given position and velocity of the car, write a function `car()` that solves the `"MountainCar-v0"` environment. Return the number of time-periods it takes before it terminates, which should be less than 180.

Run the game for a single episode and render the environment at each time-period. Then run your function for 100 episodes without rendering, and print the average number of time-periods before it terminates.

Model-Free Methods

While naïve methods like the ones we have used above can be useful, they do not help in producing an optimal policy to accomplish the given task. Model-free methods have the advantage of learning straight from experience and without a need for a model and then use this knowledge to find an optimal policy. To learn from experience, the model has to seek out actions it has not yet tried in order to find the most optimal action (what is called *exploration*). Exploration can improve our current knowledge of the environment so that we can obtain better rewards in the long run. However, to find an optimal policy, the model also has to chose from actions, typically the most optimal action, it has tried in the past and found to be effective in producing rewards (this being called *exploitation*.) Thus, a model has to balance between exploring new actions and exploiting the best actions it has found so far, a term known as the *exploration-exploitation trade-off*.

Moreover, since we have to learn from experience, we must often run the model through numerous episodes or samples of various cycles of state-action-reward to be able to have sufficient information in order to learn. To help with this latter case, we typically want to run the method for a rather large number $N \in \mathbb{N}$ of episodes. That is, we want to iterate the method for N iterations for however many timesteps there are in each iteration.

To Explore or Not to Explore?

To help balance the trade-off between exploration vs. exploitation, we can employ the *epsilon-greedy* or *ε-greedy* algorithm/policy. The ε-greedy algorithm is a simple method that helps the agent decide randomly whether to explore by taking a random action or exploit its knowledge and take the optimal action. The value $\varepsilon \in [0, 1]$ represents the agent's willingness to explore (i.e. how often we want the agent to explore), so that as $\varepsilon \to 1$, the agent cares more about exploring. Whereas, $\varepsilon \to 0$ signifies that the agent cares more about exploitation and taking the optimal action.

In the simplest implementation of the algorithm, the agent defines some constant probability $\varepsilon \in [0, 1]$ where it will act randomly and explore various actions in order to learn more about the environment through trial and error. With probability $1 - \varepsilon$, the agent will exploit the current policy it has formed and take the best action given by that policy. Thus, in this simple case and at each timestep in a given episode, we can draw some value from the standard uniform distribution and compare it to ε. If the drawn value is less than ε, we take a random action. Otherwise, we exploit our knowledge and take the best action.

The difficulty with always using the constant value ε for each timestep and for each episode is that at the beginning we do not have sufficient information to exploit so that exploring is the better option. However, as we gain more and more information as we cycle through various episodes, the agent does not have to explore as much since the model is more robust and better able to predict an optimal policy. But even in this latter case, the model could use some fine-tuning with some random exploration. Hence, we may want to use a decaying epsilon for the ε-greedy policy.

One way to employ a decaying epsilon is to use a linear decay. We can define $\delta = \frac{\varepsilon_1 - \varepsilon_N}{N}, \varepsilon_1 > \varepsilon_N$, where ε_1 signifies the starting value of epsilon in the first episode and ε_N signifies the ending value of epsilon we want at the last episode N. ε_1 should be a value that is more biased towards exploration, so either $\varepsilon_1 = 1$ or it's a value quite close to it. In contrast, ε_N should be more biased toward exploitation, so that it takes a value closer to 0, if not actually equal to 0. Then for each episode $i \in \{1, 2 \dots, N\}$, we can update the epsilon value of a given episode i with $\varepsilon_i = \varepsilon_1 - (i-1)\delta$. We can then use each ε_i as the epsilon value for each episode and, at each timestep in an episode, compare it to a randomly drawn value from the standard uniform distribution to determine whether to explore or exploit. We give a few more models of a decaying epsilon in the Additional Materials section.

The advantage of using the epsilon-greedy algorithm is that it is simple to implement and can be used in a variety of environments. This helps us balance the trade-off between exploration and exploitation and can help us find an optimal policy. However, the epsilon-greedy algorithm is not always the best method to use, and there are other methods that can be used to help balance the trade-off between exploration and exploitation. With a constant epsilon, the agent may not explore enough at the beginning and may not exploit enough at the end. With a decaying epsilon, the agent may explore too much at the beginning and exploit too much at the end since the value is so low. Moreover, the ε-greedy policy chooses actions uniformly at random, so that the worst possible action has the same probability of being chosen as the best possible action.

Problem 5. Write a function `epsilon_decay()` that accepts an integer `episode` signifying the number of the current episode, an integer `N` that defines the total number of episodes, a float `epsilon_start` that defaults to 1.0, and a float `epsilon_end` that defaults to 1e-6. Return the epsilon value for any given episode number using a linear decay.

To test your function, you can plot the epsilon value for each episode for $N = 1000$ and ensure the graph is a linear graph sloping downward to 0 starting from 1.

Temporal Difference Learning

Q-learning and SARSA(0) are model-free methods that come from the concept of *temporal-difference learning*, TD for short. Like other methods, TD learns directly from experience and does not require a model of the environment, but it also updates its estimates of the value function based in part on other estimates it has learned, without having to wait for a final outcome. Temporal difference learning is a method that uses a reward and the difference between the value of the current state and the value of the next state or states to update the value of the current state. In short, TD focuses on the differences the agent experiences in time. The main goal of TD is to learn the value function of the environment through an approximation.

The simplest equation for a TD method for some given estimate value function[12], denoted by V, is given by

$$V(s_t) = V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)), \tag{15.1}$$

[12] We give a formal definition in the next Reinforcement Learning lab.

where $r_t + \gamma V(s_{t+1})$ is the *TD-target* and is an approximation of the value of the current state that we want to approximate. The expression $r_t + \gamma V(s_{t+1}) - V(s_t)$ is called the *TD-error* and is the difference between the value of the current state, the TD-target, and the value of the next state. Notice that all of the terms in Equation 15.1 are estimates of the true value function[13]. Equation 15.1 is known as the *TD(0)*[14] or the *one-step* TD method since it updates the value of the current state at timestep t just after one timestep (i.e. in timestep $t+1$). The idea of TD(0) is the basis for the Q-learning and SARSA(0) algorithms.

Q-learning

Q-learning is a model-free reinforcement learning algorithm that uses the idea of TD to approximate the optimal action-value function $q_*(s,a)$. Q-learning creates a Q-table, which is an $n \times m$ dimensional array, a lookup table, where n is the number of observations or states and m is the number of actions. Each row in the Q-table represents a state, each column represents an action, and each cell stores the approximated value $Q(s,a)$ of taking that action in that state (i.e. the value of a state-action pair). The main idea in Q-learning is that the quality of the current state-action pair is not only based on the reward of the current state-action pair, but also on the difference between the maximum value of the next state-action pairs, for the next state, and the old approximated value of the current state-action pair. The Q-table is initialized with zeros and is updated using the following formula:

$$\begin{aligned} Q_{\text{new}}(s_t,a_t) &= Q_{\text{old}}(s_t,a_t) + \alpha \left[r_t + \gamma \max_{a \in A_{s_{t+1}}} (Q(s_{t+1},a)) - Q_{\text{old}}(s_t,a_t) \right] \qquad (15.2) \\ &= (1-\alpha) Q_{\text{old}}(s_t,a_t) + \alpha \left[r_t + \gamma \max_{a \in A_{s_{t+1}}} Q(s_{t+1},a) \right]. \end{aligned}$$

The expression $r_t + \gamma \max_{a \in A_{s_{t+1}}} (Q(s_{t+1},a))$ is the TD-target. The overall expression in the brackets of Equation 15.2 (not the second equation below) is the TD-error. Thus, we update the Q-table by taking a step in the direction of the TD-error and adding it to the old approximated value of the current state-action pair. This process is repeated for each state-action pair until the Q-table converges to the optimal policy.

The formula requires 3 hyperparameters[15]:

1. `alpha`: the *learning rate* is a value between $(0,1]$ tells the model the magnitude of the step it should take towards the solution. It influences to what extent newly acquired information overrides old learned information. As $\alpha \to 1$, the more the agent will consider the new information.

2. `gamma`: the *discount factor* is a value in the interval $[0,1]$ that determines how important the reward of the current action is compared to future rewards. As $\gamma \to 1$, the more the agent will consider the future rewards.

3. `epsilon`: the epsilon value for the epsilon-greedy algorithm.

[13] This method of using estimates of current value we want to approximate during the update step is called *bootstrapping.*

[14] This is a special case of TD(λ) and n-step TD methods.

[15] A hyperparameter is not the same as a model parameter. A hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data. It specifies how the learning process should behave and is used to find model parameters.

For each state, the optimal action is the action that maximizes the value in the Q-table. Thus, to find the optimal policy for a given state, we need only take the argmax of the row in the Q-table that corresponds to the given state. The following function will generate the optimal Q-table for a given gym-environment. **Note**: read through the code and comments to understand how to implement Q-learning because you will have to implement the SARSA(0) algorithm in problem 7.

```python
def qlearn(env, alpha=0.1, gamma=0.6, epsilon=0.1, N=70_000, decay=False):
    """ Use the Q-learning algorithm to find qvalues.
    Parameters:
        env (str): environment name (Gymnasium environment)
        alpha (float): learning rate
        gamma (float): discount factor
        epsilon (float): epsilon value for epsilon-greedy algo
        N (int): number of episodes to train for
        decay (bool): whether to decay epsilon according to epsilon_decay
    Returns:
        q_table (ndarray nxm): the Q(s,a) approximation values
    """

    # Make environment
    env=gym.make(env)
    # Make Q-table
    q_table=np.zeros((env.observation_space.n, env.action_space.n))

    # Train for N episodes
    for i in range(1,N+1):

        # Get epsilon value
        if not decay:
            epsilon=epsilon
        else:
            epsilon=epsilon_decay(i, N)

        # Reset env and get initial state; Initialize penalties, reward, done
        curr_state, info=env.reset()
        penalties, reward=0,0
        done=False

        # Keep going until the terminal state is reached
        while not done:

            # Employ epsilon-greedy algo
            if random.uniform(0,1) < epsilon: # Explore
                curr_action=env.action_space.sample()
            else:                             # Exploit
                curr_action=(q_table[curr_state]).argmax()

            # Take action and get new state and reward
            next_state, reward, done, truncated, info=env.step(curr_action)
            # Calculate new qvalue
            old_value=q_table[curr_state,curr_action]
            next_max=(q_table[next_state]).max()
            new_value=(1-alpha)*old_value+alpha*(reward+gamma*next_max)
            q_table[curr_state, curr_action]=new_value

            # Check if penalty is made; specific to Taxi env
            if reward == -10:
                penalties+=1

            # Get next observation
            curr_state=next_state

        # Print episode number
        if i % 100 == 0:
            display.clear_output(wait=True)
            print("Training model...")
            print(f"Episode: {i}")

    env.close()
    print("Training finished.")
    return q_table, penalties
```

Notice how we look at penalties in the code. The purpose of this is to see how many times the agent makes a mistake. We typically expect the agent to make mistakes at the beginning of the training process, but as the agent learns, we expect the number of mistakes to decrease. If the agent is making a lot of mistakes, it may be a sign that the agent is not learning well.

Problem 6. The environment "Taxi-v3" depicts a taxi on a city grid, as shown in Figure 15.4. The goal of this environment is to pick up a passenger in a taxi and drop them off at their destination as fast as possible. You will have to look at the environment specifications at gymnasium.farama.org/environments/toy_text/taxi/ to understand it better.

1. Initialize the environment, then randomly act until the environment is done (i.e. until the end of one episode) and print the total reward. Since the taxi is acting randomly, it often takes so long for the environment to render that Jupyter will crash, so do NOT attempt to render this environment.

2. Next, use qlearn() to calculate the optimal Q-table of the environment using the default values for the hyperparameters as given and the given default value for N. Do NOT decay the epsilon value. Then, render "Taxi-v3", use the Q-table to move through it (as described above) for one episode, and print the total reward. Hint: q_table[observation ,:] might be helpful. Note that the training time, for the default N, should take about no more than 3 minutes (depending on your computer) to complete.

3. Third, use `qlearn()` to calculate the optimal Q-table of the environment using the default values for α, γ, and `N`, but this time decay the epsilon value. Then, render `"Taxi-v3"`, use the Q-table to move through it for one episode, and print the total reward. For the default N, the training time should take about 2-6 minutes (depending on your computer) to complete. You will want to create a separate Q-table for this scenario.

4. Finally, write a function `taxi()` which initializes the `"Taxi-v3"` environment (without rendering). Run scenarios 2 and 3 as described above for 1000 episodes, and write 2-3 sentences comparing the use of a decaying epsilon value to a constant epsilon value. You may want to look at average Q-learning total reward of 1000 episodes returned by each scenario. You can also compare the penalties made in each scenario as given by the `qlearn()` function as well as look at the time it took to train each scenario (i.e. the time it took to create the Q-table). For the time, pay attention to the print statement that gives the episode number and consider what epsilon is doing as you go through the episodes.

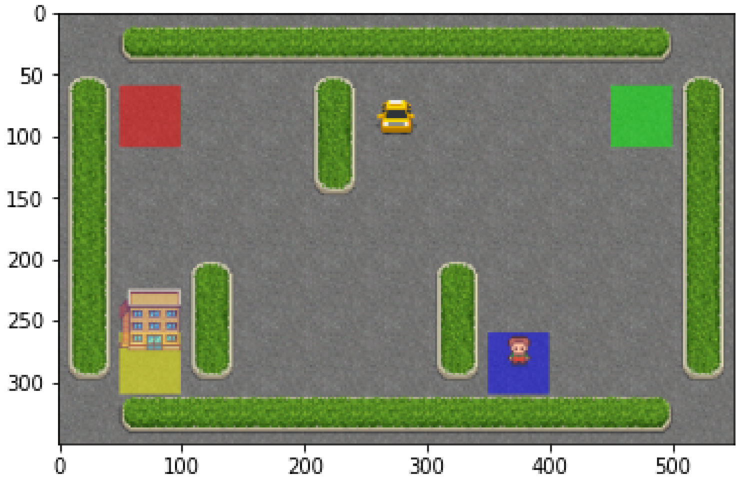


Figure 15.4: Example starting position of `"Taxi-v3"`

SARSA(0)

The main idea behind *SARSA(0)*, or just *SARSA*, is similar to Q-learning in that it approximates the optimal action-value function $q_*(s, a)$ using a Q-table. The difference between SARSA and Q-learning is that SARSA uses the quality of the next state-action pair to update the value of the current state-action pair, whereas Q-learning uses the maximum quality available in the state-action pairs of the next state to update the value of the current state-action pair. Thus, the equation for SARSA is given by

$$\begin{aligned} Q_{\text{new}}(s_t, a_t) &= Q_{\text{old}}(s_t, a_t) + \alpha \left[r_t + \gamma Q_{\text{old}}(s_{t+1}, a_{t+1}) - Q_{\text{old}}(s_t, a_t) \right] \\ &= (1-\alpha) Q_{\text{old}}(s_t, a_t) + \alpha \left[r_t + \gamma Q_{\text{old}}(s_{t+1}, a_{t+1}) \right], \end{aligned} \tag{15.3}$$

where $r_t + \gamma Q_{\text{old}}(s_{t+1}, a_{t+1})$ is the TD-target and the expression in the brackets of Equation 15.3 (not the second equation below) is the TD-error. Thus, the agent estimates the value of the current state-action pair by receiving the reward of the current state-action pair and then calculating an estimate of the value next state-action pair. This is where the name SARSA comes from as the agent uses the quintuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ to update the value of the current state-action pair using estimates of the next state-action pair.

The following pseudocode and the code block of Q-learning will help you implement the SARSA(0) algorithm. Arrows with the tail end having the word `fill` signify that the given variable is assigned a value that you must fill in. We give you some variables that may be harder to understand from just reading the `qlearn()` codeblock.

Algorithm 1 SARSA(0) Algorithm

```
procedure SARSA0(env, alpha= 0.1, gamma= 0.6, epsilon= 0.1, N= 70_000, decay=False)
    env←fill                                        ▷ Make gym env
    sarsa_tab←fill                                  ▷ Initialize Q-table (given by SARSA)
    for i in range(1,N+1) do                        ▷ Train
        if not decay then                           ▷ Get epsilon value
            epsilon←epsilon
        else
            epsilon←epsilon_decay(i, N)
        curr_s, info←fill                           ▷ Reset env and get current state s
        penalties, reward, done←fill                ▷ Initialize penalties, reward, done
        if random.uniform(0,1) < epsilon then       ▷ Get current action using epsilon-greedy algo
            curr_act←fill
        else
            curr_act←fill
        while not done do
            next_s, reward, done, trunc, info←fill  ▷ Take current action, get new state, and reward
            if random.uniform(0,1) < epsilon then   ▷ Get next action using epsilon-greedy algo
                next_act←fill
            else
                next_act←fill
            old_val←fill                            ▷ Get the value of the current state and action
            next_val←sarsa_tab[next_s, next_act]    ▷ Get the value of the next state and action
            new_val←fill                            ▷ Calculate the new value of the current state and action
            sarsa_tab[curr_s, curr_act]←new_val     ▷ Update the Q-table
            if reward == -10 then                   ▷ Check for penalty according to Taxi env
                penalties+ = 1
            curr_s←next_s                           ▷ Update the current state
            curr_act←next_act                       ▷ Update the current action
    env.close()
    return sarsa_tab, penalties
```

Before concluding the lab and moving on to the last problem, do note that both Q-learning and SARSA(0) can suffer from what is called *maximization bias*. However, we leave this off to the Additionals Materials section as it necessitates another topic we briefly touch on in the conclusion. We strongly recommend reading this section to help you know potential problems with Q-learning and SARSA(0) and how to fix them when employing these algorithms in practice.

Wrapping Up

The type of reinforcement learning we have worked with in Q-learning is called *off-policy learning*. This is a very important concept in RL, but we save it to the Additional Materials section were we discuss it in more detail along *on-policy learning*, SARSA being an example. This will help us understand the difference between the two types of learning and why we use them in different scenarios. We strongly recommend reading this section to understand the difference between off-policy and on-policy learning.

Moreover, we have worked with *online* reinforcement learning, where the agent learns from the environment in real-time. That is, the agent interacts with the environment and learns from the environment as it goes, so it can keep gathering data as it continues to interact. This is in contrast to *offline* reinforcement learning, where the agent learns from a dataset of experiences that someone has collected from the environment so that the agent does not interact with the environment in real-time and cannot gather anymore data than the one given.

Lastly, this whole process of running the algorithm for N episodes is what in machine learning is called *training* the model. Training the model is the process of feeding the model data and allowing it to learn from that data. The objective of training is to learn the proper model parameters that will allow it to increase its performance in the given task. In our case for model-free RL, training the model has the objective of exploring and exploiting the environment to learn the optimal policy.

Problem 7. You will perform the following and use the `"Taxi-v3"` environment for the following problem:

1. Implement the `sarsa0()` function as given in Algorithm 1. You will need 15.3 to help you calculate the new value of the current state-action pair as well as the given code block of Q-learning to help you implement the function. You do not need to have a section that prints out the episode number or the completion of training as in the Q-learning function, but you can if you want. Do make sure to close the environment at the end of the function. We have already given you the penalty check for the Taxi environment.

2. Then, use `sarsa0()` to calculate the optimal Q-table of the environment using the default values for the hyperparameters as given and the given default value for `N`. Employ a decaying epsilon value to calculate the Q-table. Render `"Taxi-v3"`, use the Q-table to move through it for one episode, and print the total reward. You will want to store this Q-table in a separate variable. The training time for the default N should be about the same as the training time for the Q-learning function with a decaying epsilon value.

3. Finally, write a function `compare()` which initializes the `"Taxi-v3"` environment (without rendering). This function will compare the average total reward of 1000 episodes of the Q-learning and SARSA(0) tables that were created using a decaying epsilon. Run scenarios the Taxi environment for 1000 episodes for each method using the tables you have now created. Return the average total reward for 1000 episodes for each algorithm. This is a tuple of two floats.

Additional Materials

More on the Epsilon-Greedy Algorithm

We talked about the difficulty of the trade-off between exploration and exploitation and how using the ε-greedy algorithm can help with this. Using the epsilon-greedy algorithm is a simple way to balance the trade-off between exploration and exploitation, but it is not the only way. Thompson sampling (taught in Volume 2 Chapter 17), Upper confidence bound (UCB), and Softmax Action Selection are other exploring strategies that can be used to handle the exploration-exploitation trade-off. [SB18] makes a brief mention of these strategies, but they are not covered in depth. However, there are plenty of free articles and blogs that cover these strategies in depth.

Rather than limiting ourselves to a constant epsilon value or a linear decay, we can use an exponential decay for ε. Using ε_1 as the starting value of epsilon and ε_N as the ending value of epsilon, we can define the epsilon value for each episode $i \in \{1, \ldots, N\}$ as $\varepsilon_1 * \lambda, \lambda = (\frac{\varepsilon_N}{\varepsilon_1})^{\frac{i}{N}}$. We could multiply the episode i in the numerator[16] of λ by β to control the steepness of the decay, but be careful as this can affect the value for ε_N.

Problem 8. Update the function `epsilon_decay()` as given in problem 5 to now include in the parameters a string `decay_type` that defaults to `"linear"` that allows the user to choose between a linear and exponential decay. Use `"exp"` for exponential decay. Then, update the `qlearn()` function or `sarsa0()` function (below) to also include the same parameter `decay_type`. Test the function with both linear and exponential decay. You could also try other exploration strategies like Thompson sampling, UCB, and Softmax Action Selection and compare them to the `epsilon_decay()` function.

Expected SARSA

We give a full definition of the policy π in the next lab. For now, to be more detailed, π takes in a state s and returns a probability of taking an action $a \in A_s$.

Like Q-learning and SARSA(0), *Expected SARSA* is a model-free reinforcement learning algorithm that approximates the optimal action-value function $q_*(s, a)$. Specifically, Expected SARSA uses the expected value of the next state-action pairs to update the value of the current state-action pair. That is, we take into account how likely we are to take a certain action in the next state under the current policy and use that to update the value of the current state-action pair. Thus, we move in the direction of the expectation of the next state-action pairs (hence the name Expected SARSA).

[16] There is no uniqueness of using β in the numerator as the same effect can be had by using it in the numerator.

The formula for Expected SARSA is given by

$$\begin{aligned} Q_{\text{new}}(s_t, a_t) &= Q_{\text{old}}(s_t, a_t) + \alpha\left[r_t + \gamma\mathbb{E}\big[Q_{\text{old}}(s_{t+1}, a_{t+1})\big|s_{t+1}\big] - Q_{\text{old}}(s_t, a_t)\right] \\ &= Q_{\text{old}}(s_t, a_t) + \alpha\left[r_t + \gamma\sum_{a\in A_{s_{t+1}}} \pi(a|s_{t+1})Q(s_{t+1}, a) - Q_{\text{old}}(s_t, a_t)\right]. \end{aligned}$$

In the case of a deterministic policy, the expected value of the next state-action pairs is the same as before. When we do have a stochastic policy, we can use list comprehension to calculate the expected value of the next state-action pairs. Note that Expected SARSA is an on-policy algorithm like SARSA. We leave to the reader to determine which is the TD-target and the TD-error in the above equation.

Q-learning vs. SARSA vs. Expected SARSA

Both Q-learning and SARSA are model-free reinforcement learning algorithms that approximate the optimal action-value function. One of the main differences is that Q-learning uses the maximum quality of the next state-action pair to update the value of the current state-action pair, whereas SARSA uses the quality of the next state-action pair to update the value of the current state-action pair. Thus, since Q-learning uses the best possible next action, it generally converges to the optimal policy faster than SARSA because SARSA could still take an exploratory action. Moreover, SARSA typically tends to lead to more stable solutions than Q-learning as well yielding better cumulative rewards. Both are great model-free algorithms to use.

SARSA gets the next action stochastically by, at least in our code, using the epsilon-greedy policy. This creates variance in our estimates of the value of the state-action pairs. While Expected SARSA is computationally more expensive than SARSA, it is more stable and can yield better cumulative rewards since it eliminates the variance found in SARSA. However, this is occurs when the policy is highly stochastic. In most cases, Expected SARSA is only slightly better than SARSA.

To implement this algorithm, employ the same pseudocode as given in Algorithm 1 and the given code block of Q-learning. Then just replace the equation for the new value of the current state-action pair with the Expected SARSA equation using list comprehension to calculate the expected value of the next state-action pairs. You will need to add in the parameter `policy` to the function to be able to obtain the probability of taking an action in the next state under the current policy.

On-Policy vs. Off-Policy Learning

In model-free reinforcement learning, there are two types of learning: *on-policy learning* and *off-policy learning*. The policy that the agent uses to determine its action/behavior as a response to its environment is called the *behavior policy*. Whereas, the policy that the agent uses to learn from the rewards of the actions taken and becomes the optimal policy is called the *target policy* (i.e. the policy used to update the qualities/values of state-action pairs). In *on-policy learning*, the behavior policy and the target policy are the same or similar. Whereas, in *off-policy learning*, the behavior policy and the target policy are different.

The ultimate goal is to learn the optimal policy, denoted π_*, but the way we learn π_* can be different. In on-policy learning, the agent uses a common behavior and target policy to respond to the environment and then learn from the rewards of the actions taken in order to optimize the current values of the state-action pairs so that they converge to the optimal ones. SARSA is an example of an on-policy learning algorithm. In the TD-target expression of Equation 15.3, the policy used to take actions in the environment is similar to the policy used to update the value of the state-action pairs. In our case, we used the epsilon-greedy algorithm/policy to determine the current action and the next action. Even if we used a different method for how to choose an action, say UCB or Thompson sampling, the policy used to take actions in the environment would still be the same as the policy used to update the value of the state-action pairs. You cannot use a different policy to take actions in the environment than the one you use to update the value of the state-action pairs because that will change the nature of SARSA so that it may not converge to the optimal policy.

Comparing this to Q-learning, an off-policy learning algorithm, the behavior policy and the target policy are different. In this type of learning, the agent uses the behavior policy to take actions and then uses the target policy to update the value of the state-action pairs so that it converges to π_*. In the TD-target expression of Equation 15.2, the policy the agent uses to explore or take actions in the environment, at least in our code, is the epsilon-greedy policy, but the policy the agent uses to update the value of the state-action pairs always chooses the action that maximizes the value of the next state-action pairs. Thus, regardless of whatever the behavior policy tells us to do (i.e. explore or exploit according to the epsilon-greedy algorithm), the target policy will always choose the best action to update the value of the state-action pairs. Hence, we have two different policies.

Overall, on-policy learning is a type of learning where the agent uses the same policy to take actions in the environment and to then update its estimates of a value function that will converge to the optimal value function. In contrast, off-policy learning is a type of learning where the agent uses a one policy to take actions in the environment and another policy to then update its estimates of a value function that will go to the optimal value function.

Maximization Bias & Double Q-Learning

This section is a direct connection to the previous section comparing and contrasting Q-learning, Expected SARSA, and SARSA(0). But we first had to introduce the vocabulary of on-policy and off-policy learning to understand the bias that can occur in both.

All of the given model-free methods involve maximization in the construction of their target policies. There would be no problem if we did not have to approximate the value of the state-action pairs using other estimates of the value of the state-action pairs. But the fact that we do, it leads to overestimation of the value of the state-action pairs as we take the maximum value. It is the choice of estimator, the maximum value, that leads to the overestimation of the value of the current state-action pair since it is tied to its own value. That is, we are using an estimate of the value of the next state-action pair to get an estimate of its maximum value over all possible actions in the action space to update the value of the current state-action pair. Thus, we are using the same sample to both determine the maximizing action and to estimate its value.

One way to mitigate the maximization bias is to use *Double Q-learning.* Double Q-learning is an improvement to the normal Q-learning ideas of SARSA(0), Expected SARSA, and Q-learning. The main idea is that rather than using the same sample to determine the maximizing action and to estimate its value, we use two different samples to determine the maximizing action and to estimate its value. This can be applied to Q-learning, SARSA(0), and Expected SARSA, but we will only talk about implementing this idea in Q-learning. The reader should be able to implement this idea in SARSA(0) and Expected SARSA as well.

The process of Double Q-learning is simple. We take two Q-tables, Q_1 and Q_2, and use one to determine the maximizing action and the other to estimate its value. We can use, for example, Q_1 to determine the maximizing action $a_* = \underset{a \in A_{s_{t+1}}}{\operatorname{argmax}} Q_1(s_{t+1}, a)$ and then use Q_2 to estimate its value $Q_2(s_{t+1}, a_*)$. We can then update the value of the current state-action pair using the following formula:

$$\begin{aligned} Q_1(s_t, a_t) &= Q_1(s_t, a_t) + \alpha \left[r_t + \gamma Q_2(s_{t+1}, \underset{a \in A_{s_{t+1}}}{\operatorname{argmax}} \{Q_1(s_{t+1}, a)\}) - Q_1(s_t, a_t) \right] \\ &= Q_1(s_t, a_t) + \alpha \left[r_t + \gamma Q_2(s_{t+1}, a_*) - Q_1(s_t, a_t) \right]. \end{aligned}$$

This role can be reversed where we use Q_2 to determine the maximizing action and Q_1 to estimate its value.

For implementation, we can divide the alternating between the two Q-tables by a random number generator that chooses between the two Q-tables by simulating a coin flip (i.e. a 50-50 chance). This way, we can alternate between the two Q-tables to determine the maximizing action and to estimate its value. There is nothing special about the 50-50 chance, but it is a simple way to alternate between the two Q-tables. Moreover, we can have the behavioral policy utilize the two Q-tables or just one of them. If using two, we can use an average of the two or a sum. Again, there is nothing special about the average or sum, but it is a simple way to combine the two Q-tables. We can implement as follows (note we chose an action over the sum of the two Q-tables):

Algorithm 2 DoubleQ Learning Algorithm

```
procedure DoubleQLearn(env, alpha= 0.1, gamma= 0.6, epsilon= 0.1, N= 70_000, decay=False)
    env←gym.make(env)
    q1← np.zeros((env.observation_space.n,env.action_space.n))          ▷ Initialize Q-tables
    q2← np.zeros((env.observation_space.n,env.action_space.n))
    for i in range(1,N+1) do                                             ▷ Train
        if not decay then                                                ▷ Get epsilon value
            epsilon←epsilon
        else
            epsilon←epsilon_decay(i, N)
        curr_state, info←env.reset()                       ▷ Reset env and get current state
        done←False
        while not done do
            if random.uniform(0,1) < epsilon then ▷ Get current action using epsilon-greedy algo
                curr_action←env.action_space.sample()
            else
                curr_action←(q1[curr_state]+q2[curr_state]).argmax()
            next_state, reward, done, trunc, info←env.step(curr_action)
            if random.uniform(0,1) < 0.5 then                            ▷ Update q1
                max_act←(q1[next_state]).argmax()
                qval←q1[curr_state,curr_action]
                next_qval←q2[next_state,max_act]
                q1[curr_state,curr_action]←qval+alpha*(reward+gamma*next_qval-qval)
            else                                                         ▷ Update q2
                max_act←(q2[next_state]).argmax()
                qval←q1[curr_state,curr_action]
                next_qval←q1[next_state,max_act]
                q2[curr_state,curr_action]←qval+alpha*(reward+gamma*next_qval-qval)
            curr_state←next_state
    return q1, q2
```

This double Q-learning algorithm is a great way to mitigate the maximization bias that can occur in Q-learning, SARSA(0), and Expected SARSA. The above given algorithm can be modified to work with SARSA(0) and Expected SARSA as well. Note this does increase the spatial complexity of the algorithm as we now have two Q-tables to keep track of. But, we are only using one Q-table to determine the maximizing action and the other to estimate its value so that temporal complexity is not affected. While we have mitigated the maximization bias, it can be shown double Q-learning can lead to underestimation of the value of the state-action pairs. But, this at least is better than overestimation.

16 CVXPY

Lab Objective: *CVXPY is a package of Python functions and classes designed for the purpose of convex optimization. In this lab we use these tools for linear and quadratic programming. We will solve various optimization problems using CVXPY and optimize eating healthily on a budget.*

Linear Programs

A *linear program* is a linear constrained optimization problem. Such a problem can be stated in several different forms, one of which is

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

The symbol $\preceq$ denotes that the components of $G\mathbf{x}$ are less than the components of $\mathbf{h}$. In other words, if $\mathbf{x} \preceq \mathbf{y}$, then $x_i < y_i$ for all $x_i \in \mathbf{x}$ and $y_i \in \mathbf{y}$. CVXPY accepts $\leq, \geq$, and $=$ in its constraints as long as the equations satisfy convexity requirements described later in this chapter, so we can reformulate this problem in yet another form:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{subject to} \quad & G\mathbf{x} \preceq \mathbf{h} \\ & P\mathbf{x} \succeq \mathbf{q} \\ & A\mathbf{x} = \mathbf{b}. \end{aligned}$$

CVXPY accepts NumPy arrays and SciPy sparse matrices for the constraints, but the variable $\mathbf{x}$ must be a CVXPY `Variable`.

Consider the following example:

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

We can solve this problem using the following code. Note that `cvxpy.Problem()` accepts the constraints as a single list and that >= represents both standard and elementwise greater than or equal to. The symbols <= and == are similarly versatile.

```python
>>> import cvxpy as cp
>>> import numpy as np

# First we'll initialize the objective
# We can declare x with its size and sign
# nonneg = True is equivalent to the constraint P@ x >= 0 listed below
# Both are included for demonstration but are redundant
>>> x = cp.Variable(2, nonneg = True)
>>> c = np.array([-4, -5])
>>> objective = cp.Minimize(c.T @ x)

# Then we'll write the constraints
>>> A = np.array([2, 1])
>>> G = np.array([1, 2])
>>> P = np.eye(2)
>>> constraints = [A @ x == 3, G @ x <= 3, P @ x >= 0] #This must be a list

# Assemble the problem and then solve it
>>> problem = cp.Problem(objective, constraints)
>>> print(problem.solve())
-8.999999999850528
>>> print(x.value)
array([1., 1.])
```

Achtung!

If you are having trouble with `pip install cvxpy` check the following:

- CVXPY requires a C++ compiler, most MacOs ad Linux Systems have them built in. If you are running Windows, make sure that you have the "C++ builder tools" from the Visual Studio Build Tools installed.
- CVXPY requires specific versions of packages in order to run, check that you have the right version of your packages. The most common is NumPy is not up to date.

Problem 1. Solve the following convex optimization problem:

$$\begin{aligned} \text{minimize} \quad & 2x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + 2x_2 \leq 3 \\ & x_2 - 4x_3 \leq 1 \\ & 2x_1 + 10x_2 + 3x_3 \geq 12 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{aligned}$$

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

l_1 Norm

The l_1 norm is defined

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

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

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

CVXPY includes the l_1 norm and many other useful functions. To specify a norm in CVXPY, use the syntax `cp.norm(x, a)` where a represents your choice of norm (1 in this case).

Problem 2. Write a function called `l1Min()` that accepts a matrix A and vector $\mathbf{b}$ as NumPy arrays and solves the l_1 minimization problem. Return the minimizer $\mathbf{x}$ and the primal objective value.

To test your function consider the matrix A and vector $\mathbf{b}$ below.

$$A = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & -2 & -1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

The linear system $A\mathbf{x} = \mathbf{b}$ has infinitely many solutions. Use `l1Min()` to verify that the solution which minimizes $||\mathbf{x}||_1$ is approximately $\mathbf{x} = [0., 2.571, 1.857, 0.]^\mathsf{T}$ and the minimum objective value is approximately 4.429. There's a file called `test_cvxpy_intro.py` that contains this example as a prewritten unit test that you can use to test your function for this problem.

The Transportation Problem

Consider the following transportation problem: A piano company needs to transport thirteen pianos from their three supply centers (denoted by 1, 2, 3) to two demand centers (4, 5). Transporting a piano from a supply center to a demand center incurs a cost, listed in Table 16.3. The company wants to minimize shipping costs for the pianos while meeting the demand.

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

Table 16.1: Number of pianos available at each supply center

Demand Center	Number of pianos needed
4	5
5	8

Table 16.2: Number of pianos needed at each demand center

Supply Center	Demand Center	Cost of transportation	Number of pianos
1	4	4	p_1
1	5	7	p_2
2	4	6	p_3
2	5	8	p_4
3	4	8	p_5
3	5	9	p_6

Table 16.3: Cost of transporting one piano from a supply center to a demand center

A system of constraints can be defined using the variables p_1, p_2, p_3, p_4, p_5, and p_6. First, there cannot be a negative number of pianos transported along any route. Next, use tables 16.1 and 16.2 and the variables $p_1 ... p_6$ to define a supply or demand constraint for each location. You may want to format this as a matrix. Finally, the objective function is the number of pianos shipped along each route multiplied by the respective costs (Table 16.3).

Note

Since our answers must be integers, in general this problem turns out to be an NP-hard problem. There is a whole field devoted to dealing with integer constraints, called *integer linear programming*, which is beyond the scope of this lab. Fortunately, we can treat this particular problem as a standard linear program and still obtain integer solutions.

Problem 3. Solve the piano transportation problem. Return the minimizer $\mathbf{x}$ and the primal objective value.

Quadratic Programming

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

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

where Q is a positive semidefinite symmetric matrix.

As an example, consider the quadratic function

$$f(x_1, x_2) = 2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2.$$

There are no constraints, so we only need to initialize the matrix Q and the vector $\mathbf{r}$. To find these, we first rewrite our function to match the formulation given above. If we let

$$Q = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \qquad \mathbf{r} = \begin{bmatrix} d \\ e \end{bmatrix}, \qquad \text{and} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

then

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

Thus, we see that the proper values to initialize our matrix Q and vector $\mathbf{r}$ are:

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

Now that we have the matrix Q and vector $\mathbf{r}$, we are ready to use the CVXPY function for quadratic programming, `cp.quad_form()`.

```
>>> Q = np.array([[4, 2],[2, 2]])
>>> r = np.array([1, -1])
>>> x = cp.Variable(2)
>>> prob = cp.Problem(cp.Minimize(.5 * cp.quad_form(x, Q) + r.T @ x))
>>> print(prob.solve())
[-1. 1.5]
>>> print(x.value)
-1.25
```

Problem 4. Find the minimizer and minimum of

$$g(x_1, x_2, x_3) = \frac{3}{2}x_1^2 + 2x_1x_2 + x_1x_3 + 2x_2^2 + 2x_2x_3 + \frac{3}{2}x_3^2 + 3x_1 + x_3$$

(Hint: Write the function g to match the formulation given above before coding.)

So far we have only dealt with affine constraints. When working with non-affine constraints, be aware that CVXPY comes with some Disciplined Convex Programming (DCP) rules. A minimization problem requires a convex objective function; similarly, a maximization problem requires a concave objective function. Equality constraints (`==`) must be affine. Less-than constraints (`<=`) must have the left side convex and the right side concave. Greater-than constraints (`>=`) must have the left side concave and the right side convex. This webpage provides a list of which of CVXPY's functions are concave or convex. `https://www.cvxpy.org/tutorial/functions/index.html`

Problem 5. Write a function that accepts a matrix A and vector $\mathbf{b}$ and solves the following problem.

$$\begin{array}{rl} \text{minimize} & \|A\mathbf{x} - \mathbf{b}\|_2 \\ \text{subject to} & \|\mathbf{x}\|_1 = 1 \\ & \mathbf{x} \succeq 0 \end{array}$$

To test your function, use the matrix A and vector $\mathbf{b}$ from Problem 2. The minimizer is approximately $\mathbf{x} = [0, 1, 0, 0]$ with objective value 5.099. Hint: `norm()` is a convex function, so you will have to think of a different way to take the 1-norm.

Unit Test

There is a file called `test_cvxpy_intro.py` that contains prewritten unit tests for Problem 2. There is a place for you to add your own unit tests to test your function for Problem 5, which will be graded.

Eating on a Budget

In 2009, the inmates of Morgan County jail convinced Judge Clemon of the Federal District Court in Birmingham to put Sheriff Barlett in jail for malnutrition. Under Alabama law, in order to encourage less spending, "the chief lawman could go light on prisoners' meals and pocket the leftover change."[1]. Sheriffs had to ensure a minimum amount of nutrition for inmates, but minimizing costs meant more money for the sheriffs themselves. Judge Clemon jailed Sheriff Barlett until a plan was made to use all allotted funds, $1.75 per inmate, to feed prisoners more nutritious meals. While this case made national news, the controversy of feeding prisoners in Alabama continues as of 2019[2].

[1] Nossiter, Adam, 8 Jan 2009, "As His Inmates Grew Thinner, a Sheriff's Wallet Grew Fatter", *New York Times*,`https://www.nytimes.com/2009/01/09/us/09sheriff.html`

[2] Sheets, Connor, 31 January 2019, "Alabama sheriffs urge lawmakers to get them out of the jail food business", `https://www.al.com/news/2019/01/alabama-sheriffs-urge-lawmakers-to-get-them-out-of-the-jail-food-business.html`

The problem of minimizing cost while reaching healthy nutritional requirements can be approached as a convex optimization problem. Rather than viewing this problem from the sheriff's perspective, we view it from the perspective of a college student trying to minimize food cost in order to pay for higher education, all while meeting standard nutritional guidelines.

The file `food.npy` contains a dataset with nutritional facts for 18 foods that have been eaten frequently by college students working on this text. A subset of this dataset can be found in Table 16.4, where the "Food" column contains the list of all 18 foods.

The columns of the full dataset are:

Column 1: p, price (dollars)
Column 2: s, servings per container
Column 3: c, calories per serving
Column 4: f, fat per serving (grams)
Column 5: $\hat{s}$, sugar per serving (grams)
Column 6: $\hat{c}$, calcium per serving (milligrams)
Column 7: $\hat{f}$, fiber per serving (grams)
Column 8: $\hat{p}$, protein per serving (grams)

Food	**Price**	**Servings**	**Calories**	**Fat**	**Sugar**	**Calcium**	**Fiber**	**Protein**
	p	s	c	f	$\hat{s}$	$\hat{c}$	$\hat{f}$	$\hat{p}$
	dollars			g	g	mg	g	g
Ramen	6.88	48	190	7	0	0	0	5
Potatoes	0.48	1	290	0.4	3.2	53.8	6.9	7.9
Milk	1.79	16	130	5	12	250	0	8
Eggs	1.32	12	70	5	0	28	0	6
Pasta	3.88	8	200	1	2	0	2	7
Frozen Pizza	2.78	5	350	11	5	150	2	14
Potato Chips	2.12	14	160	11	1	0	1	1
Frozen Broccoli	0.98	4	25	0	1	25	2	1
Carrots	0.98	2	52.5	0.3	6.1	42.2	3.6	1.2
Bananas	0.24	1	105	0.4	14.4	5.9	3.1	1.3
Tortillas	3.48	18	140	4	0	0	0	3
Cheese	1.88	8	110	8	0	191	0	6
Yogurt	3.47	5	90	0	7	190	0	17
Bread	1.28	6	120	2	2	60	0.01	4
Chicken	9.76	20	110	3	0	0	0	20
Rice	8.43	40	205	0.4	0.1	15.8	0.6	4.2
Pasta Sauce	3.57	15	60	1.5	7	20	2	2
Lettuce	1.78	6	8	0.1	0.6	15.5	1	0.6

Table 16.4: Subset of table containing food data

According to the FDA[1] and US Department of Health, someone on a 2000 calorie diet should have no more than 2000 calories, no more than 65 grams of fat, no more than 50 grams of sugar[2], at least 1000 milligrams of calcium[1], at least 25 grams of fiber, and at least 46 grams of protein[2] per day.

We can rewrite this as a convex optimization problem below.

$$\begin{aligned}
\text{minimize } & \sum_{i=1}^{18} p_i x_i, \\
\text{subject to } & \sum_{i=1}^{18} c_i x_i \leq 2000, \\
& \sum_{i=1}^{18} f_i x_i \leq 65, \\
& \sum_{i=1}^{18} \hat{s}_i x_i \leq 50, \\
& \sum_{i=1}^{18} \hat{c}_i x_i \geq 1000, \\
& \sum_{i=1}^{18} \hat{f}_i x_i \geq 25, \\
& \sum_{i=1}^{18} \hat{p}_i x_i \geq 46, \\
& x_i \geq 0.
\end{aligned}$$

Problem 6. Read in the file `food.npy` (this data is pickled, so you'll need to pass the keyword argument `allow_pickle=True` into `np.load()`). Use CVXPY to identify how much of each food item a college student should each to minimize cost spent each day given these simplified nutrition requirements. Return the minimizing vector and the total amount of money spent.

According to this problem, what is the food you should eat most each day? What are the three foods you should eat most each week?

(Hint: Each nutritional value must be multiplied by the number of servings to get the nutrition value of the whole product).

You can learn more about CVXPY at `https://www.cvxpy.org/index.html`.

[1] urlhttps://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/pdv.html

[2] https://www.today.com/health/4-rules-added-sugars-how-calculate-your-daily-limit-t34731

[1] 26 Sept 2018, `https://ods.od.nih.gov/factsheets/Calcium-HealthProfessional/`

[2] `https://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/protein.html`

17 Non-negative Matrix Factorization

Lab Objective: *Understand and implement the non-negative matrix factorization generator for recommendation systems with CVXPY.*

Introduction

Collaborative filtering is the process of filtering data for patterns using collaboration techniques. More specifically, it refers to making prediction about a user's interests based on other users' interests. These predictions can be used to recommend items and are why collaborative filtering is one of the common methods of creating a recommendation system.

Recommendation systems look at the similarity between users to predict what item a user is most likely to enjoy. Common recommendation systems include Netflix's "Movies you Might Enjoy" list, Spotify's "Discover Weekly" playlist, and Amazon's "Products You Might Like" suggestions.

Non-negative Matrix Factorization

Non-negative matrix factorization is one algorithm used in collaborative filtering. It can be applied to many other cases, including image processing, text mining, clustering, and community detection. The objective of non-negative matrix factorization is to take a non-negative matrix V and factor it into the product of two non-negative matrices.

For $V \in \mathbb{R}^{m \times n}, 0 \preceq W$,

$$\begin{aligned} \text{minimize} \quad & ||V - WH|| \\ \text{subject to} \quad & 0 \preceq W, 0 \preceq H \\ \text{where} \quad & W \in \mathbb{R}^{m \times k}, H \in \mathbb{R}^{k \times n} \end{aligned}$$

k is the rank of the decomposition and can either be specified or found using the Root Mean Squared Error (the square root of the MSE), SVD, Non-negative Least Squares, or cross-validation techniques.

For this lab, we will use the Frobenius norm, given by

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a|_{ij}^2}.$$

It is equivalent to the square root of the sum of the diagonal of $A^{\mathsf{H}}A$.

Problem 1. Create the `NMFRecommender` class, which will be used to implement the NMF algorithm. Initialize the class with the following parameters: `random_state` defaulting to 15, `tol` defaulting to $1e-3$, `maxiter` defaulting to 200, and `rank` defaulting to 3.

Add a method called `initialize_matrices` that takes in m and n, the dimensions of V. Set the random seed so that initializing the matrices can be replicated:

```
np.random.seed(self.random_state)
```

Initialize W and H using randomly generated numbers between 0 and 1, where $W \in \mathbb{R}^{m\times k}$ and $H \in \mathbb{R}^{k\times n}$, where `k=rank`. Store W and H as attributes and return them.

CVXPY

In order to compute the NMF of a matrix we will use the convex optimization package `CVXPY`. `CVXPY` is convex optimization package that takes a problem, converts it into standard form, calls a solver, and processes the result. Here is a basic example of how to use CVXPY

```
import cvxpy as cp

# Create two scalar optimization variables.
x = cp.Variable()
y = cp.Variable()

# Create two constraints.
constraints = [x + y == 1,
               x - y >= 1]

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve()  # Returns the optimal value.

# print the status of the solution
print(prob.status)
```

The variables `x` and `y` are updated as the problem is solved. Constraints are not required.

Vector valued variables can be created by including the dimension of the variable as an argument like so: `x = cp.Variable(10)`. Similarly matrix valued variables can be created: `x = cp.Variable((5, 5))`.

When initialized, variables have values of `None`. A value can be assigned to a variable before a problem is solved. The value can also be extracted after the optimal solution to a problem is found.

```
import numpy as np

random_matrix = np.random.random((5, 5))
x = cp.Variable((5, 5))
x.value = random_matrix
'''
solve a problem using the variable x
'''
solution_matrix = x.value
```

Problem 2. Finish the `NMF` class by adding a method `fit` that uses CVXPY to find an optimal W and H. It should accept V as a numpy array.

Constructing the problem so that it is known to be convex is important. Unfortunately, optimizing over 2 matrices being multiplied together breaks the rules of disciplined convex programming. Because of this, you must solve for W and H alternately. First solve for an optimal W given an H, then use that W to solve for an optimal H and so on. Continue until the difference between W and H and their previous values both have a Frobenius norm less than `tol`.

Finally, add a method called `reconstruct` that reconstructs and returns V by multiplying W and H.

HINT: You can build non-negativity into a `CVXPY` variable with `x = cp.Variable(n, nonneg=True)`. You can check if the solution is optimal by checking the status of your problem object with `prob.status`.

Using NMF for Recommendations

Consider the following marketing problem where we have a list of five grocery store customers and their purchases. We want to create personalized food recommendations for their next visit. We start by creating a matrix representing each person and the number of items they purchased in different grocery categories. So from the matrix, we can see that John bought two fruits and one sweet.

$$V = \begin{array}{cccccc c}
John & Alice & Mary & Greg & Peter & Jennifer & \\
0 & 1 & 0 & 1 & 2 & 2 & Vegetables \\
2 & 3 & 1 & 1 & 2 & 2 & Fruits \\
1 & 1 & 1 & 0 & 1 & 1 & Sweets \\
0 & 2 & 3 & 4 & 1 & 1 & Bread \\
0 & 0 & 0 & 0 & 1 & 0 & Coffee
\end{array}$$

After performing NMF on V, we'll get the following W and H.

$$W = \begin{array}{cccc} Component1 & Component2 & Component3 & \\ 2.19 & 0. & 0.03 & Vegetables \\ 1.53 & 3.13 & 0.11 & Fruits \\ 0.61 & 1.58 & 0. & Sweets \\ 0.01 & 0. & 1.88 & Bread \\ 0.47 & 0. & 0. & Coffee \end{array}$$

$$H = \begin{array}{ccccccc} John & Alice & Mary & Greg & Peter & Jennifer & \\ 0. & 0.43 & 0. & 0.42 & 0.96 & 0.86 & Component1 \\ 0.64 & 0.66 & 0.34 & 0. & 0.18 & 0.22 & Component2 \\ 0. & 1.06 & 1.58 & 2.12 & 0.52 & 0.53 & Component3 \end{array}$$

W represents how much each grocery feature contributes to each component; a higher weight means it's more important to that component. For example, component 1 is heavily determined by vegetables followed by fruit, then sweets, coffee and finally bread. Component 2 is represented almost entirely by fruits, while component 3 is based on fruits and bread, with a small amount of vegetables. H is similar, except instead of showing how much each grocery category affects the component, it shows a much each person belongs to the component, again with a higher weight indicating that the person belongs more in that component. We can see the John belongs in component 2, while Jennifer mostly belongs in component 1.

To get our recommendations, we reconstruct V by multiplying W and H.

$$WH = \begin{array}{ccccccc} John & Alice & Mary & Greg & Peter & Jennifer & \\ 0. & 0.9735 & 0.0474 & 0.9834 & 2.118 & 1.8993 & Vegetables \\ 2.0032 & 2.8403 & 1.238 & 0.8758 & 2.0894 & 2.0627 & Fruits \\ 1.0112 & 1.3051 & 0.5372 & 0.2562 & 0.87 & 0.8722 & Sweets \\ 0. & 1.9971 & 2.9704 & 3.9898 & 0.9872 & 1.005 & Bread \\ 0. & 0.2021 & 0. & 0.1974 & 0.4512 & 0.4042 & Coffee \end{array}$$

Most of the zeros from the original V have been filled in. This is the `collaborative filtering` portion of the algorithm. By sorting each column by weight, we can predict which items are more attractive to the customers. For instance, Mary has the highest weight for bread at 2.9704, followed by fruit at 1.238 and then sweets at 0.5372. So we would recommend bread to Mary.

Another way to interpret WH is to look at a feature and determine who is most likely to buy that item. So if we were having a sale on sweets but only had funds to let three people know, using the reconstructed matrix, we would want to target Alice, John, and Jennifer in that order. This gives us more information that V alone, which says that everyone except Greg bought one sweet.

Problem 3. Use the `NMFRecommender` class to run NMF on V, defined above, with 2 components. Return W, H, and the number of people who have higher weights in component 2 than in component 1.

Sklearn NMF

We also can compute the NMF using SkLearn. SkLearn uses a similar process as our class above. Here rank is represented by the parameter `n_components`.

```
from sklearn.decomposition import NMF

model = NMF(n_components=2, init='random', random_state=0)
W = model.fit_transform(V)
H = model.components_
```

SkLearn's NMF has other parameters that can also be included and adjusted. The `l1_ratio` is a value between 0 and 1 that gives a ratio of how much to weigh the l1 and l2 norms. When `l1_ratio` $= 1$, the l1 norm is used. When `l1_ratio` $= 0$, the l2 norm (Frobenius norm) is used. Any value between 0 and 1 is a weighted combination of the l1 and l2 norms. This ratio helps to prevent the model from over-fitting. The `alpha_W` and `alpha_H` parameters are floats that represent regularization constants; the default of 0 means that W and H are not regularized. If `alpha_H` is not specified, it will take on the value of `alpha_W` by default.

Endmember Detection using NMF

NMF can be used for analysis and identification of images. If each image corresponds to a $j \times k$ array of pixel intensities, then the data matrix is constructed by flattening the image into a vector of length jk and using these vectors as the columns of V, so V has dimensions $jk \times n$, where n is the number of images to analyze. Typically the materials in an image are referred to as the endmembers, which can be thought of as basis images which can be combined to reconstruct any image in the dataset. In the NMF decomposition $H[k, j]$ represents the abundance of the kth endmember or basis face in the jth image. $W[:, k]$ represents the spectral signature of the kth endmember. Then $V[:, j] \approxeq WH[:, j]$.

Example

This example exhibits how to use SkLearn's NMF class. The example begins with augmenting and reformatting the images; this will be done for you in the problems following the example. Pay special attention to last two blocks of code which demonstrate using NMF with properly formatted images and plotting the results.

```
from sklearn.datasets import load_sample_images
import numpy as np

# function to convert colored image to gray scale image
def gray_convert(rgb):
    r, g, b = rgb[:, :, 0], rgb[:, :, 1], rgb[:, :, 2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray

# load in sample images
dataset = load_sample_images()
# grab the first image
image = dataset.images[0]
```

```
# convert image to gray scale
image = gray_convert(image)

# get augmentations for additional images
flipLR = np.fliplr(image)
flipUP = np.flipud(image)

# create matrix V
images = [np.ravel(image), np.ravel(flipLR), np.ravel(flipUP)]
images = np.transpose(images)

# decompose using NMF
model = NMF(n_components = 5, max_iter = 1000)
W = model.fit_transform(images)
H = model.components_

# plot basis images
plt.subplots_adjust(wspace = .02, hspace = .05)
plt.figure(figsize=(20, 8))
for i in range(W.shape[1]):
    plt.subplot(1, 5, i+1)
    plt.xticks([], [])
    plt.yticks([], [])
    plt.imshow(np.reshape(W[:, i], (427, 640)), cmap = 'gray')
```

The following three flattened images now make up the columns of V.

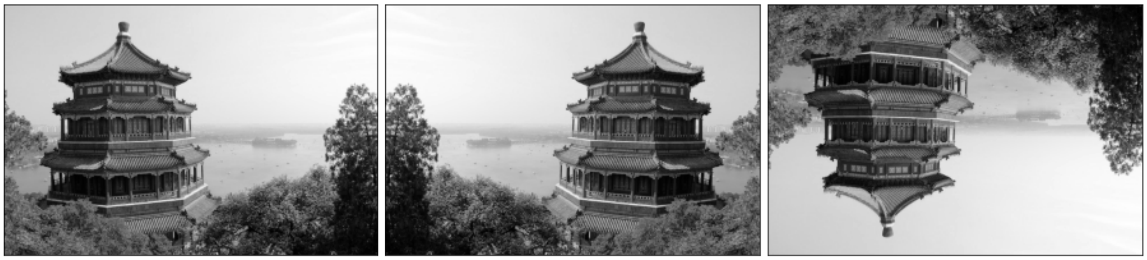


Figure 17.1: Original Images

Using a rank 5 reconstruction we see that the features used to reconstruct each image deal with the orientation of the building as well as the positive and negative space in each building.

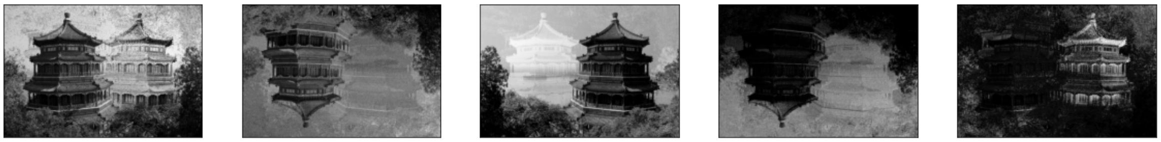


Figure 17.2: Basis images for a rank 5 deconstruction. Each basis image comes from a column of W.

For the next two problems we will be using a dataset of facial images that we can load in with the code below. Notice that `get_faces` formats the faces and returns the V matrix for the face images.

```
def get_faces(path="./faces94"):
    """Traverse the specified directory to obtain one image per subdirectory.
        Flatten and convert each image to grayscale.

        Parameters:
            path (str): The directory containing the dataset of images.

        Returns:
            ((mn, k) ndarray) An array containing one column vector per
                subdirectory. k is the number of people, and each original
                image is mxn.
        """

    # Traverse the directory and get one image per subdirectory.
    faces = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":       # Only get jpg images.
                # Load the image, convert it to grayscale,
                # and flatten it into a vector.
                faces.append(np.ravel(imread(dirpath+"/"+fname, mode="F")))
                break
    # Put all the face vectors column-wise into a matrix.
    return np.transpose(faces)

def show(image, m=200, n=180, plt_show=False):
    """Plot the flattened grayscale 'image' of width 'w' and height 'h'.

    Parameters:
        image ((mn,) ndarray): A flattened image.
        m (int): The original number of rows in the image.
        n (int): The original number of columns in the image.
        plt_show (bool): if True, call plt.show() at the end
    """
    #scale image
    image = image / 255
    #reshape image
    image = np.reshape(image, (m, n))
    #show image
    plt.imshow(image, cmap = "gray")

    if plt_show:
        plt.show()
```

Similar to the example, we have basis faces that are used to reconstruct the images in the original dataset. A sample of basis faces for a rank 75 reconstruction of the faces dataset seem to correspond to the following endmembers: forehead, glasses, hair.

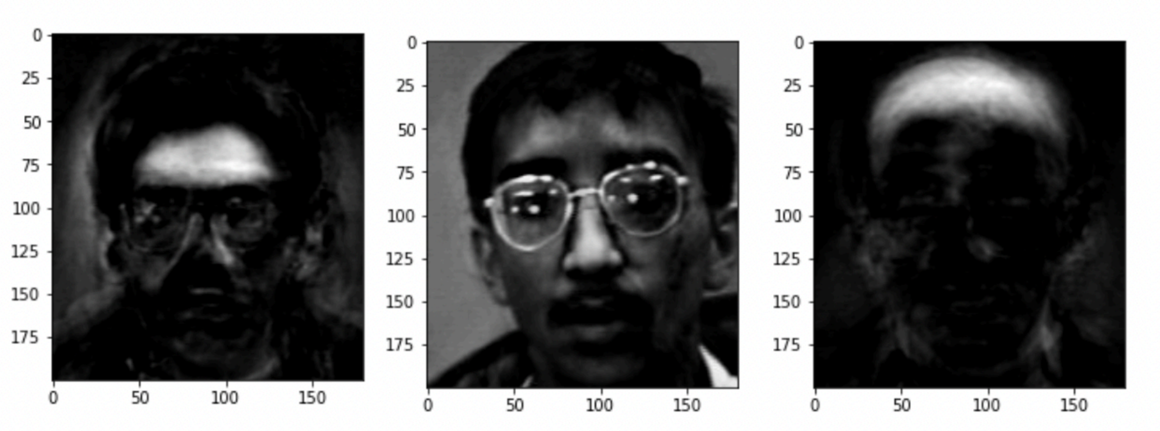


Figure 17.3: A sample of basis faces. Each basis face comes from a column of W.

Problem 4. Load in the facial dataset. SkLearn has the option to add regularization terms, `alpha_W` and `l1_ratio`, to the objective function $||V - WH||$. Reconstruct the third face in the dataset using SkLearn's NMF. Perform a grid search over the following: `n_components` $=$ [75], `alpha_W` $=$ [0, .2, .5], and `l1_ratio` $=$ [0, 10^{-5}, 1].

A grid search is a way to hone in on the best parameters by looping through a "grid" of values. It involves running the process for each combination of parameters in the grid. For example, a grid search will run NMF with `n_components` $=$ 75, `alpha_W` $=$ 0, and `l1_ratio` $=$ 0, then again with `n_components` $=$ 75, `alpha_W` $=$ 0, and `l1_ratio` $=$ 10^{-5}, and so on, with all combinations of values. Note this will take a while to run (approximately 10 minutes) since it will run and fit the NMF model 9 total times. In the NMF function, make sure to set `init = "random"` or else the function won't converge correctly.

Determine which set of parameters best reconstructs the face. These are the parameters that most closely approximate the third face with W and H. (Hint: For each set of parameters, find the norm of `faces[:,2] - W @ H[:,2]` and see which is the smallest. Due to the randomness of the algorithm, answers may vary. Additionally, notice how we only look at the third face to avoid large matrix multiplication, which is computationally expensive.)

Plot all reconstructions of the third face and put the parameters in the title; use subplots.

Problem 5. Run NMF on the facial dataset again, using the best parameters from the problem above. Next, for the second and twelfth faces in the dataset, find the 10 basis faces with the largest coefficients. Plot these basis faces along with the original image using subplots. In a markdown block write a sentence or two about differences you notice in the features of the basis faces (look closely).

18 Interior Point 1: Linear Programs

Lab Objective: *For decades after its invention, the Simplex algorithm was the only competitive method for linear programming. The past 30 years, however, have seen the discovery and widespread adoption of a new family of algorithms that rival–and in some cases outperform–the Simplex algorithm, collectively called* Interior Point methods. *One of the major shortcomings of the Simplex algorithm is that the number of steps required to solve the problem can grow exponentially with the size of the linear system. Thus, for certain large linear programs, the Simplex algorithm is simply not viable. Interior Point methods offer an alternative approach and enjoy much better theoretical convergence properties. In this lab we implement an Interior Point method for linear programs, and in the next lab we will turn to the problem of solving quadratic programs.*

Introduction

Recall that a linear program is a constrained optimization problem with a linear objective function and linear constraints. The linear constraints define a set of allowable points called the *feasible region*, the boundary of which forms a geometric object known as a *polytope*. The theory of convex optimization ensures that the optimal point for the objective function can be found among the vertices of the feasible polytope. The Simplex Method tests a sequence of such vertices until it finds the optimal point. Provided the linear program is neither unbounded nor infeasible, the algorithm is certain to produce the correct answer after a finite number of steps, but it does not guarantee an efficient path along the polytope toward the minimizer. Interior point methods do away with the feasible polytope and instead generate a sequence of points that cut through the interior (or exterior) of the feasible region and converge iteratively to the optimal point. Although it is computationally more expensive to compute such interior points, each step results in significant progress toward the minimizer. See Figure 18.1 for an example of a path using an Interior Point algorithm. In general, the Simplex Method requires many more iterations (though each iteration is less expensive computationally).

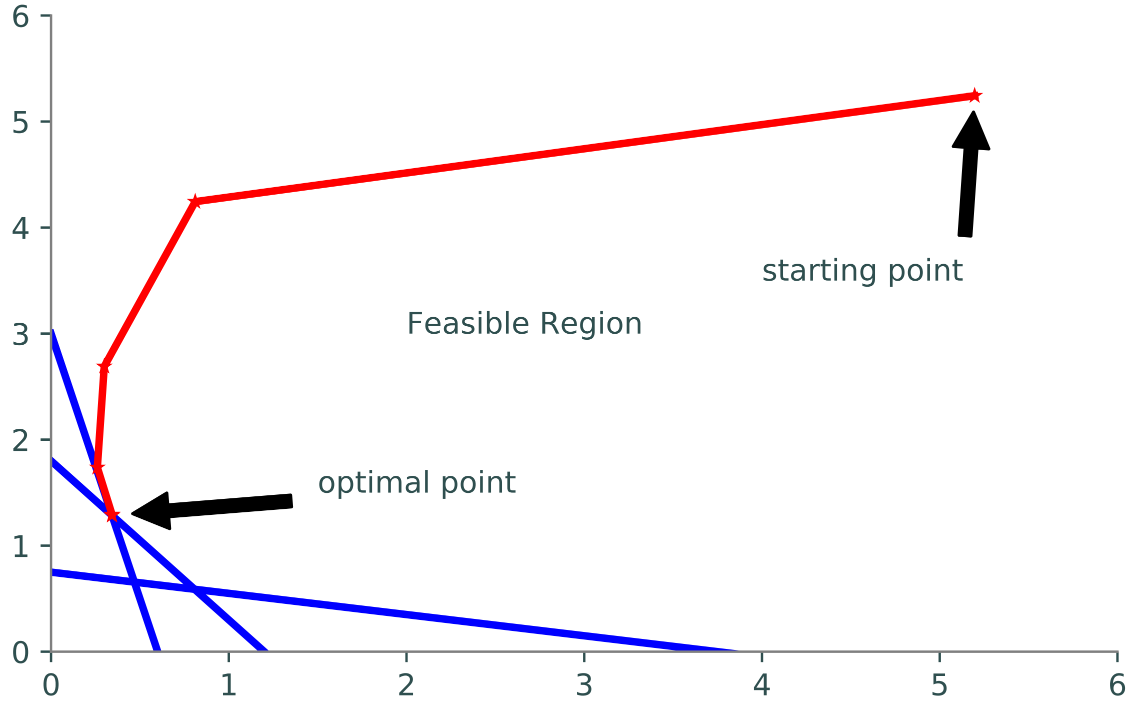


Figure 18.1: A path traced by an Interior Point algorithm.

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:

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

Here, $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m \times n}$ with full row rank. This is the *primal* problem, and its *dual* takes the form:

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

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ and $\boldsymbol{\mu} \in \mathbb{R}^n$.

KKT Conditions

The theory of convex optimization gives us necessary and sufficient conditions for the solutions to the primal and dual problems via the Karush-Kuhn-Tucker (KKT) conditions. The Lagrangian for the primal problem is as follows:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{c}^{\mathsf{T}}\mathbf{x} + \boldsymbol{\lambda}^{\mathsf{T}}(\mathbf{b} - A\mathbf{x}) - \boldsymbol{\mu}^{\mathsf{T}}\mathbf{x}$$

The KKT conditions are

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

It is convenient to write these conditions in a more compact manner, by defining an almost-linear function F and setting it equal to zero:

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

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

where $M = \operatorname{diag}(\mu_1, \mu_2, \ldots, \mu_n)$. 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.

In the next few problems, you will be writing functions within this function to solve the interior point problem one step at a time.

For this problem, within the `interiorPoint()` function, write a function for the vector-valued function F described above. This function should accept $\mathbf{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$ as parameters and return a 1-dimensional NumPy array with $2n + m$ entries.

Search Direction

A Primal-Dual Interior Point method is a line search method that starts with an initial guess $(\mathbf{x}_0^{\mathsf{T}}, \boldsymbol{\lambda}_0^{\mathsf{T}}, \boldsymbol{\mu}_0^{\mathsf{T}})$ and produces a sequence of points that converge to $(\mathbf{x}^{*\mathsf{T}}, \boldsymbol{\lambda}^{*\mathsf{T}}, \boldsymbol{\mu}^{*\mathsf{T}})$, 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 $(\triangle\mathbf{x}^{\mathsf{T}}, \triangle\boldsymbol{\lambda}^{\mathsf{T}}, \triangle\boldsymbol{\mu}^{\mathsf{T}})$ 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:

$$DF(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{bmatrix} 0 & A^{\mathsf{T}} & I \\ A & 0 & 0 \\ M & 0 & X \end{bmatrix}$$

where $X = \operatorname{diag}(x_1, x_2, \ldots, x_n)$.

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:

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

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 $\mathbf{e} = (1, 1, \ldots, 1)^{\mathsf{T}}$ 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 $(\triangle\mathbf{x}^{\mathsf{T}}, \triangle\boldsymbol{\lambda}^{\mathsf{T}}, \triangle\boldsymbol{\mu}^{\mathsf{T}})$ 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). Use the functions `lu_factor()` and `lu_solve()` from the `scipy.linalg` module to solving the system of equations efficiently.

Step Length

Now that we have our search direction, it remains to choose our step length. We wish to step nearly as far as possible without violating the problem's constraints, thus remaining in the interior of the feasible region. First, we calculate the maximum allowable step lengths for $\mathbf{x}$ and $\boldsymbol{\mu}$, respectively:

$$\alpha_{\max} = \min\{-\mu_i/\triangle\mu_i \mid \triangle\mu_i < 0\}$$
$$\delta_{\max} = \min\{-x_i/\triangle x_i \mid \triangle x_i < 0\}$$

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

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

[1] ν is the Greek letter for n, pronounced "nu."

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

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

Problem 3. Within `interiorPoint()`, write a subroutine to compute the step size after the search direction has been computed. Avoid using loops when computing $\alpha_{\max}$ and $\delta_{\max}$ (use masking and NumPy functions instead).

Initial Point

Finally, the choice of initial point $(\mathbf{x}_0, \boldsymbol{\lambda}_0, \boldsymbol{\mu}_0)$ is an important, nontrivial one. A naïvely or randomly chosen initial point may cause the algorithm to fail to converge. The following function will calculate an appropriate initial point.

```python
def starting_point(A, b, c):
    """Calculate an initial guess to the solution of the linear program
    min c\trp  x, Ax = b, x>=0.
    Reference: Nocedal and Wright, p. 410.
    """
    # Calculate x, lam, mu of minimal norm satisfying both
    # the primal and dual constraints.
    B = la.inv(A @ A.T))
    x = A.T @ B @ b
    lam = B @ A @ c
    mu = c - (A.T @ lam)

    # Perturb x and s so they are nonnegative.
    dx = max((-3./2)*x.min(), 0)
    dmu = max((-3./2)*mu.min(), 0)
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    # Perturb x and mu so they are not too small and not too dissimilar.
    dx = .5*(x*mu).sum()/mu.sum()
    dmu = .5*(x*mu).sum()/x.sum()
    x += dx*np.ones_like(x)
    mu += dmu*np.ones_like(mu)

    return x, lam, mu
```

Problem 4. Complete the implementation of `interiorPoint()`.

Use the function `starting_point()` provided above to select an initial point, then run the iteration `niter` times, or until the duality measure is less than `tol`. Return the optimal point $\mathbf{x}^*$ and the optimal value $\mathbf{c}^\mathsf{T}\mathbf{x}^*$.

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.

```python
def randomLP(j, k):
    """Generate a linear program min c\trp  x s.t. Ax = b, x>=0.
    First generate m feasible constraints, then add
    slack variables to convert it into the above form.
    Inputs:
        j (int >= k): number of desired constraints.
        k (int): dimension of space in which to optimize.
    Outputs:
        A ((j, j+k) ndarray): Constraint matrix.
        b ((j,) ndarray): Constraint vector.
        c ((j+k,), ndarray): Objective function with j trailing 0s.
        x ((k,) ndarray): The first 'k' terms of the solution to the LP.
    """
    A = np.random.random((j, k))*20 - 10
    A[A[:, -1]<0] *= -1
    x = np.random.random(k)*10
    b = np.zeros(j)
    b[:k] = A[:k, :] @ x
    b[k:] = A[k:, :] @ x + np.random.random(j-k)*10
    c = np.zeros(j+k)
    c[:k] = A[:k, :].sum(axis=0)/k
    A = np.hstack((A, np.eye(j)))
    return A, b, -c, x
```

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

Unit Test

There is a file called `test_interior_point_linear.py` that contains a place for you to write unit tests to test your function from Problems 1-4. Use the `randomLP()` function to create test cases to use in your function. The tests you write will be graded.

Least Absolute Deviations (LAD)

We now return to the familiar problem of fitting a line (or hyperplane) to a set of data. We have previously approached this problem by minimizing the sum of the squares of the errors between the data points and the line, an approach known as *least squares*. The least squares solution can be obtained analytically when fitting a linear function, or through a number of optimization methods (such as Conjugate Gradient) when fitting a nonlinear function.

The method of *least absolute deviations* (LAD) also seeks to find a best fit line to a set of data, but the error between the data and the line is measured differently. In particular, suppose we have a set of data points $(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_m, \mathbf{x}_m)$, where $y_i \in \mathbb{R}$, $\mathbf{x}_i \in \mathbb{R}^n$ for $i = 1, 2, \ldots, m$. 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:

$$y_i = \boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b, \qquad i = 1, 2, \ldots, m,$$

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

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

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 $\boldsymbol{\beta}^\mathsf{T}\mathbf{x}_i + b - y_i$ 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.

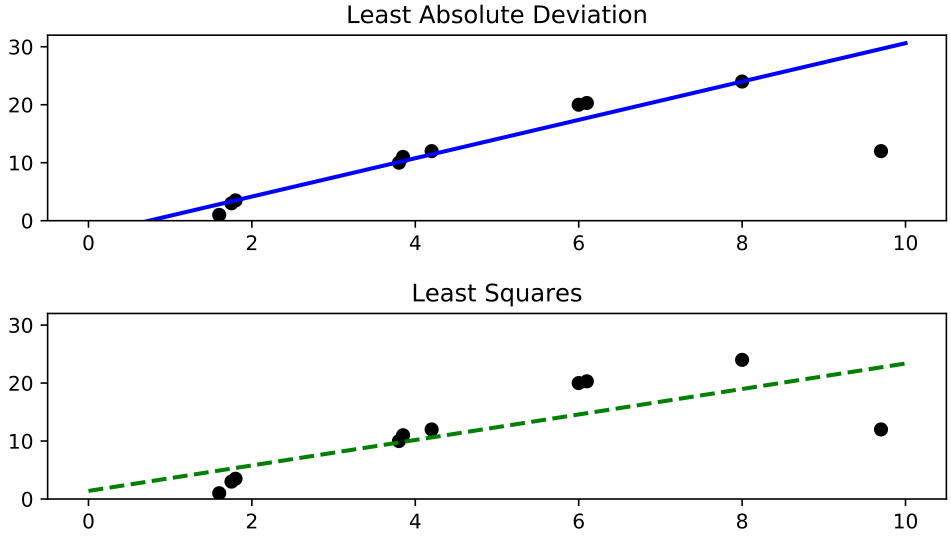


Figure 18.2: Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

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

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

The u_i are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

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

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:

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

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:

$$\begin{aligned}
\boldsymbol{\beta} &= \boldsymbol{\beta}_1 - \boldsymbol{\beta}_2,\\
b &= b_1 - b_2,\\
\boldsymbol{\beta}_1, \boldsymbol{\beta}_2 &\succeq \mathbf{0}; b_1, b_2 \geq 0.
\end{aligned}$$

Substituting these values into our constraints, we have the following system of constraints:

$$\begin{aligned}
u_i - \boldsymbol{\beta}_1^\mathsf{T}\mathbf{x}_i + \boldsymbol{\beta}_2^\mathsf{T}\mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i,\\
u_i + \boldsymbol{\beta}_1^\mathsf{T}\mathbf{x}_i - \boldsymbol{\beta}_2^\mathsf{T}\mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i,\\
\boldsymbol{\beta}_1, \boldsymbol{\beta}_2 \succeq \mathbf{0}; u_i, b_1, b_2, s_{2i-1}, s_{2i} &\geq 0.
\end{aligned}$$

Writing $\mathbf{y} = (-y_1, y_1, -y_2, y_2, \ldots, -y_m, y_m)^\mathsf{T}$ and $\boldsymbol{\beta}_i = (\beta_{i,1}, \ldots, \beta_{i,n})^\mathsf{T}$ for $i = \{1, 2\}$, we can aggregate all of our variables into one vector as follows:

$$\mathbf{v} = (u_1, \ldots, u_m, \beta_{1,1}, \ldots, \beta_{1,n}, \beta_{2,1}, \ldots, \beta_{2,n}, b_1, b_2, s_1, \ldots, s_{2m})^\mathsf{T}.$$

Defining $\mathbf{c} = (1, 1, \ldots, 1, 0, \ldots, 0)^\mathsf{T}$ (where only the first m entries are equal to 1), we can write our objective function as

$$\sum_{i=1}^{m} u_i = \mathbf{c}^\mathsf{T}\mathbf{v}.$$

Hence, the final form of our linear program is:

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

where A is a matrix containing the coefficients of the constraints. Our constraints are now equalities, and the variables are all nonnegative, so we are ready to use our interior point method to obtain the solution.

LAD Example

Consider the following example. We start with an array `data`, each row of which consists of the values $y_i, x_{i,1}, \ldots, x_{i,n}$, where $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})^\mathsf{T}$. We will have $3m + 2(n + 1)$ variables in our linear program. Below, we initialize the vectors $\mathbf{c}$ and $\mathbf{y}$.

```
>>> m = data.shape[0]
>>> n = data.shape[1] - 1
>>> c = np.zeros(3*m + 2*(n + 1))
>>> c[:m] = 1
>>> y = np.empty(2*m)
>>> y[::2] = -data[:, 0]
>>> y[1::2] = data[:, 0]
>>> x = data[:, 1:]
```

The hardest part is initializing the constraint matrix correctly. It has $2m$ rows and $3m+2(n+1)$ columns. Try writing out the constraint matrix by hand for small m, n, and make sure you understand why the code below is correct.

```
>>> A = np.ones((2*m, 3*m + 2*(n + 1)))
>>> A[::2, :m] = np.eye(m)
>>> A[1::2, :m] = np.eye(m)
>>> A[::2, m:m+n] = -x
>>> A[1::2, m:m+n] = x
>>> A[::2, m+n:m+2*n] = x
>>> A[1::2, m+n:m+2*n] = -x
>>> A[::2, m+2*n] = -1
>>> A[1::2, m+2*n+1] = -1
>>> A[:, m+2*n+2:] = -np.eye(2*m, 2*m)
```

Now we can calculate the solution by calling our interior point function.

```
>>> sol = interiorPoint(A, y, c, niter=10)[0]
```

However, the variable `sol` holds the value for the vector

$$\mathbf{v} = (u_1, \ldots, u_m, \beta_{1,1}, \ldots, \beta_{1,n}, \beta_{2,1}, \ldots, \beta_{2,n}, b_1, b_2, s_1, \ldots, s_{2m+1})^\mathsf{T}.$$

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 2: Quadratic Programs

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

Quadratic Optimization Problems

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

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

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

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

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

The Lagrangian function for this problem is:

$$\mathcal{L}(\mathbf{x},\boldsymbol{\mu}) = \frac{1}{2}\mathbf{x}^\mathsf{T}Q\mathbf{x}+\mathbf{c}^\mathsf{T}\mathbf{x}-\boldsymbol{\mu}^\mathsf{T}(A\mathbf{x}-\mathbf{b}), \tag{19.1}$$

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

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

Then the complete set of KKT conditions are:

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

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

$$F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q\mathbf{x} - A^{\mathsf{T}}\boldsymbol{\mu} + \mathbf{c} \\ A\mathbf{x} - \mathbf{y} - \mathbf{b} \\ YM\mathbf{e} \end{bmatrix} = \mathbf{0},$$

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

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

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

and the duality measure ν for this problem is

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

Search Direction

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

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) \begin{bmatrix} \triangle\mathbf{x} \\ \triangle\mathbf{y} \\ \triangle\boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \sigma\nu\mathbf{e} \end{bmatrix}, \tag{19.2}$$

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

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

In this function, calculate the search direction. Create F and DF as described above, and calculate the search direction $(\triangle\mathbf{x}^{\mathsf{T}}, \triangle\mathbf{y}^{\mathsf{T}}, \triangle\boldsymbol{\mu}^{\mathsf{T}})$ by solving Equation 19.2. Use $\sigma = \frac{1}{10}$ for the centering parameter.

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

Step Length

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

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

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. This may speed up convergence in some cases.

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

$$\beta_{\max} = \min\{-\mu_i/\triangle\mu_i \mid \triangle\mu_i < 0\}$$
$$\delta_{\max} = \min\{-y_i/\triangle y_i \mid \triangle y_i < 0\}$$

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

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

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

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

```
def startingPoint(G, c, A, b, guess):
    """
    Obtain an appropriate initial point for solving the QP
    .5 x\trp  Gx + x\trp  c s.t. Ax >= b.
    Parameters:
        G -- symmetric positive semidefinite matrix shape (n, n)
        c -- array of length n
        A -- constraint matrix shape (m, n)
        b -- array of length m
        guess -- a tuple of arrays (x, y, l) of lengths n, m, and m, resp.
    Returns:
        a tuple of arrays (x0, y0, l0) of lengths n, m, and m, resp.
    """
    m, n = A.shape
    x0, y0, l0 = guess

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

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

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

    return x0, y0, l0
```

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

```
procedure Interior Point Method for QP
    Choose initial point $(\mathbf{x}_0, \mathbf{y}_0, \boldsymbol{\mu}_0)$.
    while $k <$ niters and $\nu \geq$ tol: do
        Calculate the duality measure $\nu$.
        Solve 19.2 for the search direction $(\triangle\mathbf{x}_k, \triangle\mathbf{y}_k, \triangle\boldsymbol{\mu}_k)$.
        Calculate the step length $\alpha$.
        $(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \boldsymbol{\mu}_k) + \alpha(\triangle\mathbf{x}_k, \triangle\mathbf{y}_k, \triangle\boldsymbol{\mu}_k)$.
```

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

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 $\left[\frac{2}{3}, \frac{4}{3}\right]$.

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

Note

The Interior Point methods presented in this and the preceding labs are only special cases of the more general Interior Point algorithm. The general version can be used to solve many convex optimization problems, provided that one can derive the corresponding KKT conditions and duality measure 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 of an organism or by engineers to design tensile structures. Often we can describe configurations of elastic membranes as a solution to an optimization problem. As a simple example, we will find the shape of a large circus tent by solving a quadratic constrained optimization problem using our Interior Point method.

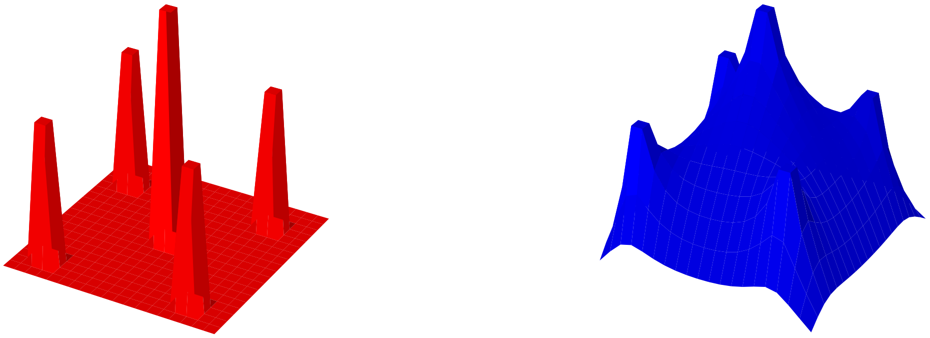


Figure 19.1: Tent pole configuration (left) and optimal elastic tent (right).

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 constraint:

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

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

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

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:

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

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

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

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

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

Application: Markowitz Portfolio Optimization

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

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

Stated more precisely, our portfolio consists of n risky assets together with an allocation vector $\mathbf{x} = (x_1, \ldots, x_n)^\mathsf{T}$, 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 ith 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:

$$\sqrt{\mathbf{x}^\mathsf{T} Q \mathbf{x}}.$$

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 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:

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

Note that we have slightly altered our objective function for convenience, as minimizing $\frac{1}{2}\mathbf{x}^\mathsf{T} Q\mathbf{x}$ is equivalent to minimizing $\sqrt{\mathbf{x}^\mathsf{T} Q\mathbf{x}}$. 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:

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

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.

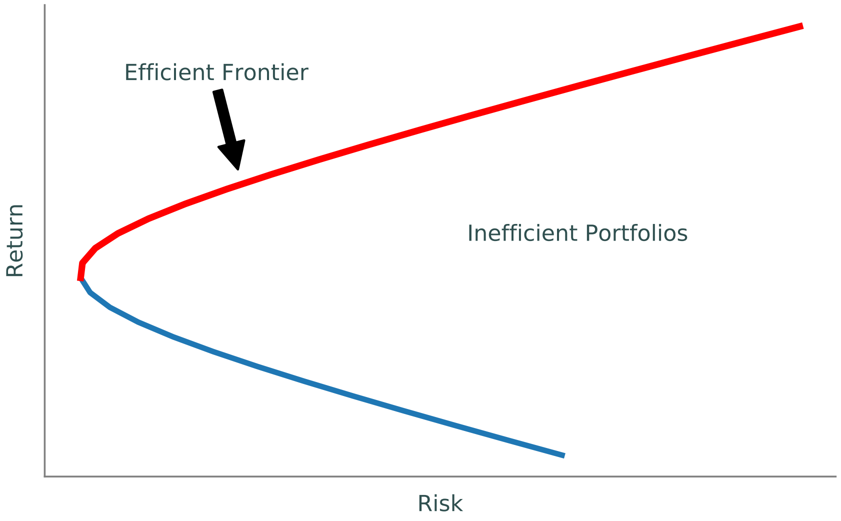


Figure 19.2: Efficient frontier on the risk-return plane.

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

$$y^i = [y^i_1, \ \ldots, \ y^i_k]^\mathsf{T}.$$

We estimate the expected rate of return for asset i by simply taking the average of $y_1, \ldots, y_k$, 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 $y^1, \ldots, y^n$. In this way, we obtain estimated values for each μ_i and Q.

Problem 4. The text file `portfolio.txt` contains historical stock data for several assets (U.S. bonds, gold, S&P 500, etc). In particular, the first column gives the years corresponding to the data, and the remaining eight columns give the historical returns of eight assets over the course of these years. Use this data to estimate the covariance matrix Q as well as the expected rates of return 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 Dynamic Programming

Lab Objective: *Sequential decision making problems are a class of problems in which the current choice depends on future choices. They are a subset of Markov decision processes, an important class of problems with applications in business, robotics, and economics. Dynamic programming is a method of solving these problems that optimizes the solution by breaking the problem down into steps and optimizing the decision at each time period. In this lab we use dynamic programming to solve two classic dynamic optimization problems.*

The Marriage Problem

Many dynamic optimization problems can be classified as *optimal stopping* problems, where the goal is to determine at what time to take an action to maximize the expected reward. For example, how many people should you date before you get married? Or when hiring a secretary, how many people should you interview before hiring the current interviewee? These problems try to determine at which person t to stop in order to maximize the chance of getting the best candidate.

For instance, let N be the number of people you could date. After dating each person, you can either marry them or move on; you can't resume a relationship once it ends. In addition, you can rank your current relationship to all of the previous options, but not to future ones. The goal is to find the policy that maximizes the probability of choosing the best marriage partner. This policy may not always choose the best candidate, but it should get an almost-best candidate most of the time.

Let $V(t-1)$ be the probability that the best marriage partner is person t, assuming we didn't choose the first $t-1$ candidates while using an optimal policy. In other words, after dating $t-1$ people, you want to know the probability that the t^{th} person is the one you should marry. Note that the probability that the t^{th} person is the best candidate is $\frac{1}{t}$ and the probability that they aren't is $\frac{t-1}{t}$. If the t^{th} person is not the best out of the first t, then the probability they are the best overall is 0 and the probability they are not is $V(t)$. If the t^{th} person is the best out of the first t, then the probability they are the best overall is $\frac{t}{N}$ and the probability they are not is $V(t)$.

By Bellman's optimality equations,

$$V(t-1) = \frac{t-1}{t}\max\left\{0, V(t)\right\} + \frac{1}{t}\max\left\{\frac{t}{N}, V(t)\right\} = \max\left\{\frac{t-1}{t}V(t) + \frac{1}{N}, V(t)\right\}. \qquad (20.1)$$

Notice that (20.1) implies that $V(t-1) \geq V(t)$ for all $t \leq N$. Hence, the probability of selecting the best match $V(t)$ is non-increasing. Conversely, $P(t \text{ is best overall} | t \text{ is best out of the first } t) = \frac{t}{N}$ is strictly increasing. Therefore, there is some t_0, called the *optimal stopping point*, such that $V(t) \leq \frac{t}{N}$ for all $t \geq t_0$. After t_0 relationships, we choose the next partner who is better than all of the previous ones. We can write (20.1) as

$$V(t-1) = \begin{cases} V(t_0) & t < t_0, \\ \frac{t-1}{t}V(t) + \frac{1}{N} & t \geq t_0. \end{cases}$$

The goal of an optimal stopping problem is to find t_0, which we can do by backwards induction. We start at the final candidate, who always has probability 0 of being the best overall if they are not the best so far, and work our way backwards, computing the expected value $V(t)$, for $t = N, N-1, \ldots, 1$.

If $N = 4$, we have

$$\begin{aligned} V(4) &= 0, \\ V(3) &= \max\left\{\frac{3}{4}V(4) + \frac{1}{4}, 0\right\} = .25, \\ V(2) &= \max\left\{\frac{2}{3}V(3) + \frac{1}{4}, .25\right\} = .4166, \\ V(1) &= \max\left\{\frac{1}{2}V(2) + \frac{1}{4}, .4166\right\} = .4583. \end{aligned}$$

In this case, the maximum expected value is .4583 and the stopping point is $t = 1$. It is also useful to look at the percent of possible candidates you should pass over before selecting a candidate. This is called the optimal stopping percentage, which in this case is $1/4 = .25$. After passing over (and observing) the optimal percentage of candidates, the first candidate that is better than the candidates you observed should be selected[1].

Problem 1. Write a function that accepts a number of candidates N. Calculate the expected values of choosing candidate t for $t = 1, 2, \ldots, N$.

Return the highest expected value $V(t_0)$ and the optimal stopping point t_0. (Hint: Python starts indices at 0, so you may need to adjust your indexing before returning the optimal stopping point.)

There is a file called `test_dynamic_programming.py` that contains a prewritten unit test for this problem. You can use it to make sure your function works for $N = 4$.

Problem 2. Write a function that takes in an integer M and runs your function from Problem 1 for each $N = 3, 4, \ldots, M$. Graph the optimal stopping percentage of candidates (t_0/N) to interview and the maximum probability $V(t_0)$ against N. Return the optimal stopping percentage for M.

The optimal stopping percentage for $M = 1000$ is .368.

[1] For examples and a more comprehensive explanation, see `https://www.americanscientist.org/article/knowing-when-to-stop`.

Both the stopping time and the probability of choosing the best person converge to $\frac{1}{e} \approx .36788$. Then to maximize the chance of having the best marriage, you should date at least $\frac{N}{e}$ people before choosing the next best person. This famous problem is also known as the *secretary problem*, the *sultan's dowry problem*, and the *best choice problem*. For more information, see `https://en.wikipedia.org/wiki/Secretary_problem`.

The Cake Eating Problem

Imagine you are given a cake. How do you eat it to maximize your enjoyment? Some people may prefer to eat all of their cake at once and not save any for later. Others may prefer to eat a little bit at a time. If we are to consume a cake of size W over $T+1$ time periods, then our consumption at each step is represented as a vector

$$\mathbf{c} = \begin{bmatrix} c_0 & c_1 & \cdots & c_T \end{bmatrix}^\mathsf{T},$$

where

$$\sum_{i=0}^{T} c_i = W.$$

This vector is called a *policy vector* and describes how much cake is eaten at each time period. The enjoyment of eating a slice of cake is represented by a utility function. For some amount of consumption $c_i \in [0, W]$, the utility gained is given by $u(c_i)$.

For this lab, we assume the utility function satisfies $u(0) = 0$, that $W = 1$, and that W is cut into N equally-sized pieces so that each c_i must be of the form $\frac{i}{N}$ for some integer $0 \leq i \leq N$.

Discount Factors

A person or firm typically has a time preference for saving or consuming. For example, a dollar today can be invested and yield interest, whereas a dollar received next year does not include the accrued interest. Since cake gets stale as it gets older, we assume that cake in the present yields more utility than cake in the future. We can model this by multiplying future utility by a discount factor $\beta \in (0, 1)$. For example, if we were to consume c_0 cake at time 0 and c_1 cake at time 1, with $c_0 = c_1$ then the utility gained at time 0 is larger than the utility at time 1:

$$u(c_0) > \beta u(c_1).$$

The total utility for eating the cake is

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

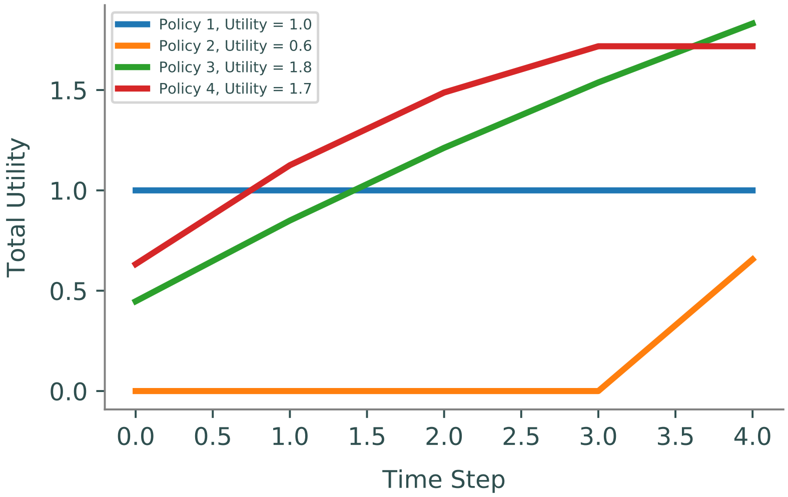


Figure 20.1: Plots for various policies with $u(x) = \sqrt{x}$ and $\beta = 0.9$. Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same amount of cake at each step, while policy 4 begins by eating .4 of the cake, then .3, .2, and .1.

The Value Function

The cake eating problem is an optimization problem where we maximize utility.

$$\begin{aligned} \max_{\mathbf{c}} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to } \sum_{t=0}^{T} c_t = W \\ c_t \geq 0. \end{aligned} \tag{20.2}$$

One way to solve it is with the value function. The value function $V(a, b, W)$ gives the utility gained from following an optimal policy from time a to time b.

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

$V(0, T, W)$ gives how much utility we gain in T days and is the same as Equation 20.2.

Let W_t represent the total amount of cake left at time t. Observe that $W_{t+1} \leq W_t$ for all t, because our problem does not allow for the creation of more cake. Notice that $V(t+1, T, W_{t+1})$ can be represented by $\beta V(t, T-1, W_{t+1})$, which is the value of eating W_{t+1} cake later. Then we can express the value function as the sum of the utility of eating $W_t - W_{t+1}$ cake now and W_{t+1} cake later.

$$V(t, T, W_t) = \max_{W_{t+1}} \left(u(W_t - W_{t+1}) + \beta V(t, T-1, W_{t+1}) \right) \tag{20.3}$$

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

Let $\mathbf{w} = \begin{bmatrix} 0 & \frac{1}{N} & \cdots & \frac{N-1}{N} & 1 \end{bmatrix}^\mathsf{T}$. We define the *consumption matrix* C by $C_{ij} = u(w_i - w_j)$. Note that C is an $(N+1) \times (N+1)$ lower triangular matrix since we assume $j \leq i$; we can't consume more cake than we have. The consumption matrix will help solve the value function by calculating all possible value of $u(W_t - W_{t+1})$ at once. At each time t, W_t can only have $N+1$ values, which will be represented as $w_i = \frac{i}{N}$, which is i pieces of cake remaining. For example, if $N = 4$, then $w = [0, .25, .5, .75, 1]^\mathsf{T}$, and $w_3 = 0.75$ represents having three pieces of cake left. In this case, we get the following consumption matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ u(0.25) & 0 & 0 & 0 & 0 \\ u(0.5) & u(0.25) & 0 & 0 & 0 \\ u(0.75) & u(0.5) & u(0.25) & 0 & 0 \\ u(1) & u(0.75) & u(0.5) & u(0.25) & 0 \end{bmatrix}.$$

Problem 3. Write a function that accepts the number of equal sized pieces N that divides the cake and a utility function $u(x)$. Assume $W = 1$. Create a partition vector $\mathbf{w}$ whose entries correspond to possible amounts of cake. Return the consumption matrix.

Solving the Optimization Problem

Initially we do not know how much cake to eat at $t = 0$: should we eat one piece of cake (w_1), or perhaps all of the cake (w_N)? It may not be obvious which option is best and that option may change depending on the discount factor β. Instead of asking how much cake to eat at some time t, we ask how valuable w_i cake is at time t. As mentioned above, $V(t, T-1, W_{t+1})$ in 20.3 is a new value function problem with $a = t, b = T-1$, and $W = W_{t+1}$, making 20.3 a recursion formula. By using the optimal value of the value function in the future, $V(t, T-1, W_{t+1})$, we can determine the optimal value for the present, $V(t, T, W_t)$. $V(t, T, W_t)$ can be solved by trying each possible W_{t+1} and choosing the one that gives the highest utility.

The $(N+1) \times (T+1)$ matrix A that solves the value function is called the *value function matrix*. A_{ij} is the value of having w_i cake at time j. $A_{0j} = 0$ because there is never any value in having w_0 cake, i.e. $u(w_0) = u(0) = 0$.

We start at the last time period. Since there is no value in having any cake left over when time runs out, the decision at time T is obvious: eat the rest of the cake. The amount of utility gained from having w_i cake at time T is given by $u(w_i)$. So $A_{iT} = u(w_i)$. Written in the form of (20.3),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} \left(u(w_i - w_j) + \beta V(0, -1, w_j) \right) = u(w_i). \tag{20.4}$$

This happens because $V(0,-1,w_j)=0$. As mentioned, there is no value in saving cake so this equation is maximized when $w_j=0$. All possible values of w_i are calculated so that the value of having w_i cake at time T is known.

Achtung!

Given a time interval from $t=0$ to $t=T$ the utility of waiting until time T to eat w_i cake is actually $\beta^{\mathsf{T}}u(W_i)$. However, through backwards induction, the problem is solved backwards by beginning with $t=T$ as an isolated state and calculating its value. This is why the value function above is $V(0,0,W_i)$ and not $V(T,T,W_i)$.

For example, the following matrix results with $T=3$, $N=4$, and $\beta=0.9$.

$$\begin{bmatrix} 0 & 0 & 0 & u(0) \\ 0 & 0 & 0 & u(0.25) \\ 0 & 0 & 0 & u(0.5) \\ 0 & 0 & 0 & u(0.75) \\ 0 & 0 & 0 & u(1) \end{bmatrix}.$$

Problem 4. Write a function that accepts a stopping time T, a number of equal sized pieces N that divides the cake, a discount factor β, and a utility function $u(x)$. Return the value function matrix A for $t=T$ (the matrix should have zeros everywhere except the last column). Return a matrix of zeros for the policy matrix P.

Next, we use the fact that $A_{jT}=V(0,0,w_j)$ to evaluate the $T-1$ column of the value function matrix, $A_{i(T-1)}$, by modifying (20.4) as follows,

$$A_{i(T-1)}=V(0,1,w_i)=\max_{w_j}\left(u(w_i-w_j)+\beta V(0,0,w_j)\right)=\max_{w_j}\left(u(w_i-w_j)+\beta A_{jT}\right). \qquad (20.5)$$

Remember that there is a limited set of possibilities for w_j, and we only need to consider options such that $w_j\leq w_i$. Instead of doing these one by one for each w_i, we can compute the options for each w_i simultaneously by creating a matrix. This information is stored in an $(N+1)\times(N+1)$ matrix known as the *current value matrix*, or CV^t, where the (ij)th entry is the value of eating w_i-w_j pieces of cake at time t and saving j pieces of cake until the next period. For $t=T-1$,

$$CV^{T-1}_{ij}=u(w_i-w_j)+\beta A_{jT}. \qquad (20.6)$$

The largest entry in the ith row of CV^{T-1} is the optimal value that the value function can attain at $T-1$, given that we start with w_i cake. The maximal values of each row of CV^{T-1} become the column of the value function matrix, A, at time $T-1$.

Achtung!

The notation CV^t does not mean raising the matrix to the tth power; rather, it indicates what time period we are in. All of the CV^t could be grouped together into a three-dimensional matrix, CV, that has dimensions $(N+1) \times (N+1) \times (T+1)$. Although this is possible, we will not use CV in this lab, and will instead only consider CV^t for any given time t.

The following matrix is CV^2 where $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$. The maximum value of each row, circled in red, is used in the 3^{rd} column of A. Remember that A's column index begins at 0, so the 3^{rd} column represents $j = 2$.

$$CV^2 = \begin{bmatrix} \boxed{0} & 0 & 0 & 0 & 0 \\ \boxed{0.5} & 0.45 & 0 & 0 & 0 \\ 0.707 & \boxed{0.95} & 0.636 & 0 & 0 \\ 0.866 & \boxed{1.157} & 1.136 & 0.779 & 0 \\ 1 & 1.316 & \boxed{1.343} & 1.279 & 0.9 \end{bmatrix}$$

Now that the column of A corresponding to $t = T-1$ has been calculated, we repeat the process for $T-2$ and so on until we have calculated each column of A. In summary, at each time step t, find CV^t and then set A_{it} as the maximum value of the ith row of CV^t. Generalizing (20.5) and (20.6) shows

$$CV^t_{ij} = u(w_i - w_j) + \beta A_{j(t+1)}. \qquad A_{it} = \max_j \left(CV^t_{ij}\right). \tag{20.7}$$

The full value function matrix corresponding to the example is below. The maximum value in the value function matrix is the maximum possible utility to be gained.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 0.95 & 0.95 & 0.95 & 0.707 \\ 1.355 & 1.355 & 1.157 & 0.866 \\ 1.7195 & 1.562 & 1.343 & 1 \end{bmatrix}.$$

Figure 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 5. Complete your function from Problem 4 so it returns the entire value function matrix. Starting from the next to last column, iterate backwards by

- calculating the current value matrix for time t using (20.7),
- finding the largest value in each row of the current value matrix, and
- filling in the corresponding column of A with these values.

(Hint: Use `axis` arguments.)

Solving for the Optimal Policy

With the value function matrix constructed, the optimization problem is solved in some sense. The value function matrix contains the maximum possible utility to be gained. However, it is not immediately apparent what policy should be followed by only inspecting the value function matrix A. The $(N+1)\times(T+1)$ policy matrix, P, is used to find the optimal policy. The (ij)th entry of the policy matrix indicates how much cake to eat at time j if we have i pieces of cake. Like A and CV, i and j begin at 0.

The last column of P is calculated similarly to last column of A. $P_{iT} = w_i$, because at time T we know that the remainder of the cake should be eaten. Recall that the column of A corresponding to t was calculated by the maximum values of CV^t. The column of P for time t is calculated by taking $w_i - w_j$, where j is the smallest index corresponding to the maximum value of CV^t,

$$P_{it} = w_i - w_j.$$

$$\text{where } j = \{\, \min\{j\} \mid CV^t_{ij} \geq CV^t_{ik}\ \forall\, k \in [0, 1, \ldots, N]\,\}$$

Recall CV^2 in our example with $T = 3$, $\beta = .9$, $N = 4$, and $u(x) = \sqrt{x}$ above.

$$CV^2 = \begin{bmatrix} \boxed{0} & 0 & 0 & 0 & 0 \\ \boxed{0.5} & 0.45 & 0 & 0 & 0 \\ 0.707 & \boxed{0.95} & 0.636 & 0 & 0 \\ 0.866 & \boxed{1.157} & 1.136 & 0.779 & 0 \\ 1 & 1.316 & \boxed{1.343} & 1.279 & 0.9 \end{bmatrix}$$

To calculate P_{12}, we look at the second row $(i = 1)$ in CV^2. The maximum, .5, occurs at CV^2_{10}, so $j = 0$ and $P_{12} = w_1 - w_0 = .25 - 0 = .25$. Similarly, $P_{42} = w_4 - w_2 = 1 - .5 = .5$. Continuing in this manner,

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & \boxed{0.25} \\ 0.25 & 0.25 & \boxed{0.25} & 0.5 \\ 0.25 & \boxed{0.25} & 0.5 & 0.75 \\ \boxed{0.25} & 0.5 & 0.5 & 1. \end{bmatrix}$$

Given that the rows of P are the slices of cake available and the columns are the time intervals, we find the policy by starting in the bottom left corner, P_{N0}, where there are N slices of cake available and $t = 0$. This entry tells us what percentage of the N slices of cake we should eat. In the example, this entry is 0.25, telling us we should eat 1 slice of cake at $t = 0$. Thus, when $t = 1$ we have $N - 1$ slices of cake available, since we ate 1 slice of cake. We look at the entry at $P_{(N-1)1}$, which has value 0.25. So we eat 1 slice of cake at $t = 1$. We continue this pattern to find the optimal policy $\mathbf{c} = \begin{bmatrix} .25 & .25 & .25 & .25 \end{bmatrix}$.

Achtung!

The optimal policy will not always be a straight diagonal in the example above. For example, if the bottom left corner had value .5, then we should eat 2 pieces of cake instead of 1. Then the next entry we should evaluate would be $P_{(N-2)1}$ in order to determine the optimal policy.

To verify the optimal policy found with P, we can use the value function matrix A. By expanding the entires of A, we can see that the optimal policy does give the maximum value.

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

Problem 6. Modify your function from Problem 4 to determine the policy matrix. Initialize the matrix as zeros and fill it in starting from the last column at the same time that you calculate the value function matrix.
(Hint: You may find `np.argmax()` useful.)

Unit Test

There is a file called `test_dynamic_programming.py` that contains a prewritten unit test for Problem 1. There is a spot for you to add your own unit test for your function from Problem 6 to make sure it produces the correct matrices from the example. This will be graded.

Problem 7. Write a function `find_policy()` that will find the optimal policy for the stopping time T, a cake of size 1 split into N pieces, a discount factor β, and the utility function u.

21 Reinforcement Learning 2: Markov Decision Process

Lab Objective: *We introduce the Markov decision process and its properties and connection to model-based reinforcement learning. We demonstrate two model-free methods and use them to solve a Markov decision process. This will connect to chapter 16 and 17 of the Volume 2 textbook.* ***Note*** *that we first present a rather good amount of theory before the start of the lab. This will help you understand the problem and the solutions better.*

A *Markov decision process* (MDP) is a mathematical framework used to model decision-making in situations where outcomes are partly random and partly under the control of a decision maker. Thus, the decision maker has the ability to take actions that affect the outcome of the process but does not have full control over the outcome. Generally, an MDP is a discrete-time stochastic[1] control process that contains at its core a *Markov chain/process* and follows the *Markov property*.

A sequence of random variables $X_1, X_2, \ldots, X_n$, known as a *stochastic* or *random process*, has the Markov property if the conditional probability distribution of future states of the process depends only upon the present state and not on the sequence of events that preceded it. That is, a stochastic process has the Markov property if $P(X_{n+1} = x_{n+1}|X_n = x_n, \ldots, X_1 = x_1) = P(X_{n+1} = x_{n+1}|X_n = x_n)$. Or in simpler terms, the future only depends on the present and not the past. A Markov chain or process is a stochastic model that describes a sequence of possible events that follow the Markov property.

What makes an MDP different from a Markov chain is that rather than determining event movement using only probabilities, event movement is determined based on probabilities, actions, and rewards. They are formulated as follows:

- $\mathbb{T}$ is a set of discrete time-periods called *decision epochs*. In this lab, $\mathbb{T} = \{0, 1, \ldots, T\}$, where T is the final time-period so that our MDP is finite, but it can also be infinite.

- S is the set of possible states, which is finite by definition of a finite MDP.

- A is a set of actions, which is also finite, where $\forall s \in S$, $A_s \subset A$ is the set of allowable actions that state s can take.

[1] By stochastic we mean probabilistic or randomly determined. Thus, there is some implied probability distribution over the domain, so that given the same input, the outcome is not always the same. *Deterministic* means that the outcome is fully predictable and always returns the same output given the same input. An MDP can be stochastic or deterministic.

- $g(s_t, a_t) = s_{t+1}$ is a transition function[2] that determines the state s_{t+1} at time $t+1$ based on the previous state s_t and action a_t.

- $\mathcal{R}$ is a set of rewards, which also is finite. The reward can be received after an action is taken, after several actions are taken, or at the end of the process.

- The time discount factor $\beta \in [0, 1]$ determines how much the reward function decreases in value with time. That is, a reward received at some time k in the future is worth β^{k-1} times as much as the same reward received today, so β accounts for this decrease in value. Thus, $\beta \to 0$ means we care more about immediate rewards, while $\beta \to 1$ means we care more about future rewards.

One important definition of an MDP is what is usually termed the *dynamics function* p given by

$$p(s', r|, s, a) = P(S_{t+1} = s', R_t = r|S_t = s, A_t = a). \tag{21.1}$$

This function[3] defines a probability distribution for each $s \in S$ and for each $a \in A_s$ (i.e. for each choice of s and a). That is, we have

$$\sum_{s' \in S} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \forall s \in S, \forall a \in A_s. \tag{21.2}$$

The dynamics function p tells us the probability of transitioning to state s' and receiving reward r after taking action a in state s at time t. Since this is an MDP, the dynamics function p is Markovian (i.e. p satisfies the Markov property as given earlier). Furthermore, the dynamics function p gives rise to the *state-transition probability*, or *transition probability* for short, at timestep t,

$$p(s'|s, a) = p_t(s'|s, a) = P(S_{t+1} = s'|S_t = s, A_t = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a). \tag{21.3}$$

This transition probability is the probability of ending up in state s' at timestep $t+1$ (i.e. next state being s') given that the process was in state s at time step t and action a was taken.

With p, we also get the reward function of three inputs $r : S \times A \times S \to \mathbb{R}$,

$$r(s', s, a) = r_t(s', s, a) = \mathbb{E}[R_t|S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{21.4}$$

which is the reward or expected reward for ending in state s' at timestep $t+1$ if the process is currently in state s at timestep t and action a is taken. Note that this equation[4] for the reward function is deterministic[5].

[2] This is also called the *transition model* and is usually denoted by the function of three inputs $T(s_t, a_t, s_{t+1})$. This definition is exactly as we have defined above, just different notation.

[3] We use capital letters with a time subscript to denote random variables and lowercase letters to denote specific values of those random variables.

[4] This uses the definition of conditional expectation which is very similar to normal expectation as taught in chapter 5 of the Volume 2 textbook with the difference being, roughly speaking, the use of conditional probability.

[5] The lone r as given in the definition of p is a stochastic function we talk about in the Additional Materials Section.

Moreover, the *dynamics* of an MDP, or the properties that govern the behavior of the MDP, are defined by the transition function g (or at least knowing the transition probabilities) and the reward function. Once we have these functions, we can start to solve the MDP without having to take actions in the process we are looking at since we can compute the expected reward using the transition probabilities. The objective in an MDP is to find a "policy" that specifies which action to take in each state that will maximize some cumulative function of the rewards, which is typically the sum of discounted rewards (see below). Thus, the dynamic optimization problem, assuming a finite horizon and a deterministic reward function of three variables, is

$$\begin{aligned} \max_{\mathbf{a}} \sum_{t=0}^{T} \beta^t r(s_{t+1}, s_t, a_t) \\ \text{subject to } s_{t+1} = g(s_t, a_t) \ \forall t. \end{aligned} \tag{21.5}$$

The cake eating problem follows this format where S consists of the possible amounts of remaining cake ($\frac{i}{W}$), c_t is the amount of cake we can eat, and the amount of cake remaining $s_{t+1} = g(s_t, a_t)$ is $w_t - c_t$, where w_t is the amount of cake we have left and c_t is the amount of cake we eat at time t. This is an example of a deterministic Markov process.

RL Connection & Definitions

Recall from the previous RL lab that, as a problem, RL dealt with an agent being given some task to complete in an environment where it only knows it can take some actions but is not told what to do at any given time. Thus, an MDP is the mathematical framework that models the environment in which the agent operates. This is the formalization that allows us to give equations to the value functions we attempted to estimate using model-free methods. Since an MDP follows the Markov property, we only need to rely on the current state or state-action pairs to predict the future. This implies that under an MDP the agent obtains a full observation of a given state so that the observation space equals the state space. Moreover, the fact we have the dynamics of the environment allows to make very precise estimations of the values of any state or state-action pairs. This is why we call this model-based RL.

Before continuing, we give two more definitions that are used in RL. Refer back to the previous RL lab for a refresh on other definitions we now assume. The new definitions are:

- For this lab[6], the *policy/strategy*, denoted by π, is a mapping that goes from the state space to the action space. Specifically, the policy is a deterministic rule by which the agent selects actions as a function of states so that $\pi(s) = a, a \in A_s$. We can think of the policy as a rule that tells the agent which action to take in each state.

- The *state-value function* or *value function* for a policy π, denoted by $v_\pi(s)$ or just $v(s)$, is a function of states that returns the *value* of a state s as the expected future rewards of starting at state s and following policy π thereafter. That is, given a state s and a policy π, the value of a state, $v(s)$, is the future rewards that the agent can receive by enacting π having started in s at some period t. This is different to the action-value function $q_\pi(s, a)$ as $v_\pi(s)$ only returns the value of a state and not a state with a specific action.

[6] We give a more general definition in the Additionals Materials Section.

Rewriting the Optimization Problem

Recall that in model-free RL the agent mainly learned by trial and error as it responded to the observed rewards of taken actions and then adjusted its approximation. Since we no longer have to do that, how does modeling the environment as an MDP help the agent learn? The MDP model allows us to use the equations of the value functions because these specify what is good in the long run, not just in the immediate sense like the reward does. Much like in Q-learning or SARSA, we will build estimations of the value functions, but in this case, we can now incorporate the decision-making since we have transition probabilities and a reward function to better our estimates. Our overall goal is to ***find the policy that maximizes a value function, for all of its inputs***, be it states or state-action pairs, which in turn will maximize the future rewards and lead the agent to accomplishing the given task.

Bellman Equation

Since we now have a model for the environment, we can give proper equations to the value functions. Given that we will be working primarily with the state value function, we will omit an equation for the action value function $q_\pi(s,a)$. For the remainder of this lab, assume we are working in the finite horizon setting and are under one episode with a finite set of timesteps $\mathbb{T} = \{1, \dots, T\}$.

For all $s \in S$, the state-value function $v_\pi(s)$ for a deterministic policy π can be defined as

$$v_\pi(s) = v(s) = \mathbb{E}\left[\sum_{k=0}^{T} \beta^k r_k \middle| S_0 = s\right] \tag{21.6a}$$

$$= \sum_{s' \in S^+} p(s'|s,a)\Big[r(s',s,a) + \beta v_\pi(s')\Big]. \tag{21.6b}$$

Equation 21.6b is the *Bellman equation* for the state-value function. Theis equation expresses the relationship between the value of a state and the value of the next state [7]. Do notice how this uses the dynamics of the environment. Unlike in model-free RL where we literally ave to choose some action to enact, model-based RL only needs to use the dynamics of the environment to compute an estimate.

Bellman Optimality Equation

Since we want to maximize the value function, we are essentially trying to find a policy that maximizes the expected future reward that the agent can receive. A policy π is defined to be better than another policy π' (i.e. $\pi \geq \pi'$) if and only if $v_\pi(s) \geq v_{\pi'}(s), \forall s \in S$. There is always at least one policy that is better than or equal to all other policies, so we call this the *optimal policy*, denoted by π_*. All optimal policies have the same *optimal state-value function*, denoted by v_*, which is defined as $v_*(s) = \max_\pi v_\pi(s), \forall s \in S$. They also share the same *optimal state-action value* or *optimal action-value function*, denoted by q_*, which is defined as $q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S, \forall a \in A_s$.

The optimal state-value function is still a value function for a policy, so it satisfies the Bellman equation in 21.6b. However, the fact v_* is the optimal state-value function allows us to rewrite it independently of any particular policy. We have

$$v_*(s) = \max_{a \in A_s} \sum_{s' \in S^+} p(s'|s,a)\Big[r(s',s,a) + \beta v_*(s')\Big] \tag{21.7}$$

[7] Note that by definition, the value of a terminal state is 0.

Equation 21.7 is the Bellman equation for v_* called the *Bellman optimality equation* for the state-value function under a deterministic policy. The Bellman optimality equation says that the value of a state under the optimal policy must equal the expected future rewards that the agent can receive by taking the best action in that state and following the optimal policy thereafter.

Thus, to solve the MDP, we need only get the actions that maximize the value function for each state. We have that the optimal policy π_* is given by $\pi_* = \pi_*(s) = \underset{a \in A_s}{\operatorname{argmax}}\, v_*(s)$.

Iterative Methods and Lab Notation

Iterative methods can be powerful ways to solve dynamic optimization problems without computing the exact solution. Often we can iterate very quickly to the true solution, or at least within some ε error of the solution. These methods are significantly faster than computing the exact solution using dynamic programming. When we compute the value functions with iterative methods, we typically use capital letters to denote the approximations of the value functions, e.g. V for the state-value function v and Q for the state-action value function q.

We let $N_{s,a}$ be the set of all possible next states for a given state-action pair (s, a). That is, $N_{s,a}$ represents all possible future states that can be obtained by taking action a during state s. Then, in the case of a deterministic Markov process, $N_{s,a}$ has one element for all state-action pairs since the transition probability is always 1. In a stochastic Markov process, there can be multiple possible next states for a given state-action pair since the transition probability is less than or equal to 1. As a result, $N_{s,a}$ may have multiple elements for each (s, a). Note that this new notation and assumption changes the definition of the value functions in the Bellman equations from having a sum iterating over all states s' in S^+ to just one sum over $s' \in N_{s,a}$.

Furthermore, we define a dictionary P to represent the decision process. This dictionary contains all of the information about the states, actions, probabilities, and rewards. Each dictionary key is a state-action combination and each dictionary value is a list of tuples. That is, P is a dictionary whose keys are states and whose values are dictionaries (i.e. a nested dictionary). The keys of the nested dictionary are actions and the values are lists of tuples. This goes as follows:

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

Note the slight notation change from $(s'|s, a)$ to $(s, a, \bar{s})$. In the dictionary, s is the current state, a is the action, $\bar{s} \in N_{s,a}$ is the next state if action a is taken, and $is\_terminal$ indicates if $\bar{s}$ is a terminal state. In addition, $p(s, a, \bar{s}) = p(\bar{s})$ is the probability of taking action a while in state s and ending in state $\bar{s}$, and $r(s, a, \bar{s}) = r(\bar{s})$ is the reward for taking action a while in state s and ending up in state $\bar{s}$.

Lastly, we will be making the optimal policy deterministic, so that the policy will always choose one action that maximizes the value function for a given state.

Lab Example: Moving on a Grid

The following example can be used to test all of your problems in this lab, except the last problem. This will be our working example for the lab.

Consider an $N \times N$ grid. Assume that a robot moves around the grid, one space at a time, until it reaches the lower right hand corner and stops. Each square is a state, so that the state space (including the terminal state) is $S^+ = \{0, 1, \ldots, N^2 - 1\}$, and the action space is $A = \{Left, Down, Right, Up\}$. For this lab, $Left = 0$, $Down = 1$, $Right = 2$, and $Up = 3$, so that $A = \{0, 1, 2, 3\}$. If you take the action $a = 1$, then you move *Down* on the grid. Thus, the action automatically determines the next state, so that the transition probability is deterministic.

Let $N = 2$ and label the squares as displayed below. In this example, we define the reward to be -1 if the robot moves into state 2, -1 if the robot moves into state 0 from state 1, and 1 when it reaches the terminal state, state 3. All other transitions have a reward of 0. We define the reward function to be $r(\bar{s})$. Since this is a deterministic model, $p(\bar{s}) = 1$ for all possible state-action pairs (s, a).

0	1
2	3

A_s is the set of actions that keep the robot on the grid. If the robot is in the top left hand corner, the only allowed actions are *Down* and *Right* so $A_0 = \{1, 2\}$. The transition function $g(s, a) = \bar{s}$ can be explicitly defined for each s, a where $\bar{s}$ is the new state after moving.

All of this information is encapsulated in P. We define $P[s][a]$ for all states and actions, even if they are not possible. This simplifies coding the algorithm but is not necessary.

```
P[0][0] = [(0, 0, 0, False)]    P[2][0] = [(0, 2, -1, False)]
P[0][1] = [(1, 2, -1, False)]   P[2][1] = [(0, 2, -1, False)]
P[0][2] = [(1, 1, 0, False)]    P[2][2] = [(1, 3, 1, True)]
P[0][3] = [(0, 0, 0, False)]    P[2][3] = [(1, 0, 0, False)]
P[1][0] = [(1, 0, -1, False)]   P[3][0] = [(0, 0, 0, True)]
P[1][1] = [(1, 3, 1, True)]     P[3][1] = [(0, 0, 0, True)]
P[1][2] = [(0, 0, 0, False)]    P[3][2] = [(0, 0, 0, True)]
P[1][3] = [(0, 0, 0, False)]    P[3][3] = [(0, 0, 1, True)]
```

For the sake of clarity, we will do a quick example using the above dictionary. We first assume that we start in state 0 corresponding to the 0 in the above grid. Next, we move *Down* the grid to state 2. This corresponds to taking action 1. To get the correct values from the dictionary, we look at $P[s][a]$ or in this case $P[0][1] = [(1, 2, -1, False)]$. So, when we move *Down* from state 0 to state 2, $p(\bar{s}) = 1$, $u(\bar{s}) = -1$, and $\bar{s} = 2$. As a final note, when the action is not possible $p(\bar{s}) = 0$, as shown in the dictionary above.

Foundation of Model-Based Methods: Policy Evaluation/Prediction

Policy evaluation or the *prediction problem* is the process of determining the value function v_π for a given arbitrary policy π. That is, the goal is to measure how well a policy performs by predicting the value of each state under the given policy[8]. Using the Bellman equation 21.6b, we can write the iterative update rule for policy evaluation as

$$v_{k+1}(s) = \sum_{s' \in S^+} p(s'|s, a)\Big[r(s', s, a) + \beta v_k(s')\Big], \forall s \in S. \tag{21.8}$$

As we take a sequence of value functions $v_0, v_1, \ldots$ and update them according to the above rule (i.e. $k \to \infty$), the sequence will converge to v_π. This process is called *iterative policy evaluation.* The convergence of the sequence is guaranteed since it can be shown that the Bellman equations follow Blackwell's theorem (see Volume 2 chapter 16.3).

[8] Note this is the same thing we did when running the equations for Q-learning or SARSA.

Value Iteration

Value iteration is a method used to solve the optimal value function by taking an initial estimate and updating the estimate of the optimal value function iteratively using the Bellman optimality equation as the update rule. Using our new notation, we can write the value iteration algorithm as follows for a given $s \in S$ (note the k is the iteration number and not an exponent):

$$V_*^{k+1}(s) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(s, a, \bar{s}) \cdot (r(s, a, \bar{s}) + \beta V_*^k(s)) \right\}. \tag{21.9}$$

This update rule is very similar to the iterative policy evaluation update rule in Equation 21.8 except that we now take the maximum value over all possible actions. Much like iterative policy evaluation, the estimated optimal value function V_*^k will converge to the optimal value function v_* as $k \to \infty$. However, we stop the update rule when the value function changes only by some small amount ε.

The summation of 21.9 occurs when it is a stochastic Markov process. For example, if the robot is in the top left corner, and we want it to move right, we could have the probability of the robot actually moving right as 0.5. In this case, $P[0][2] = [(0.5, 1, 0, False), (0.5, 2, -1, False)]$. This type of process will occur later in the lab.

Lab Example: Value Iteration

As an example, let $V_*^0 = [0, 0, 0, 0]$ and $\beta = 1$, where each entry of V_*^0 represents the maximum value at that state, and $V_*^0(s) = V_*^0[s]$ if we are using the array or list form of the value function. We calculate $V_*^1(s)$ from the robot example above. For $V_*^1(0)$, we choose the `max` of the possible outcomes, states 1 or 2, after moving. Thus, we use $P[0][2]$ for state 1 because moving from state 0 to state 1 requires going right, action 2.

$$\begin{aligned}
V_*^1(0) &= \max_{a \in A_0} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) \cdot (r(\bar{s}) + V_*^0(\bar{s})) \right\} \\
&= \max\{p(1) \cdot (r(1) + V_*^0(1)), p(2) \cdot (r(2) + V_*^0(2)))\} \\
&= \max\{1(0 + 0), 1(-1 + 0)\} \\
&= \max\{0, -1\} \\
&= 0 \\
V_*^1(1) &= \max\{p(0) \cdot (r(0) + V_*^0(0)), p(3) \cdot (r(3) + V_*^0(3))\} \\
&= \max\{1(-1 + 0), 1(1 + 0)\} \\
&= 1 \\
V_*^1(2) &= \max\{p(0) \cdot (r(0) + V_*^0(0)), p(3) \cdot (r(3) + V_*^0(3))\} \\
&= \max\{1(0 + 0), 1(1 + 0)\} \\
&= 1 \\
V_*^1(3) &= \max\{p(1) \cdot (r(1) + V_*^0(1)), p(2) \cdot (r(2) + V_*^0(2))\} \\
&= \max\{1(0 + 0), 1(0 + 0)\} \\
&= 0
\end{aligned}$$

This calculation gives $V_*^1 = [0,1,1,0]$. Repeating the process yields $V_*^2 = [1,1,1,0]$. Repeating a third time gives $V_*^3 = [1,1,1,0]$, which is the same as V_*^2, so the process has converged. This means that the solution is $[1,1,1,0]$. Thus, the total maximum reward the robot can achieve by starting on square i is the ith entry of the solution $[1,1,1,0]$.

When implementing functions in this lab, instead of only looking at possible actions $a \in A_s$, we can consider all of the actions. This will not affect the results, because $p(\bar{s}) = 0$ when an action is not possible. This simplifies the coding significantly. For example, when calculating $V_*^{k+1}(s_i)$ consider the following lines of code.

```
# Outside loop over all states s
state_action_vector = np.zeros(nA) # initial values for each action
for a in range(nA):
    for tuple_info in P[s][a]:
        # tuple_info is a tuple of (probability, next state, reward, done)
        p, next_s, r, _ = tuple_info
        # sums up the possible end states and rewards with given action
        state_action_vector[a] += (p * (r + beta * V_old[next_s]))
#add the max value to the value function
V_new[s] = state_action_vector.max()
```

Problem 1. Write a function called `value_iteration()` that will accept a dictionary P representing the decision process, an integer for the number of states, an integer for the number of actions, a float for the discount factor $\beta \in [0,1]$ defaulting to 1, a float for the tolerance amount ε defaulting to `1e-8`, and an integer for the maximum number of iterations `maxiter` defaulting to 3,000. Perform value iteration until $\|V_*^{k+1} - V_*^k\| < \varepsilon$ or $k >$ `maxiter`. Return the final vector representing $V_* \approx v_*$ and the number of iterations that were needed for convergence. Test your code on the example given above.

Foundation of Model-Based Methods: Policy Improvement/Control

One reason to evaluate a policy using any of the value functions is to improve it. Assuming we have a policy π, we want to know if we need to change the policy so that it now can choose some action $\bar{a} \neq \pi(s)$. $v_\pi(s)$ already tells us how good it is to follow the current policy π from state s, but would it be better if we changed to some new policy π' where we take some action $\bar{a}$ not given by the current policy that could potentially be better? This is where the *Policy Improvement theorem* can help. The theorem states that given two policies[9] $\pi(s)$ and $\pi'(s)$, if for all s in S, we have that $q_\pi(s, \bar{a} = \pi'(s)) \geq v_\pi(s)$[10], then $v_{\pi'}(s) \geq v_\pi(s), \forall s \in S$. This means that policy π' is at least as good as policy π. Moreover, if there is a strict inequality for at least one state s, then π' is strictly better than π.

[9] Note that $\pi(s)$ and $\pi'(s)$ are very much identical except that $\pi(s) \neq \bar{a} = \pi'(s)$ (i.e. π does not produce $\bar{a}$ for a specific s whereas the other can for that same s).

[10] We used the relationship equation 21.16.

Thus, to extend this process for all states and all actions, we can select at each given state the best action a according to the action-value function $q_\pi(s,a)$. That is, we consider the new greedy policy π' such that $\pi'(s) = \underset{a\in A_s}{\operatorname{argmax}}\, q_\pi(s,a) = \underset{a\in A_s}{\operatorname{argmax}} \sum_{s'\in S^+} p(s'|s,a)[r(s',s,a)+\beta v_\pi(s')]$. The process of making a new policy that improves the current one by using the current policy's value function is called *policy improvement*.

This whole process of finding an optimal policy without having a fixed policy but having a fixed value function is what is known the *control problem*. In the control problem, we are trying to find the optimal policy that maximizes the value function. Whereas in the prediction problem, we are trying to find the value function for a given policy assuming we stick to that policy (i.e. the policy is fixed).

Achtung!

Recall that the definition of argmax is the set of all actions that maximize the value function. Functions like `np.argmax()` return the index of the first occurrence of the maximum value and not a set. Thus, we need to be careful when using these functions to find the optimal action.

Calculating the Optimal Policy with Value Iteration (Policy Improvement)

While knowing the maximum expected value is helpful, it is usually more important to know the policy that generates the most value. Value iteration tells the robot what reward it can expect, but not how to get it. Recall that π is the strategy the agent uses to choose an action a given a state s as the input. Thus, the optimal deterministic policy chooses the action that maximizes the value function. Then, by 21.20 and 21.9 we have, for a deterministic reward function (using the lab notation),

$$\pi_* = \underset{a\in A_s}{\operatorname{argmax}} \left\{ \sum_{\bar{s}\in N_{s,a}} p(\bar{s})\cdot(r(\bar{s})+\beta\cdot V_*(\bar{s})) \right\}. \tag{21.10}$$

Note that we do need to know the estimate V_* of the optimal value function to find the optimal policy. This is because the optimal policy is the one that maximizes the value function. Thus, we can use the value function to find the optimal policy. This process of extracting the optimal policy using the optimal value function is the control portion (i.e. the policy improvement step).

Lab Example: Obtaining the Optimal Deterministic Policy

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

$$\begin{aligned}
\pi_*(0) &= \underset{\{1,2\}}{\operatorname{argmax}}\ \{p(2)\cdot(r(2)+V_*(2)), p(1)\cdot(r(1)+V_*(1))\}\\
&= \operatorname{argmax}\{1\cdot(-1+1), 1\cdot(0+1)\}\\
&= \operatorname{argmax}\{0,1\}\\
&= 2
\end{aligned}$$

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

$$\begin{aligned}\pi_*(1) &= \underset{\{0,1\}}{\text{argmax}}\ \{1\cdot(-1+1), 1\cdot(1+0)\}\\ &= \text{argmax}\{0,1\} = 1\\ \pi_*(2) &= \underset{\{2,3\}}{\text{argmax}}\ \{1\cdot(1+0), 1\cdot(0+1)\}\\ &= \text{argmax}\{1,1\} = 2\end{aligned}$$

Since state 3 is terminal, it doesn't matter what $\pi_*(3)$ is, but we'll set it to 0 for convenience. Thus, the optimal policy corresponding to the optimal reward is $[2,1,2,0]$. The robot should move to state 3 if possible, avoiding state 2 because it has a negative reward.

Note

Note that π_* gives the optimal action a to take at each state s. It does not give a sequence of actions to take in order to maximize the policy. In other words, π_* does not give a specific ordering of best actions to take in order to maximize the policy. It only tells us what to do at each state and not how to organize the actions across the whole state space.

Problem 2. Write a function called `policy_improvement()` that will accept a dictionary P representing the decision process, the number of states, the number of actions, an array V representing some value function (optimal or not), and a discount factor $\beta \in [0,1]$ defaulting as before. Return the deterministic policy vector π corresponding to V.

In order to use Equation 21.10, you will need to run a similar method to the one in Problem 1 but with the modification as shown in the equation and using the estimated value function V rather than some old value of a previous iteration. Note this is a function that will perform the control problem on any array V. It is not specific to V_*. You can test your code on the example with V_* and $\beta = 1$.

Policy Iteration

For dynamic programming problems, it can be shown that value function iteration converges relative to the discount factor β. As $\beta \to 1$, the number of iterations increases dramatically. As mentioned earlier β is usually close to 1, which means this algorithm can converge slowly. In value iteration, we used an initial guess, V_*^0, for the optimal value function, and used Equation 21.7 to make an iterative method towards the true value function resulting in Equation 21.9. Once we achieved a good enough approximation for v_*, we recovered the optimal policy π_*.

Instead of iterating on our value function, we can similarly make an initial guess, π_*^0 (the zero is an iteration number not an exponent), for the optimal policy and use this to iterate toward the true π_*. We do so by taking advantage of the definition of the value function, where we assume that the current policy is optimal. We can then compute $V_{\pi_*^0}$ and use this to find a better policy using the policy improvement theorem. This then gives us a new policy π_*^1 so that we then repeat the process of performing policy evaluation and policy improvement until we reach the optimal policy. This iterative process of using policy evaluation and policy improvement is called *policy iteration*.

$$\pi^0 \xrightarrow{\text{evaluation}} V_\pi^0 \xrightarrow{\text{improvement}} \pi^1 \xrightarrow{\text{evaluation}} V_\pi^1 \xrightarrow{\text{improvement}} \pi^2 \cdots \xrightarrow{\text{improvement}} \pi_*$$

Figure 21.1: The policy iteration process as given by [Hu23].

Policy Evaluation (Step 1)

Given any arbitrary policy π_k, we can modify Equation 21.8 by assuming that the given policy actually returns an optimal action $a = \pi_k(s)$. Because the policy is optimal, we have that the action the transition probability uses is the action that the policy π_k returns. Thus, $p(s'|s, a) = p(s'|s, \pi(s))$. We then get the following equation for an iterative policy evaluation:

$$V_{k+1}(s) = \sum_{\bar{s} \in N_{s,\pi(s)}} p(\bar{s}) \cdot (r(\bar{s}) + \beta \cdot V_k(\bar{s})). \tag{21.11}$$

This equation is very similar to the iterative policy evaluation equation 21.8, except that we assume that the policy is optimal.

Then, in order to compute the iterative policy evaluation, we can do something similar to the given code block right before Problem 1 with a few modifications. One, we no longer need to loop over all actions since we are assuming the policy is optimal, but we still have to create a state-action vector for all actions in the action space. Next, rather than looking at the tuple information of `P[s][a]`, we look at the tuple of `P[s][policy[s]]`. Lastly, since we are not computing V_*, we can just add all the values of the state-action vector together to calculate the new `V_new[s]`. Consider the following

```
for s in range(nS):
    state_action_vector = np.zeros(nA)
    for tuple_info in P[s][policy[s]]:
        # The rest is the same as before
    V_new[s] = state_action_vector.sum()
```

Problem 3. Write a function called `iter_policy_eval()` that accepts a dictionary P representing the decision process, the number of states, the number of actions, an array called `policy` representing some chosen deterministic policy, a discount factor $\beta \in [0, 1]$, and a tolerance amount ε, these last two defaulting as before. Use the process above to return an approximated value function, $V_{k+1} \approx v$, corresponding to some given policy. That is, return an array called V where $V[s]$ is the approximated value of state s.

You may want to cast the value `policy[s]` to an integer to avoid any indexing errors. Note that your code should be very similar to the code in Problem 1 except for the modifications mentioned above. Also, notice that you are not given a maximum number of iterations in this problem.

Note you are not computing an approximation to the optimal value function v_*. You are iteratively computing the value function for any given policy under the assumption that the policy is optimal. Whether the actual given policy is optimal or not, it does not matter as we only want its value function v_π. You should not be taking any maximums or argmaxes in this problem.

Test your code on the policy vector generated from `policy_improvement()` for the example. The result should be the same value function array from `value_iteration()`.

Policy Improvement (Step 2)

Now that we have the value function for our policy, we can take the value function and find a better policy. This step employs the same method used in value iteration to find the policy. In other words, this step uses the `policy_improvement()` method from Problem 2 with the newly computed value function V_{k+1}.

Policy iteration starts with an initial estimation π_*^0 and iterates using iterative policy evaluation and policy improvement successively until the desired tolerance is reached. The algorithm for policy iteration, using two of the functions that you previously implemented, can be summarized as follows:

Algorithm 1 Policy Iteration

```
procedure POLICY ITERATION($P, nS, nA, \beta, tol$, maxiter)
    $\pi_*^0 \leftarrow [\pi_*^0(s_0), \pi_*^0(s_1), \ldots, \pi_*^0(s_N)]$        ▷ Initialize $\pi_*$ as array of ones of length nS
    for $k = 0, 1, \ldots,$ maxiter do        ▷ Iterate only maxiter times at most
        $V_*^{k+1} =$ iter_policy_eval($\pi_*^k$)        ▷ Policy evaluation step
        $\pi_*^{k+1} =$ policy_improvement($V_*^{k+1}$)        ▷ Policy improvement step
        if $||\pi_*^{k+1} - \pi_*^k|| < \varepsilon$ then
            break        ▷ Stop iterating if the policy doesn't change enough
    return $V_*^{k+1}, \pi_*^{k+1}$
```

Note that algorithm uses functions whose assumptions are that we have a deterministic optimal policy and deterministic reward function. You should modify the functions when dealing with stochastic policies and rewards.

Problem 4. Write a function called `policy_iteration()` that will accept a dictionary P representing the decision process, the number of states, the number of actions, a discount factor $\beta \in [0, 1]$ (defaulting as before), the tolerance amount ε (defaulting as before), and the maximum number of iterations `maxiter` defaulting to 200. Perform policy iteration until $\|\pi_*^{k+1} - \pi_*^k\| < \varepsilon$ or $k >$ `maxiter`. Return the final vector representing V_*^k, the optimal deterministic policy π_*^k, and the number of iterations required for convergence. Test your code on the example given above and compare your answers to the results from Problems 1 and 2.

The Frozen Lake Problem

For the rest of this lab, we will be using the Gymnasium environment `'FrozenLake-v1'`. Gymnasium can be installed using the following code.

```
>>> pip install gymnasium
>>> # You may also need to install these dependencies
>>> pip install gymnasium[all]
>>> pip install gymnasium[classic-control]
```

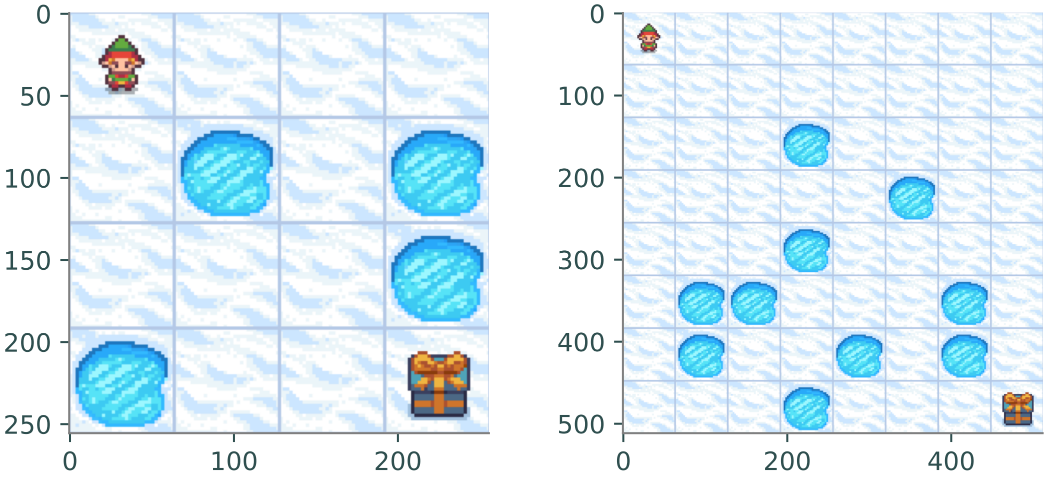


Figure 21.2: Default starting positions of the 4×4 and 8×8 versions of "FrozenLake-v1"

In the Frozen Lake problem, an elf attempts to cross a treacherous frozen lake to obtain a present. The lake is divided into an $N \times N$ grid where the top left corner is the start, the bottom right corner is the end, and the other squares are either frozen or holes. To retrieve the present, the elf must successfully navigate around the melted ice without falling through a hole. The possible actions are left, right, up, and down, but since the ice is slippery, the elf won't always move in the intended direction. Hence, this is a stochastic MDP (i.e. $p(s'|s, a) \leq 1$). The reward for falling is 0, and the reward for obtaining the present is 1. There are two scenarios with $N = 4$ and $N = 8$.

Using Gymnasium

The 'FrozenLake-v1' environment has 3 important attributes: P, observation_space.n, and action_space.n. We can calculate the optimal policy of 'FrozenLake-v1' with value iteration or policy iteration using these 3 attributes. Since the ice is slippery, this policy will not always result in a reward of 1.

```
>>> import gymnasium as gym

>>> # Initialize environment for 4x4 scenario
>>> env = gym.make('FrozenLake-v1', desc=None, map_name='4x4', is_slippery=True↩
    )
>>> # Find number of states and actions
>>> env.observation_space.n
16
>>> env.action_space.n
4
>>> # Get the dictionary with all the states and actions
>>> dictionary_P = env.P

>>> env.close()  # Always close the environment!
```

The attribute `P` is similar to the dictionary we used in the previous problems. As already mentioned, $p(s'|s,a) \leq 1$, which means the set $N_{s,a}$ has more than one value. **NOTE** if you did not implement the functions in this lab to account for this, they will not work as intended on this dictionary, which we will use for the remainder of this lab.

Problem 5. Note first that this problem and the next are linked so you may want to read through both before starting.

Write a function called `frozen_lake()` that accepts a boolean `basic_case` defaulting to `True`, an integer M defaulting to 1000 that indicates how many episodes of `"FrozenLake-v1"` to run, and a boolean `render` defaulting to `False`. If `basic_case` is `True`, run the 4×4 scenario. If not, run the 8×8 scenario. If `render` is `True`, render the environment by applying the argument `render_mode='human'` when initializing the environment. Close the environment at the end of the function.

For each model-based algorithm, your created `frozen_lake()` function will also return the optimal value function array V_*, the optimal deterministic policy array π_*, and the average discounted reward sum for M episodes. Thus, your output is a tuple of 6 elements. Note the two model-based algorithms are value iteration and policy iteration. (Hint: Rendering this environment can take a long time, so only render it with small values of M.)

Remember that policy iteration already returns both V_* and π_* whereas value iteration only returns V_*. Once you have both optimal policies, use problem 6 to help you obtain the average.

Gymnasium environments have built-in functions that allow us to simulate each step of the scenario. Before running a simulation in Gymnasium, always revert it to the starting position by calling the `reset()` function. The function `step()` moves the simulation to the next state.

```
>>> import gymnasium as gym
>>> # Initialize environment for 4x4 scenario
>>> env = gym.make('FrozenLake-v1', desc=None, map_name='4x4', is_slippery=True↩
    )

>>> # Put environment in starting state
>>> observation, info = env.reset()
>>> # Take a step in the optimal direction and update variables
>>> observation, reward, done, trunc, info = env.step(int(policy[observation]))

>>> env.close()  # Always close the environment!
```

The function `step()` takes integers representing different actions and returns: `observation`, `reward`, `done`, `truncated`, and `info`. When we take an action, we get a new `observation`, or state, as well as the `reward` for taking that action. If the elf falls into a hole or reaches the present, the simulation terminates (`done=True`). The `truncated` and `info` values will not be used in this lab. For more information about this environment, visit `gymnasium.farama.org/environments/toy_text/frozen_lake/`.

Problem 6. Write a function `run_simulation()` that takes in an environment `env`, a deterministic policy array `policy`, and a discount factor β. Calculate the total discounted reward sum of the policy for one episode of the environment (i.e. step through the environment until `done=True`). This function will be called by `frozen_lake()` in 5, which both initializes and closes the environment, so do not call `close()` in this function. However, you should call `reset()` at the beginning of this function, to revert the environment back to its starting position. (Hint: When calculating the dicounted reward, use β^k as shown in Equation 21.5.)

Next, modify `frozen_lake()` to call `run_simulation()` for both the value iteration and policy iteration for M episodes. Then, modify `frozen_lake()` to return the actual values of the mean total discounted reward for both policies. (Hint: Even though you run the simulation M times, you should only calculate the policies once, because each policy depends on the dictionary P, which does not change.)

Wrapping Up Reinforcement Learning

There a few more things to discuss before we finish this lab and move on to the last problem. We do reserve some other important ideas to the Additional Materials section so that we can focus on the last problem. We strongly encourage you to read through the Additional Materials section to get a better understanding of reinforcement learning.

Value Iteration vs. Policy Iteration

An *expected update* is the term used to describe the single update of value of a single state $s \in S$. A *sweep* is the term used to describe a complete iteration of an expected update for all states $s \in S$ (i.e. iterating through all states and updating each once). Both value iteration and policy iteration use sweeps to find the optimal policy. Value iteration first performs various sweeps using iterative policy evaluation. It stops once we are within a certain tolerance of the true value function. We then perform a single sweep using policy improvement to find the optimal policy.

On the other hand, policy iteration first performs various sweeps using policy evaluation to obtain an estimated optimal value function V_*. We then perform a single sweep using policy improvement to find the optimal policy. If the estimated optimal policy is within a certain tolerance of the previous policy, we stop. Else, we repeat the process by performing more sweeps to first improve the estimate of the optimal value function and then to improve the policy. This whole process of finding the optimal policy through independent alternations of policy evaluation and policy improvement (for value iteration or policy iteration) is called *generalized policy iteration.*

Given these differences, policy iteration is generally more computationally expensive than value iteration. However, policy iteration tends to yield a better policy than what value iteration yields. Value iteration converges faster than policy iteration and is easier to implement. Nonetheless, both algorithms are great at solving MDPs and are used in practice.

Do keep in mind that using generalized policy iteration, employing dynamic programming as we have done, while guaranteed to converge to the optimal policy, is not always the best method for solving MDPs. We are required to sweep several times through the state space to find the optimal policy.

Model-Based vs. Model-Free

In this lab, we have used a model-based approach to solve the `'FrozenLake-v1'` environment. This means that we have used the dynamics function p to find the optimal policy as well as were able to use a reward function. This is a great way to solve MDPs when we have access to the dynamics function and reward function and know that the agent gets a full observation of the state. However, in many cases, we do not have access to these functions. This is where model-free methods come in. Model-free methods still try to solve the underlying MDP, but they do not use the dynamics function or reward function to do so since they use experience to learn the optimal policy. However, we still rely on the fact that the agent gets a full observation of the current state. When the observation is not full, we have to use a different method called *partially observable MDPs* (POMDPs).

Moreover, both methods use what is called *bootstrapping*, which is the process of using estimates of value functions to improve the estimates of the same value functions we want. These two also look ahead to the future state, compute a value, and use that value to improve the current value we are seeing. The two method types are best for small finite MDPs. One thing to note is that both also are computationally expensive so that they may not be the best methods for solving large MDPs. The methods shown in these two RL labs are collectively called *tabular methods* since we used tables or arrays to store the value functions and policies.

Lastly, for these two labs, we have worked with a *stationary* MDP, which means that the dynamics function and reward function do not change over time. Even when the environment is stochastic, the probabilities do not change or at least change slow enough so that the agent can learn the optimal policy. We also assumed that we worked with a *stationary* policy, which means that the policy does not change over time. RL can get quite complicated when we have to deal with non-stationary MDPs and policies and even more so when we have to deal with POMDPs.

Problem 7. You will be comparing the model-free Q-learning algorithm and the model-based Value Iteration algorithm.

Write a function called `model_comparison()` that accepts a parameter `episodes` defaulting to 1000. Run both algorithms on the `'FrozenLake-v1'` environment using the 8×8 grid and `is_slippery=True` for however many episodes `model_comparison()` is given. For each algorithm, calculate the average total discounted reward sum for however many episodes `model_comparison()` is given. Then, write a string of 2-4 sentences comparing the two algorithms using your knowledge of RL or the lab material that explains the differences in the results. Return a tuple of 3 elements consisting of the average total discounted reward for both algorithms and the final element being the string that you wrote. For consistency, keep both `beta` parameters at `1.0` to calculate the total discounted reward. On Q-learning, employ a linear decaying epsilon.

To help, we have created a py file called `model_free_rl.py` that contains the functions `run_q_learn`, `run_simulation_table()`, and `epsilon_decay()`. The first function runs Q-learning on the `'FrozenLake-v1'` environment and saves a Q-table as npy file that will be used in the `run_simulation_table()` function to solve the `'FrozenLake-v1'` environment. There is no need to store anything from this function, so you can just run it. We give a breakdown of the parameters at the end. After you execute `run_q_learn`, you can comment out the line of code that executes the Q-learning algorithm since you no longer need to train it. Ensure the npy file was created before commenting it out. The npy file is named `q_table.npy`.

The `run_simulation_table()` function is much like the problem 6 function. It accepts the arguments `env` (Gym environment) and `beta` a float representing the discount factor that is defaulted to 1.0. This function will return the total discounted reward for having followed the policy given by Q-learning for one episode of the environment. The code of the function already loads the npy file to extract the table, so execute this function only after you have run the Q-learning algorithm and have the npy file.

Remember that `value_iteration()` only returns the value function. As such, you need to use another one of your functions to be able to extract the optimal policy from the value function array. Lastly, you will need to run the function `run_simulation()`, from problem 6, to be able to calculate the total discounted reward of the policy produced by the value iteration algorithm. You may find `ndarray.mean()` and list comprehension useful to compute the two 2 floats.

You may experiment with various model hyperparameters, but when submitting the lab, ensure that all inputs use their defaults values (except for epsilon as you will linearly decay it). The function `run_q_learn` has the following 7 inputs (in order):

- `env` (`str`): The Gym environment.
- `alpha` (`float`): The learning rate. Defaulted to `0.1`.
- `gamma` (`float`): The discount factor. Defaulted to `0.6`.
- `epsilon` (`float`): The epsilon value for the epsilon-greedy policy. Defaulted to `0.1`.
- `N` (`int`): The number of episodes to train for. Defaulted to `70_000`.
- `decay` (`bool`): whether or not to decay the epsilon value. Defaulted to `False`.
- `decay_type` (`str`): the type of model given to `epsilon_decay()` in order to calculate a decaying epsilon value (`'linear'` or `'exp'` for exponential). Defaulted to `'linear'`.

Additional Materials

We give a brief overview of some of the topics we did not cover in any of the labs that are important to reinforcement learning.

Stochastic Dynamic Programming

Dynamic programming, DP for short, is a method used to solve complex problems by breaking them down into simpler subproblems over time. DP requires that a solution to the problem is able to be constructed from solutions to its subproblems[11] and that the solved subproblems are reused several times[12]. *Dynamic optimization*, or also called dynamic programming, is the process of simplyfying a decision-making problem by using these two principles of DP. In *stochastic dynamic optimization*, or stochastic dynamic programming, we use the principles of DP to solve a decision-making problem where the outcomes are partly random and partly under the control of a decision maker. The goal in stochastic dynamic programming is to get a strategy on how to act in the face of uncertainty. This is where the MDP comes in. Using any of the Bellman equations, we can see that the value functions have both properties[13], so we can use DP to solve the MDP. This is why value iteration and policy iteration are considered DP methods.

Convergence of Value Iteration

We proof the convergence of the value iteration we used in this lab. A more general value iteration that uses more stochastic components can also be proved in a similar fashion.

A function f that is a contraction mapping has a *fixed point* p such that $f(p) = p$. Blackwell's contraction theorem can be used to show that Bellman's equation is a "fixed point" (it actually acts more like a fixed function in this case) for an operator $T : L^\infty(X;\mathbb{R}) \to L^\infty(X;\mathbb{R})$ where $L^\infty(X;\mathbb{R})$ is the set of all bounded functions:

$$T[f](s) = \max_{a\in A_s}\left\{\sum_{\bar{s}\in N_{s,a}} p(\bar{s})\cdot(r(\bar{s}) + \beta f(\bar{s}))\right\} \tag{21.12}$$

It can be shown that Equation 21.5 is the fixed "point" of our operator T. A result of contraction mappings is that there exists a unique solution to Equation 21.12, namely

$$V_*^{k+1}(s_i) = T[V_*^k](s_i) = \max_{a\in A_s}\left\{\sum_{\bar{s}\in N_{s,a}} p(\bar{s})\cdot\big(r(\bar{s}) + \beta V_*^k(\bar{s})\big)\right\} \tag{21.13}$$

where an initial guess for $V_*^0(s)$ is used. As $k \to \infty$, it is guaranteed that $(V_*^k(s)) \to v_*(s)$. Because of the contraction mapping, if $V_*^{k+1}(s) = V_*^k(s)\ \forall\, s$, we have found the true optimal value function, $v_*(s)$.

Stochastic vs Deterministic Policy

A *deterministic* policy is a function $\pi : S \to A$ that maps a state s to a single action $a \in A_s \subset A$ with certainty. That is, the agent will always take the same action a for a given s when using a deterministic policy π. This is why we can write $\pi(s) = a$ since only one action will be produced time and time again for a given state. The main advantage of a deterministic policy is the easiness in implementation and interpretation. This type of policy is best suited for environments where the agent should take the same action for a given state every single time it comes to that state or for tasks requiring precise control.

[11] This property is called the *optimal substructure* property.

[12] This property is called the *overlapping subproblems* property.

[13] Mainly, the calculation of the next state becomes the subproblem, and the fact that we can store that value for later use when it becomes the next state for another state is what meets the criteria.

On the other hand, a *stochastic* policy is a function $\pi : S \times A \to [0,1]$ where $\pi(A|S=s)$ is a possibly distinct probability distribution over the action space for a fixed state s. Thus, each state can have its own probabilistic rule for selecting actions from its own action set. Hence, $\pi(A=a|S=s) = \pi(a|s)$[14] is a value in the interval $[0,1]$ denoting the probability of taking action a in state s. The advantage of a stochastic policy is that it can capture the uncertainty of the environment but with the downside of having to learn a probability distribution for each state. Nevertheless, this allows us to learn tasks where randomness or exploration are more common.

Do note that although we denote an action a or the action space A as an input for a stochastic policy, it is an output of the policy. We use it in the input to emphasize the fact the action is uncertain as the stochastic policy can choose a completely different action. Lastly, we can think of a deterministic policy as a stochastic policy where $\pi(a|s) = 1$.

Achtung!

Note that the transition probability is not the same as the probability given by some policy. The former is a property of the environment that cannot be changed while the latter is a property of the agent that can be changed.

Stochastic Processes in RL

In this lab, we worked with a stochastic environment where the outcomes of the agent's actions are not fully in the agent's control. In the `'FrozenLake-v1'` environment, the agent's chosen direction is not always the direction the agent moves. We did not consider the agent's actions to be stochastic nor did we consider the rewards to be stochastic. The general process that can be used for more complex environments and tasks employs a stochastic policy and a stochastic reward function.

There can be a few different types of stochastic processes in RL:

- The environment: This gives us either a deterministic or stochastic transition probability. This means that the next state is always the same for a given state-action pair or that there is a probability distribution over the next states for a given state-action pair, respectively. Thus, $p(s'|s,a) = 1, \forall(s,a)$, for deterministic, and $p(s'|s,a) \leq 1, \forall(s,a)$, for stochastic.

- The policy π: This means that the agent can either have a fixed action for a given state which implies $\pi(a|s) = P(A_t = a|S_t = s) = \pi(s) = 1, \forall(s,a)$. Or, there is a probability distribution over actions for a given state so that $\pi(a|s) = P(A_t = a|S_t = s) \leq 1, \forall s \in S$.

- The reward function: In the deterministic case, the reward function returns the same reward for the same input and is either dependent on just the current state-action pair (s,a) or is dependent on the next state and current state-action pair which is the triple (s', s, a). For stochastic, the reward function returns a probability distribution over rewards for a given (s,a) or (s', s, a).

The dynamics function p is meant to capture the true stochastic nature of the world or environments we live in. That is, it works with a stochastic MDP, policy, and reward. When dealing with a deterministic reward function, we typically use the transition probability rather than the dynamics function.

[14] This probability can also be written as $\pi(s,a)$.

With a true stochastic environment, we get the following Bellman equation for $v_\pi(s)$ as

$$\begin{aligned} v_\pi(s) = v(s) &= \mathbb{E}\left[\sum_{k=0}^{T} \beta^k r_k \middle| S_0 = s\right] && (21.14a)\\ &= \sum_{a\in A_s} \pi(a|s) \sum_{s'\in S^+} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \beta v_\pi(s')\Big]. && (21.14b)\end{aligned}$$

Similary, the state-action quality function $q_\pi(s, a)$ can be defined as

$$\begin{aligned} q_\pi(s, a) = q(s, a) &= \mathbb{E}\left[\sum_{k=0}^{T} \beta^k r_t \middle| S_0 = s, A_0 = a\right] && (21.15a)\\ &= \sum_{s'\in S^+} \sum_{r\in\mathcal{R}} p(s', r|s, a)\left[r + \beta \sum_{a'\in A_{s'}} \pi(a'|s') q_\pi(s', a')\right]. && (21.15b)\end{aligned}$$

With these two, we can form the relationship

$$v_\pi(s) = \sum_{a\in A_s} \pi(a|s) q_\pi(s, a). \tag{21.16}$$

We can then get the Bellman optimality for v_*

$$\begin{aligned} v_*(s) &= \max_{a\in A_s} q_{\pi_*}(s, a)\\ &= \max_{a\in A_s} \sum_{s'\in S^+} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \beta v_*(s')\Big], && (21.17)\end{aligned}$$

and the action-value as

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[r_0 + \beta \max_{a'\in A_s} q_*(s_1, a')|S_0 = s, A_0 = a]\\ &= \sum_{s'\in S^+} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \beta \max_{a'\in A_{s'}} q_*(s', a')\Big]. && (21.18)\end{aligned}$$

The remainder of the other functions for a stochastic process that are similar to those used in the lab can be derived from these equations.

Deterministic Reward Function

In the lab, we assumed that the reward function was deterministic. Specifically, we worked with a function that had a reward dependent not only on the current state-action pair (s, a) but also on the next state s'. This was the reward function of three inputs $r(s', s, a)$.

In other cases, the reward function can be dependent only on the current state-action pair (s, a). Using the dynamics function p, we can also compute the reward function of two inputs $r : S \times A \to \mathbb{R}$ as

$$r(s, a) = r_t(s, a) = \mathbb{E}[R_t|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in S} p(s', r|s, a). \tag{21.19}$$

This tells us the expected reward for taking action a in state s at timestep t.

Do note that $r(s', s, a)$ and $r(s, a)$ are not necessarily the same as they have different uses. When we care about going to a specific state after taking a specific action, we use $r(s', s, a)$. When we care about the reward for taking a specific action in a specific state, we use $r(s, a)$.

Solving the Optimization Problem

For each state, there may be multiple actions that maximize the value function in the Bellman optimality equation (recall the definition of argmax). Thus, any policy that assigns a nonzero probability to these actions is an optimal policy. That is, any policy that is *greedy*[15] with respect to the optimal value function is an optimal policy. A greedy policy only selects the action that maximizes the value function only in the short term, specifically only optimizing the expected future rewards of the next timestep, but the nature of v_* already considers the long-term rewards of all possible future behavior making it available to each state immediately. Hence, given the Bellman optimality equation for $v_*(s)$ (21.17), we can simply use

$$\pi_* = \pi_*(s) = \underset{a \in A_s}{\text{argmax}}\ v_*(s). \tag{21.20}$$

When we have the optimal action-value function, we can use

$$\pi_* = \pi_*(s) = \underset{a \in A_s}{\text{argmax}}\ q_*(s,a). \tag{21.21}$$

In this case, there is no need to perform some greedy search since the optimal action-value function selects optimal actions without having to know anything about possible successor states and their respective values. In general, the optimal policy is usually deterministic since we only care about choosing one of the many optimal actions available (or the only optimal action available). Should we have started with a stochastic policy and want it to remain stochastic, we need only assign each optimal action any probability we want as long as the suboptimal actions are given a probability of 0. This will keep the policy stochastic but still optimal. But, we can always just choose one of the optimal actions and assign it a probability of 1 to make the stochastic policy deterministic.

Closer Look Into Policy Improvement

In the section on policy improvement, we mentioned that policy improvement finds the optimal policy. We had supposed that there was a new policy π' that was better than the old policy π. But what if there was a new policy that is just as good as the old policy but not better than the current policy π? Then, $v_\pi = v_{\pi'}$, so that from

$$\pi'(a|s) = \underset{a \in A_s}{\text{argmax}}\ q_\pi(s,a) = \underset{a \in A_s}{\text{argmax}} \sum_{s' \in S^+} \sum_{r \in \mathcal{R}} p(s', r|s,a)[r + \beta v_\pi(s')]$$

, we have

$$v_{\pi'} = \max_{a \in A_s} \sum_{s' \in S^+} \sum_{r \in \mathcal{R}} p(s', r|s,a)[r + \gamma v_{\pi'}(s')].$$

This is the Bellman optimality equation so that π' is an optimal policy as is π. Thus policy improvement does find the optimal policy except in the case where the original policy is optimal so that we do not have to improve the current policy.

When Should I Use RL?

Tom Mitchel defined machine learning as

[15] By greedy we mean any search or decision procedure that selects alternatives based on local or immediate considerations without considering the possibility that such a selection may prevent future access to even better alternatives.

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

There are, in general, three types of learning: supervised learning, unsupervised learning, and reinforcement learning.

In *supervised learning*, the computer is given a set of inputs and outputs that have been already been given some context of a relationship and is asked to learn the mapping between the two. Whereas in *unsupervised learning*, the computer is given a set of inputs and outputs and is asked to find the structure or patterns in the data without having any context of any sort of relationship in the data. While these two types of learning are great for many tasks, they do not work well when the computer has to interact with the environment to learn a task. In RL, we are concerned about finding any structure or patterns nor are we concerned about finding a mapping between inputs and outputs. We are strictly concerned with how the computer ought to take actions in an environment to maximize some notion of a cumulative reward.

Typically RL is better used when the algorithm needs to make sequential decisions in an environment. This is because RL is great at learning how to make decisions in an environment where the agent can interact with the environment. Typically RL is best in these types of environments where we have an existing decision-making model that we want to improve or when we have a decision-making model that we want to learn from scratch through interaction. RL is also great when the agent can receive feedback from the environment in the form of a reward. Thus, we need to be able to define a reward function that tells the agent how well it is doing and not run into the *reward engineering problem*.

Morever, RL is great when the agent can learn from its mistakes and improve its decision-making process. Thus, when we can afford to have the agent make mistakes and learn from them, online RL is a great tool to use. But when we cannot afford to have the agent make mistakes, offline RL is a better tool to use so we must be able to have a dataset of the environment that the agent can learn from.

Challenges in RL

We talked about the exploration-exploitation trade-off in the last lab. RL faces other problems that must be dealt with whenever you are formulating a problem as an RL problem. Here are some of the challenges in RL:

- *Credit Assignment Problem*: This is the challenge of determining which actions an agent took that led to a particular reward. This is a problem because the agent may have taken many actions before receiving a reward. The agent must determine which of these actions led to the reward. It could be that the reward was due to a combination of actions or that the reward was due to an action that was taken a long time ago. Moreover, this also does not mean that other actions were not important as those actions could have set up the agent to receive the reward while not directly giving the reward.

- *Reward Engineering Problem*: This is the process of designing a good reward function that encourages the desired behavior in the agent. The reward should reflect the desired goal we want the agent to achieve. The reward is not a place to give the agent new information about the environment. It should only be used to tell the agent how well it is doing.

- *Generalization Problem*: This is the ability of an agent to apply what it has learned to new and previously unseen situations. This is a problem because the agent may have learned a policy that works well in the training environment but does not work well in the testing environment. It also arises when the agent has to learn a policy in an arbitrarily large state space. The agent must be able to generalize its policy to all states in the state space but may not have visited all states during training.

- *Sample Efficiency Problem*: This refers to the ability of an RL agent to learn an optimal policy with a limited number of interactions with the environment. Many RL algorithms require a large number of interactions with the environment (or a large dataset of the environment) to learn an optimal policy. This is a problem because in many real-world applications, the agent cannot interact with the environment an unlimited number of times.

Futher Reading and Resources

Reinforcement learning is a vast field with many different algorithms and techniques and applications. The following are some resources that can help you learn more about reinforcement learning. Note that some of these are available online for free through various universities or by the publishers or authors themselves.

- *Reinforcement Learning: An Introduction*, 2nd edition, 2018, by Richard S. Sutton and Andrew G. Barto is a classic book covering the basics of reinforcement learning and algorithms. It also gives a good overview of the field and the math behind it. The only prerequisite knowledge is the material you have learned in Volume 2 textbook.

- *Algorithms for Reinforcement Learning*, 2009, by Csaba Szepesvári is a book that covers many different algorithms in reinforcement learning strictly and rigorously from a mathematical perspective. It does not contain many implementations or applications, but it does cover the strengths and weaknesses of many algorithms as well as the math behind them and what is known or unknown about them. The material therein is more advanced than the Sutton and Barto book, so knowledge of Volume 1 and 2 textbooks is recommended.

- *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, 2nd edition, 2022, by Steven L. Brunton and J. Nathan Kutz is a book that brings together machine learning, mathematics, and physics to integrate modeling of dynamical systems. The book contains chapters on reinforcement learning and deep reinforcement learning, as well as many other topics in machine learning. Dr. Brunton has a YouTube channel where he covers some of the material in the book, including a playlist on reinforcement learning.

- *Reinforcement Learning and Optimal Control*, 2019, by Dimitri P. Bertsekas is a book that covers reinforcement learning and optimal control from a mathematical perspective and using dynamic programming.

Part II

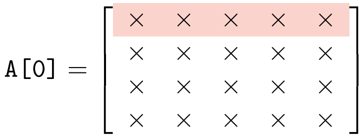
Appendices

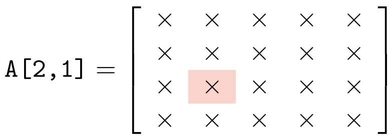
A NumPy Visual Guide

Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

Data Access

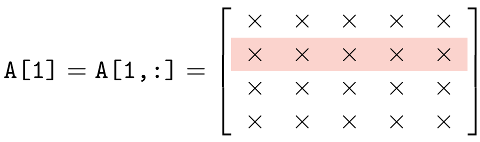
The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

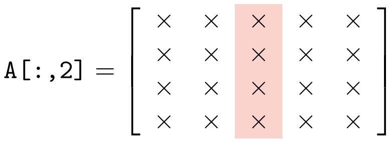


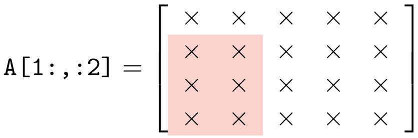


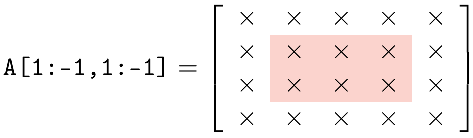
Slicing

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."









Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$\texttt{A} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad\qquad \texttt{B} = \begin{bmatrix} \ast & \ast & \ast \\ \ast & \ast & \ast \\ \ast & \ast & \ast \end{bmatrix}$$

$$\texttt{np.hstack((A,B,A))} = \begin{bmatrix} \times & \times & \times & \ast & \ast & \ast & \times & \times & \times \\ \times & \times & \times & \ast & \ast & \ast & \times & \times & \times \\ \times & \times & \times & \ast & \ast & \ast & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((A,B,A))} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \ast & \ast & \ast \\ \ast & \ast & \ast \\ \ast & \ast & \ast \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

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()`.

$$\texttt{x} = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad\qquad \texttt{y} = \begin{bmatrix} \ast & \ast & \ast & \ast \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & \ast & \ast & \ast & \ast & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ \ast & \ast & \ast & \ast \\ \times & \times & \times & \times \end{bmatrix} \qquad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & \ast & \times \\ \times & \ast & \times \\ \times & \ast & \times \\ \times & \ast & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

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

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

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

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`.

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

$$\texttt{A.sum(axis=0)} = \left[\begin{array}{c|c|c|c} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{array}\right] = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \left[\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \end{array}\right] = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

B Matplotlib Syntax and Customization Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

Axes objects are usually managed through the functions plt.subplot() and plt.subplots(). The function subplot() is used as plt.subplot(nrows, ncols, plot_number). Note that if the inputs for plt.subplot() are all integers, the commas between the entries can be omitted. For example, plt.subplot(3,2,2) can be shortened to plt.subplot(322).

The function subplots() is used as plt.subplots(nrows, ncols), and returns a Figure object and an array of Axes. This array has the shape (nrows, ncols), and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

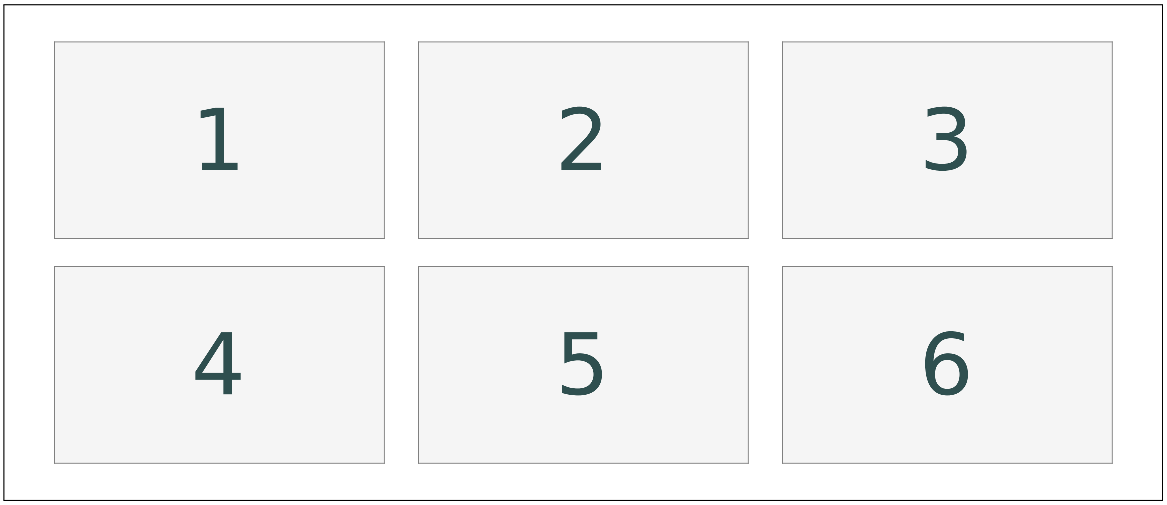


Figure B.1: The layout of subplots with plt.subplot(2,3,i) (2 rows, 3 columns), where i is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

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

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

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

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

Achtung!

Be careful not to mix up the following similarly-named functions:

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

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

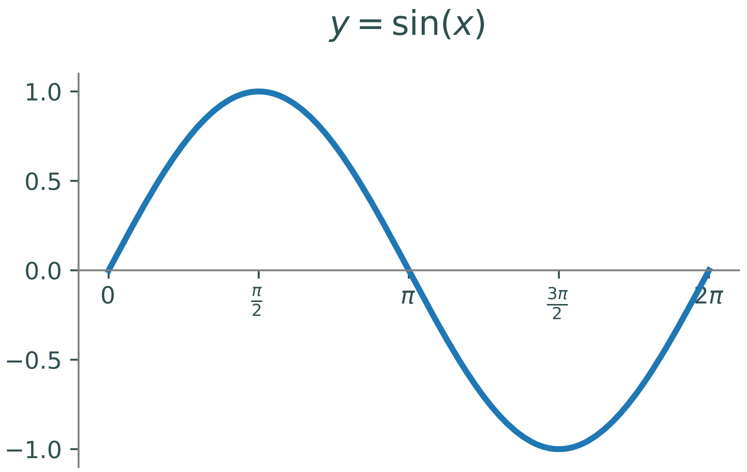


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

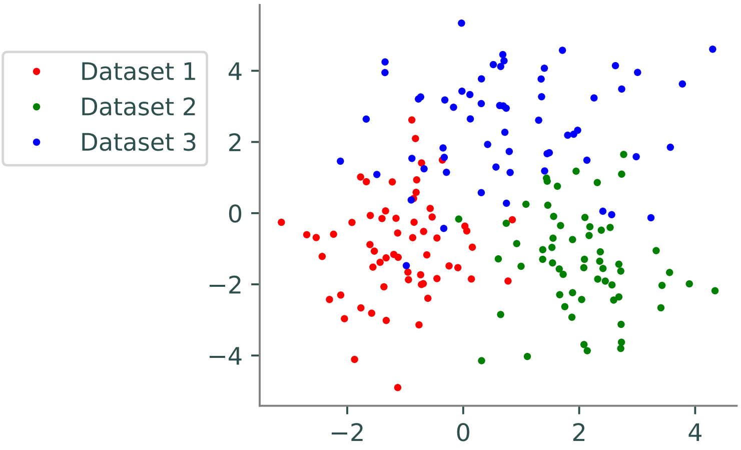


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white
'C0' - 'C9'	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate(’text’,(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc=’best’)
`text()`	Add text at a given position on the plot	text(x,y,’text’)
`title()`	Add a title to the plot	title(’text’)
`suptitle()`	Add a title to the figure	suptitle(’text’)
`xlabel()`	Add a label to the x-axis	xlabel(’text’)
`ylabel()`	Add a label to the y-axis	ylabel(’text’)

Table B.4: Text and annotation functions in Matplotlib

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')
```

[1] See `http://www.latex-project.org/` for more information.

```
# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
ˆ	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
```

```
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

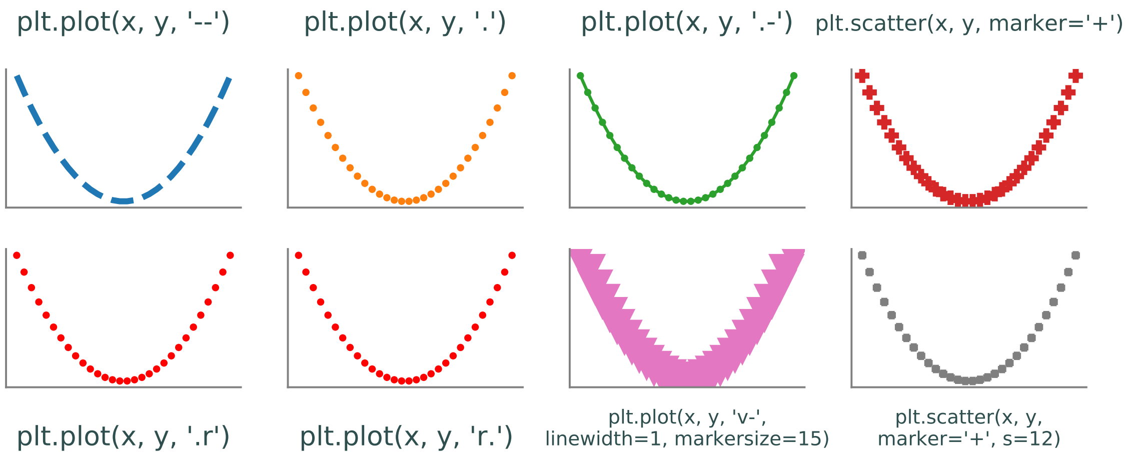


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to ’center’ or ’edge’ to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using boxplot() with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting vert=False.

Scatter and hexbin plots

Scatterplots can be created using either plot() or scatter(). Generally, it is simpler to use plot(), although there are some cases where scatter() is better. In particular, scatter() allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the c or s arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function hexbin(). Like plot() and scatter(), this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the pcolormesh() function. Contour plots are created using contour() or contourf(), with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function np.meshgrid():

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the cmap parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a matplotlib.colors.LogNorm object in the norm parameter of these functions.

With pcolormesh(), it is also necessary to pass shading='auto' or shading='nearest' to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

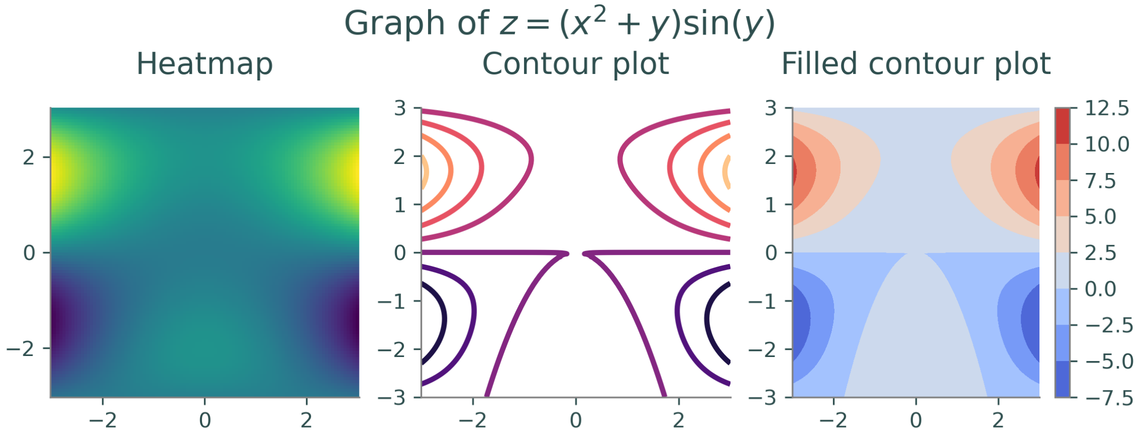


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

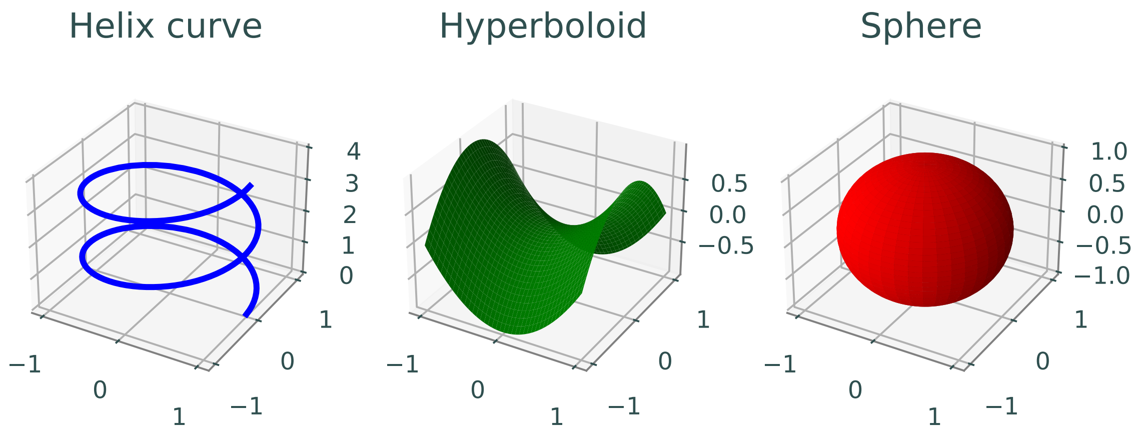


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[ADH$^+$01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.

[BK22] Steven L. Brunton and J. Nathan Kutz. *Data-driven science and engineering: Machine learning, Dynamical Systems, and Control (2nd edition).* Cambridge University Press, 2022.

[BL04] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[Gei60] Theodor Seuss Geisel. *Green eggs and ham.* Beginner Books, 1960.

[Hu23] Michael Hu. *The Art of Reinforcement Learning: Fundamentals, Mathematics, and Implementations with Python.* Apress, 2023.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[KM72] Victor Klee and George J. Minty. How good is the simplex algorithm? In *Inequalities*, volume 3, pages 159–175. Academic Press, 1972.

[Nas00] J.C. Nash. The (dantzig) simplex method for linear programming. *Computing in Science and Engineering*, 2(1):29–31, 2000.

[Oli06] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction (2nd edition).* MIT Press, 2018.

[VD10] Guido VanRossum and Fred L Drake. *The python language reference.* Python software foundation Amsterdam, Netherlands, 2010.

[VHL06] Philipp Von Hilgers and Amy N Langville. The five greatest applications of markov chains. In *Proceedings of the Markov Anniversary Meeting, Boston Press, Boston, MA*. Citeseer, 2006.