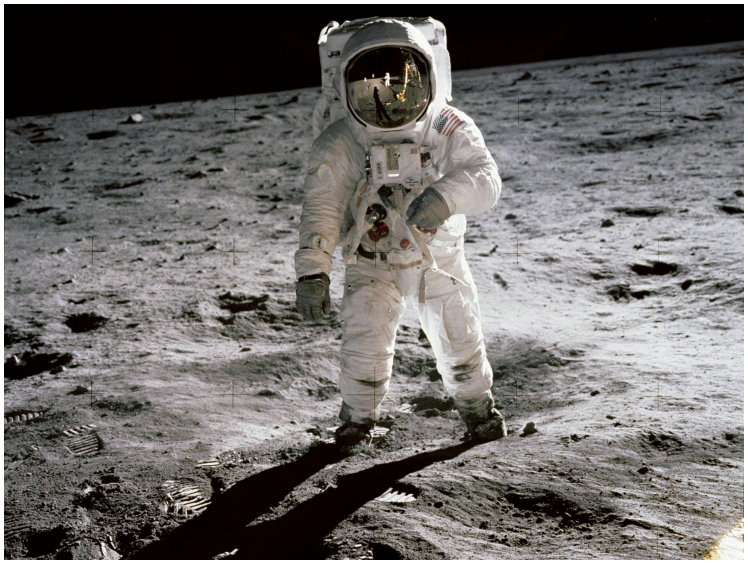
Labs for Foundations of Applied Mathematics

Volume I
Mathematical Analysis



List of Contributors

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

J. Bejarano
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

A. Frandsen
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Fuhriman
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

K. Jacobson
Brigham Young University

J. Leete
Brigham Young University

J. Lytle
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

C. Robertson
Brigham Young University

R. Sandberg
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

T. Thompson
Brigham Young University

M. Victors
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

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

Contents

Part I

Labs

1 Linear Transformations

Lab Objective: *Linear transformations are the most basic and essential operators in vector space theory. In this lab we visually explore how linear transformations alter points in $\mathbb{R}^2$. We then empirically explore the computational cost of applying linear transformations via matrix multiplication.*

Linear Transformations

A *linear transformation* is a mapping between vector spaces that preserves addition and scalar multiplication. More precisely, let V and W be vector spaces over a common field $\mathbb{F}$. A map $L : V \to W$ is a linear transformation from V into W if

$$L(a\mathbf{x}_1 + b\mathbf{x}_2) = aL\mathbf{x}_1 + bL\mathbf{x}_2$$

for all vectors $\mathbf{x}_1,\ \mathbf{x}_2 \in V$ and scalars $a,\ b \in \mathbb{F}$.

Every linear transformation L from an m-dimensional vector space into an n-dimensional vector space can be represented by an $m \times n$ matrix A, called the *matrix representation* of L. To apply L to a vector $\mathbf{x}$, left multiply by its matrix representation. This results in a new vector $\mathbf{x}'$, where each component is some linear combination of the elements of $\mathbf{x}$. For linear transformations from $\mathbb{R}^2$ to $\mathbb{R}^2$, this process has the following form.

$$A\mathbf{x} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{x}'$$

Linear transformations can be interpreted geometrically. To demonstrate this, we examine a set of points that collectively form a picture of a horse, stored in the file `horse.npy`. The coordinate pairs are organized by column, so the array has two rows: one for x-coordinates, and one for y-coordinates.

$$\begin{bmatrix} x_1 & x_2 & x_3 & \dots \\ y_1 & y_2 & y_3 & \dots \end{bmatrix}$$

Use `np.load()` to extract the array from the `.npy` file, then plot the points as individual pixels. See Figure 1.1 for the result.

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

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

# Load the array from the .npy file.
>>> data = np.load("horse.npy")

# Plot the x row against the y row with black pixels.
>>> plt.plot(data[0], data[1], 'k,')

# Set the window limits to [-1, 1] by [-1, 1] and make the window square.
>>> plt.axis([-1,1,-1,1])
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```

Types of Linear Transformations

Linear transformations from $\mathbb{R}^2$ into $\mathbb{R}^2$ can be classified in a few ways.

- **Stretch**: Stretches or compresses the vector along each axis. The matrix representation is diagonal:

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$$

  If $a = b$, the transformation is called a *dilation*. The stretch in Figure 1.1 uses $a = \frac{1}{2}$ and $b = \frac{6}{5}$ to compress the x-axis and stretch the y-axis.

- **Shear**: Slants the vector by a scalar factor horizontally or vertically. There are two matrix representations:

$$\text{horizontal shear: } \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \qquad \text{vertical shear: } \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix}$$

  Horizontal shears skew the x-coordinate of the vector while vertical shears skew the y-coordinate. Figure 1.1 has a horizontal shear with $a = \frac{1}{2}$.

- **Reflection**: Reflects the vector about a line that passes through the origin. The reflection about the line spanned by the vector $[a, b]^\mathsf{T}$ has the matrix representation

$$\frac{1}{a^2 + b^2} \begin{bmatrix} a^2 - b^2 & 2ab \\ 2ab & b^2 - a^2 \end{bmatrix}.$$

  The reflection in Figure 1.1 reflects the image about the y-axis ($a = 0$, $b = 1$).

- **Rotation**: Rotates the vector around the origin. A counterclockwise rotation of θ radians has the following matrix representation:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

  A negative value of θ performs a clockwise rotation. Choosing $\theta = \frac{\pi}{2}$ produces the rotation in Figure 1.1.

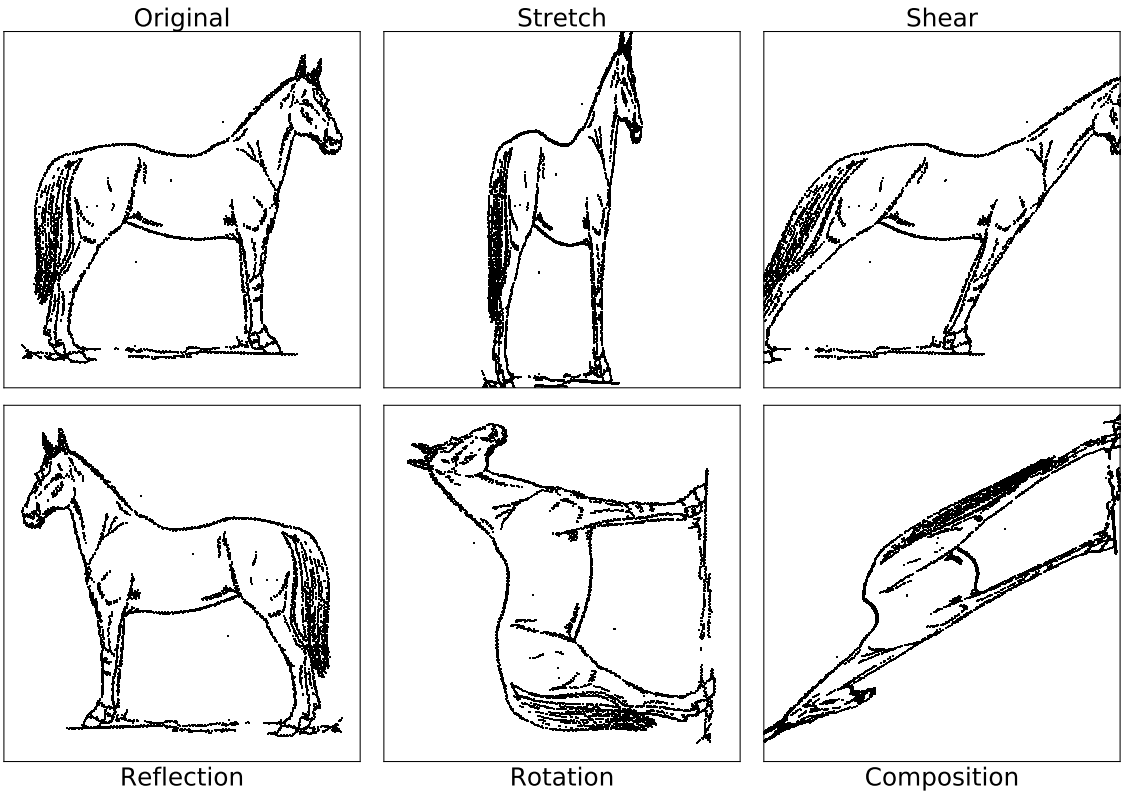


Figure 1.1: The points stored in `horse.npy` under various linear transformations.

Problem 1. Write a function for each of the linear transformations listed above. Each function should accept an array to transform and the scalars that define the transformation (a and b for stretch, shear, and reflection, and θ for rotation). Construct the matrix representation and left multiply it with the input array. Return the transformed array.

To test your functions, consider writing a separate function that plots two arrays (the original and the transformed) for a side-by-side comparison.

Note

Reflections and rotations are two ways to implement the QR decomposition, an important matrix factorization that will be studied in another lab.

Compositions of Linear Transformations

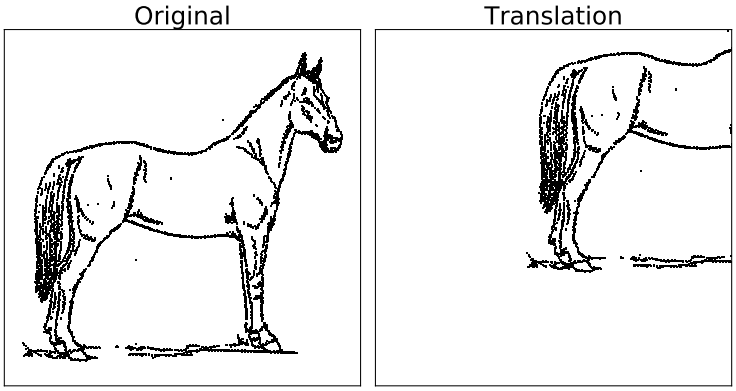
Let V, W, and Z be finite-dimensional vector spaces. If $L : V \to W$ and $K : W \to Z$ are linear transformations with matrix representations A and B, respectively, then the *composition* $KL : V \to Z$ is also a linear transformation, and its matrix representation is the matrix product BA.

For example, if S is a matrix representing a shear and R is a matrix representing a rotation, then RS represents a shear followed by a rotation. In fact, any linear transformation $L : \mathbb{R}^2 \to \mathbb{R}^2$ is a composition of the four transformations discussed above. Figure 1.1 displays the composition of all four previous transformations, applied in order (stretch, shear, reflection, then rotation).

Affine Transformations

All linear transformations map the origin to itself. An *affine transformation* is a mapping between vector spaces that preserves the relationships between points and lines, but that may not preserve the origin. Every affine transformation T can be represented by a matrix A and a vector $\mathbf{b}$. To apply T to a vector x, we calculate $A\mathbf{x} + \mathbf{b}$. If $\mathbf{b} = \mathbf{0}$ then the transformation is linear, and if $A = I$ but $\mathbf{b} \neq \mathbf{0}$ then it is called a *translation*.

For example, if T is the translation with $\mathbf{b} = \left[\frac{3}{4}, \frac{1}{2}\right]^\mathsf{T}$, then applying T to an image will shift it right by $\frac{3}{4}$ and up by $\frac{1}{2}$. This translation is illustrated below.



Affine transformations include all compositions of stretches, shears, rotations, reflections, and translations. For example, if S represents a shear and R a rotation, and if $\mathbf{b}$ is a vector, then $RS\mathbf{x}+\mathbf{b}$ shears, then rotates, then translates $\mathbf{x}$.

Modeling Motion with Affine Transformations

Affine transformations can be used to model particle motion, such as a planet rotating around the sun. Let the sun be the origin, the planet's location at time t be given by the vector $\mathbf{p}(t)$, and suppose the planet has angular momentum ω (a measure of how fast the planet goes around the sun). To find the planet's position at time t given the planet's initial position $\mathbf{p}(0)$, rotate the vector $\mathbf{p}(0)$ around the origin by $t\omega$ radians. Thus if $R(\theta)$ is the matrix representation of the linear transformation that rotates a vector around the origin by θ radians, then

$$\mathbf{p}(t) = R(t\omega)\mathbf{p}(0).$$

Composing the rotation with a translation shifts the center of rotation away from the origin, yielding more complicated motion.

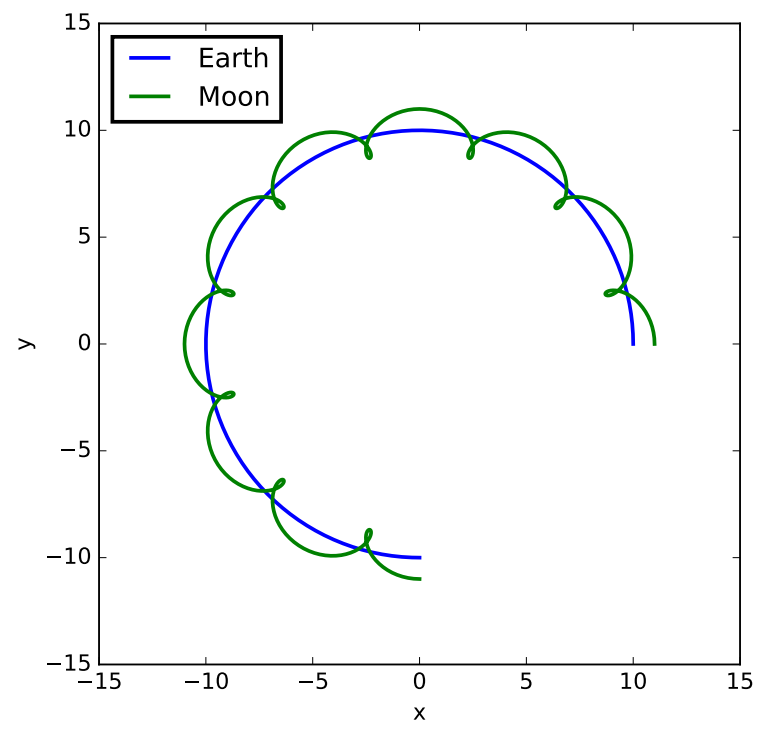
Problem 2. The moon orbits the earth while the earth orbits the sun. Assuming circular orbits, we can compute the trajectories of both the earth and the moon using only linear and affine transformations.

Assume an orientation where both the earth and moon travel counterclockwise, with the sun at the origin. Let $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ be the positions of the earth and the moon at time t, respectively, and let ω_e and ω_m be each celestial body's angular momentum. For a particular time t, we calculate $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ with the following steps:

1. Compute $\mathbf{p}_e(t)$ by rotating the initial vector $\mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_e$ radians.
2. Calculate the position of the moon relative to the earth at time t by rotating the vector $\mathbf{p}_m(0) - \mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_m$ radians.
3. To compute $\mathbf{p}_m(t)$, translate the vector resulting from the previous step by $\mathbf{p}_e(t)$.

Write a function that accepts a final time T and the angular momenta ω_e and ω_m. Assuming initial positions $\mathbf{p}_e(0) = (10, 0)$ and $\mathbf{p}_m(0) = (11, 0)$, plot $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ over the time interval $t \in [0, T]$.

The moon travels around the earth approximately 13 times every year. With $T = \frac{3\pi}{2}$, $\omega_e = 1$, and $\omega_m = 13$, your plot should resemble the following figure (use `plt.gca().set_aspect("equal")` to fix the aspect ratio).



Timing Matrix Operations

Linear transformations are easy to perform through simple matrix multiplication. However, performing matrix multiplication with very large matrices can strain a machine's time and memory constraints. For the remainder of this lab we take an empirical approach in exploring how much time and memory different matrix operations require.

Timing Code

The `time` module in the standard library include functions for dealing with time. The module's `time()` function measures the number of seconds from a fixed starting point, called "the Epoch" (January 1, 1970 for Unix machines).

```
>>> import time
>>> time.time()
1466609325.819298
```

The `time()` function[1] is useful for measuring how long it takes for code to run: record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed.

```
>>> def time_for_loop(iters):
...     """Time how long it takes to go through 'iters' iterations of nothing.
    """
...     start = time.time()         # Clock the starting time.
...     for _ in range(int(iters)):
...         pass
...     end = time.time()           # Clock the ending time.
...     return end - start          # Report the difference.
...
>>> time_for_loop(1e5)              # 1e5 = 100000.
0.007936954498291016
>>> time_for_loop(1e7)              # 1e7 = 10000000.
0.8008430004119873
```

The standard library's `timeit` module is built specifically to time code and has more sophisticated tools than the `time` module. The `timeit()` function accepts a function handle (the name of the function to run) and the number of times to run it. Additionally, in IPython the quick command `%timeit` uses `timeit.timeit()` to quickly time a single line of code.

```
In [1]: import timeit
In [2]: def for_loop():
   ...:     """Go through 1e7 iterations of nothing."""
   ...:     for _ in range(int(1e7)):
   ...:         pass

In [3]: timeit.timeit(for_loop, number=5) / 5.
Out[3]: 0.8081045627593995
```

[1]The `clock()` function is similar to `time()`, but it records more precision on Windows machines.

```
In [4]: %timeit for_loop()
1 loop, best of 3: 801 ms per loop
```

Timing an Algorithm

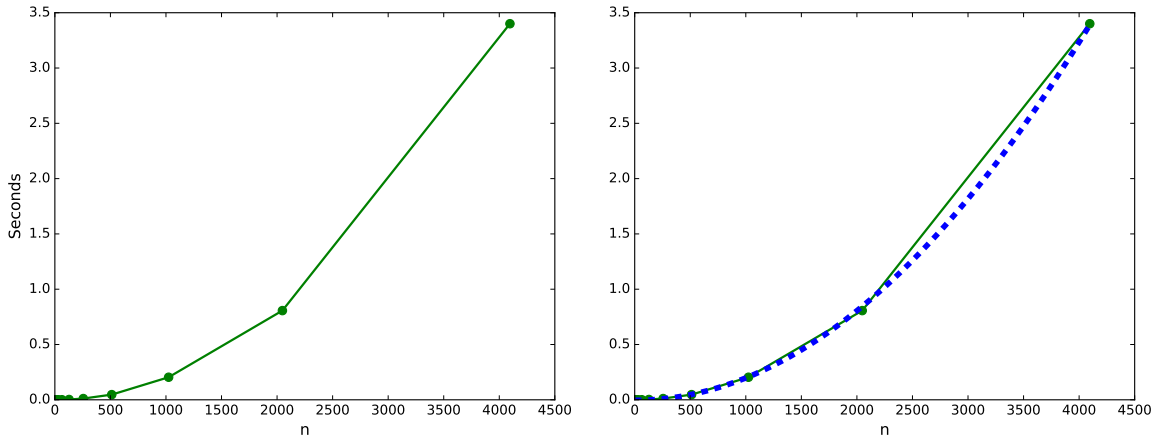
Most algorithms have at least one input that dictates the size of the problem to be solved. For example, the following functions take in a single integer n and produce a random vector of length n as a list or a random $n \times n$ matrix as a list of lists.

```
from random import random
def random_vector(n):       # Equivalent to np.random.random(n).tolist()
    """Generate a random vector of length n as a list."""
    return [random() for i in xrange(n)]

def random_matrix(n):       # Equivalent to np.random.random((n,n)).tolist()
    """Generate a random nxn matrix as a list of lists."""
    return [[random() for j in xrange(n)] for i in xrange(n)]
```

Executing `random_vector(n)` calls `random()` n times, so doubling n should about double the amount of time `random_vector(n)` takes to execute. By contrast, executing `random_matrix(n)` calls `random()` n^2 times (n times per row with n rows). Therefore doubling n will likely more than double the amount of time `random_matrix(n)` takes to execute, especially if n is large.

To visualize this phenomenon, we time `random_matrix()` for $n = 2^1,\ 2^2,\ \ldots,\ 2^{12}$ and plot n against the execution time. The result is displayed below on the left.

```
>>> domain = 2**np.arange(1,13)
>>> times = []
>>> for n in domain:
...     start = time.time()
...     random_matrix(n)
...     times.append(time.time() - start)
...
>>> plt.plot(domain, times, 'g.-', linewidth=2, markersize=15)
>>> plt.xlabel("n", fontsize=14)
>>> plt.ylabel("Seconds", fontsize=14)
>>> plt.show()
```



The figure on the left shows that the execution time for `random_matrix(n)` increases quadratically in n. In fact, the blue dotted line in the figure on the right is the parabola $y = an^2$, which fits nicely over the timed observations. Here a is a small constant, but it is much less significant than the exponent on the n. To represent this algorithm's growth, we ignore a altogether and write `random_matrix(n)` $\sim n^2$.

Problem 3. Let A be an $m \times n$ matrix with entries a_{ij}, $\mathbf{x}$ be an $n \times 1$ vector with entries x_k, and B be an $n \times p$ matrix with entries b_{ij}.

The matrix-vector product $A\mathbf{x} = \mathbf{y}$ is a new $m \times 1$ vector and the matrix-matrix product $AB = C$ is a new $m \times p$ matrix. The entries y_i of $\mathbf{y}$ and c_{ij} of C are determined by the following formulas:

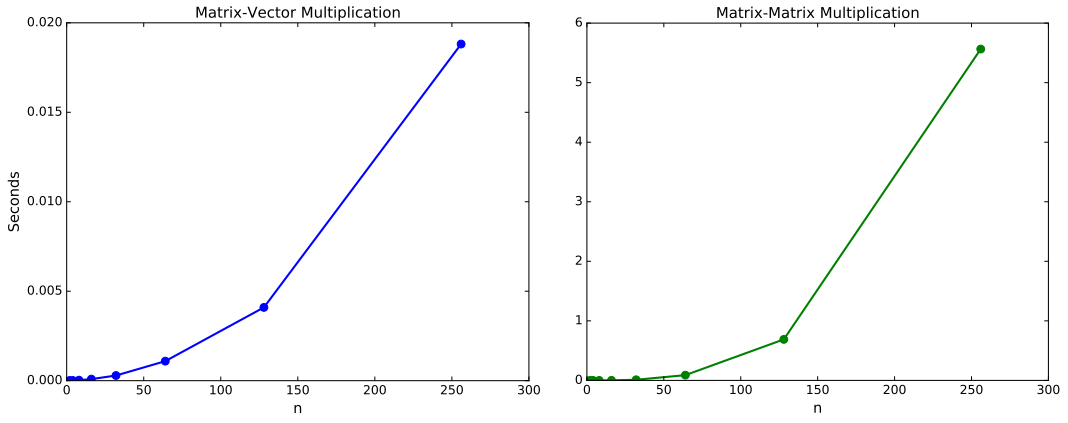
$$y_i = \sum_{k=1}^{n} a_{ik} x_k \qquad\qquad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Below, we implement these multiplication formulas without using NumPy.

```
def matrix_vector_product(A, x):    # Equivalent to np.dot(A,x).tolist()
    """Compute the matrix-vector product Ax as a list."""
    m, n = len(A), len(x)
    return [sum([A[i][k] * x[k] for k in range(n)]) for i in range(m)]

def matrix_matrix_product(A, B):    # Equivalent to np.dot(A,B).tolist()
    """Compute the matrix-matrix product AB as a list of lists."""
    m, n, p = len(A), len(B), len(B[0])
    return [[sum([A[i][k] * B[k][j] for k in range(n)])
                                    for j in range(p) ]
                                    for i in range(m) ]
```

Use `time.time()`, `timeit.timeit()`, or `%timeit` to time each of these functions with increasingly large inputs. Generate the inputs A, $\mathbf{x}$, and B with `random_matrix()` and `random_vector()` (so each input will be $n \times n$ or $n \times 1$). Only time the multiplication functions, not the generating functions.

Report your findings in a single figure with two subplots: one with matrix-vector times, and one with matrix-matrix times. Choose a domain for n so that your figure accurately describes the growth, but avoid values of n that lead to execution times of more than 1 minute. Your figure should resemble the following plots.



Logarithmic Plots

The two plots from Problem 3 look similar, but the actual execution times differ greatly. To adequately compare the two, we need to view the results differently.

A *logarithmic plot* uses a logarithmic scale—with values that increase exponentially, such as 10^1, 10^2, 10^3, ...—on one or both of its axes. The three kinds of log plots are listed below.

- **log-lin**: the x-axis uses a logarithmic scale but the y-axis uses a linear scale.
  Use `plt.semilogx()` instead of `plt.plot()`.
- **lin-log**: the x-axis is uses a linear scale but the y-axis uses a log scale.
  Use `plt.semilogy()` instead of `plt.plot()`.
- **log-log**: both the x and y-axis use a logarithmic scale.
  Use `plt.loglog()` instead of `plt.plot()`.

Since the domain $n = 2^1, 2^2, \ldots$ is a logarithmic scale and the execution times increase quadratically, we visualize the results of the previous problem with a log-log plot. The default base for the logarithmic scales on logarithmic plots in Matplotlib is 10. To change the base to 2 on each axis, specify the keyword arguments `basex=2` and `basey=2`.

Suppose the domain of n values are stored in `domain` and the corresponding execution times for `matrix_vector_product()` and `matrix_matrix_product()` are stored in `vector_times` and `matrix_times`, respectively. The following code produces Figure 1.5.

```
>>> plt.subplot(121)        # Plot both curves on a lin-lin plot.
>>> plt.plot(domain, vector_times, 'b.-', lw=2, ms=15, label="Matrix-Vector")
>>> plt.plot(domain, matrix_times, 'g.-', lw=2, ms=15, label="Matrix-Matrix")
>>> plt.legend(loc="upper left")
```

```
>>> plot.subplot(122)       # Plot both curves on a base 2 log-log plot.
>>> plt.loglog(domain, vector_times, 'b.-', basex=2, basey=2, lw=2, ms=15)
>>> plt.loglog(domain, matrix_times, 'g.-', basex=2, basey=2, lw=2, ms=15)
>>> plt.show()
```

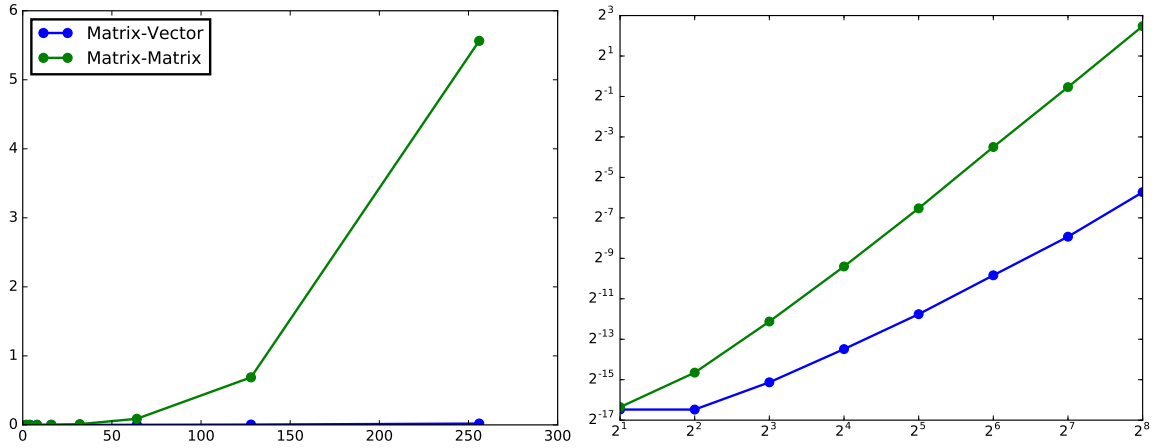


Figure 1.5

In the log-log plot, the slope of the `matrix_matrix_product()` line is about 3 and the slope of the `matrix_vector_product()` line is about 2. This reflects the fact that matrix-matrix multiplication (which uses 3 loops) is $\sim n^3$, while matrix-vector multiplication (which only has 2 loops) is only $\sim n^2$.

Problem 4. NumPy is built specifically for fast numerical computations. Repeat the experiment of Problem 3, timing the following operations:

- matrix-vector multiplication with `matrix_vector_product()`.
- matrix-matrix multiplication with `matrix_matrix_product()`.
- matrix-vector multiplication with `np.dot()`.
- matrix-matrix multiplication with `np.dot()`.

Create a single figure with two subplots: one with all four sets of execution times on a regular linear scale, and one with all four sets of execution times on a log-log scale. Compare your results to Figure 1.5.

NOTE

Problem 4 shows that **matrix operations are significantly faster in NumPy than in plain Python**. Matrix-matrix multiplication grows cubically regardless of the implementation; however, with lists the times grows at a rate of an^3 while with NumPy the times grow at a rate of bn^3, where a is much larger than b. NumPy is more efficient for several reasons:

1. Iterating through loops is very expensive. Many of NumPy's operations are implemented in C, which are much faster than Python loops.
2. Arrays are designed specifically for matrix operations, while Python lists are general purpose.
3. NumPy takes careful advantage of computer hardware, efficiently using different levels of computer memory.

However, in Problem 4, the execution times for matrix multiplication with NumPy seem to increase somewhat inconsistently. This is because the fastest layer of computer memory can only handle so much information before the computer has to begin using a larger, slower layer of memory.

Additional Material

Image Transformation as a Class

Consider organizing the functions from Problem 1 into a class. The constructor might accept an array or the name of a file containing an array. This structure would makes it easy to do several linear or affine transformations in sequence.

```
>>> horse = ImageTransformer("horse.npy")
>>> horse.stretch(.5, 1.2)
>>> horse.shear(.5, 0)
>>> horse.relect(0, 1)
>>> horse.rotate(np.pi/2.)
>>> horse.translate(.75, .5)
>>> horse.display()
```

Animating Parametrizations

The plot in Problem 2 fails to fully convey the system's evolution over time because time itself is not part of the plot. The following function creates a simple Matplotlib animation for the earth and moon trajectories.

```
from matplotlib.animation import FuncAnimation

def solar_system_animation(earth, moon):
    """Animate the moon orbiting the earth and the earth orbiting the sun.
    Inputs:
        earth ((2,N) ndarray): The earth's postion with x-coordinates on the
            first row and y coordinates on the second row.
        moon ((2,N) ndarray): The moon's postion with x-coordinates on the
            first row and y coordinates on the second row.
    """
    fig = plt.figure()                              # Make a figure explicitly.
    plt.axis([-15,15,-15,15])                       # Set the window limits.
    plt.gca().set_aspect("equal")                   # Make the window square.

    earth_dot,  = plt.plot([],[], 'bo', ms=10)      # Blue dot for the earth.
    earth_path, = plt.plot([],[], 'b-')             # Blue line for the earth.
    moon_dot,   = plt.plot([],[], 'go', ms=5)       # Green dot for the moon.
    moon_path,  = plt.plot([],[], 'g-')             # Green line for the moon.
    plt.plot([0],[0],'y*', ms=30)                   # Yellow star for the sun.

    def animate(index):
        """Update the four earth and moon plots."""
        earth_dot.set_data(earth[0,index], earth[1,index])
        earth_path.set_data(earth[0,:index], earth[1,:index])
        moon_dot.set_data(moon[0,index], moon[1,index])
        moon_path.set_data(moon[0,:index], moon[1,:index])
        return earth_dot, earth_path, moon_dot, moon_path,
```

```
a = FuncAnimation(fig, animate, frames=earth.shape[1], interval=25)
plt.show()
```

See `http://matplotlib.org/1.5.1/examples/animation/index.html` for other examples of animations in Matplotlib.

2 Linear Systems

Lab Objective: *The fundamental problem of linear algebra is solving the linear system $A\mathbf{x} = \mathbf{b}$, if it is even possible. There are many approaches to solving this problem, each with different pros and cons. In this lab we implement the LU decomposition and use it to solve square linear systems. We also introduce SciPy, together with its libraries for linear algebra and working with sparse matrices.*

Gaussian Elimination

The standard approach for solving the linear system $A\mathbf{x} = \mathbf{b}$ on paper is reducing the augmented matrix $[A \mid \mathbf{b}]$ to row-echelon form (REF) via *Gaussian elimination*, then using back substitution. The matrix is in REF when the leading non-zero term in each row is the diagonal term, so the matrix is upper triangular.

At each step of the process, there are three possible operations: swapping two rows, multiplying one row by a scalar value, or adding a scalar multiple of one row to another. Many systems, like the one displayed below, can be reduced to REF using only the third type of operation. First use multiples of the first row to get zeros below the diagonal in the first column, then use a multiple of the second row to get zeros below the diagonal in the second column.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 4 & 7 & 8 & 9 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 3 & 4 & 5 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array}\right]$$

Each of these operations is mathematically equivalent to left-multiplying by a *type III elementary matrix*, the identity with one non-diagonal non-zero term. If row operation k corresponds to matrix E_k, the following equation is $E_3E_2E_1A = U$.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array}\right]$$

However, matrix multiplication is an inefficient way to implement row reduction. Instead, modify the matrix *in place* (without making a copy), changing only those entries that are affected by each row operation.

```
>>> import numpy as np
>>> A = np.array([[1, 1, 1, 1],
...               [1, 4, 2, 3],
...               [4, 7, 8, 9]], dtype=np.float)

# Reduce the 0th column to zeros below the diagonal.
>>> A[1,0:] -= (A[1,0] / A[0,0]) * A[0]
>>> A[2,0:] -= (A[2,0] / A[0,0]) * A[0]

# Reduce the 1st column to zeros below the diagonal.
>>> A[2,1:] -= (A[2,1] / A[1,1]) * A[1,1:]
>>> print(A)
[[ 1.  1.  1.  1.]
 [ 0.  3.  1.  2.]
 [ 0.  0.  3.  3.]]
```

Note that the final row operation modifies only part of the third row to avoid spending the computation time of adding 0 to 0.

If a 0 appears on the main diagonal during any part of row reduction, the approach given above tries to divide by 0. Swapping the current row with one below it that does not have a 0 in the same column solves this problem. This is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

Problem 1. Write a function which reduces a square matrix A to REF. You may assume that A is invertible and that a 0 will never appear on the main diagonal (so only use type III row reductions, not type II). Avoid operating on entries that you know will be 0 before and after a row operation.

Consider generating small random matrices as test cases with NumPy's `random` module (`np.random.randint()` may be particularly useful).

Achtung!

Gaussian elimination is not always numerically stable. Suppose that, due to roundoff error, we have a matrix with a very small entry on the diagonal.

$$A = \begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix}$$

10^{-15} is essentially zero, but instead of swapping the first and second rows to put the A in REF, a computer might multiply the first row by 10^{15} and add it to the second row to eliminate the -1. The resulting matrix is far from what we would expect on paper.

$$\begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 10^{-15} & 1 \\ 0 & 10^{15} \end{bmatrix}$$

Round-off error can propagate through many steps in a calculation. NumPy's routines that employ row reduction use several tricks to minimize the impact of round-off errors, but beware that these tricks can't fix every matrix.

The LU Decomposition

The *LU decomposition* of a square matrix A is a factorization $A = LU$ where U is the **upper** triangular REF of A and L is the **lower** triangular product of the type III elementary matrices whose inverses reduce A to U. The LU decomposition of A exists when A can be reduced to REF using only type III elementary matrices (no row swaps). However, the rows of A can always be permuted in a way such that the decomposition exists. If P is a permutation matrix encoding the appropriate row swaps, then the decomposition $PA = LU$ always exists.

Suppose A has an LU decomposition (not requiring row swaps). Then A can be reduced to REF with k row operations, corresponding to left-multiplying the type III elementary matrices $E_1, \dots, E_k$. Because there were no row swaps, each E_i is lower triangular, so each inverse E_i^{-1} is also lower triangular. Furthermore, since the product of lower triangular matrices is lower triangular, L is lower triangular.

$$\begin{aligned} E_k \dots E_2 E_1 A = U \quad \longrightarrow \quad A &= (E_k \dots E_2 E_1)^{-1} U \\ &= E_1^{-1} E_2^{-1} \dots E_k^{-1} U \\ &= LU \end{aligned}$$

We can thus compute L by right-multiplying the identity by the matrices used to reduce U. However, in this special situation, each right-multiplication only changes one entry of L, so we can avoid matrix multiplication altogether. The entire process, only slightly different than row reduction, is summarized below.

Algorithm 2.1

```
procedure LU Decomposition(A)
    m, n ← shape(A)                    ▷ Store the dimensions of A.
    U ← copy(A)                        ▷ Make a copy of A with np.copy().
    L ← I_m                            ▷ The m × m identity matrix.
    for j = 0 ... n − 1 do
        for i = j + 1 ... m − 1 do
            L_{i,j} ← U_{i,j} / U_{j,j}
            U_{i,j:} ← U_{i,j:} − L_{i,j} U_{j,j:}
    return L, U
```

Problem 2. Write a function that finds the LU decomposition of a square matrix. You may assume the decomposition exists and requires no row swaps.

Forward and Backward Substitution

If $A\mathbf{x} = \mathbf{b}$ and $PA = LU$, then $LU\mathbf{x} = PA\mathbf{x} = P\mathbf{b}$. This system can be solved by first solving $L\mathbf{y} = P\mathbf{b}$, then $U\mathbf{x} = \mathbf{y}$. Since L and U are both triangular, these systems can be solved with backward and forward substitution. We can thus compute the LU factorization of A once, then use substitution to efficiently solve $A\mathbf{x} = \mathbf{b}$ for various values of $\mathbf{b}$.

Since the diagonal entries of L are all 1, the triangular system $L\mathbf{y} = \mathbf{b}$ has the following form:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Matrix multiplication yields the following equations:

$$\begin{aligned} y_1 &= b_1 & y_1 &= b_1 \\ l_{21}y_1 + y_2 &= b_2 & y_2 &= b_2 - l_{21}y_1 \\ &\vdots & &\vdots \\ \sum_{j=1}^{k-1} l_{kj}y_j + y_k &= b_k & y_k &= b_k - \sum_{j=1}^{k-1} l_{kj}y_j \end{aligned} \tag{2.1}$$

The triangular system $U\mathbf{x} = \mathbf{b}$ yields similar equations, but in reverse order:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

$$\begin{aligned} u_{nn}x_n &= y_n & x_n &= \frac{1}{u_{nn}}y_n \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} & x_{n-1} &= \frac{1}{u_{n-1,n-1}}\left(y_{n-1} - u_{n-1,n}x_n\right) \\ &\vdots & &\vdots \\ \sum_{j=k}^{n} u_{kj}x_j &= y_k & x_k &= \frac{1}{u_{kk}}\left(y_k - \sum_{j=k+1}^{n} u_{kj}x_j\right) \end{aligned} \tag{2.2}$$

Problem 3. Write a function that, given A and $\mathbf{b}$, solves the square linear system $A\mathbf{x} = \mathbf{b}$. Use the function from Problem 2 to compute L and U, then use Equations 2.1 and 2.2 to solve for $\mathbf{y}$, then $\mathbf{x}$. You may again assume that there are no row swaps (so $P = I$ in this case).

SciPy

SciPy is a powerful scientific computing library built upon NumPy. It includes high-level tools for linear algebra, statistics, signal processing, integration, optimization, machine learning, and more.

SciPy is typically imported with the convention `import scipy as sp`. However, SciPy is set up in a way that requires its submodules to be imported separately.[1]

Linear Algebra

NumPy and SciPy both have a linear algebra module, each called `linalg`, but SciPy's module is the larger of the two. Some of SciPy's common `linalg` functions are listed below. See `http://docs.scipy.org/doc/scipy/reference/linalg.html` for more documentation.

Function	Returns
`det()`	The determinant of a square matrix.
`eig()`	The eigenvalues and eigenvectors of a square matrix.
`inv()`	The inverse of an invertible matrix.
`norm()`	The norm of a vector or matrix norm of a matrix.
`solve()`	The solution to $A\mathbf{x} = \mathbf{b}$ (the system may not be square).

This library also includes routines for computing matrix decompositions.

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

# Make a random matrix and a random vector.
>>> A = np.random.random((1000,1000))
>>> b = np.random.random(1000)

# Compute the LU decomposition of A, including pivots.
>>> lu, piv = la.lu_factor(A)

# Use the LU decomposition to solve Ax = b.
>>> x = la.lu_solve((lu,piv), b)

# Check that the solution is legitimate.
>>> np.allclose(A.dot(x), b)
True
```

As with NumPy, SciPy's routines are all highly optimized. However, some algorithms are, by nature, faster than others.

Problem 4. Write a function that times different `scipy.linalg` functions for solving square linear systems.

For various values of n, generate a random $n \times n$ matrix A and a random $n \times 1$ vector $\mathbf{b}$ using `np.random.random()`. Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

[1] SciPy modules like `linalg` are really *packages*, which need to be initialized separately.

1. Invert A with `la.inv()` and left-multiply the inverse to $\mathbf{b}$.
2. Use `la.solve()`.
3. Use `la.lu_factor()` and `la.lu_solve()` to solve the system with the LU decomposition.
4. Use `la.lu_factor()` and `la.lu_solve()`, but only time `la.lu_solve()`.

Plot the system size n versus the execution times. Use log scales if needed.

Note

Numerically inverting matrices is so costly that there is hardly ever a good reason to do it. Use a specific solver, like `la.lu_solve()`, whenever possible.

Sparse Matrices

Large linear systems can have tens of thousands of parameters. Storing the corresponding matrices in memory can be difficult: a 100000×100000 system requires around 40 GB to store in a NumPy array (4 bytes per entry $\times$ 10^{10} entries). This is well beyond the amount of RAM in a normal laptop.

In applications where systems of this size arise, it is common that such matrices are *sparse*, meaning that most of the entries are 0. SciPy's `sparse` module provides tools for efficiently constructing and manipulating 1- and 2-D sparse matrices. A `sparse` matrix only stores the nonzero values and the positions of these values. For sufficiently sparse matrices, storing the matrix as a `sparse` matrix may only take megabytes, rather than gigabytes.

For example, diagonal matrices are sparse. Storing an $n \times n$ diagonal matrix in the naïve way means storing n^2 values in memory. It is almost always more efficient to instead store the diagonal entries in a 1-D array of n values. In addition to using less storage space, this allows for much faster matrix operations: the standard algorithm to multiply a matrix by a diagonal matrix involves n^3 steps, but most of these are multiplying by or adding 0. A smarter algorithm can accomplish the same task much faster.

SciPy has seven `sparse` matrix types. Each type is optimized either for storing sparse matrices whose nonzero entries follow certain patterns, or for performing certain computations.

Name	Description	Advantages
`bsr_matrix`	Block Sparse Row	Specialized structure.
`coo_matrix`	Coordinate Format	Conversion among sparse formats.
`csc_matrix`	Compressed Sparse Column	Column-based operations and slicing.
`csr_matrix`	Compressed Sparse Row	Row-based operations and slicing.
`dia_matrix`	Diagonal Storage	Specialized structure.
`dok_matrix`	Dictionary of Keys	Element access, incremental construction.
`lil_matrix`	Row-based Linked List	Incremental construction.
		Conversion to CSR or CSC format.

Creating Sparse Matrices

A regular, non-sparse matrix is called *full* or *dense*. Full matrices can be converted to each of the sparse matrix formats listed above. However, it is more memory efficient to never create the full matrix in the first place. There are really three different approaches for creating sparse matrices from scratch.

- **Coordinate Format**: When all of the nonzero values and their positions are known, create the entire sparse matrix at once as a `coo_matrix`. All nonzero values are stored as a location (or coordinate) and value. This format also supports very fast conversion between the other sparse matrix types.

```
>>> from scipy import sparse

# Define the rows, columns, and values separately.
>>> rows = np.array([0, 1, 0])
>>> cols = np.array([0, 1, 1])
>>> vals = np.array([3, 5, 2])
>>> A = sparse.coo_matrix((vals, (rows,cols)), shape=(3,3))
>>> print(A)
  (0, 0)    3
  (1, 1)    5
  (0, 1)    2

# The toarray() method casts the sparse matrix as a NumPy array.
# However, using regular arrays forfeits all sparse-related optimizations.
>>> print(A.toarray())
[[3 2 0]
 [0 5 0]
 [0 0 0]]
```

- **DOK and LIL Formats**: If the matrix values and their locations are not known a priori, construct the matrix incrementally with `dok_matrix` or `lil_matrix`. Indicate the size of the matrix, then change individual values with regular slicing syntax.

```
>>> B = sparse.lil_matrix((2,6))
>>> B[0,2] = 4
>>> B[1,3:] = 9

>>> print(B.toarray())
[[ 0.  0.  4.  0.  0.  0.]
 [ 0.  0.  0.  9.  9.  9.]]
```

- **BSR and DIA Formats**: When the matrix has a particular sparsity pattern, such as nonzero blocks or diagonals, use `bsr_matrix` or `dia_matrix`. The functions `sparse.bmat()` and `sparse.diags()` are convenient ways to create these structures.

```
# Use sparse.bmat() to create a block matrix. Use 'None' for zero blocks.
>>> A = sparse.coo_matrix(np.ones((2,2)))
```

```
>>> B = sparse.coo_matrix(np.full((2,2), 2.))
>>> print(sparse.bmat([[  A , None,  A  ],
                       [None,  B  , None]], format='bsr').toarray())
[[ 1.,  1.,  0.,  0.,  1.,  1.],
 [ 1.,  1.,  0.,  0.,  1.,  1.],
 [ 0.,  0.,  2.,  2.,  0.,  0.],
 [ 0.,  0.,  2.,  2.,  0.,  0.]]

# Use sparse.diags() to create a matrix with diagonal entries.
>>> diagonals = [[1,2],[3,4,5],[6]]     # List the diagonal entries.
>>> offsets   = [ -1  ,   0   , 3 ]     # Specify the diagonal they go on.
>>> print(sparse.diags(diagonals, offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  4.  0.  0.]
 [ 0.  2.  5.  0.]]

# If all of the diagonals have the same entry, specify the entry alone.
>>> print(sparse.diags([1,3,6], offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  3.  0.  0.]
 [ 0.  1.  3.  0.]]
```

Note

A *banded* matrix is a square matrix whose only non-zero entries are on the main diagonal and on some diagonals on either side. If the nonzero entries are confined to the three central diagonals, the matrix is also called *tri-diagonal*. Banded matrices arise naturally in many applications, including numerical methods for solving differential equations.

Problem 5. Write a function that accepts an integer n and returns a sparse $n \times n$ tri-diagonal array A with 2's along the main diagonal and -1's along the first subdiagonal and the first superdiagonal.

$$A = \begin{bmatrix} 2 & -1 & \cdots & 0 & 0 \\ -1 & 2 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}$$

Sparse Matrix Operations

Once a sparse matrix has been constructed, it should be converted to a `csr_matrix` or `csc_matrix` with the matrix's `tocsr()` or `tocsc()` method. The CSR and CSC formats are optimized for row or column operations, respectively. To choose the correct format to use, determine what direction the matrix will be traversed.

For example, in the matrix-matrix multiplication AB, A is traversed row-wise, but B is traversed column-wise. Thus A should be converted to a `csr_matrix` and B should be converted to a `csc_matrix`.

```
# Initialize a sparse matrix incrementally as a lil_matrix.
>>> A = sparse.lil_matrix((10000,10000))
>>> for k in xrange(10000):
...     A[np.random.randint(0,9999), np.random.randint(0,9999)] = k
...
>>> A
<10000x10000 sparse matrix of type '<type 'numpy.float64'>'
    with 9999 stored elements in LInked List format>

# Convert A to CSR and CSC formats to compute the matrix product AA.
>>> Acsr = A.tocsr()
>>> Acsc = A.tocsc()
>>> Acsr.dot(Acsc)
<10000x10000 sparse matrix of type '<type 'numpy.float64'>'
    with 10142 stored elements in Compressed Sparse Row format>
```

Beware that row-based operations on a `csc_matrix` are very slow, and similarly, column-based operations on a `csr_matrix` are very slow.

Achtung!

Many familiar NumPy operations have analogous routines in the `sparse` module. These methods take advantage of the sparse structure of the matrices and are, therefore, usually significantly faster. However, Scipy's `sparse` matrices behave a little differently than NumPy arrays.

Operation	`numpy`	`scipy.sparse`
Component-wise Addition	`A + B`	`A + B`
Scalar Multiplication	`2 * A`	`2 * A`
Component-wise Multiplication	`A * B`	`A.multiply(B)`
Matrix Multiplication	`A.dot(B)`	`A * B`, `A.dot(B)`

Note in particular the difference between `A * B` for NumPy arrays and SciPy sparse matrices. Do **not** use `np.dot()` to try to multiply sparse matrices, as it may treat the inputs incorrectly. The syntax `A.dot(B)` is safest in most cases.

Finally, SciPy has a linear algebra library specifically for sparse matrices, called `scipy.sparse.linalg`. Like other SciPy modules, it must be imported specifically.

```
>>> from scipy.sparse import linalg as spla
```

Problem 6. Write a function that times regular and sparse linear system solvers.

For various values of n, generate the $n \times n$ matrix A described in Problem 5 and a random $n \times 1$ vector $\mathbf{b}$. Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

1. Convert A to CSR format, then use `scipy.sparse.linalg.spsolve()`.
2. Convert A to a NumPy array, then use `scipy.linalg.solve()`.

Plot the system size n versus the execution times. As always, use log scales where appropriate and use a legend to label each line.

Achtung!

Because there are fast algorithms for solving a tri-diagonal linear system, you may think that there are fast algorithms for inverting a tri-diagonal matrix. In fact this is not true, and the inverse of a sparse matrix is usually not sparse. There is rarely a good reason to invert a matrix, sparse or dense.

Note

One way to view a sparse matrix is to convert it to a NumPy array with the `toarray()` method. If a matrix is to large to fit in memory as an array, it can still be visualized with Matplotlib's `plt.spy()`, which colors in the locations of the non-zero entries of the matrix. Try the following code for visualizing a simple banded matrix.

```
>>> from matplotlib import pyplot as plt
>>> A = sparse.diags([1, 2, 3, 4, 5], [-5000, -1000, 0, 1000, 5000],
...                                                shape=(10000, 10000))
>>> plt.spy(A, markersize=1)
>>> plt.show()
```

See `http://docs.scipy.org/doc/scipy/reference/sparse.html` for additional details on SciPy's `sparse` module.

Additional Material

Improvements on the LU Decomposition

Vectorization

Algorithm 2.1 uses two loops to compute the LU decomposition. With a little vectorization, the process can be reduced to a single loop.

Algorithm 2.2

procedure FAST LU DECOMPOSITION(A)
    $m, n \leftarrow \text{shape}(A)$
    $U \leftarrow \text{copy}(A)$
    $L \leftarrow I_m$
    for $k = 0 \dots n-1$ **do**
        $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$
        $U_{k+1:,k:} \leftarrow U_{k+1:,k:} - L_{k+1:,k} U_{k,k:}^{\mathsf{T}}$
    return L, U

Note that step 7 is an *outer product*, not the regular dot product ($\mathbf{x}\mathbf{y}^{\mathsf{T}}$ instead of the usual $\mathbf{x}^{\mathsf{T}}\mathbf{y}$). Use `np.outer()` instead of `np.dot()` to get the desired result.

Pivoting

Gaussian elimination iterates through the rows of a matrix, using the diagonal entry $x_{k,k}$ of the matrix at the kth iteration to zero out all of the entries in the column below $x_{k,k}$ ($x_{i,k}$ for $i \geq k$). This diagonal entry is called the *pivot*. Unfortunately, Gaussian elimination, and hence the LU decomposition, can be very numerically unstable if at any step the pivot is a very small number. Most professional row reduction algorithms avoid this problem via *partial pivoting*.

The idea is to choose the largest number (in magnitude) possible to be the pivot by swapping the pivot row[2] with another row before operating on the matrix. For example, the second and fourth rows of the following matrix are exchanged so that the pivot is -6 instead of 2.

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

A row swap is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix}$$

For the LU decomposition, if the permutation matrix at step k is P_k, then $P = P_k \dots P_2 P_1$ yields $PA = LU$. The complete algorithm is given below.

[2] *Complete pivoting* involves row and column swaps, but doing both operations is usually considered overkill.

Algorithm 2.3

procedure LU Decomposition with Partial Pivoting(A)
  $m, n \leftarrow \text{shape}(A)$
  $U \leftarrow \text{copy}(A)$
  $L \leftarrow I_m$
  $P \leftarrow [0,\ 1,\ \ldots,\ n-1]$ ▷ See tip 2 below.
  for $k = 0 \ldots n-1$ **do**
    Select $i \geq k$ that maximizes $|U_{i,k}|$
    $U_{k,k:} \leftrightarrow U_{i,k:}$ ▷ Swap the two rows.
    $L_{k,:k} \leftrightarrow L_{i,:k}$ ▷ Swap the two rows.
    $P_k \leftrightarrow P_i$ ▷ Swap the two entries.
    $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$
    $U_{k+1:,k:} \leftarrow U_{k+1:,k:} - L_{k+1:,k} U_{k,k:}^{\mathsf{T}}$
  return L, U, P

The following tips may be helpful for implementing this algorithm:

1. Since NumPy arrays are mutable, use `np.copy()` to reassign the rows of an array simultaneously.

2. Instead of storing P as an $n \times n$ array, fancy indexing allows us to encode row swaps in a 1-D array of length n. Initialize P as the array $[0, 1, \ldots, n]$. After performing a row swap on A, perform the same operations on P. Then the matrix product PA will be the same as `A[P]`.

```
>>> A = np.zeros(3) + np.vstack(np.arange(3))
>>> P = np.arange(3)
>>> print(A)
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

# Swap rows 1 and 2.
>>> A[1], A[2] = np.copy(A[2]), np.copy(A[1])
>>> P[1], P[2] = P[2], P[1]

>>> print(A)                        # A with the new row arrangement.
[[ 0.  0.  0.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]

>>> print(P)                        # The permutation of the rows.
[0 2 1]

>>> print(A[P])                     # A with the original row arrangement.
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]
```

There are potential cases where even partial pivoting does not eliminate catastrophic numerical errors in Gaussian elimination, but the odds of having such an amazingly poor matrix are essentially zero. The numerical analyst J.H. Wilkinson captured the likelihood of encountering such a matrix in a natural application when he said, "Anyone that unlucky has already been run over by a bus!"

In Place

The LU decomposition can be performed in place (overwriting the original matrix A) by storing U on and above the main diagonal of the array and storing L below it. The main diagonal of L does not need to be stored since all of its entries are 1. This format saves an entire array of memory, and is how `scipy.linalg.lu_factor()` returns the factorization.

More Applications of the LU Decomposition

The LU decomposition can also be used to compute inverses and determinants.

- **Inverse**: $(PA)^{-1} = (LU)^{-1} \longrightarrow A^{-1}P^{-1} = U^{-1}L^{-1} \longrightarrow LUA^{-1} = P$. Solve $LU\mathbf{a}_i = \mathbf{p}_i$ with forward and backward substitution (as in Problem 3) for every column $\mathbf{p}_i$ of P. Then

$$A^{-1} = \left[\begin{array}{c|c|c|c} & & & \\ \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \\ & & & \end{array}\right],$$

  the matrix where $\mathbf{a}_k$ is the kth column.

- **Determinant**: $\det(A) = \det(P^{-1}LU) = \frac{\det(L)\det(U)}{\det(P)}$. The determinant of a triangular matrix is the product of its diagonal entries. Since every diagonal entry of L is 1, $\det(L) = 1$. Also, P is just a row permutation of the identity matrix (which has determinant 1), and a single row swap negates the determinant. Thus

$$\det(A) = (-1)^S \prod_{i=1}^{n} u_{ii},$$

  where S is the number of row swaps.

The Cholesky Decomposition

A square matrix A is called *positive definite* if $\mathbf{z}^\mathsf{T} A\mathbf{z} > 0$ for all nonzero vectors $\mathbf{z}$. In addition, A is called *Hermitian* if $A = A^\mathsf{H} = \overline{A^\mathsf{T}}$. If A is Hermitian positive definite, it has a *Cholesky Decomposition* $A = U^\mathsf{H}U$ where U is upper triangular with real, positive entries on the diagonal. This is the matrix equivalent to taking the square root of a positive real number.

The Cholesky decomposition takes advantage of the conjugate symmetry of A to simultaneously reduce the columns *and* rows of A to zeros (except for the diagonal). It thus requires only half of the calculations and memory of the LU decomposition. Furthermore, the algorithm is *numerically stable*, which means that round-off errors do not propagate throughout the computation. Because of its efficiency and stability, this decomposition is used when possible to solve least squares, optimization, and state estimation problems.

Algorithm 2.4

```
1: procedure CHOLESKY DECOMPOSITION(A)
2:     U ← A                                          ▷ Copy A if desired.
3:     for i = 0 ... n − 1 do
4:         for j = i + 1 ... n − 1 do
5:             U_{j,j:} ← U_{j,j:} − U_{i,j:} conj(U_{ij}) / U_{ii}
6:         U_{i,i:} ← U_{i,i:} / sqrt(U_{ii})
7:     return U
```

As with the LU decomposition, SciPy's `linalg` module has optimized routines, `la.cho_factor()` and `la.cho_solve()`, for using the Cholesky decomposition.

3 The QR Decomposition

Lab Objective: *The QR decomposition is a fundamentally important matrix factorization. It is straightforward to implement, is numerically stable, and provides the basis of several important algorithms. In this lab, we explore several ways to produce the QR decomposition and implement a few immediate applications.*

We restrict our discussion to real matrices. However, the results and algorithms presented here can be extended to complex matrices by replacing "transpose" with "hermitian conjugate" and "symmetric matrix" with "hermitian matrix."

The QR decomposition of a matrix A is a factorization $A = QR$, where Q is has orthonormal columns and R is upper triangular. Every $m \times n$ matrix A of rank $n \leq m$ has a QR decomposition, with two main forms.

- **Reduced QR**: Q is $m \times n$, R is $n \times n$, and the columns of Q $\{\mathbf{q}_j\}_{j=1}^{n}$ form an orthonormal basis for the column space of A.

- **Full QR**: Q is $m \times m$ and R is $m \times n$. In this case, the columns of Q $\{\mathbf{q}_j\}_{j=1}^{m}$ form an orthonormal basis for all of $\mathbb{F}^m$, and the last $m - n$ rows of R only contain zeros. If $m = n$, this is the same as the reduced factorization.

We distinguish between these two forms by writing $\widehat{Q}$ and $\widehat{R}$ for the reduced decomposition and Q and R for the full decomposition.

$$\underbrace{\begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n & \mathbf{q}_{n+1} & \cdots & \mathbf{q}_m \end{bmatrix}}_{Q\ (m \times m)} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix}}_{R\ (m \times n)} = A\ (m \times n)$$

QR via Gram-Schmidt

The *classical Gram-Schmidt algorithm* takes a linearly independent set of vectors and constructs an orthonormal set of vectors with the same span. Applying Gram-Schmidt to the columns of A, which are linearly independent since A has rank n, results in the columns of Q.

Let $\{\mathbf{x}_j\}_{j=1}^n$ be the columns of A. Define

$$\mathbf{q}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|}, \qquad \mathbf{q}_k = \frac{\mathbf{x}_k - \mathbf{p}_{k-1}}{\|\mathbf{x}_k - \mathbf{p}_{k-1}\|}, \quad k = 2,\ \ldots,\ n,$$

$$\mathbf{p}_0 = \mathbf{0},\ \text{and} \qquad \mathbf{p}_{k-1} = \sum_{j=1}^{k-1} \langle \mathbf{q}_j, \mathbf{x}_k \rangle \mathbf{q}_j, \quad k = 2,\ \ldots,\ n.$$

Each $\mathbf{p}_{k-1}$ is the projection of $\mathbf{x}_k$ onto the span of $\{\mathbf{q}_j\}_{j=1}^{k-1}$, so $\mathbf{q}_k' = \mathbf{x}_k - \mathbf{p}_{k-1}$ is the residual vector of the projection. Thus $\mathbf{q}_k'$ is orthogonal to each of the $\{\mathbf{q}_j\}_{j=1}^{k-1}$. Therefore, normalizing each $\mathbf{q}_k'$ produces an orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$.

To construct the reduced QR decomposition, let $\widehat{Q}$ be the matrix with columns $\{\mathbf{q}_j\}_{j=1}^n$, and let $\widehat{R}$ be the upper triangular matrix with the following entries:

$$r_{kk} = \|\mathbf{x}_k - \mathbf{p}_{k-1}\|, \qquad r_{jk} = \langle \mathbf{q}_j, \mathbf{x}_k \rangle = \mathbf{q}_j^\mathsf{T} \mathbf{x}_k,\ j < k.$$

This clever choice of entries for $\widehat{R}$ reverses the Gram-Schmidt process and ensures that $\widehat{Q}\widehat{R} = A$.

Modified Gram-Schmidt

If the columns of A are close to being linearly dependent, the classical Gram-Schmidt algorithm often produces a set of vectors $\{\mathbf{q}_j\}_{j=1}^n$ that are not even close to orthonormal due to rounding errors. The *modified Gram-Schmidt algorithm* is a slight variant of the classical algorithm which more consistently produces a set of vectors that are "very close" to orthonormal.

Let $\mathbf{q}_1$ be the normalization of $\mathbf{x}_1$ as before. Instead of making just $\mathbf{x}_2$ orthogonal to $\mathbf{q}_1$, make *each* of the vectors $\{\mathbf{x}_j\}_{j=2}^n$ orthogonal to $\mathbf{q}_1$:

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_1, \mathbf{x}_k \rangle \mathbf{q}_1, \quad k = 2,\ \ldots,\ n.$$

Next, define $\mathbf{q}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|}$. Proceed by making each of $\{\mathbf{x}_j\}_{j=3}^n$ orthogonal to $\mathbf{q}_2$:

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_2, \mathbf{x}_k \rangle \mathbf{q}_2, \quad k = 3, \ldots, n.$$

Since each of these new vectors is a linear combination of vectors orthogonal to $\mathbf{q}_1$, they are orthogonal to $\mathbf{q}_1$ as well. Continuing this process results in the desired orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$. The entire modified Gram-Schmidt algorithm is described below in Algorithm 3.1.

Algorithm 3.1

1: **procedure** MODIFIED GRAM-SCHMIDT(A)
2: $m, n \leftarrow \text{shape}(A)$ ▷ Store the dimensions of A.
3: $Q \leftarrow \text{copy}(A)$ ▷ Make a copy of A with `np.copy()`.
4: $R \leftarrow \text{zeros}(n, n)$ ▷ An $n \times n$ array of all zeros.
5: **for** $i = 0 \dots n-1$ **do**
6: $R_{i,i} \leftarrow \|Q_{:,i}\|$
7: $Q_{:,i} \leftarrow Q_{:,i}/R_{i,i}$ ▷ Normalize the ith column of Q.
8: **for** $j = i+1 \dots n-1$ **do**
9: $R_{i,j} \leftarrow Q_{:,j}^{\mathsf{T}} Q_{:,i}$
10: $Q_{:,j} \leftarrow Q_{:,j} - R_{i,j} Q_{:,i}$ ▷ Orthogonalize the jth column of Q.
11: **return** Q, R

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n. Use Algorithm 3.1 to compute the reduced QR decomposition of A.

Consider the following tips for implementing the algorithm.

- In Python, the operation `a = a + b` can also be written as `a += b`.
- Use `scipy.linalg.norm()` to compute the norm of the vector in step 6.
- Note that steps 7 and 10 employ scalar multiplication or division, while step 9 uses vector multiplication.

To test your function, generate test cases with NumPy's `np.random` module. Verify that R is upper triangular, Q is orthonormal, and $QR = A$. You may also want to compare your results to SciPy's QR factorization algorithm.

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

# Generate a random matrix and get its reduced QR decomposition via SciPy.
>>> A = np.random.random((6,4))
>>> Q,R = la.qr(A, mode="economic") # Use mode="economic" for reduced QR.
>>> print A.shape, Q.shape, R.shape
(6,4) (6,4) (4,4)

# Verify that R is upper triangular, Q is orthonormal, and QR = A.
>>> np.allclose(np.triu(R), R)
True
>>> np.allclose(np.dot(Q.T, Q), np.identity(4))
True
>>> np.allclose(np.dot(Q, R), A)
True
```

Consequences of the QR Decomposition

The special structures of Q and R immediately provide some simple applications.

Determinants

Let A be $n \times n$. Then Q and R are both $n \times n$ as well.[1] Since Q is orthonormal and R is upper-triangular,

$$\det(Q) = \pm 1 \qquad\qquad \det(R) = \prod_{i=1}^{n} r_{i,i}$$

Then since $\det(AB) = \det(A)\det(B)$, we have the following:

$$|\det(A)| = |\det(QR)| = |\det(Q)\det(R)| = \left|\prod_{k=1}^{n} r_{kk}\right|$$

Problem 2. Write a function that accepts an invertible $n \times n$ matrix A. Use the QR decomposition of A to calculate $|\det(A)|$.

You may use your QR decomposition algorithm from Problem 1 or SciPy's QR routine. Can you implement this function in a single line?

Linear Systems

The LU decomposition is usually the matrix factorization of choice to solve the linear system $A\mathbf{x} = \mathbf{b}$ because the triangular structures of L and U facilitate forward and backward substitution. However, the QR decomposition avoids the potential numerical issues that come with Gaussian elimination.

Since Q is orthonormal, $Q^{-1} = Q^\mathsf{T}$. Therefore, solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving the system $R\mathbf{x} = Q^\mathsf{T}\mathbf{b}$. Since R is upper-triangular, $R\mathbf{x} = Q^\mathsf{T}\mathbf{b}$ can be solved quickly with back substitution.[2]

Problem 3. Write a function that accepts an invertible $n \times n$ matrix A and a vector $\mathbf{b}$ of length n. Use the QR decomposition to solve $A\mathbf{x} = \mathbf{b}$ in the following steps:

1. Compute Q and R.
2. Calculate $\mathbf{y} = Q^\mathsf{T}\mathbf{b}$.
3. Use back substitution to solve $R\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

[1] An $n \times n$ orthonormal matrix is sometimes called *unitary* in other texts.

[2] See Problem 3 of the Linear Systems lab for a refresher on back substitution.

QR via Householder

The Gram-Schmidt algorithm orthonormalizes A using a series of transformations that are stored in an upper triangular matrix. Another way to compute the QR decomposition is to take the opposite approach: triangularize A through a series of orthonormal transformations. Orthonormal transformations are numerically stable, meaning that they are less susceptible to rounding errors. In fact, this approach is usually faster and more accurate than Gram-Schmidt methods.

The idea is for the kth orthonormal transformation Q_k to map the kth column of A to the span of $\{\mathbf{e}_j\}_{j=1}^{k}$, where the $\mathbf{e}_j$ are the standard basis vectors in $\mathbb{R}^m$. In addition, to preserve the work of the previous transformations, Q_k should not modify any entries of A that are above or to the left of the kth diagonal term of A. For a 4×3 matrix A, the process can be visualized as follows.

$$Q_3Q_2Q_1 \begin{bmatrix} \boxed{\begin{matrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{matrix}} \end{bmatrix} = Q_3Q_2 \begin{bmatrix} * & * & * \\ 0 & \boxed{\begin{matrix} * & * \\ * & * \\ * & * \end{matrix}} \\ 0 & \end{bmatrix} = Q_3 \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & \boxed{*} \\ 0 & 0 & * \end{bmatrix} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{bmatrix}$$

Thus $Q_3Q_2Q_1A = R$, so that $A = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}R$ since each Q_k is orthonormal. Furthermore, the product of square orthonormal matrices is orthonormal, so setting $Q = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}$ yields the full QR decomposition.

How to correctly construct each Q_k isn't immediately obvious. The ingenious solution lies in one of the basic types of linear transformations: reflections.

Householder Transformations

The *orthogonal complement* of a nonzero vector $\mathbf{v} \in \mathbb{R}^n$ is the set of all vectors $\mathbf{x} \in \mathbb{R}^n$ that are orthogonal to $\mathbf{v}$, denoted $\mathbf{v}^\perp = \{\mathbf{x} \in \mathbb{R}^n \mid \langle \mathbf{x}, \mathbf{v} \rangle = 0\}$. A *Householder transformation* is a linear transformation that reflects a vector $\mathbf{x}$ across the orthogonal complement $\mathbf{v}^\perp$ for some specified $\mathbf{v}$.

The matrix representation of the Householder transformation corresponding to $\mathbf{v}$ is given by $H_\mathbf{v} = I - 2\frac{\mathbf{v}\mathbf{v}^\mathsf{T}}{\mathbf{v}^\mathsf{T}\mathbf{v}}$. Since $H_\mathbf{v}^\mathsf{T}H_\mathbf{v} = I$, Householder transformations are orthonormal.

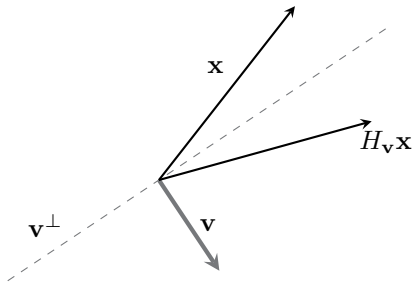


Figure 3.1: The vector $\mathbf{v}$ defines the orthogonal complement $\mathbf{v}^\perp$. Applying the Householder transformation $H_\mathbf{v}$ to $\mathbf{x}$ reflects $\mathbf{x}$ across $\mathbf{v}^\perp$.

Householder Triangularization

The *Householder algorithm* uses Householder transformations for the orthonormal transformations in the QR decomposition process described on the previous page. The goal in choosing Q_k is to send $\mathbf{x}_k$, the kth column of A, to the span of $\{\mathbf{e}_j\}_{j=1}^{k}$. In other words, if $Q_k\mathbf{x}_k = \mathbf{y}_k$, the last $m-k$ entries of $\mathbf{y}_k$ should be 0.

$$Q_k\mathbf{x}_k = Q_k \begin{bmatrix} z_1 \\ \vdots \\ z_k \\ z_{k+1} \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{y}_k$$

To begin, decompose $\mathbf{x}_k$ into $\mathbf{x}_k = \mathbf{x}_k' + \mathbf{x}_k''$, where $\mathbf{x}_k'$ and $\mathbf{x}_k''$ are of the form

$$\mathbf{x}_k' = [z_1 \quad \cdots \quad z_{k-1} \quad 0 \quad \cdots \quad 0]^\mathsf{T} \qquad \text{and} \qquad \mathbf{x}_k'' = [0 \quad \cdots \quad 0 \quad z_k \quad \cdots \quad z_m]^\mathsf{T}.$$

Because $\mathbf{x}_k'$ represents elements of A that lie above the diagonal, only $\mathbf{x}_k''$ needs to be altered by the reflection.

The two vectors $\mathbf{x}_k'' \pm \|\mathbf{x}_k''\|\mathbf{e}_k$ both yield Householder transformations that send $\mathbf{x}_k''$ to the span of $\mathbf{e}_k$ (see Figure 3.2). Between the two, the one that reflects $\mathbf{x}_k''$ further is more numerically stable. This reflection corresponds to

$$\mathbf{v}_k = \mathbf{x}_k'' + \operatorname{sign}(z_k)\|\mathbf{x}_k''\|\mathbf{e}_k,$$

where z_k is the first nonzero component of $\mathbf{x}_k''$ (the kth component of $\mathbf{x}_k$).

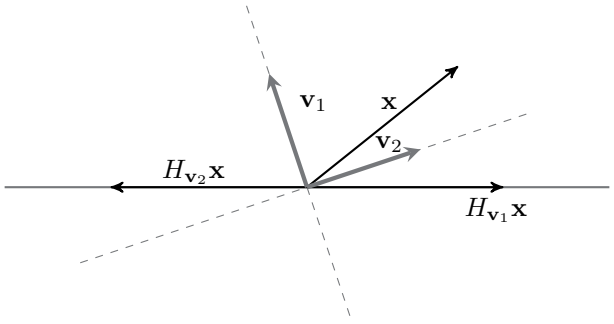


Figure 3.2: There are two possible reflections that map $\mathbf{x}$ into the span of $\mathbf{e}_1$, defined by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. In this illustration, $H_{\mathbf{v}_2}$ is the more stable transformation since it reflects $\mathbf{x}$ further than $H_{\mathbf{v}_1}$.

After choosing $\mathbf{v}_k$, set $\mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}$. Then $H_{\mathbf{v}_k} = I - 2\frac{\mathbf{v}_k\mathbf{v}_k^\mathsf{T}}{\|\mathbf{v}_k\|^2} = I - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T}$, and hence Q_k is given by the following block matrix.

$$Q_k = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & I_{m-k+1} - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T} \end{bmatrix}$$

Here I_p denotes the $p \times p$ identity matrix, and thus each Q_k is $m \times m$.

It is apparent from its form that Q_k does not affect the first $k-1$ rows and columns of any matrix that it acts on. Then by starting with $R = A$ and $Q = I$, at each step of the algorithm we need only multiply the entries in the lower right $(m-k+1) \times (m-k+1)$ submatrices of R and Q by $I - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T}$. This completes the Householder algorithm, detailed below.

Algorithm 3.2

1: **procedure** HOUSEHOLDER(A)
2: $m, n \leftarrow \text{shape}(A)$
3: $R \leftarrow \text{copy}(A)$
4: $Q \leftarrow I_m$ ▷ The $m \times m$ identity matrix.
5: **for** $k = 0 \dots n-1$ **do**
6: $\mathbf{u} \leftarrow \text{copy}(R_{k:,k})$
7: $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$ ▷ u_0 is the first entry of $\mathbf{u}$.
8: $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$ ▷ Normalize $\mathbf{u}$.
9: $R_{k:,k:} \leftarrow R_{k:,k:} - 2\mathbf{u}\left(\mathbf{u}^\mathsf{T} R_{k:,k:}\right)$ ▷ Apply the reflection to R.
10: $Q_{k:,:} \leftarrow Q_{k:,:} - 2\mathbf{u}\left(\mathbf{u}^\mathsf{T} Q_{k:,:}\right)$ ▷ Apply the reflection to Q.
11: **return** Q^T, R

Problem 4. Write a function that accepts as input a $m \times n$ matrix A of rank n. Use Algorithm 3.2 to compute the full QR decomposition of A.

Consider the following implementation details.

- NumPy's `np.sign()` is an easy way to implement the sign() operation in step 7. However, `np.sign(0)` returns 0, which will cause a problem in the rare case that $u_0 = 0$ (which is possible if the top left entry of A is 0 to begin with). The following code defines a function that returns the sign of a single number, counting 0 as positive.

```
sign = lambda x: 1 if x >= 0 else -1
```

- In steps 9 and 10, the multiplication of $\mathbf{u}$ and $(\mathbf{u}^\mathsf{T}X)$ is an *outer product* ($\mathbf{x}\mathbf{y}^\mathsf{T}$ instead of the usual $\mathbf{x}^\mathsf{T}\mathbf{y}$). Use `np.outer()` instead of `np.dot()` to handle this correctly.

As with Problem 1, use NumPy and SciPy to generate test cases and validate your function.

```
>>> A = np.random.random((5, 3))
>>> Q,R = la.qr(A)                  # Get the full QR decomposition.
>>> print A.shape, Q.shape, R.shape
(5,3) (5,5) (5,3)
>>> np.allclose(Q.dot(R), A)
True
```

Upper Hessenberg Form

An *upper Hessenberg matrix* is a square matrix that is nearly upper triangular, with zeros below the first subdiagonal. Every $n \times n$ matrix A can be written $A = QHQ^\mathsf{T}$ where Q is orthonormal and H, called the *Hessenberg form* of A, is an upper Hessenberg matrix. Putting a matrix in upper Hessenberg form is an important first step to computing its eigenvalues numerically.

This algorithm also uses Householder transformations. To find orthogonal Q and upper Hessenberg H such that $A = QHQ^\mathsf{T}$, it suffices to find such matrices that satisfy $Q^\mathsf{T}AQ = H$. Thus, the strategy is to multiply A on the left and right by a series of orthonormal matrices until it is in Hessenberg form.

Using the same Q_k as in the kth step of the Householder algorithm introduces $n-k$ zeros in the kth column of A, but multiplying Q_kA on the right by Q_k^T destroys all of those zeros. Instead, choose a Q_1 that fixes $\mathbf{e}_1$ and reflects the first column of A into the span of $\mathbf{e}_1$ and $\mathbf{e}_2$. The product Q_1A then leaves the first row of A alone, and the product $(Q_1A)Q_1^\mathsf{T}$ leaves the first column of (Q_1A) alone.

$$\underset{A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1A}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix}} \xrightarrow{Q_1^\mathsf{T}} \underset{(Q_1A)Q_1^\mathsf{T}}{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix}}$$

Continuing the process results in the upper Hessenberg form of A.

$$Q_3Q_2Q_1AQ_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

This implies that $A = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}HQ_3Q_2Q_1$, so setting $Q = Q_1^\mathsf{T}Q_2^\mathsf{T}Q_3^\mathsf{T}$ results in the desired factorization $A = QHQ^\mathsf{T}$.

Constructing the Reflections

Constructing the Q_k uses the same approach as in the Householder algorithm, but shifted down one element. Let $\mathbf{x}_k = \mathbf{y}_k' + \mathbf{y}_k''$ where $\mathbf{y}_k'$ and $\mathbf{y}_k''$ are of the form

$$\mathbf{y}_k' = [z_1 \quad \cdots \quad z_k \quad 0 \quad \cdots \quad 0]^\mathsf{T} \qquad \text{and} \qquad \mathbf{y}_k'' = [0 \quad \cdots \quad 0 \quad z_{k+1} \quad \cdots \quad z_m]^\mathsf{T}.$$

Because $\mathbf{y}_k'$ represents elements of A that lie above the first subdiagonal, only $\mathbf{y}_k''$ needs to be altered. This suggests using the following reflection.

$$\mathbf{v}_k = \mathbf{y}_k'' + \operatorname{sign}(z_k)\|\mathbf{y}_k''\|\mathbf{e}_k \qquad\qquad \mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}$$

$$Q_k = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & I_{m-k} - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T} \end{bmatrix}$$

The complete algorithm is given below. Note how similar it is to Algorithm 3.2.

Algorithm 3.3

procedure HESSENBERG(A)
  $m, n \leftarrow \text{shape}(A)$
  $H \leftarrow \text{copy}(A)$
  $Q \leftarrow I_m$
  for $k = 0 \dots n-3$ **do**
    $\mathbf{u} \leftarrow \text{copy}(H_{k+1:,k})$
    $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$
    $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$
    $H_{k+1:,k:} \leftarrow H_{k+1:,k:} - 2\mathbf{u}(\mathbf{u}^\mathsf{T} H_{k+1:,k:})$ ▷ Apply Q_k to H.
    $H_{:,k+1:} \leftarrow H_{:,k+1:} - 2(H_{:,k+1:}\mathbf{u})\mathbf{u}^\mathsf{T}$ ▷ Apply Q_k^T to H.
    $Q_{k+1:,:} \leftarrow Q_{k+1:,:} - 2\mathbf{u}(\mathbf{u}^\mathsf{T} Q_{k+1:,:})$ ▷ Apply Q_k to Q.
  return H, Q^T

Problem 5. Write a function that accepts a nonsingular $n \times n$ matrix A. Use Algorithm 3.3 to compute its upper Hessenberg form, upper Hessenberg H and orthogonal Q satisfying $A = QHQ^\mathsf{T}$.

Test your function and compare your results to `scipy.linalg.hessenberg()`.

```
# Generate a random matrix and get its upper Hessenberg form via SciPy.
>>> A = np.random.random((8,8))
>>> H, Q = la.hessenberg(A, calc_q=True)

# Verify that H has all zeros below the first subdiagonal and QHQ^T = A.
>>> np.allclose(np.triu(H, -1), H)
True
>>> np.allclose(np.dot(np.dot(Q, H), Q.T), A)
True
```

NOTE

When A is symmetric, its upper Hessenberg form is a tridiagonal matrix. This is because the Q_k's zero out everything below the first subdiagonal of A and the Q_k^T's zero out everything to the right of the first superdiagonal. Tridiagonal matrices make computations fast, so the computing the Hessenberg form of a symmetric matrix is very useful.

Additional Material

Complex QR Decomposition

The QR decomposition also exists for matrices with complex entries. The standard inner product in $\mathbb{R}^m$ is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\mathsf{T}\mathbf{y}$, but the (more general) standard inner product in $\mathbb{C}^m$ is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\mathsf{H}\mathbf{y}$. The $^\mathsf{H}$ stands for the *Hermitian conjugate*, the conjugate of the transpose. Making a few small adjustments in the implementations of Algorithms 3.1 and 3.2 accounts for using the complex inner product.

1. Replace any transpose operations with the conjugate of the transpose.

```
>>> A = np.reshape(np.arange(4) + 1j*np.arange(4), (2,2))
>>> print(A)
[[ 0.+0.j  1.+1.j]
 [ 2.+2.j  3.+3.j]]

>>> print(A.T)                      # Regular transpose.
[[ 0.+0.j  2.+2.j]
 [ 1.+1.j  3.+3.j]]

>>> print(A.conj().T)               # Hermitian conjugate.
[[ 0.-0.j  2.-2.j]
 [ 1.-1.j  3.-3.j]]
```

2. Conjugate the first entry of vector or matrix multiplication before multiplying with `np.dot()`.

```
>>> x = np.arange(2) + 1j*np.arange(2)
>>> print(x)
[ 0.+0.j  1.+1.j]

>>> np.dot(x, x)                    # Standard real inner product.
2j

>>> np.dot(x.conj(), y)             # Standard complex inner product.
(2 + 0j)
```

3. In the complex plane, there are infinitely many reflections that map a vector $\mathbf{x}$ into the span of $\mathbf{e}_k$, not just the two displayed in Figure 3.2. Using $\text{sign}(z_k)$ to choose one is still a valid method, but it requires updating the `sign()` function so that it can handle complex numbers.

```
sign = lambda x: 1 if np.real(x) >= 0 else -1
```

QR with Pivoting

The LU decomposition can be improved by employing Gaussian elimination with partial pivoting, where the rows of A are strategically permuted at each iteration. The QR factorization can be similarly improved by permuting the columns of A at each iteration. The result is the factorization $AP = QR$, where P is a permutation matrix that encodes the column swaps. SciPy's `scipy.linalg.qr()` can compute the pivoted QR decomposition.

```
# Get the decomposition AP = QR for a random matrix A.
>>> A = np.random.random((8,10))
>>> Q,R,P = la.qr(A, pivoting=True)

# P is returned as a 1-D array that encodes column ordering,
# so A can be reconstructed with fancy indexing.
>>> np.allclose(Q.dot(R), A[:,P])
True
```

QR via Givens

The Householder algorithm uses reflections to triangularize A. However, A can also be made upper triangular using rotations. To illustrate the idea, recall that the following matrix represents a counterclockwise rotation of θ radians.

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

This transformation is orthonormal. Given $\mathbf{x} = [a, b]^\mathsf{T}$, if θ is the angle between $\mathbf{x}$ and $\mathbf{e}_1$, then $R_{-\theta}$ maps $\mathbf{x}$ to the span of $\mathbf{e}_1$.

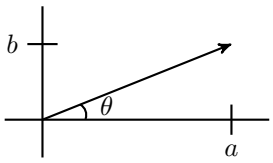


Figure 3.3: Rotating clockwise by θ sends the vector $[a, b]^\mathsf{T}$ to the span of $\mathbf{e}_1$.

In terms of a and b, $\cos\theta = \frac{a}{\sqrt{a^2+b^2}}$ and $\sin\theta = \frac{b}{\sqrt{a^2+b^2}}$. Therefore,

$$R_{-\theta}\mathbf{x} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \frac{a}{\sqrt{a^2+b^2}} & \frac{b}{\sqrt{a^2+b^2}} \\ -\frac{b}{\sqrt{a^2+b^2}} & \frac{a}{\sqrt{a^2+b^2}} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sqrt{a^2+b^2} \\ 0 \end{bmatrix}.$$

The matrix R_θ above is an example of a 2×2 *Givens rotation matrix*. In general, the Givens matrix $G(i, j, \theta)$ represents the orthonormal transformation that rotates the 2-dimensional span of $\mathbf{e}_i$ and $\mathbf{e}_j$ by θ radians. The matrix representation of this transformation is a generalization of R_θ.

$$G(i, j, \theta) = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}$$

Here I represents the identity matrix, $c = \cos\theta$, and $s = \sin\theta$. The c's appear on the i^{th} and j^{th} diagonal entries.

Givens Triangularization

As demonstrated, θ can be chosen such that $G(i, j, \theta)$ rotates a vector so that its jth-component is 0. Such a transformation will only affect the i^{th} and j^{th} entries of any vector it acts on (and thus the i^{th} and j^{th} rows of any matrix it acts on).

To compute the QR decomposition of A, iterate through the subdiagonal entries of A in the order depicted by Figure 3.4. Zero out the ij^{th} entry with a rotation in the plane spanned by $\mathbf{e}_{i-1}$ and $\mathbf{e}_i$, represented by the Givens matrix $G(i-1, i, \theta)$.

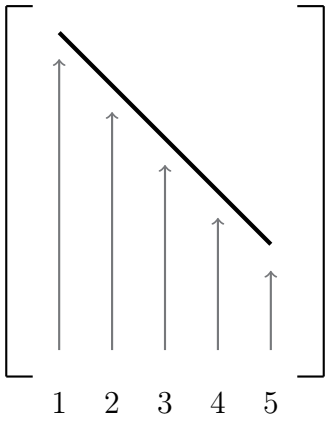


Figure 3.4: The order in which to zero out subdiagonal entries in the Givens triangularization algorithm. The heavy black line is the main diagonal of the matrix. Entries should be zeroed out from bottom to top in each column, beginning with the leftmost column.

On a 2×3 matrix, the process can be visualized as follows.

$$\begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix} \xrightarrow{G(2,3,\theta_1)} \begin{bmatrix} * & * \\ \boxed{\begin{matrix} * & * \\ 0 & * \end{matrix}} \end{bmatrix} \xrightarrow{G(1,2,\theta_2)} \begin{bmatrix} \boxed{\begin{matrix} * & * \\ 0 & * \end{matrix}} \\ 0 \quad * \end{bmatrix} \xrightarrow{G(2,3,\theta_3)} \begin{bmatrix} * & * \\ 0 & \boxed{*} \\ 0 & \boxed{0} \end{bmatrix}$$

At each stage, the boxed entries are those modified by the previous transformation. The final transformation $G(2, 3, \theta_3)$ operates on the bottom two rows, but since the first two entries are zero, they are unaffected.

Assuming that at the ij^{th} stage of the algorithm a_{ij} is nonzero, Algorithm 3.4 computes the Givens triangularization of a matrix. Notice that the algorithm does not actually form the entire matrices $G(i, j, \theta)$; instead, it modifies only those entries of the matrix that are affected by the transformation.

Algorithm 3.4

procedure GIVENS TRIANGULARIZATION(A)
  $m, n \leftarrow \text{shape}(A)$
  $R \leftarrow \text{copy}(A)$
  $Q \leftarrow I_m$
  for $j = 0 \dots n-1$ **do**
    for $i = m-1 \dots j+1$ **do**
      $a, b \leftarrow R_{i-1,j}, R_{i,j}$
      $G \leftarrow [[a, b], [-b, a]]/\sqrt{a^2 + b^2}$
      $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$
      $Q_{i-1:i+1,:} \leftarrow GQ_{i-1:i+1,:}$
  return Q^T, R

QR of a Hessenberg Matrix via Givens

The Givens algorithm is particularly efficient for computing the QR decomposition of a matrix that is already in upper Hessenberg form, since only the first subdiagonal needs to be zeroed out. Algorithm 3.5 details this process.

Algorithm 3.5

procedure GIVENS TRIANGULARIZATION OF HESSENBERG(H)
  $m, n \leftarrow \text{shape}(H)$
  $R \leftarrow \text{copy}(H)$
  $Q \leftarrow I_m$
  for $j = 0 \dots \min\{n-1, m-1\}$ **do**
    $i = j + 1$
    $a, b \leftarrow R_{i-1,j}, R_{i,j}$
    $G \leftarrow [[a, b], [-b, a]]/\sqrt{a^2 + b^2}$
    $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$
    $Q_{i-1:i+1,:i+1} \leftarrow GQ_{i-1:i+1,:i+1}$
  return Q^T, R

4 Least Squares and Computing Eigenvalues

Lab Objective: *Because of its numerical stability and convenient structure, the QR decomposition is the basis of many important and practical algorithms. In this lab, we introduce linear least squares problems, tools in Python for computing least squares solutions, and two fundamental eigenvalue algorithms.*

As in the previous lab, we restrict ourselves to real matrices and therefore use the transpose in place of the Hermitian conjugate.

Least Squares

A linear system $A\mathbf{x} = \mathbf{b}$ is *overdetermined* if it has more equations than unknowns. In this situation, there is no true solution, and $\mathbf{x}$ can only be approximated.

The *least squares solution* of $A\mathbf{x} = \mathbf{b}$, denoted as $\widehat{\mathbf{x}}$, is the "closest" vector to a solution, meaning it minimizes the quantity $\|A\widehat{\mathbf{x}} - \mathbf{b}\|_2$. In other words, $\widehat{\mathbf{x}}$ is the vector such that $A\widehat{\mathbf{x}}$ is projection of $\mathbf{b}$ onto the range of A, and can be calculated by solving the *normal equation*:[1]

$$A^\mathsf{T} A\widehat{\mathbf{x}} = A^\mathsf{T}\mathbf{b}$$

If A is full rank, which it usually is in applications, its QR decomposition provides an efficient way to solve the normal equation. Let $A = \widehat{Q}\widehat{R}$ be the reduced QR decomposition of A, so $\widehat{Q}$ is $m \times n$ with orthonormal columns and $\widehat{R}$ is $n \times n$, invertible, and upper triangular. Since $\widehat{Q}^\mathsf{T}\widehat{Q} = I$, and since $\widehat{R}^\mathsf{T}$ is invertible, the normal equation can be reduced as follows (we omit the hats on $\widehat{Q}$ and $\widehat{R}$ for clarity):

$$\begin{aligned} A^\mathsf{T} A\widehat{\mathbf{x}} &= A^\mathsf{T}\mathbf{b} \\ (QR)^\mathsf{T} QR\widehat{\mathbf{x}} &= (QR)^\mathsf{T}\mathbf{b} \\ R^\mathsf{T} Q^\mathsf{T} QR\widehat{\mathbf{x}} &= R^\mathsf{T} Q^\mathsf{T}\mathbf{b} \\ R^\mathsf{T} R\widehat{\mathbf{x}} &= R^\mathsf{T} Q^\mathsf{T}\mathbf{b} \\ R\widehat{\mathbf{x}} &= Q^\mathsf{T}\mathbf{b} \end{aligned} \tag{4.1}$$

Thus $\widehat{\mathbf{x}}$ is the least squares solution to $A\mathbf{x} = \mathbf{b}$ if and only if $\widehat{R}\widehat{\mathbf{x}} = \widehat{Q}^\mathsf{T}\mathbf{b}$. Since $\widehat{R}$ is upper triangular, this equation can be solved quickly with back substitution.

[1] See Volume I Chapter 3 for a formal derivation of the normal equation.

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n and a vector $\mathbf{b}$ of length n. Use the QR decomposition and Equation 4.1 to solve the normal equation corresponding to $A\mathbf{x} = \mathbf{b}$.

You may use either SciPy's QR routine or one of your own routines from the previous lab. In addition, you may use `la.solve_triangular()`, SciPy's optimized routine for solving triangular systems.

Fitting a Line

The least squares solution can be used to find the best fit curve of a chosen type to a set of points. Consider the problem of finding the line $y = ax + b$ that best fits a set of m points $\{(x_k, y_k)\}_{k=1}^m$. Ideally, we seek a and b such that $y_k = ax_k + b$ for all k. The following linear system simultaneously represents all of these equations.

$$A\mathbf{x} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b} \tag{4.2}$$

Note that A has full column rank as long as not all of the x_k values are the same.

Because this system has two unknowns, it is guaranteed to have a solution if it has two or fewer equations. However, if there are more that two data points, the system is overdetermined if any set of three points are not collinear. We therefore seek a least squares solution, which in this case means finding the slope $\widehat{a}$ and y-intercept $\widehat{b}$ such that the line $y = \widehat{a}x + \widehat{b}$ best fits the data.

Figure 4.1 is a typical example of this idea where $\widehat{a} \approx \frac{1}{2}$ and $\widehat{b} \approx -3$.

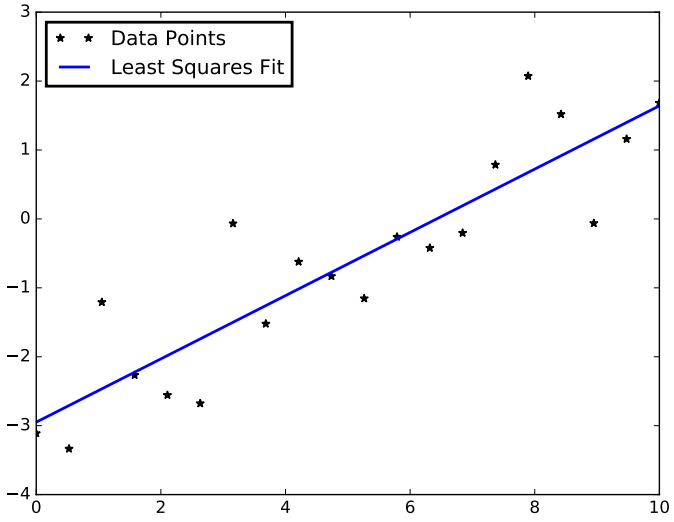


Figure 4.1

Problem 2. The file `housing.npy` contains the purchase-only housing price index, a measure of how housing prices are changing, for the United States from 2000 to 2010.[a] Each row in the array is a separate measurement; the columns are the year and the price index, in that order. To avoid large numerical computations, the year measurements start at 0 instead of 2000.

Find the least squares line that relates the year to the housing price index (i.e., let year be the x-axis and index the y-axis).

1. Construct the matrix A and the vector $\mathbf{b}$ described by Equation 4.2.
   (Hint: the functions `np.vstack()`, `np.column_stack()`, and/or `np.ones()` may be helpful.)
2. Use your function from Problem 1 to find the least squares solution.
3. Plot the data points as a scatter plot.
4. Plot the least squares line with the scatter plot.

[a] See `http://www.fhfa.gov/DataTools/Downloads/Pages/House-Price-Index.aspx`.

Note

The least squares problem of fitting a line to a set of points is often called *linear regression*, and the resulting line is called the *linear regression line*. SciPy's specialized tool for linear regression is `scipy.stats.linregress()`. This function takes in an array of x-coordinates and a corresponding array of y-coordinates, and returns the slope and intercept of the regression line, along with a few other statistical measurements.

For example, the following code produces Figure 4.1.

```
>>> import numpy as np
>>> from scipy.stats import linregress

# Generate some random data close to the line y = .5x - 3.
>>> x = np.linspace(0, 10, 20)
>>> y = .5*x - 3 + np.random.randn(20)

# Use linregress() to calculate m and b, as well as the correlation
# coefficient, p-value, and standard error. See the documentation for
# details on each of these extra return values.
>>> a, b, rvalue, pvalue, stderr = linregress(x, y)

>>> plt.plot(x, y, 'k*', label="Data Points")
>>> plt.plot(x, a*x + b, 'b-', lw=2, label="Least Squares Fit")
>>> plt.legend(loc="upper left")
>>> plt.show()
```

Fitting a Polynomial

Least squares can also be used to fit a set of data to the best fit polynomial of a specified degree. Let $\{(x_k, y_k)\}_{k=1}^m$ be the set of m data points in question. The general form for a polynomial of degree n is as follows:

$$p_n(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_2 x^2 + c_1 x + c_0$$

Note that the polynomial is uniquely determined by its $n+1$ coefficients $\{c_k\}_{k=0}^n$. Ideally, then, we seek the set of coefficients $\{c_k\}_{k=0}^n$ such that

$$y_k = c_n x_k^n + c_{n-1}x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all values of k. These m linear equations yield the following linear system:

$$A\mathbf{x} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b} \tag{4.3}$$

If $m > n+1$ this system is overdetermined, requiring a least squares solution.

Working with Polynomials in NumPy

The $m \times (n+1)$ matrix A of Equation 4.3 is called a *Vandermonde matrix*.[2] NumPy's `np.vander()` is a convenient tool for quickly constructing a Vandermonde matrix, given the values $\{x_k\}_{k=1}^m$ and the number of desired columns.

```
>>> print(np.vander([2, 3, 5], 2))
[[2 1]
 [3 1]
 [5 1]]

>>> print(np.vander([2, 3, 5, 4], 3))
[[ 4  2  1]
 [ 9  3  1]
 [25  5  1]
 [16  4  1]]
```

NumPy also has powerful tools for working efficiently with polynomials. The class `np.poly1d` represents a 1-dimensional polynomial. Instances of this class are callable like a function.[3] The constructor accepts the polynomial's coefficients, from largest degree to smallest.

Table 4.1 lists the attributes and methods of the `np.poly1d` class. See `http://docs.scipy.org/doc/numpy/reference/routines.polynomials.html` for a list of NumPy's polynomial routines.

[2] Vandermonde matrices have many special properties and are useful for many applications, including polynomial interpolation and discrete Fourier analysis.

[3] Class instances can be made callable by implementing the `__call__()` magic method.

Attribute	Description
coeffs	The $n+1$ coefficients, from greatest degree to least.
order	The polynomial degree (n).
roots	The $n-1$ roots.
Method	Returns
deriv()	The coefficients of the polynomial after being differentiated.
integ()	The coefficients of the polynomial after being integrated (with $c_0 = 0$).

Table 4.1: Attributes and methods of the `np.poly1d` class.

```
# Create a callable object for the polynomial f(x) = (x-1)(x-2) = x^2 - 3x + 2.
>>> f = np.poly1d([1, -3, 2])
>>> print(f)
   2
1 x - 3 x + 2

# Evaluate f(x) for several values of x in a single function call.
>>> f([1, 2, 3, 4])
array([0, 0, 2, 6])

# Evaluate f(x) at 1, 2, 3, and 4 without creating f(x) explicitly.
>>> np.polyval([1, -3, 2], [1, 2, 3, 4])
array([0, 0, 2, 6])
```

Problem 3. The data in `housing.npy` is nonlinear, and might be better fit by a polynomial than a line.

Write a function that uses Equation 4.3 to calculate the polynomials of degree 3, 6, 9, and 12 that best fit the data. Plot the original data points and each least squares polynomial together in individual subplots.
(Hint: define a separate, more refined domain with `np.linspace()` and use this domain to smoothly plot the polynomials.)

Instead of using Problem 1 to solve the normal equation, you may use `scipy.linalg.lstsq()`, demonstrated below.

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

# Define A and b appropriately.

# Solve the normal equation using SciPy's least squares routine.
# The least squares solution is the first of four return values.
>>> x = la.lstsq(A, b)[0]
```

Compare your results to `np.polyfit()`. This function receives an array of x values, an array of y values, and an integer for the polynomial degree, and returns the coefficients of the best fit polynomial of that degree.

ACHTUNG!

Having more parameters in a least squares model is not always better. For a set of m points, the best fit polynomial of degree $m-1$ *interpolates* the data set, meaning that $p(x_k) = y_k$ exactly for each k. In this case there are enough unknowns that the system is no longer overdetermined. However, such polynomials are highly subject to numerical errors and are unlikely to accurately represent true patterns in the data.

Choosing to have too many unknowns in a fitting problem is (fittingly) called *overfitting*, and is an important issue to avoid in any statistical model.

Fitting a Circle

Suppose the set of m points $\{(x_k, y_k)\}_{k=1}^m$ are arranged in a nearly circular pattern. The general equation of a circle with radius r and center (c_1, c_2) is as follows:

$$(x-c_1)^2 + (y-c_2)^2 = r^2. \tag{4.4}$$

The circle is uniquely determined by r, c_1, and c_2, so these are the parameters that should be solved for in a least squares formulation of the problem. However, Equation 4.4 is not linear in any of these variables.

$$\begin{aligned} (x-c_1)^2 + (y-c_2)^2 &= r^2 \\ x^2 - 2c_1x + c_1^2 + y^2 - 2c_2y + c_2^2 &= r^2 \\ x^2 + y^2 &= 2c_1x + 2c_2y + r^2 - c_1^2 - c_2^2 \end{aligned} \tag{4.5}$$

The quadratic terms x^2 and y^2 are acceptable because the points $\{(x_k, y_k)\}_{k=1}^m$ are given. To eliminate the nonlinear terms in the unknown parameters r, c_1, and c_2, define a new variable $c_3 = r^2 - c_1^2 - c_2^2$. Then for each point (x_k, y_k), Equation 4.5 becomes the following:

$$2c_1x_k + 2c_2y_k + c_3 = x_k^2 + y_k^2$$

These m equations are linear in c_1, c_2, and c_3, and can be written as a linear system.

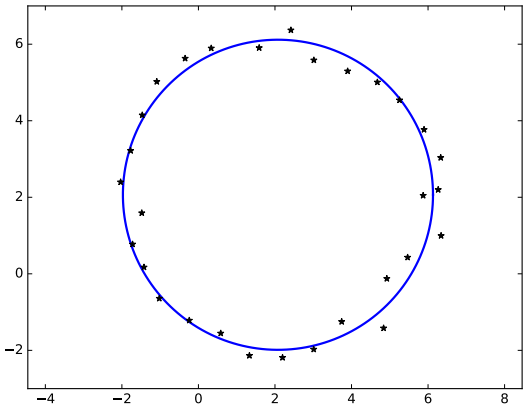
$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_m & 2y_m & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_m^2 + y_m^2 \end{bmatrix} \tag{4.6}$$

After solving for the least squares solution, r can be recovered with the relation $r = \sqrt{c_1^2 + c_2^2 + c_3}$. Finally, plotting a circle is best done with polar coordinates. Using the same variables as before, the circle can be represented in polar equations with the following equations:

$$x = r\cos(\theta) + c_1 \qquad\qquad y = r\sin(\theta) + c_2,$$

where $\theta \in [0, 2\pi]$.

```
# Load some data and construct the matrix A and the vector b.
>>> xk, yk = np.load("circle.npy").T
>>> A = np.column_stack((2*x, 2*y, np.ones_like(x)))
>>> b = xk**2 + yk**2

# Calculate the least squares solution and calculate the radius.
>>> c1, c2, c3 = la.lstsq(A, b)[0]
>>> r = np.sqrt(c1**2 + c2**2 + c3)

# Plot the circle using polar coordinates.
>>> theta = np.linspace(0, 2*np.pi, 200)
>>> x = r*np.cos(theta) + c1
>>> y = r*np.sin(theta) + c2
>>> plt.plot(x, y, '-', lw=2)
>>> plt.plot(xk, yk, 'k*')
>>> plt.axis("equal")
```



Problem 4. The general equation for an ellipse is

$$ax^2 + bx + cxy + dy + ey^2 = 1.$$

Write a function that calculates the parameters for the ellipse that best fits the data in the file `ellipse.npy`. Plot the original data points and the ellipse together, using the following function to plot the ellipse.

```
def plot_ellipse(a, b, c, d, e):
    """Plot an ellipse of the form ax^2 + bx + cxy + dy + ey^2 = 1."""
    theta = np.linspace(0, 2*np.pi, 200)
    cos_t, sin_t = np.cos(theta), np.sin(theta)
    A = a*(cos_t**2) + c*cos_t*sin_t + e*(sin_t**2)
```

```
    B = b*cos_t + d*sin_t
    r = (-B + np.sqrt(B**2 + 4*A))/(2*A)

    plt.plot(r*cos_t, r*sin_t, lw=2)
    plt.gca().set_aspect("equal", "datalim")
```

Computing Eigenvalues

The eigenvalues of an $n \times n$ matrix A are the roots of its characteristic polynomial $\det(A - \lambda I)$. Thus, finding the eigenvalues of A amounts to computing the roots of a polynomial of degree n. However, for $n \geq 5$, it is provably impossible to find an algebraic closed-form solution to this problem.[4] In addition, numerically computing the roots of a polynomial is a famously ill-conditioned problem, meaning that small changes in the coefficients of the polynomial (brought about by small changes in the entries of A) may yield wildly different results. Instead, eigenvalues must be computed with iterative methods.

The Power Method

The *dominant eigenvalue* of the $n \times n$ matrix A is the unique eigenvalue of greatest magnitude, if such an eigenvalue exists. The *power method* iteratively computes the dominant eigenvalue of A and its corresponding eigenvector.

Begin by choosing a vector $\mathbf{x}_0$ such that $\|\mathbf{x}_0\| = 1$, and define the following:

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{\|A\mathbf{x}_k\|}$$

If A has a dominant eigenvalue λ, and if the projection of $\mathbf{x}_0$ onto the subspace spanned by the eigenvectors corresponding to λ is nonzero, then the sequence of vectors $\{\mathbf{x}_k\}_{k=0}^{\infty}$ converges to an eigenvector $\mathbf{x}$ of A corresponding to λ.

Since $\mathbf{x}$ is an eigenvector of A, $A\mathbf{x} = \lambda\mathbf{x}$. Left multiplying by $\mathbf{x}^\mathsf{T}$ on each side gives $\mathbf{x}^\mathsf{T}A\mathbf{x} = \lambda\mathbf{x}^\mathsf{T}\mathbf{x}$, and hence $\lambda = \frac{\mathbf{x}^\mathsf{T}A\mathbf{x}}{\mathbf{x}^\mathsf{T}\mathbf{x}}$. This ratio is called the *Rayleigh quotient*. However, since each $\mathbf{x}_k$ is normalized, $\mathbf{x}^\mathsf{T}\mathbf{x} = \|\mathbf{x}\|^2 = 1$, so $\lambda = \mathbf{x}^\mathsf{T}A\mathbf{x}$.

The entire algorithm is summarized below.

Algorithm 4.1

1: **procedure** POWER METHOD(A)
2: $m, n \leftarrow \text{shape}(A)$ ▷ A is square so $m = n$.
3: $\mathbf{x}_0 \leftarrow \text{random}(n)$ ▷ A random vector of length n
4: $\mathbf{x}_0 \leftarrow \mathbf{x}_0/\|\mathbf{x}_0\|$ ▷ Normalize $\mathbf{x}_0$
5: **for** $k = 1,\ 2,\ \ldots,\ N-1$ **do**
6: $\mathbf{x}_{k+1} \leftarrow A\mathbf{x}_k$
7: $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1}/\|\mathbf{x}_{k+1}\|$
8: **return** $\mathbf{x}_N^\mathsf{T}A\mathbf{x}_N,\ \mathbf{x}_N$

[4] This result, called *Abel's impossibility theorem*, was first proven by Niels Heinrik Abel in 1824.

The power method is limited by a few assumptions. First, not all square matrices A have a dominant eigenvalue. However, the Perron-Frobenius theorem guarantees that if all entries of A are positive, then A has a dominant eigenvalue. Second, there is no way to choose an $\mathbf{x}_0$ that is guaranteed to have a nonzero projection onto the span of the eigenvectors corresponding to λ, though a random $\mathbf{x}_0$ will almost surely satisfy this condition. Even with these assumptions, a rigorous proof that the power method converges is most convenient with tools from spectral calculus. See Chapter ?? of the Volume I text for details.

Problem 5. Write a function that accepts an $n \times n$ matrix A, a maximum number of iterations N, and a stopping tolerance `tol`. Use Algorithm 4.1 to compute the dominant eigenvalue of A and a corresponding eigenvector.

Continue the loop in step 5 until either $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|$ is less than the tolerance `tol`, or until iterating the maximum number of times N.

Test your function on square matrices with all positive entries, verifying that $A\mathbf{x} = \lambda\mathbf{x}$. Use SciPy's eigenvalue solver, `scipy.linalg.eig()`, to compute all of the eigenvalues and corresponding eigenvectors of A and check that λ is the dominant eigenvalue of A.

```
# Construct a random matrix with positive entries.
>>> A = np.random.random((10,10))

# Compute the eigenvalues and eigenvectors of A via SciPy.
>>> eigs, vecs = la.eig(A)

# Get the dominant eigenvalue and eigenvector of A.
# The eigenvector of the kth eigenvalue is the kth column of 'vecs'.
>>> loc = np.argmax(eigs)
>>> lamb, x = eigs[loc], vecs[:,loc]

# Verify that Ax = lambda x.
>>> np.allclose(A.dot(x), lamb*x)
True
```

The QR Algorithm

An obvious shortcoming of the power method is that it only computes the largest eigenvalue and a corresponding eigenvector. The QR algorithm, on the other hand, attempts to find all eigenvalues of A.

Let $A_0 = A$, and for arbitrary k let $Q_k R_k = A_k$ be the QR decomposition of A_k. Since A is square, so are Q_k and R_k, so they can be recombined in reverse order.

$$A_{k+1} = R_k Q_k$$

This recursive definition establishes an important relation between the A_k.

$$Q_k^{-1} A_k Q_k = Q_k^{-1}(Q_k R_k)Q_k = (Q_k^{-1}Q_k)(R_k Q_k) = A_{k+1}$$

Thus A_k is orthonormally similar to A_{k+1}, and similar matrices have the same eigenvalues. The series of matrices $\{A_k\}_{k=0}^{\infty}$ converges to the following block matrix.

$$S = \begin{bmatrix} S_1 & * & \cdots & * \\ \mathbf{0} & S_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ \mathbf{0} & \cdots & \mathbf{0} & S_m \end{bmatrix}$$

Each S_i is either a 1×1 or 2×2 matrix.[5] Since S is block upper triangular, its eigenvalues are the eigenvalues of its diagonal S_i blocks. Then because A is similar to each A_k, those eigenvalues of S are the eigenvalues of A.

When A has real entries but complex eigenvalues, 2×2 S_i blocks appear in S. Finding eigenvalues of a 2×2 matrix is equivalent to finding the roots of a 2nd degree polynomial, which has a closed form solution via the quadratic equation. This implies that complex eigenvalues come in conjugate pairs.

$$\begin{aligned} \det(S_i - \lambda I) = \begin{vmatrix} a-\lambda & b \\ c & d-\lambda \end{vmatrix} &= (a-\lambda)(d-\lambda) - bc \\ &= \lambda^2 - ad\lambda + (ad - bc) \end{aligned} \tag{4.7}$$

Hessenberg Preconditioning

The QR algorithm works more accurately and efficiently on matrices that are in upper Hessenberg form, as upper Hessenberg matrices are already close to triangular. Furthermore, if $H = QR$ is the QR decomposition of upper Hessenberg H then RQ is also upper Hessenberg, so the almost-triangular form is preserved at each iteration. Putting a matrix in upper Hessenberg form before applying the QR algorithm is called *Hessenberg preconditioning*.

With preconditioning in mind, the entire QR algorithm is as follows.

[5] If all of the S_i are 1×1 matrices, then the upper triangular S is called the *Schur form* of A. If some of the S_i are 2×2 matrices, then S is called the *real Schur form* of A.

Algorithm 4.2

1: **procedure** QR ALGORITHM(A)
2: $m, n \leftarrow \text{shape}(A)$
3: $S \leftarrow \text{hessenberg}(A)$ ▷ Put A in upper Hessenberg form.
4: **for** $k = 0,\ 1,\ \ldots,\ N-1$ **do**
5: $Q, R \leftarrow \text{qr}(S)$ ▷ Get the QR decomposition of A_k.
6: $S \leftarrow RQ$ ▷ Recombine R_k and Q_k into A_{k+1}.
7: `eigs` $\leftarrow$ `[]` ▷ Initialize an empty list of eigenvalues.
8: $i \leftarrow 0$
9: **while** $i < n$ **do**
10: **if** S_i is 1×1 **then**
11: Append S_i to `eigs`
12: **else if** S_i is 2×2 **then**
13: Calculate the eigenvalues of S_i
14: Append the eigenvalues of S_i to `eigs`
15: $i \leftarrow i + 1$
16: $i \leftarrow i + 1$ ▷ Move to the next S_i.
17: **return** `eigs`

Problem 6. Write a function that accepts an $n \times n$ matrix A, a number of iterations N, and a tolerance `tol`. Use Algorithm 4.2 to implement the QR algorithm with Hessenberg preconditioning, returning the eigenvalues of A.

Consider the following implementation details.

- Use `scipy.linalg.hessenberg()` to reduce A to upper Hessenberg form in step 3, or use your own Hessenberg function from the previous lab.
- The loop in step 4 should run for N total iterations.
- Use `scipy.linalg.qr()` to compute the QR decomposition of S in step 5, or use your own QR factorization routine from the previous lab (since S is upper Hessenberg, Givens rotations are the most efficient way to produce Q and R).
- Assume that S_i is 1×1 in step 10 if one of two criteria hold:
  1. S_i is the last diagonal entry of S.
  2. The absolute value of element below the ith main diagonal entry of S (the lower left element of the 2×2 block) is less than `tol`.
- If S_i is 2×2, use the quadratic formula and Equation 4.7 to compute its eigenvalues. Use the function `cmath.sqrt()` to correctly compute the square root of a negative number.

Test your function on small random symmetric matrices, comparing your results to `scipy.linalg.eig()`. To construct a random symmetric matrix, note that $A + A^\mathsf{T}$ is always symmetric.

Note

Algorithm 4.2 is theoretically sound, but can still be greatly improved. Most modern computer packages instead use the *implicit QR algorithm*, an improved version of the QR algorithm, to compute eigenvalues.

For large matrices, there are other iterative methods besides the power method and the QR algorithm for efficiently computing eigenvalues. They include the Arnoldi iteration, the Jacobi method, the Rayleigh quotient method, and others. We will return to this subject after studying spectral calculus and Krylov subspaces.

5 Image Segmentation

Lab Objective: *Understand some basic applications of eigenvalues to graph theory. Learn how to calculate the Laplacian matrix of a graph. Apply the Laplacian matrix to determine connectivity of a graph and segment an image.*

Graph Theory

Graph theory is a branch of mathematics dealing with mathematical structures called graphs. Graphs represent relationships between objects. An *undirected graph* is a set of nodes (or vertices) and edges, where each edge connects exactly two nodes (see Figure 5.1). In a *directed graph*, edges are directional. Each edge only goes one way, usually visualized as an arrow pointing from one node to another. In this lab, we will only consider undirected graphs, which we will simply call graphs (unless we wish to emphasize the fact that they are undirected).

A *weighted* graph is a graph with a weight attached to each edge. For example, a weighted graph could represent a collection of cities with roads connecting them. The vertices would be cities, the edges would be roads, and weight of an edge would be the length of a road. Such a graph is depicted in Figure 5.2.

Any unweighted graph can be thought of as a weighted graph by assigning a weight of 1 to each edge.

Adjacency, Degree, and Laplacian Matrices

We will now introduce three matrices associated with a graph. Throughout this section, assume we are working with a weighted undirected graph with N nodes, and that w_{ij} is the weight attached to the edge connecting node i and node j. We first define the adjacency matrix.

Definition 5.1. *The* adjacency matrix *is an $N \times N$ matrix whose (i,j)-th entry is*

$$\begin{cases} w_{ij} & \text{if an edge connects node } i \text{ and node } j \\ 0 & \text{otherwise.} \end{cases}$$

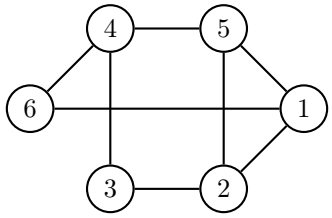


Figure 5.1: An undirected graph that is connected.

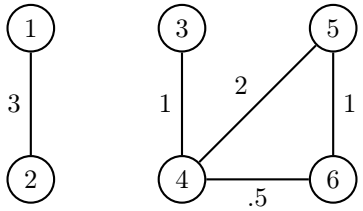


Figure 5.2: A weighted undirected graph that is not connected.

For example, the graph in Figure 5.1 has the adjacency matrix A_1 and the graph in Figure 5.2 has the adjacency matrix A_2, where

$$A_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & .5 \\ 0 & 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & .5 & 1 & 0 \end{pmatrix}.$$

Notice that these adjacency matrices are symmetric. This will always be the case for undirected graphs.

The second matrix is the degree matrix.

Definition 5.2. *The* degree matrix *is an $N \times N$ diagonal matrix whose (i,i)-th entry is*

$$\sum_{j=1}^{N} w_{ij}.$$

This quantity is the sum of the weight of each edge leaving node i.

We call the (i,i)-th entry of the degree matrix the *degree* of node i. As an example, the degree matrices of the graphs in Figures 5.1 and 5.2 are D_1 and D_2, respectively.

$$D_1 = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}. \qquad D_2 = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.5 \end{pmatrix}$$

Finally, we can combine the degree matrix and the adjacency matrix to get the Laplacian matrix.

Definition 5.3. *The* Laplacian matrix *of a graph is*

$$D - A$$

where D is the degree matrix and A is the adjacency matrix of the graph.

For example, the Laplacian matrix of the graphs in Figures 5.1 and 5.2 are L_1 and L_2, respectively, where

$$L_1 = \begin{pmatrix} 3 & -1 & 0 & 0 & -1 & -1 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ -1 & 0 & 0 & -1 & 0 & 2 \end{pmatrix}. \qquad L_2 = \begin{pmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{pmatrix}$$

In this lab we will learn about graphs by studying their Laplacian matrices. While the Laplacian matrix seems simple, we can learn surprising things from its eigenvalues.

Problem 1. Write a function that accepts the adjacency matrix of a graph as an argument and returns the Laplacian matrix. Test your function on the graphs in Figures 5.1 and 5.2.

Hint: You can compute the diagonal of the degree matrix in one line by summing over an axis (see Lab 2 and NumPy Visual Guide).

Connectivity: First Application of Laplacians

A *connected graph* is a graph where every vertex is connected to every other vertex by at least one path. The graph in Figure 5.1 is connected, whereas the graph in Figure 5.2 is not. It is often important to know if a graph is connected. A naive approach to determine connectivity of a graph is to search every possible path from each vertex. While this works for very small graphs, most interesting graphs will have thousands of vertices, and for such graphs this approach is not feasible.

Instead of the naive approach, we can use an interesting result from algebraic graph theory. This result relates the connectivity of a graph to its Laplacian.

The Laplacian of any graph always has at least one zero eigenvalue. Why is this true? If L is the Laplacian matrix of a graph, then the rows of L must sum to 0. (Think about how L was created.) Since this is true, L cannot have full rank, so $\lambda = 0$ must be an eigenvalue of L.

Furthermore, if L represents a graph that is *not* connected, more than one of the eigenvalues of L will be zero. To see this, let $J \subset \{1 \dots N\}$ such that the vertices $\{v_j\}_{j \in J}$ form a connected component of the graph. Define a vector $\mathbf{x}$ such that

$$\mathbf{x}_j = \begin{cases} 1, & j \in J \\ 0, & j \notin J \end{cases}$$

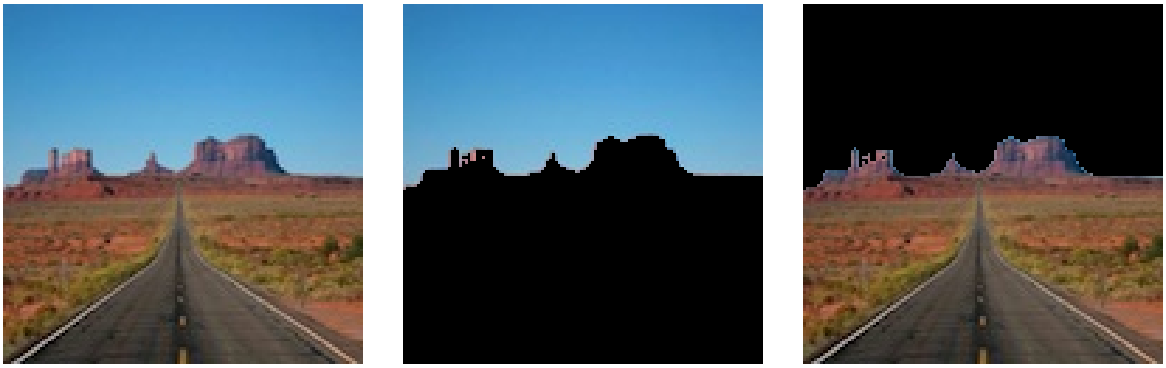


Figure 5.3: An image and its segments.

Then $\mathbf{x}$ is an eigenvector of L corresponding to the eigenvalue $\lambda = 0$. (Look at the Laplacian matrix in Figure 5.2 and consider the product $L_2\mathbf{x}$.) In other words, for each connected component, 0 appears at least once as an eigenvalue.

In fact, it can be rigorously proven that the number of zero eigenvalues of the Laplacian exactly *equals* the number of connected components. If we can solve for the eigenvalues of L, this makes it simple to calculate how many connected components are in the graph.

L is always a positive semi-definite matrix when all weights in the graph are positive, so all of its eigenvalues are greater than or equal to 0. The second smallest eigenvalue of L is known as the *algebraic connectivity* of the graph. It is clearly 0 for non-connected graphs. For connected graphs, the algebraic connectivity can give us useful information about the sparsity or "connectedness" of a graph. A higher algebraic connectivity indicates that the graph is more strongly connected.

Problem 2. Compute the number of connected components in a graph. Write a function that accepts the adjacency matrix of a graph and returns two arguments: the number of connected components, and the algebraic connectivity of the graph (second smallest eigenvalue of the Laplacian).

Use the `scipy.linalg` package to compute the eigenvalues. Note that this package will return complex eigenvalues (with negligible imaginary parts). Keep only the real parts. Your function should also accept a tolerance value, such that all eigenvalues less than this value are assumed to be zero. This should default to `tol=1e-8`.

Image Segmentation: Second Application of Laplacians

Image segmentation is the process of finding natural boundaries in an image (see Figure 5.3). This is a simple task for humans, who can easily pick out portions of an image that "belong together." In this lab, you will learn one way for a computer to segment images using graph theory.

The algorithm we will present comes from a paper by Jianbo Shi and Jitendra Malik in 2000 ([**Shi2000**]). They segment an image using the following steps.

1. **Represent the image as a weighted graph of pixels.** An image is made up of individual pixels, each having a brightness and a location. (Here, *brightness* is equivalent to the grayscale value of the pixel.) To define a graph representing the image, we let each pixel be a vertex. The weight of the edges between two pixels is determined by their distance apart and their similarity in brightness. We define the graph so that two similar pixels (i.e., with similar brightness and location) will be connected by a strong edge.

2. **Calculate the Laplacian.** We calculate the adjacency and degree matrices of the graph representing the image, and use these to obtain the Laplacian.

3. **Choose the best cut.** The graph we have created is connected, but not all edges have the same weight. Dissimilar pixels will be connected by edges that have a low weight. We can split the graph into two connected components by "cutting" it along the low-weight edges. These components are the image segments we are looking for. (We can also cut an image multiple times to segment it into more than two pieces.) As we will see later, Shi and Malik's algorithm uses spectral information (eigenvectors) of the Laplacian to minimize the weight of the cut edges.

Defining the Graph and Adjacency Matrix

We now define the graph that represents an image. In our $M \times N$ image, the associated graph will have MN nodes, one representing each pixel. We let w_{ij} be the weight of the edge connecting pixels i and j, and define

$$w_{ij} = \begin{cases} \exp(-\frac{|I(i)-I(j)|}{\sigma_I^2} - \frac{d(i,j)}{\sigma_d^2}) & \text{for } d(i,j) < r \\ 0 & \text{otherwise,} \end{cases} \tag{5.1}$$

where

- $d(i,j)$ is the Euclidean distance between pixel i and pixel j
- $|I(i) - I(j)|$ is the difference in brightness of pixels i and j
- r, σ_I and σ_d are constants that we choose

With this definition, pixels that are farther apart than radius r will never be connected. Pixels within r will be strongly connected if they are similar in brightness ($|I(i) - I(j)|$ is small) and close together ($d(i,j)$ is small). Highly contrasting pixels ($|I(i) - I(j)|$ is large) will be weakly connected. This gives us a graph with the properties that we intuitively want.

The adjacency matrix W is $(MN) \times (MN)$ and has w_{ij} as its ijth entry. W will be sparse as long as r is small. Figure 5.4 shows a visualization of the adjacency matrix for a 4×4 image when $r = 1.2$.

Computing the Adjacency Matrix

We will now write a function to compute the adjacency matrix for a given image. The function will accept a filename and constants `radius`, `sigma_I`, and `sigma_d`, and return the adjacency matrix and the diagonal of the corresponding degree matrix.

The basic approach is straightforward. For each pixel in the image, compare it to every other pixel, use (5.1) to calculate the weight of the edge between them, and fill in the weight in the adjacency matrix. This section will discuss how to implement this efficiently in Python.

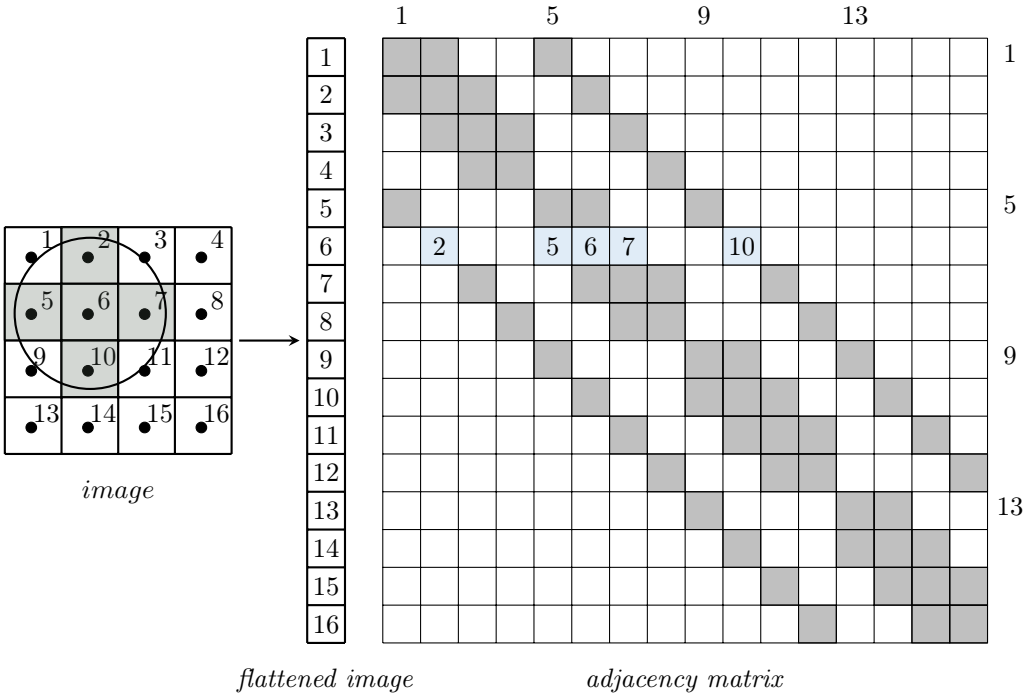


Figure 5.4: The grid on the left represents a 4×4 (or $M \times N$) image with 16 pixels. At right is the corresponding 16×16 (or $(MN) \times (MN)$) adjacency matrix with all nonzero entries shaded. For example, in the 6^{th} row, entries 2, 5, 6, 7, and 10 are nonzero because those pixels are within radius r of pixel 6 (here $r = 1.2$).

First, load an image and convert it to grayscale. The included function `getImage` helps handle color images. It accepts a filename and returns a 3-D representation of the image and a 2-D representation of the brightness values of the image.

It will be useful to flatten the $M \times N$ image. This converts it into a 1-D array of pixels of length MN, essentially giving each pixel an index. We can use `img.flatten` to flatten the array `img`.

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

Next, initialize empty adjacency and degree matrices to fill in. As in Figure 5.4, the adjacency matrix will be sparse, so initialize it as a sparse matrix `W`. Use the sparse matrix type `lil_matrix`, which is optimized for filling in a matrix one entry at a time. Since we only need to store the diagonal of the degree matrix, initialize this diagonal as a regular 1-dimensional NumPy array `D`.

We now fill in `W` according to our definition in Equation 5.1. Each pixel in the image corresponds to one row in `W`. For each pixel:

- Find its neighbors (the pixels in the image that are within distance r of it)
- Use 5.1 to calculate the weights connecting the pixel to each of its neighbors
- Fill in these weights in `W`, leaving the rest of the values in the row as zeros. [1]

[1] Note that `W` will be a symmetric matrix. We could potentially speed up this algorithm by taking advantage of this

The sum of the entries of a row in `W` will be the corresponding entry in `D`.

You may choose to use the provided helper function `getNeighbors` to find the neighbors of a given pixel. The function accepts the index of a pixel in the flattened image, along with a value for r and the original image dimensions. It returns two flat arrays: `indices` and `distances`. The array `indices` contains the indices of the neighbor pixels within distance r of the input pixel. The array `distances` contains the corresponding distances of those pixels from the input pixel. Using Figure 5.4 as an example, with the inputs 6, 1.2, 4, and 4 the outputs would be `indices = array([2,5,6,7,10])` and `distances = array([1,1,0,1,1])`. Try running the function with different inputs to build intuition about what it does.

Finally, convert `W` to the sparse matrix type `csc_matrix`, which is faster for computations. Then return `W` and `D`.

Problem 3. Write the function `adjacency` described in this section. Accept an image a filename and constants `radius`, `sigma_I`, and `sigma_d`. Return the corresponding sparse adjacency matrix `W` and the diagonal of the degree matrix `D`. Use (5.1) to compute the weights in the adjacency matrix. For speed, try to compute an entire row of `W` at once, instead of filling in `W` entry by entry.

Minimizing the 'Cut'

As stated earlier, the goal is to split the image into two segments, while minimizing the weight of the edges that are 'cut' in the corresponding graph. Let L be the Laplacian of the adjacency matrix defined in 5.1 and let D be the degree matrix. Shi and Malik proved that using the second smallest eigenvalue of $D^{-1/2}LD^{-1/2}$, we can minimize the 'cut' . Both D and L will be symmetric matrices, so all eigenvalues of $D^{-1/2}LD^{-1/2}$ will be real, therefore the second smallest one is well-defined. (Note that $D^{-1/2}$ refers not to matrix, but element-wise, exponentiation.)

The eigenvector associated to the second smallest eigenvalue is the key to segmenting the image. This eigenvector will have MN entries. Shi and Malik proved that the indices of its *positive* entries are the indices of the pixels in the flattened image which belong in one segment. Likewise, the indices of its *negative* entries are the indices of the pixels which belong in the other segment.

To compute the segments, reshape this eigenvector as an $M \times N$ array, and set the positive entries to True and the negative entries to False. We can multiply this True-False mask entry-wise by the image. This zeros out the pixels in the image corresponding to the `False` entries in the mask, without affecting the pixels corresponding to `True` entries. We can negate the mask using the tilde operator, which lets us compute the other segment of the image. Finally we return the two segments.

Problem 4. Write the function `segment` to segment an image. The function should accept an image filename and return both of the segments. You should call the code you wrote in Problem 3. Use sparse matrices where possible.

Test on the image `dream.png`. Your segments should look like the segments in Figure 5.5 (the original image is on the left).

Hints:

---

fact.

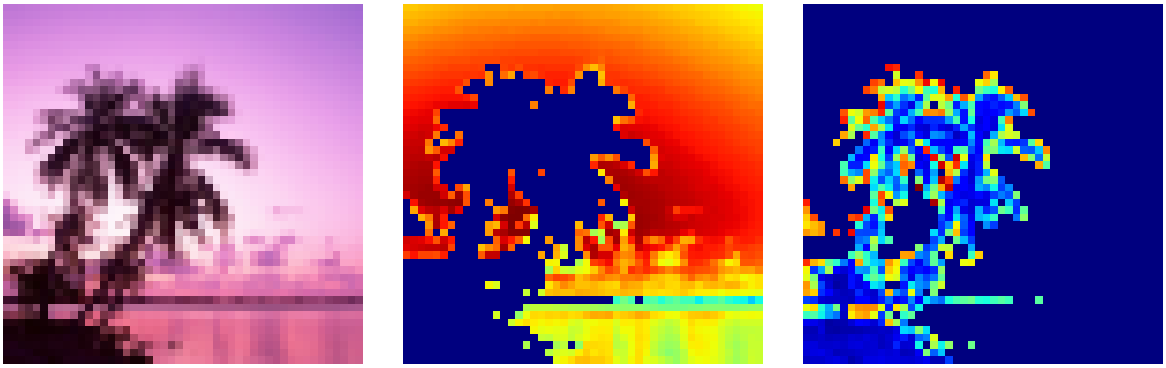


Figure 5.5: Segments of `dream.png`

1. After defining $D^{-1/2}$, convert D and $D^{-1/2}$ into sparse matrices using `scipy.sparse.spdiags`.
2. Since are now dealing with sparse instead of dense matrices, you shouldn't use your solution to Problem 1 to calculate the Laplacian.
3. Multiply sparse matrices with `A.dot(B)`.
4. Use `scipy.sparse.eigsh` to calculate the eigenvector. This is a sparse eigenvalue solver optimized for symmetric matrices. Set the keyword `which = "SM"` to return the smallest eigenvalues.
5. The provided function `displayPosNeg` can be used to plot your images.

6 The SVD and Image Compression

Lab Objective: *Learn how to compute the compact SVD. Explore the SVD as a method of matrix approximation, and use it to perform image compression.*

The *Singular Value Decomposition* or *SVD* is a matrix decomposition that is widely used in both theoretical and applied mathematics. Originally discovered by theoretical mathematicians, it is a canonical way to decompose a matrix. Its practical use became apparent later on when Erhard Schmidt showed that the SVD could be a computational tool for providing low-rank matrix approximations. Modern developments continue to confirm the importance of the SVD in both computational and theoretical applications.

Computing the SVD

The Singular Value Decomposition decomposes an $m \times n$ matrix A into the form

$$A = U\Sigma V^{\mathsf{H}}$$

where U and V are square and unitary of sizes m and n respectively, and Σ is diagonal and of size $m \times n$. The values along the diagonal of Σ are called the *singular values* of A. These are also the square roots of the eigenvalues of $A^{\mathsf{H}}A$. Commonly the singular values are listed in decreasing order. Thus we have

$$\Sigma = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ are the singular values of A.

If A is of rank r, then A has exactly r nonzero singular values. The first r columns of U span the range of A, and the last $n - r$ columns span the null space of A^{H}. Likewise, the first r columns of V span the range of A^{H} and the last $m - r$ span the null space of A.

For a more in-depth definition and proof that the SVD exists for every matrix, refer to the section on the SVD in the text. Here we will focus on computing the SVD.

First, we define two modifications of the regular SVD. In the *compact SVD*, we only keep the r nonzero singular values. Only r column vectors of U and r row vectors of V^{H}, corresponding to the r singular values, are calculated. The compact SVD takes the form $A = U_r\Sigma_r V_r^{\mathsf{H}}$ where U_r is $m \times r$, Σ_r is $r \times r$ and diagonal, and V_r^{H} is $r \times n$. Although we drop the decompositions of the nullspaces, by calculating $U_1\Sigma_1 V_1^{\mathsf{H}}$ we can still recover the full matrix A.

The *truncated SVD* is similar to the compact SVD, but instead of keeping all the nonzero singular values, we only keep the k largest. While this saves space, it means that we cannot recover the whole matrix. Instead we end up with $\widehat{A}_k = U_k \Sigma_k V_k^\mathsf{H}$ where $\widehat{A}_k$ is a rank k approximation of A, U_k is $m \times k$, Σ is $k \times k$ and diagonal, and V_k^H is $k \times n$.

The components of the compact or truncated SVD can be calculated as follows:

- The singular values of A, which form the diagonal of Σ, are the square roots of the eigenvalues of $A^\mathsf{H}A$. These are sorted in descending order. For the compact SVD, keep all of the nonzero singular values. For the truncated SVD, keep only the largest k.

- The columns of V are the eigenvectors of $A^\mathsf{H}A$, where the ith column V_i matches the ith singular value.

- The columns of U are $U_i = \frac{1}{\sigma_i} AV_i$.

Problem 1. Write a function `truncated_svd` that accepts a matrix A and an optional integer `k = None`. If `k` is `None`, calculate the compact SVD. If `k` is an integer, calculate the truncated SVD, keeping only the `k` largest singular values. (Note: if there are fewer than `k` nonzero singular values, the truncated SVD will come out the same as the compact SVD.) Since the only difference between these two processes is the number of singular values we keep, we only need to write one function.

Here's an outline to follow:

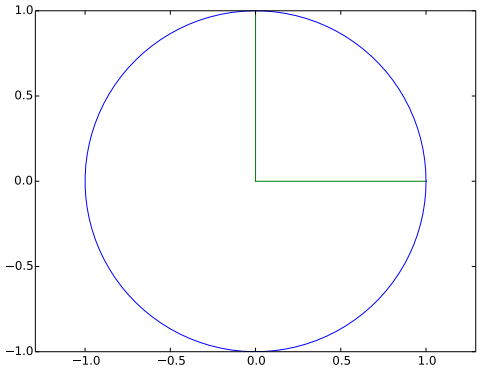
1. Find the eigenvalues and eigenvectors of $A^\mathsf{H}A$.
2. Find the singular values of A. Keep only the greatest `k`, and discard any that are equal to zero.
3. Calculate V.
4. Calculate U.

Return U, the diagonal of Σ, and V. Check your function by calculating the compact SVD and seeing if $U\Sigma V^\mathsf{H} = A$ using `np.allclose()`.

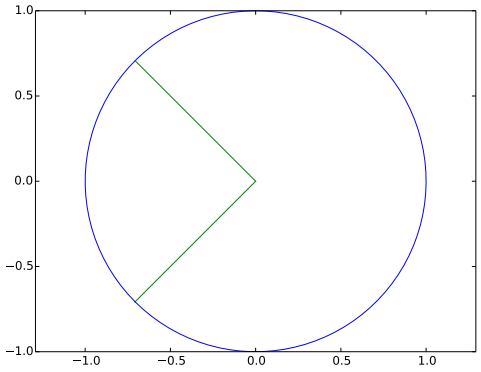
Hint: When calculating the SVD, you will need to sort the eigenvalues while keeping track of their associated eigenvectors. Consider using the function `np.argsort` for keeping the eigenvalues and eigenvectors in the same order while sorting.

Note

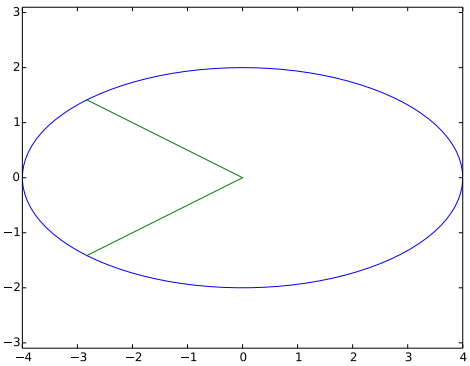
In practice, calculating $A^\mathsf{H}A$ in order to find its eigenvalues is unstable. We use this method here because it is mathematically the simplest. However, industrial SVD solvers use different methods that avoid computing $A^\mathsf{H}A$.



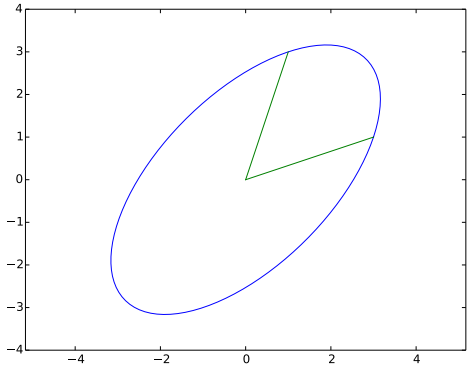
(a) The unit circle, S, with two unit vectors.



(b) $V^{\mathsf{H}}S$



(c) $\Sigma V^{\mathsf{H}}S$



(d) $U\Sigma V^{\mathsf{H}}S$

Figure 6.1: Each step in transforming the unit circle and two unit vectors using the matrix A.

Visualizing the SVD

Recall that a matrix is a way to express a linear transformation. An $m \times n$ matrix defines a linear transformation that sends points from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Intuitively, the SVD can be thought of as breaking a linear transformation into more basic steps. The SVD decomposes a given matrix into two rotations and a scaling. V^{H} represents a rotation, Σ represents a rescaling along the principal axes, and U represents another rotation.

Problem 2. In this problem we will use the SVD to visualize how the matrix

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

acts on points in $\mathbb{R}^2$. Given a set of points S in $\mathbb{R}^2$, we can calculate the transformation AS in steps by using the SVD $A = U\Sigma V^{\mathsf{H}}$.

Specifically, let S be a set of points on the unit circle. To generate the x- and y-coordinates of S, recall the equation for the unit circle in polar coordinates:

$$x = \cos(\theta) \qquad\qquad y = \sin(\theta),$$

where $\theta \in [0, 2\pi]$.

Plot four separate subplots to demonstrate each step of the transformation, plotting S, $V^{\mathsf{H}}S$, $\Sigma V^{\mathsf{H}}S$, then $U\Sigma V^{\mathsf{H}}$. Do the same for the standard basis vectors $\mathbf{e}_1 = [1, 0]^{\mathsf{T}}$ and $\mathbf{e}_2 = [0, 1]^{\mathsf{T}}$. Your solution should look similar to Figure 6.1.
(Hint: Force the plot to use the same scale on each of the axes with `plt.axis("equal")`. Otherwise, the circle will appear elliptical.)

The SVD and Data Compression

We now turn to computational uses of the SVD. We will explore how the SVD is useful for matrix approximations, and use it to compress images.

Low-Rank Matrix Approximation

If the rank r of a matrix A is significantly smaller than its dimensions, the compact SVD offers a way to store A with less memory. Storing an $m \times n$ matrix requires storing mn values. By decomposing the original matrix into the compact SVD, U_r, Σ_r and V_r together require $mr + r + nr$ values. This is an efficient storage method if r is much smaller than both m and n. For example, suppose $m = 100$, $n = 200$ and $r = 20$. Then the original matrix would require storing $20,000$ values, whereas the compact SVD only requires storing 6020.

The truncated SVD allows even greater efficiency. By only keeping the first k singular values, we can create an approximation $\widehat{A}_k = U_k \Sigma_k V_k^{\mathsf{H}}$. This requires storing only $mk + k + nk$ values. As we make k small, we eventually require very little storage. This comes at the cost of losing information from the original matrix.

The beauty of the SVD is that it makes it easy to only keep the information that is most important. Larger singular values correspond to columns of U and V that contain more information, so dropping the smallest singular values retains as much information as possible. In mathematical terms, given a matrix A and its rank-k truncated SVD approximation $\widehat{A}_k = U_k \Sigma_k V_k^{\mathsf{H}}$, the matrix $\widehat{A}_k$ is the *best rank k approximation* to A (with respect to the induced 2-norm and Frobenius norm). This is a very significant concept in applied mathematics, appearing in areas including signal processing, statistics, semantic indexing, and control theory.

Implementation

We can use SciPy's linear algebra module to create low-rank SVD approximations or a given matrix. The code below computes the SVD of `A`.

```
>>> import numpy as np
>>> import scipy.linalg as la
>>> A = np.array([[1,1,3,4], [5,4,3,7], [9,10,10,12], [13,14,15,16], ↩
    [17,18,19,20]])
>>> U,s,Vh = la.svd(A, full_matrices=False)
```

In the last line of code, we included the keyword argument `full_matrices=False` to calculate the compact SVD rather than the full SVD. The arrays `U` and `Vh` correspond to the matrices U_r and V_r^H discussed earlier. The array `s` gives the nonzero singular values of the matrix `A`, and we can find the rank of `A` by inspecting the number of entries in `s` (here we have a rank 4 matrix).

Next, we calculate a rank 3 approximation. We take the first three singular values, first three columns of `U`, and first three rows of `Vh`. We omit the last singular value from the calculation along with the last column of `U` and last row of `Vh`.

```
>>> S = np.diag(s[:3])
>>> Ahat = U[:,:3].dot(S).dot(Vh[:3,:])
>>> la.norm(A-Ahat)
```

Note that $\widehat{A}$ is "close" to the original matrix A, but that its rank is 3 instead of 4.

Problem 3. Write a function `svd_approx` that takes as input a matrix A and a positive integer k and returns the best rank k approximation to A (with respect to the induced 2-norm and Frobenius norm). Use `scipy.linalg.svd`.

Error of Low-Rank Approximations

Recall that the error between the best rank s approximation $\widehat{A_s}$ of A with respect to the induced 2-norm is given by

$$\|A - \widehat{A_s}\|_2 = \sigma_{s+1},$$

where σ_{s+1} is the $(s+1)$-th singular value of A. (See the proof of the Schmidt-Eckard-Young-Mirsky theorem in the text).

This offers a way to approximate a matrix within an error tolerance: choose the truncated SVD approximation such that the largest discarded singular value is less than the error tolerance.

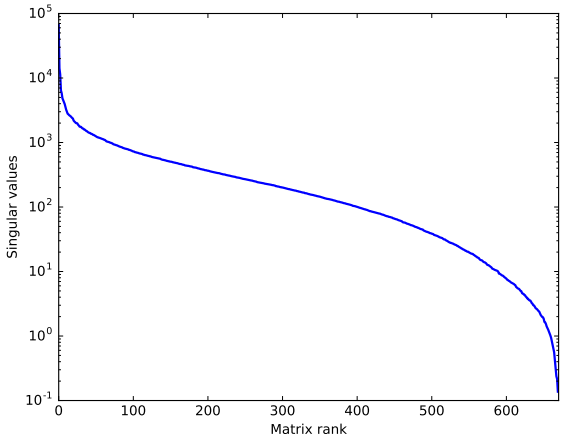
Problem 4. Using `scipy.linalg.svd`, write a function `lowest_rank_approx` that takes as input a matrix A and a positive number e and returns the lowest rank approximation of A with error less than e (with respect to the induced 2-norm). You should only calculate the SVD once.

Application to Image Compression

Sometimes there is not enough available bandwidth to transmit a full resolution photograph. Suppose you need to transmit an image from a remote location. You might aim to reduce the amount of data being sent, while also minimizing the loss of detail in the image.

This can be done using the SVD. An image is just a matrix of pixel values, which means it has a singular value decomposition. Computing and sending a low-rank SVD approximation of the image can considerably reduce the amount of data sent, while retaining a high level of image detail. Additionally, successive levels of detail can be sent after the inital low-rank approximation by sending additional singular values and their corresponding columns of V and U.

(a) NGC 3603 (Hubble Space Telescope).



(b) Singular values from greatest to smallest on a log scale

Figure 6.2: An image and its singular values.

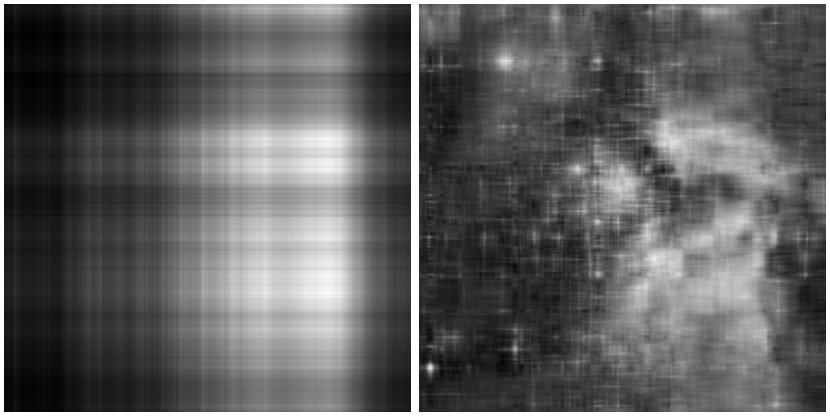
Examining the singular values of an image gives us an idea of how low-rank the approximation can be. Figure 6.2 presents an image and a log plot of its singular values from greatest to least. The plot in 6.2b is typical for a photograph—the singular values start out large but drop off rapidly. In this rank 670 image, 624 of the singular values are 50 or more times smaller than the largest singular value. By discarding these relatively small singular values, we can retain all but the finest image details, while storing only a rank 46 image! This is a huge reduction in data size.

Figure 6.3 shows several low-rank approximations of the image in Figure 6.2a. Even at a low rank the image is recognizable. By rank 40, the approximation visibly differs very little from the original.

The following code demonstrates how to use `plt.imread` and `plt.imshow` to read in an image, convert it to black and white, and show it. The function from Problem 3 can then be used to calculate an approximation of `X`.

```
>>> import matplotlib.pyplot as plt
>>> # Take only one layer (layer 0) of the image
>>> X = plt.imread('hubble_image.jpg')[:,:,0].astype(float)
% >>> X.nbytes     #number of bytes needed to store X  <---- this seems ↩
    superfluous
>>> plt.imshow(X, cmap="gray")
>>> plt.show()
```

Problem 5. Using the `svd_approx` function from Problem 3, write a function `compress_img` that accepts two parameters `filename` and `k`. The function should plot the original image and the best rank k approximation of the original image.



(a) Rank 1 (b) Rank 14

(c) Rank 27 (d) Rank 40

Figure 6.3: Different rank approximations for SVD-based compression. Notice that higher rank is needed to resolve finer detail.

While `svd_approx` worked for grayscale images, the `compress_img` function should work on color images. You may split the image into its three RGB layers and approximate each layer separately, then recombine them. Test your function on `hubble_image.jpg` Your output should be similar to Figure 6.4.

Hints:

- Sometimes `plt.imshow` does not behave as expected when being passed RGB values between 0 and 255. It behaves much better when being passed values between 0 and 1.
- Since the SVD provides an approximation, it is possible that the SVD will generate values slightly outside the valid range of RGB values. To fix this, use fancy indexing (as discussed in the NumPy and SciPy lab) to set values greater than 1 to 1 and values less than 0 to 0.

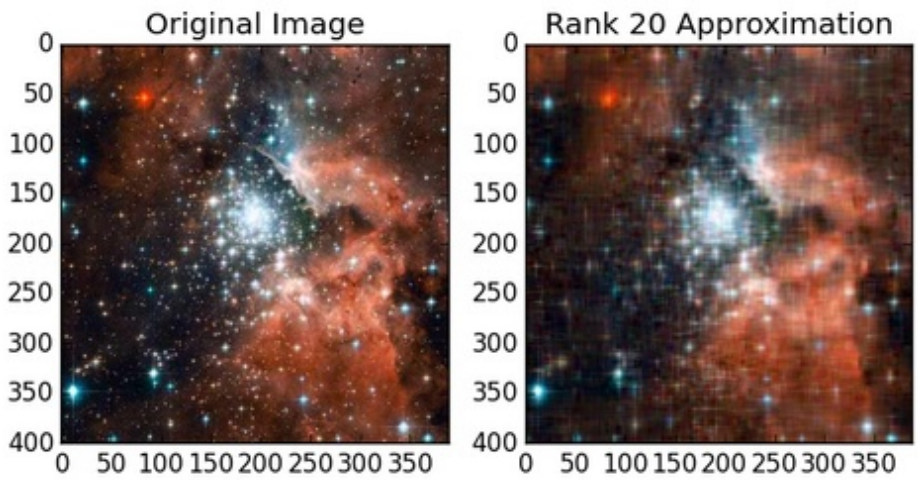


Figure 6.4: Correct output for the best rank 20 approximation.

7 Facial Recognition using Eigenfaces

Lab Objective: *Use the singular value decomposition to implement a simple facial recognition system.*

Suppose we have a large database containing images of human faces. We would like to identify people by comparing their pictures to those in the database. This task is called *facial recognition*.

One way to automate the comparison process is know as the *eigenfaces* method. As the name suggests, this method uses eigenvectors of matrices related to the collection of face images. The method essentially projects face images to a low-dimensional subspace, in a way that preserves their distinguishing characteristics. Comparing the images in fewer dimensions is faster and allows us to store the images using less data.

The idea of projecting to fewer dimensions is not unique to the eigenfaces method. This method is an example of *principal component analysis*, where data is compared based on its principal components in a lower-dimensional vector space. Principal component analysis can be applied to many computing problems besides facial recognition.

Load the Data

The first step is to obtain a dataset of face images. Recall that a digital image may be stored as an $m \times n$ array of pixels. In this lab, we will store the images as mn-vectors by concatenating the rows of the $m \times n$ arrays.

Problem 1. In this lab we will use the `faces94` face image dataset found at `http://cswww.essex.ac.uk/mv/allfaces/faces94.html` This problem will make sure that you can load and display the images from the dataset.

1. Download the `faces94` dataset from the link above and extract the files. You should now have a directory named "faces94" which contains photographs of 153 people, organized into folders by person.

Figure 7.1: The mean face.

2. The function `getFaces()` is given in the appendix. It constructs a set of face images by selecting exactly one face image for each person in the directory. It should return an array whose columns are flattened face images. Feel free to modify the given code. You may have to replace the parameter `"./faces94"` with the location of the directory `faces94` on your machine.

   Test this function to make sure it runs without errors. Check that the return value `F` is a 36000×153 array. The columns of this array are 153 flattened face images of 153 different people.

3. Use `plt.imshow()` to display one of the faces. The original image dimensions are 200×180. You may find it useful to write a helper function that accepts a flattened image and displays it.

Shift By the Mean

The facial recognition algorithm is more robust if we first *shift by the mean*. When we shift a set of data by the mean, the distinguishing features are exaggerated. Therefore, in the context of facial recognition, shifting by the mean accentuates the unique features of the face. Suppose we have a collection of k face images represented as vectors $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k$ of length mn. Define the *mean face* $\boldsymbol{\mu}$ to be the average of the $\mathbf{f}_i$:

$$\boldsymbol{\mu} = \frac{1}{k} \sum_{i=1}^{k} \mathbf{f}_i.$$

Problem 2.

The facial recognition method you will write in this lab will be structured as a class. An outline of the `FacialRec` class is provided in the appendix. You will write the methods of the class.

When initialized, the `FacialRec` object first loads the face images using `getFaces` and stores the result. The next step is to compute the mean face.

Figure 7.2: Three mean-shifted faces from the dataset.

1. In your class definition, implement the method `FacialRec.initMeanImage()`. Compute the mean face and store it as `self.mu`. This can be done in one line of code using NumPy.
2. Display the mean face. Your result should match Figure 7.1.

For each $i = 1, \ldots, k$, define $\bar{\mathbf{f}}_i := \mathbf{f}_i - \boldsymbol{\mu}$. The mean-shifted face vector $\bar{\mathbf{f}}_i$ is the deviation of the i-th face from the mean, and thus captures the unique features of the face. Now form the $mn \times k$ matrix $\bar{F}$ whose columns are given by the mean-shifted face vectors, i.e.

$$\bar{F} = \begin{bmatrix} \bar{\mathbf{f}}_1 & \bar{\mathbf{f}}_2 & \cdots & \bar{\mathbf{f}}_k \end{bmatrix}.$$

Problem 3.

1. In your class definition, implement the method `FacialRec.initDifferences()`. Compute $\bar{F}$ and store it as `self.Fbar`. This can be done in one line using array broadcasting.
2. Display one of the mean-shifted faces. The output should be similar to the faces in Figure 7.2.

Project to a Subspace

Now suppose we are given a new face vector $\mathbf{g}$. We first shift $\mathbf{g}$ by the mean of the dataset, giving us $\bar{\mathbf{g}} = \mathbf{g} - \boldsymbol{\mu}$. The closest match to $\bar{\mathbf{g}}$ is the vector $\bar{\mathbf{f}}_i$ that minimizes $\|\bar{\mathbf{g}} - \bar{\mathbf{f}}_i\|_2$. If there are k images in the dataset, we find this match by comparing $\bar{\mathbf{g}}$ to every element of $\{\bar{\mathbf{f}}_1, \ldots \bar{\mathbf{f}}_k\}$.

However, comparing the original face images pixel by pixel is computationally expensive and inefficient. The vectors $\bar{\mathbf{f}}_i$ and $\bar{\mathbf{g}}$ are length mn, which in our case equals 36000 and in practice may be many times larger. Computing the difference between face vectors of this length is time consuming, especially when the dataset is very large. It also requires us to use mn values to store each face, which is an inefficient use of space. In addition, pixel by pixel comparison is not very robust to small changes in individual pixels.

Instead, in order to store and compare our face vectors, we would like to represent each one with fewer than mn values. We can do this by projecting to a subspace. Mathematically, we want to choose s, $s \ll mn$, and project the face vectors to an s-dimensional subspace of the original mn-dimensional space of images. We can then use just s values to store each face in terms of the basis vectors of the new subspace.

The "best" subspace to project to is the one that is closest in the least squares sense (i.e., such that the sum of the squared errors between $\{\bar{\mathbf{f}}_1, \dots \bar{\mathbf{f}}_k\}$ and their projections is minimized). Let $U\Sigma V^T$ be an SVD of $\bar{F}$, with $\mathbf{u}_i$ the columns of U. As we will prove below, the s-dimensional subspace that minimizes the squared error is the span of $\mathbf{u}_1, \dots, \mathbf{u}_s$. Note that $\mathbf{u}_1, \dots, \mathbf{u}_s$ is an orthonormal basis for this subspace.

The projection matrix is $P_s = U_s U_s^T$ where $U_s = [\mathbf{u}_1 \ \dots \ \mathbf{u}_s]$. This matrix projects the original face vectors into the optimal s-dimensional subspace.

The Proof: SVD as a Least Squares Solution

Theorem 7.1. *Let $\mathbf{f}_1, \dots, \mathbf{f}_k$ be vectors on $\mathbb{R}^{mn}$, and let $\bar{F} = [\bar{f}_1 \ \dots \ \bar{f}_k]$. Suppose $U\Sigma V^T$ is an SVD for $\bar{F}$. Then the s-dimensional subspace that solves the least squares problem for $\mathbf{f}_1, \dots, \mathbf{f}_k$ is the span of the first s columns of U. If U_s is the first s columns of U, then the matrix $U_s U_s^T$ is the projection onto this subspace.*

Proof. We seek a rank-s projection matrix P_s so that $\sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2$ is minimized—i.e., the sum of the squares of the "errors" is minimal when we project $\bar{\mathbf{f}}_i$ via P_s. But minimizing this quantity is the same as minimizing its square, which happens to equal the Frobenius norm of $P_s\bar{F} - \bar{F}$. Written mathematically,

$$\begin{aligned}\inf_{\text{rank}(P_s)=s} \sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2 &= \inf_{\text{rank}(P_s)=s} \left(\sum_{i=1}^k \|P_s\bar{\mathbf{f}}_i - \bar{\mathbf{f}}_i\|_2^2\right)^2 \\ &= \inf_{\text{rank}(P_s)=s} \|P_s\bar{F} - \bar{F}\|_F.\end{aligned}$$

Now let $U\Sigma V^T$ be an SVD of $\bar{F}$ with $\mathbf{u}_i$ the columns of U, $\mathbf{v}_i$ the columns of V, and σ_i the singular values of $\bar{F}$. If $P_s = \sum_{i=1}^s \mathbf{u}_i\mathbf{u}_i^T$, then

$$\begin{aligned}P_s\bar{F} &= \left(\sum_{i=1}^s \mathbf{u}_i\mathbf{u}_i^T\right)\left(\sum_{j=1}^k \sigma_j\mathbf{u}_i\mathbf{v}_i^T\right) = \sum_{i=1}^s\sum_{j=1}^k \sigma_j\mathbf{u}_i\mathbf{u}_i^T\mathbf{u}_j\mathbf{v}_j^T \\ &= \sum_{i=1}^s\sum_{j=1}^k \sigma_j\mathbf{u}_i\delta_{ij}\mathbf{v}_j^T = \sum_{i=1}^s \sigma_i\mathbf{u}_i\mathbf{v}_i^T.\end{aligned}$$

In fact, the Schmidt-Eckart-Young-Mirsky Theorem from Lab 6 tells us that $X = \sum_{i=1}^s \sigma_i\mathbf{u}_i\mathbf{v}_i^T$ is exactly the rank-s matrix that minimizes $\|X - \bar{F}\|_F$. Since $P_s\bar{F}$ will always have rank s or less, the projection $P_s = \sum_{i=1}^s \mathbf{u}_i\mathbf{u}_i^T$ is the one we seek. If we let $U_s = [\mathbf{u}_1 \ \dots \ \mathbf{u}_s]$, then we may write $P_s = U_sU_s^T$. Notice that P_s is projection onto the subspace spanned by the columns of U_s. □

The s basis vectors $\mathbf{u}_1, \dots, \mathbf{u}_s$ are eigenvectors of $\bar{F}\bar{F}^T$. They also resemble face images. Hence, they are commonly called the "eigenfaces."

Figure 7.3: The top three eigenfaces.

Problem 4.

1. In your class definition, implement the method `FacialRec.initEigenfaces()`.

   Compute the SVD (`scipy.linalg.svd()` is a good implementation to use) and store the array `U` containing the eigenfaces as its columns. Because we will only use the first few columns of `U`, specify the keyword parameter `full_matrices=False` to compute only the compact SVD.

2. Plot the first eigenface (i.e. the first column of `U`). It should resemble the first eigenface shown in Figure 7.3.

Change Basis

The projection matrix $P_s = U_s U_s^T$ projects a face vector into the s-dimensional subspace spanned by the eigenfaces, but still keeps it as a vector in $\mathbb{R}^{mn}$. The change-of-basis matrix U_s^T both projects the face vector and and changes the basis. The resulting vector has length s and represents the face in terms of eigenfaces.

To represent any face vector in terms of the first s eigenfaces, multiply by U_s^T. To change back to a full length-mn projection, multiply again by U_s.

Problem 5.

1. Implement the method `FacialRec.project()` in your class definition. This should accept a flattened image or an array with flattened images as its columns. It should also accept a value for s. The function should project the image or images into the appropriate s-dimensional subspace and change basis, then return the result.

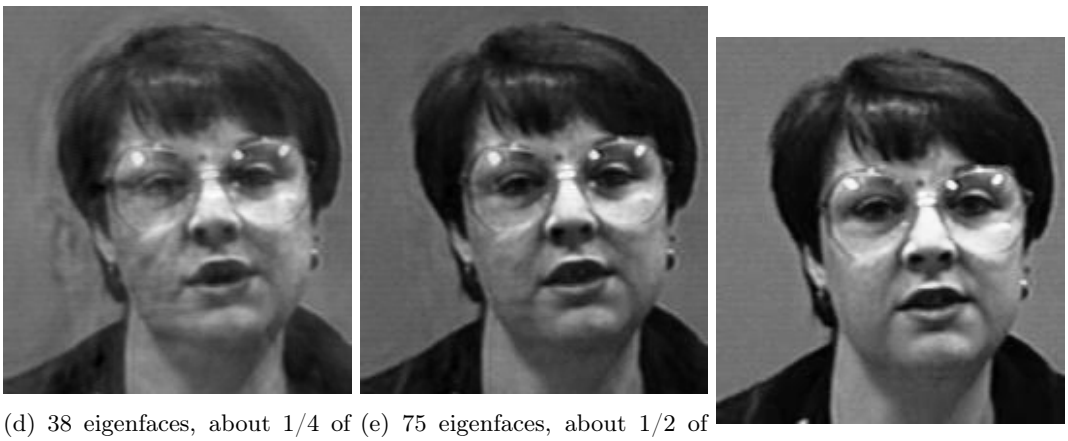
2. Let `face` be the first mean-shifted face (the first column of `facialRec.Fbar`). Do the following:

   (a) Project `face` so that it is represented in terms of the first 19 eigenfaces.

(a) 5 eigenfaces, about 1/32 of the total.

(b) 9 eigenfaces, or 1/16 of the total.

(c) 19 eigenfaces, about 1/8 of the total.



(d) 38 eigenfaces, about 1/4 of the total.

(e) 75 eigenfaces, about 1/2 of the total.

(f) All 153 of the eigenfaces.

Figure 7.4: Image rebuilt with various numbers of eigenfaces. The image is somewhat recognizable when it is reconstructed with only 1/8 of the eigenfaces.

(b) Change basis again back to the standard basis on $\mathbb{R}^{mn}$.

(c) Add back the mean face `facialRec.mu`.

(d) Plot the resulting image.

Your image should match Figure 7.4c.

Match Faces

Finally, we are ready to identify which mean-shifted image $\bar{\mathbf{f}}_i$ is closest to an input image, $\bar{\mathbf{g}}$. We begin by projecting all vectors to some s-dimensional subspace and writing them in terms of the basis vectors, which are the eigenfaces. This is done by multiplying by the change-of-basis matrix:

$$\widehat{\mathbf{f}}_i = U_s^T(\mathbf{f}_i - \boldsymbol{\mu}) \qquad \widehat{\mathbf{g}} = U_s^T(\mathbf{g} - \boldsymbol{\mu}).$$

Next, we compute which $\widehat{\mathbf{f}}_i$ is closest to $\widehat{\mathbf{g}}$. Since the columns of U_s are an orthonormal basis, we get the same result doing the computation in this basis as we would in the standard Euclidean basis. Define

$$i^* = \text{argmin}_i \|\widehat{\mathbf{f}}_i - \widehat{\mathbf{g}}\|_2.$$

Then the i^*-th face image is the best match for $\mathbf{g}$.

Problem 6.

1. Implement the method `FacialRec.findNearest()` as follows.

```
def findNearest(self, image, s=38):
    Fhat = # Project Fbar, producing a matrix whose columns are the f↩
        -hat defined above
    ghat = # Shift 'image' by the mean and project, producing g-hat ↩
        as defined above
    # for both Fhat and ghat, use your project function from the ↩
        previous problem

    # Return the index that minimizes ||fhat_i - ghat||_2.
```

   The functions `np.linalg.norm()` and `np.argmin()` will be useful for the last line. When using `np.linalg.norm`, make sure you indicate the correct axis.

2. Test your facial recognition system on faces selected randomly from the `faces94` dataset. The function `sampleFaces(n_tests, path)` at the end of this lab will build an array of `n_tests` random faces from the `faces94` dataset.

   Plot the random face beside the face returned by your facial recognition code to see if your system is accurately recognizing faces.

By this point, you have created a basic facial recognition system. We can extend the system to detect when a face doesn't match anything currently in the dataset, and then add this new face. We can also make the system more robust by including multiple pictures of the same face with different expressions and lighting conditions.

Although there are other approaches to facial recognition that utilize more complex techniques, the method of eigenfaces remains a wonderfully simple and effective solution, illustrating another application of the singular value decomposition.

Appendix: Helper Code

This section contains some functions to help you implement the facial recognition class outlined in the problems of this lab.

```
import numpy as np
from scipy import linalg as la
from os import walk
from scipy.ndimage import imread
from matplotlib import pyplot as plt
```

```
from random import sample

def getFaces(path="./faces94"):
    """Traverse the directory specified by 'path' and return an array ↩
       containing
    one column vector per subdirectory.

    For the faces94 dataset, this gives an array with just one column for each
    face in the dataset. Each column corresponds to a flattened grayscale image↩
       .
    """

    # Traverse the directory and get one image per subdirectory.
    faces = []
    for (dirpath, dirnames, filenames) in walk(path):
        for f in filenames:
            if f[-3:]=="jpg": # only get jpg images
                # load image, convert to grayscale, flatten into vector
                face = imread(dirpath+"/"+f).mean(axis=2).ravel()
                faces.append(face)
                break

    # put all the face vectors column-wise into a matrix.
    F = np.array(faces).T
    return F


def sampleFaces(n_tests,path = "./faces94"):
    """Return an array containing a sample of n_tests images contained
    in the path as flattened images in the columns of the output.
    """
    files = []
    for (dirpath, dirnames, filenames) in walk(path):
        for f in filenames:
            if f[-3:]=="jpg": # only get jpg images
                files.append(dirpath+"/"+f)

    #Get a sample of the images
    test_files = sample(files, n_tests)
    #Flatten and average the pixel values
    images = np.array([imread(f).mean(axis=2).ravel() for f in test_files]).T
    return images
```

The following is the outline of the Facial Recognition class.

```
class FacialRec:
    ##########Members##########
    #   F, mu, Fbar, and U
    ############################
```

```
    def __init__(self,path):
        self.initFaces(path)
        self.initMeanImage()
        self.initDifferences()
        self.initEigenfaces()

    def initFaces(self, path):
        self.F = getFaces(path)
    def initMeanImage(self):
        pass
    def initDifferences(self):
        pass
    def initEigenfaces(self):
        pass
    def project(self, A, s=38):
        pass
    def findNearest(self, image, s=38):
        pass
```

8 Numerical Differentiation

Lab Objective: *Understand and implement finite difference approximations of the derivative in single and multiple dimensions. Evaluate the accuracy of these approximations.*

Derivative Approximations in One Dimension

The derivative of a function f at a point x_0 is

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}. \tag{8.1}$$

In this lab, we will investigate one way a computer can calculate $f'(x_0)$.

Forward Difference Quotient

Suppose that in Equation (8.1), instead of taking a limit, we just pick a small value for h. Then we would expect $f'(x_0)$ to be close to the quantity

$$\frac{f(x_0 + h) - f(x_0)}{h}. \tag{8.2}$$

This quotient is called the *first order forward difference approximation* of the derivative. Because $f'(x_0)$ is the limit of such quotients, we expect that when h is small, this quotient is close to $f'(x_0)$. We can use Taylor's formula to find just how close.

By Taylor's formula,

$$f(x_0 + h) = f(x_0) + f'(x_0)h + R_2(h),$$

where $R_2(h) = \left(\int_0^1 (1-t)f''(x_0 + th)dt\right) h^2$. (This is called the *integral form* of the remainder for Taylor's Theorem; see Volume 1 Chapter 6). When we solve this equation for $f'(x_0)$, we get

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{R_2(h)}{h}. \tag{8.3}$$

Thus, the error in using the first order forward difference quotient to approximate $f'(x_0)$ is

$$\left|\frac{R_2(h)}{h}\right| \leq |h| \int_0^1 |1-t||f''(x_0 + th)|dt.$$

If we assume f'' is continuous, then for any δ, set $M = \sup_{x\in(x_0-\delta,x_0+\delta)} f''(x)$. Then if $|h| < \delta$, we have

$$\left|\frac{R_2(h)}{h}\right| \leq |h| \int_0^1 M dt = M|h| \in O(h).$$

Therefore, the error in using (8.2) to approximate $f'(x_0)$ grows like h.

Centered Difference Quotient

In fact, we can approximate $f'(x_0)$ to the second order with another difference quotient, called the *centered difference quotient*. We begin by trying to find the *backward difference quotient*. Evaluate Taylor's formula at $x_0 - h$ to derive

$$f'(x_0) = \frac{f(x_0) - f(x_0 - h)}{h} + \frac{R_2(-h)}{h}. \tag{8.4}$$

The first term on the right hand side of (8.4) is called the *backward difference quotient*. This quotient also approximates $f'(x_0)$ to first order, so it is not the quotient we are looking for. When we add (8.3) and (8.4) and solve for $f'(x_0)$ (by dividing by 2), we get

$$f'(x_0) = \frac{\frac{1}{2}f(x_0 + h) - \frac{1}{2}f(x_0 - h)}{h} + \frac{R_2(-h) - R_2(h)}{2h} \tag{8.5}$$

The *centered difference quotient* is the first term of the right hand side of (8.5). Let us investigate the remainder term to see how accurate this approximation is. Recall from the proof of Taylor's theorem that $R_k = \frac{f^{(k)}(x_0)}{k!}h^k + R_{k+1}$. Therefore,

$$\begin{aligned}
\frac{R_2(-h) - R_2(h)}{2h} &= \frac{1}{2h}\left(\frac{f''(x_0)}{2}h^2 + R_3(-h) - \frac{f''(x_0)}{2}h^2 - R_3(h)\right) \\
&= \frac{1}{2h}(R_3(-h) - R_3(h)) \\
&= \frac{1}{2h}\left(\left(\int_0^1 \frac{(1-t)^2}{2}f'''(x_0 + th)dt\right)h^3 - \left(\int_0^1 \frac{(1-t)^2}{2}f'''(x_0 - th)dt\right)h^3\right) \\
&= \left(\int_0^1 \frac{(1-t)^2}{4}(f'''(x_0 + th) - f'''(x_0 - th))\right)h^2 \\
&\in O(h^2)
\end{aligned}$$

once we restrict h to some δ-neighborhood of 0. So the error in using the centered difference quotient to approximate $f'(x_0)$ grows like h^2, which is smaller than h when $|h| < 1$.

Accuracy of Approximations

Let us discuss what step size h we should plug into the difference quotients to get the best approximation to $f'(x_0)$. Since f' is defined as a limit as $h \to 0$, you may think that it is best to choose h as small as possible, but this is not the case. In fact, dividing by very small numbers causes errors in floating point arithmetic. This means that as we decrease $|h|$, the error between $f'(x_0)$ and the difference quotient will first decrease, but then increase when $|h|$ gets too small because of floating point arithmetic.

Here is an example with the function $f(x) = e^x$. A quick way to write f as a function in Python is with the `lambda` keyword.

h	1e-1	1e-3	1e-5	1e-7	1e-9	1e-11
Error	5e-3	5e-7	6e-11	6e-11	7e-9	1e-5

Table 8.1: This table shows that it is best not to choose h too small when you approximate derivatives with difference quotients. Here, "Error" equals the absolute value of $f'(1) - f_{app}(1)$ where $f(x) = e^x$ and f_{app} is the centered difference approximation to f'.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> f = lambda x: np.exp(x)
```

In general, the line `f = lambda <params> : <expression>` is equivalent to defining a function `f` that accepts the parameters `params` and returns `expression`.

Next we fix a step size `h` and define an approximation to the derivative of `f` using the *centered difference quotient*.

```
>>> h = 1e-1
>>> Df_app = lambda x: .5*(f(x+h)-f(x-h))/h
```

Finally, we check the accuracy of this approximation at $x_0 = 1$ by computing the difference between `Df_app(1)` and the actual derivative evaluated at 1.

```
# Since f(x) = e^x, the derivative of f(x) is f(x)
>>> np.abs( f(1)-Df_app(1) )
0.0045327354883726301
```

We note that our functions `f` and `Df_app` behave as expected when they are passed a NumPy array.

```
>>> h = np.array([1e-1, 1e-3, 1e-5, 1e-7, 1e-9, 1e-11])
>>> np.abs( f(1)-Df_app(1) )
array([  4.53273549e-03,   4.53046679e-07,   5.85869131e-11,
         5.85873572e-11,   6.60275079e-09,   1.04294937e-05])
```

These results are summarized in Table 8.1.

Thus, the optimal value of h is one that is small, but not too small. A good choice is `h = 1e-5`.

Problem 1. Write a function that accepts as input a callable function object `f`, an array of points `pts`, and a keyword argument `h` that defaults to `1e-5`. Return an array of the *centered difference quotients* of `f` at each point in `pts` with the specified value of `h`.

Problem 2. Write a function that accepts as input a callable function object `f`, the derivative `df` of the function `f`, an array of points `pts`, and a keyword argument `h` that defaults to `1e-5`. Return an array of the errors for the *centered difference quotients* at each point in `pts` with the specified value of `h`.

Problem 3. Use the *centered difference quotient* to approximate the derivative of $f(x) = (\sin(x)+1)^x$ at $x = \frac{\pi}{3}, \frac{\pi}{4}$, and $\frac{\pi}{6}$. Calculate the error of the approximations.

You may wonder if the forward or backward difference quotients are ever used, since the centered difference quotient is a more accurate approximation of the derivative. In fact, there are some functions that in practice do not behave well under centered difference quotients. In these cases, one must use the forward or backward difference quotient.

Finally, we remark that forward, backward, and centered difference quotients can be used to approximate higher-order derivatives of f. However, taking derivatives is an *unstable* operation. This means that taking a derivative can amplify the arithmetic error in your computation. For this reason, difference quotients are not generally used to approximate derivatives higher than second order.

Problem 4. The radar stations A and B, separated by the distance a = 500 m, track the plane C by recording the angles α and β at one-second intervals (See figure 8.1). Three successive readings are give in table 8.2. Use centered difference quotients to calculate the speed v of the plane at t = 10 s. The coordinates of the plane can be shown to be

$$x = a\frac{\tan(\beta)}{\tan(\beta)-\tan(\alpha)} \tag{8.6}$$

$$y = a\frac{\tan(\beta)\tan(\alpha)}{\tan(\beta)-\tan(\alpha)} \tag{8.7}$$

(Kiusalaas, Jaan. Numerical Methods in Engineering with Python 3)

Derivative Approximations in Multiple Dimensions

Finite difference methods can also be used to calculate derivatives in higher dimensions. Recall that the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ at a point $x_0 \in \mathbb{R}^n$ is the $m \times n$ matrix $J = (J_{ij})$ defined component-wise by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(x_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$J = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{pmatrix}.$$

The Jacobian is useful in many applications. For example, the Jacobian can be used to find zeros of functions in multiple variables.

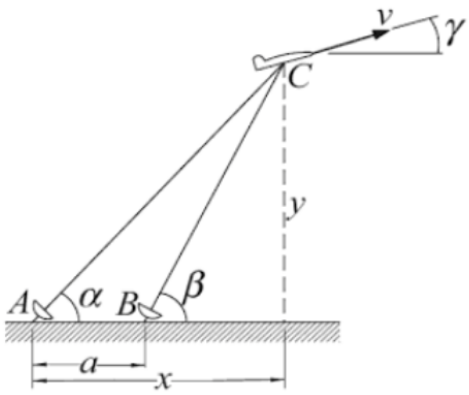


Figure 8.1: Radar stations in relation to plane

t (s)	9	10	11
α	54.80°	54.06°	53.34°
β	65.59°	64.59°	63.62°

Figure 8.2: Angles α and β at one-second intervals

The forward difference quotient for approximating a partial derivative is

$$\frac{\partial f}{\partial x_j}(x_0) \approx \frac{f(x_0 + he_j) - f(x_0)}{h},$$

where e_j is the j^{th} standard basis vector. Similarly, the centered difference approximation is

$$\frac{\partial f}{\partial x_j}(x_0) \approx \frac{\frac{1}{2}f(x_0 + he_j) - \frac{1}{2}f(x_0 - he_j)}{h}.$$

Problem 5. Write a function that accepts

1. a function handle `f`,
2. an integer `n` that is the dimension of the domain of `f`,
3. an integer `m` that is the dimension of the range of `f`,
4. an `1 x n`-dimensional NumPy array `pt` representing a point in $\mathbb{R}^n$, and
5. a keyword argument `h` that defaults to `1e-5`.

Return the approximate Jacobian matrix of `f` at `pt` using the centered difference quotient.

Problem 6.

Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by

$$f(x, y) = \begin{bmatrix} e^x \sin(y) + y^3 \\ 3y - \cos(x) \end{bmatrix}$$

Find the error between your Jacobian function and the analytically computed derivative on the square $[-1, 1] \times [-1, 1]$ using ten thousand grid points (100 per side). You may apply your Jacobian function to the points one at a time using a double `for` loop. Once you get the error matrix for a given point, calculate the Frobenius norm of this matrix (`la.norm` defaults to the Frobenius norm). This norm will be your total error for that point. What is the maximum error of your Jacobian function over all points in the square?

Hint: The following code defines the function $f(x, y) = \begin{bmatrix} x^2 \\ x + y \end{bmatrix}$.

```
# f accepts a length-2 NumPy array
>>> f = lambda x: np.array([x[0]**2, x[0]+x[1]])
```

9 Symbolic and Automatic Differentiation

Lab Objective: *Python can compute derivatives symbolically and automatically. SymPy uses symbolic computations to calculate exact derivatives. Autograd automatically differentiates Python and NumPy code. In this lab, we learn how to use these packages for differentiation. We will also compare the computation cost and accuracy of differentiating between SymPy, Autograd, and numerical differentiation.*

SymPy

Symbolic Manipulation

It is simple to numerically compute $2x^5 + 4xy + 3x$ given any specific values of x and y. In some circumstances, however, getting the entire algebraic expression is more desirable. We can use SymPy to create symbolic representations of expressions.

To symbolically represent expressions, symbolic variables must first be defined. To define the symbol x, we write `x = sy.symbols('x')`. Multiple variables can also be defined at once with `sy.symbols('x,y,z')`. Combining symbolic objects results in a SymPy *expression*.

```
>>> import sympy as sy

# Define the symbolic variables x and y.
>>> x, y  = sy.symbols('x, y')

# Create the expression 2x^5 + 4xy + 3x.
>>> expr = 2*x**5 + 4*x*y + 3*x
>>> print(expr)
2*x**5 + 4*x*y + 3*x
```

Be careful that the function `sy.symbols()` is spelled correctly. The command `sy.Symbol()` will still work for a single symbolic variable, but `sy.symbol` is a submodule and cannot be called at all.

ACHTUNG!

Beware of integer division: the expression `(3/4)*sy.sin(x)` will evaluate to 0. Be careful about using floating points as well. The expression `(1/3.)*sy.sin(x)` will evaluate to `0.333333333333333*sy.sin(x)`.

To keep $\frac{3}{4}$ symbolic, use `3*sy.sin(x)/4` or `sy.Rational(3,4)*sy.sin(x)` instead.

Problem 1. Write a function that creates the expression $\frac{2}{5}e^{x^2-y}\cosh(x+y) + \frac{3}{7}\log(xy+1)$ symbolically with SymPy. The Sympy syntax for $\cosh(x)$ and $\log(x)$ are`sy.cosh(x)` and `sy.log(x)`, respectively. Make sure that your constants are not floating point numbers.

SymPy can be used to solve difficult expressions for given variables. Consider the following equation:

$$\frac{w}{w-x} + \frac{x}{x-y} + \frac{y}{y-w} = 0$$

The explicit solution of w,written in terms of x and y, would be tedious to compute by hand but SymPy can solve for this explicit solution.

```
>>> w, x, y = sy.symbols('w, x, y')
>>> expr = w/(w-x) + x/(x-y) + y/(y-w)

# Solve for the expression in terms of w.
>>> sy.solve(expr,w)
[(x**2 + 3*x*y - 2*y**2 + (-x + y)*sqrt(x**2 - 8*x*y + 4*y**2))/(2*(2*x - y)), ↩
      (x**2 + 3*x*y - 2*y**2 + (x - y)*sqrt(x**2 - 8*x*y + 4*y**2))/(2*(2*x - y↩
    ))]
```

Note that `sy.solve()` returns a list of expressions that solves for w. To use the expression solved, one must take that expression from the list.

```
# Take the first expression that solves for w in terms of x and y.
>>> sy.solve(expr,w)[0]
(x**2 + 3*x*y - 2*y**2 + (-x + y)*sqrt(x**2 - 8*x*y + 4*y**2))/(2*(2*x - y))
```

In the example above, a symbolic expression is used instead of an equation. This is because SymPy automatically sets the expression equal to zero, solves for the indicated variable, and then returns the result. If there is an equation given, subtract everything to one side of the equality.

Another way to use `sy.solve()` is to declare an equation. An equation can be defined using `sy.Eq`. To represent the equation $y = 3x + 2$, declare `equation = sy.Eq(3*x+2,y)`. To solve for x, use `sy.solve(equation,x)`.

Problem 2. Write a function that uses Sympy to solve the equation $y = \sqrt{5 - e^{x^2}}$ for x. Return the list of expressions that solves for x. The SymPy syntax for $\sqrt{x}$ is `sy.sqrt(x)`.

Symplify

SymPy can also be used to expand and simplify different symbolic expressions. As an example we will simplify the expression

$$\frac{wx^2y^2 - wx^2 - wy^2 + w - x^2y^2z + 2x^2y^2 + x^2z - 2x^2 + y^2z - 2y^2 - z + 2}{wxy - wx - wy + w - xyz + 2xy + xz - 2x + yz - 2y - z + 2}.$$

```
>>> w, x, y, z=sy.symbols('w, x, y, z')
>>> expr = (w*x**2*y**2 - w*x**2 - w*y**2 + w - x**2*y**2*z + 2*x**2*y**2 + x↩
    **2*z - 2*x**2 + y**2*z - 2*y**2 - z + 2)/(w*x*y - w*x - w*y + w - x*y*z + ↩
    2*x*y + x*z - 2*x + y*z - 2*y - z + 2)
>>> expr.simplify()
x*y + x + y + 1
```

`simplify()` is the general simplification method for a SymPy expression. It can be called as a function from the module by using `sy.simplify()` or it can be used as a method for an object as in the example above. SymPy can compute specific types of simplification. For example, to factor an expression, use `factor()`. To expand an expression, use `expand()`. To focus purely on simplifying the trigonometric aspects of the expression, use `trigsimp()`. To cancel variable expressions in the numerator and denominator of all rational sub-expressions, use `cancel()`. There are several other kinds of algebraic manipulations in SymPy, see the documentation for a more comprehensive list. Many of these more important functions in SymPy are also available as methods to expressions. This is the case with all of the above examples.

Refer to the Additional Material or `http://docs.sympy.org/0.7.2/tutorial.html` for other useful features.

Differentiation using SymPy

SymPy can be used to take closed form derivatives without doing it by hand. Derivatives can be taken in Sympy using the `sy.Derivative()` function:

```
>>> x, y = sy.symbols('x,y')
>>> expr = sy.sin(x)*sy.cos(x)*sy.exp(y)*(x**3+y)**4
# Find the derivative of expr in terms of x.
>>> sy.Derivative(expr,x).doit()
12*x**2*(x**3 + y)**3*exp(y)*sin(x)*cos(x) - (x**3 + y)**4*exp(y)*sin(x)**2 + (↩
    x**3 + y)**4*exp(y)*cos(x)**2
```

The `.doit()` method tells SymPy to evaluate the derivative of the expression.

Equivalently, one can use the `.diff()` method of a SymPy expression.

```
>>> expr.diff(x)
```

```
12*x**2*(x**3 + y)**3*exp(y)*sin(x)*cos(x) - (x**3 + y)**4*exp(y)*sin(x)**2 + (↩
    x**3 + y)**4*exp(y)*cos(x)**2
```

To find the derivative of the expression above at $x = 0$, substitute a symbolic variable for a value by using the `.sub()` method shown below:

```
# Save the derivative as an expression named d_expr.
>>> d_expr = expr.diff(x)
# Substitute x with 0.
>>> d_expr.subs(x,0)
y**4*exp(y)
```

To substitute values in for multiple variables, either use a dictionary or a list of tuples for each variable and value.

```
# Substitute multiple variables with a dictionary.
>>> d_expr.subs({x:1.,y:2.})
839.384133011589
# Substitute multiple variables with a list of tuples.
>>> d_expr.subs([(x,1.),(y,2.)])
839.384133011589
```

Make sure that the substitutions you implemented are floating points. If they are not, Sympy will not evaluate the expression. You may use the method `.evalf( )` to evaluate the expression.

```
#  Replace y with the integer 2.
>>> d_expr.subs([(x,1.),(y,2)])
113.598289385442*exp(2)
# Use .evalf() to evaluate e^2.
>>> d_expr.subs([(x,1.),(y,2)]).evalf()
839.384133011589
```

Note that these methods return a modified version of the expression, but do not actually change the original expression.

Problem 3. Use SymPy to calculate the derivative of $e^{\sin(\cos(x))}$ at $x = 1$. Use the second order centered difference quotient from the previous lab to calculate the same derivative. Compare the performance of Sympy and the centered difference quotient. Print out the total time each method takes to compute the derivative as well as the error of the approximation.

The error is denoted as $|Df(x_0) - D\tilde{f}(x_0)|$ where $D\tilde{f}(x_0)$ is the approximated derivative and $Df(x_0) = -e^{\sin(cos(x))} \sin(x) \cos(\cos(x))$.

Return the SymPy approximation as a float.

Hint: Recall that the second order centered difference quotient is

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

For this approximation, let $h = 1 \times 10^{-5}$. The result should reveal that SymPy has an error of zero but the centered difference quotient will compute it faster.

From the previous problem, we could see that SymPy is very accurate in finding derivatives. However the cost of having accurate solutions is the time the method takes to compute the derivative. Finding the right method to compute differentiations will depend on which factor is more prominent. In cases where stability is more important, Sympy would be the better option. If accuracy is not as important but less computation time is important, then numerical differentiation (like centered difference quotient) might be the better choice. Later, in this lab, we will explore another differentiable tool called Autograd.

Useful tools for Differentiation with SymPy

When using `.diff()`, the method defaults to solving the first derivative of the function. Given a function that is differentiable n times, you can find the nth derivative of the function by having n as the second argument in `.diff()`.

The following equation takes the 20th partial derivative with respect to x of

$$\prod_{i=1}^{23} (x + iy)$$

```
>>> x, y, i = sy.symbols('x, y, i')
# Define the expression.
>>> expr = sy.product((x+i*y), (i, 1, 23))
# Expand the product.
>>> expr = expr.expand()
# Take the 20th derivative with respect to x.
>>> expr.diff(x, 20)
4308669456480829440000*(x**3 + 36*x**2*y + 426*x*y**2 + 1656*y**3)
```

Another effective tool in Sympy is to transform a sympy expression to a lambda function through `sy.lambdify()`. This can be used to calculate numerical values very fast. Consider the following example:

```
>>> x = sy.symbols('x')
# Assign the second derivative of sin(x)^2 to expr.
>>> expr = (sy.sin(x)**2).diff(x,2)
# Turn the expression into a lambda function with x as the variable.
>>> f = sy.lambdify(x,expr)
>>> f(0)
2.0
>>> f(np.pi/2)
-2.0
>>> f(np.pi)
2.0
```

There are many advantages of transforming a SymPy expression into a lambda function. Rather than substituting values by using the `.subs()` method, evaluating the lambda functions is cheaper to compute. According to the Lambdify documentation, using `sy.lambdify()` to do numerical evaluations "takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs()` method."

Problem 4. Compare the time it takes to evaluate the 3rd derivative of $\tanh(x)$ at 10,000 randomly generated points using the `.subs()` method versus turning a Sympy expression into a lambda function through `sy.lambdify()`. Print out the times it takes to evaluate the 3rd derivative of $\tanh(x)$ at 10,000 random points by `.subs()` and through `sy.lambdify()`.

By default, `sy.lambdify()` uses the `math` library. However, `sy.lambdify()` supports many other libraries including NumPy. By including `"numpy"` as the third argument of the function, the function generated by `sy.lambdify()` can have access to vectorized functions. Consider the following example:

```
#Generate an numpy array of points.
>>> points = np.linspace(0,2*np.pi,10)
>>> expr = sy.sin(2*x).diff(x,2)
# Turn expr in to a lambda function with the numpy attribute.
>>> f = sy.lambdify(x,expr,"numpy")
# Evaluate f with multiple points at once.
>>> f(points)
array([ -0.00000000e+00,  -3.93923101e+00,  -1.36808057e+00,
         3.46410162e+00,   2.57115044e+00,  -2.57115044e+00,
        -3.46410162e+00,   1.36808057e+00,   3.93923101e+00,
         1.95943488e-15])
```

SymPy can also be used to compute the Jacobian of a matrix using the `.jacobian()` method, which takes in either a list or a matrix of the variables. The Jacobian of $f(x,y) = \begin{bmatrix} x^2 \\ x+y \end{bmatrix}$ is found by doing the following:

```
# Create a matrix of symbolic variables.
>>> x,y = sy.symbols('x,y')
>>> F = sy.Matrix([x**2,x+y])

# Find the jacobian of the matrix with respect to x and y.
>>> F_jac = F.jacobian([x,y])
>>> F_jac
Matrix([
[2*x, 0],
[  1, 1]])

# Evaluate the Jacobian at (1,1).
>>> F_jac.subs([(x,1.),(y,1.)])
Matrix([
[2.0, 0],
```

```
[  1, 1]])
```

In addition, SymPy includes several integral transforms, such as the Laplace, Fourier, sine, and cosine transforms. SymPy also allows you to do simple separation of variables on PDEs, Taylor Series, Laurent Series, Fourier Series, and many, *many* other things.

Autograd

Autograd is a package that allows for efficient automatic differentiation using NumPy codes. Unlike SymPy ,which has many diverse applications, autograd is solely used in differentiation. Because Autograd works on ordinary NumPy code, it is very useful to calculate gradients automatically rather than deriving the code by hand. Due to this feature, it can be very useful in machine learning.

Autograd is installed by running the following command in the terminal:

```
pip install autograd
```

See `https://github.com/HIPS/autograd` for more complete installation instructions.

The following code computes the derivative of $e^{\sin(\cos(x))}$ at $x = 1$ using Autograd:

```
>>> from autograd import grad
>>> import autograd.numpy as anp        # Use autograd's own version of NumPy

>>> g = lambda x: anp.exp(anp.sin(anp.cos(x)))
>>> grad_g = grad(g)
>>> grad_g(1.)
-1.20697770398
```

To support most of the NumPy features[1], autograd uses a thinly-wrapped version of Numpy called `autograd.numpy`. This lab will denote the Autograd's version of Numpy as `anp`. Use `anp` the way NumPy is used.

The function `grad()` returns a function that computes the gradient of your original function. This new function, which returns the gradient, accepts the same parameters as the original function.

When there are multiple variables, the parameter `argnum` allows you to specify with respect to which variable you are computing the gradient.

```
>>> f = lambda x,y: 3*x*y + 2*y - x
>>> grad_f = grad(f, argnum=0) #gradient with respect to the first variable (x)
>>> grad_f(.25,.5)
0.5
>>> grad_f = grad(f, argnum=1) #gradient with respect to the second variable (y↩
    )
>>> grad_fun(.25,.5)
2.75
```

Finding the gradient with respect to multiple variables can by done using `multigrad()` by specifying which variables in the `argnums` parameter.

[1]For a list of Numpy features that Autograd does not support, please refer to `https://github.com/HIPS/autograd/blob/701ed8518140ffa4246e7ef18256a71ed639045b/docs/tutorial.md`.

```
>>> grad_fun = autograd.multigrad(function, argnums=[0,1])
>>> grad_fun(.25,.5)
(0.5, 2.75)
```

Problem 5. Use Autograd to compute the derivative of $f(x) = \ln\sqrt{\sin\left(\sqrt{x}\right)}$ at $x = \frac{\pi}{4}$. Compare how long it takes to compute this derivative among Autograd, SymPy, and the second order centered difference quotient and record the error each approximation. Print the computation time and error for each method.

The error is denoted as $|Df(x_0) - D\tilde{f}(x_0)|$ where $Df(x_0) = \frac{\cot\left(\sqrt{x}\right)}{4\sqrt{x}}$ and $D\tilde{f}(x_0)$ is the approximated derivative.

SymPy will have the exact derivative of the function yielding zero error. However, SymPy will also have the longest computation time. Centered difference quotient will have the least amount of time with the greatest error. Autograd will have shorter computation time than SymPy and a smaller error than centered difference quotient.

As shown in the previous problem, Autograd can be an efficient tool in differentiation. Although Autograd does not calculate exact derivatives, the resulting error is relatively small with less computational time than SymPy.

Autograd allows users to differentiate a function as many times as desired.

```
>>> f = lambda x: anp.sin(x) + 3**anp.cos(x)
# Calculate the first derivative.
>>> df = grad(f)
# Calculate the second derivative and so forth.
>>> df2 = grad(df)
>>> df3 = grad(df2)
>>> df3(1.)
2.6834458987503522
```

Although `grad()` is very efficient, it does not allow for array broadcasting. However, Autograd has another function `elementwise_grad()` that does.

```
>>> from autograd import elementwise_grad
>>> f = lambda x: anp.sin(x) + 3**anp.cos(x)
>>> f_grad = elementwise_grad(f)
>>> f_grad(np.array([1.,2.,3.,]))
array([-1.13338111, -1.04855565, -1.04224253])
```

Problem 6. Use `elementwise_grad()` to graph $f(x) = \frac{1}{\cosh(x)}$ and it's next five derivatives where $x \in [-7, 7]$. Display the plots in multiple subplots.

While the `grad()` function can only output scalar-valued functions, `jacobian()` can allow you to find the gradient of vectors. The following shows how to find the Jacobian of $f(x,y) = \begin{bmatrix} x^2 \\ x+y \end{bmatrix}$ evaluated at (1,1).

```
from autograd import jacobian
>>> f = lambda x: anp.array([x[0]**2, x[0]+x[1]])
>>> jacobian_f = jacobian(f)
>>> jacobian_f(anp.array([1.,1.]))
array([[ 2.,  0.],
       [ 1.,  1.]])
```

Problem 7.

Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by

$$f(x,y) = \begin{bmatrix} e^x \sin(y) + y^3 \\ 3y - \cos(x) \end{bmatrix}$$

Find the Jacobian function using SymPy and Autograd. Print out the time it takes to compute each Jacobian at $(x,y) = (1,1)$.

Notice that Autograd computes the Jacobian a lot faster than SymPy.

To learn more about Autograd visit `https://github.com/HIPS/autograd`.

Additional Materials

Displaying SymPy Expressions

SymPy includes a simplified plotting wrapper around Matplotlib.

```
>>> x = sy.symbols('x')
>>> expr = sy.sin(x) * sy.exp(x)
>>> sy.plot(expr, (x, -3, 3))
```

The code above plots $\sin(x)e^x$ for values of x from -3 to 3.

SymPy also has several nice options for printing equations. To know what the equation looks like use `sy.pprint()`. It can interface with the IPython Notebook to display the formula more clearly as well. IPython Notebook can enable pretty printing by loading the extension that comes with SymPy. In SymPy 7.2, this is done as follows:

```
%load_ext sympy.interactive.ipythonprinting
```

The syntax is written a little differently in SymPy 7.3.

```
import sympy as sy
sy.init_printing()
```

Figure 9.1 shows a screenshot of SymPy's special printing in the IPython notebook. If at some point you need to write a formula in LaTeX, the function `sy.latex()` can convert a SymPy symbolic expression to LaTeX.

```
In [1]: import sympy as sy
        sy.init_printing()
        x, y, z, theta = sy.symbols("x,y,z,\\theta")

In [2]: expr = sy.sin(theta)*sy.exp(y)*sy.log(z)*(x+y*theta)**4
        expr = sy.Integral(expr,(x,0,2))
        expr = sy.Integral(expr,(y,-1,1))
        expr = sy.Integral(expr,(z,-2,0))
        expr = sy.Derivative(expr,theta)
        expr

Out[2]: $$\frac{d}{d\theta}\int_{-2}^{0}\int_{-1}^{1}\int_{0}^{2}(\theta y+x)^4 e^y \log(z)\sin(\theta)\,dx\,dy\,dz$$
```

Figure 9.1: A screenshot showing how SymPy can interface with the IPython Notebook to display equations.

Basic Number Types

SymPy has some good built in datatypes which can be used to represent rational numbers and arbitrary precision floating point numbers. Arbitrary precision floating point operations are supported through the package `mpmath`. These can be useful if computation to a very high precision is needed. It can also avoid possible overflow error in computation. They are, however, much more costly to compute.

You can declare a rational number $\frac{a}{b}$ using `sy.Rational(a, b)`. A real number r of precision n can be declared using `sy.Float(r,n)`.

A nice example of the use of these datatypes is the following function which computes π to the nth digit.

```
def mypi(n):
    #Calculates pi to n decimal points.
    tot = sy.Rational(0, 1)
    term = 1
    bound = sy.Rational(1, 10)**(n+1)
    i = 0
    while bound <= term:
        term = 6 * sy.Rational(sy.factorial(2*i), 4**i*(sy.factorial(i))**2*(2*↩
            i+1)*2**(2*i+1))
        tot += term
        i += 1
    return sy.Float(tot, n)
```

This function works by evaluating the Taylor series for $6 \arcsin\left(\frac{1}{2}\right)$. We used a rather crude error estimate to ensure that we were close enough to break the loop.

More Sympy Substitution

Substitution also can be used (to some extent) to substitute one expression for another. For example, if you want to apply the double angle identity to replace products of sines and cosines, you could use the following:

```
>>> expr.subs(sy.sin(x) * sy.cos(x), sy.sin(2*x)/2)
(x**3 + y)**4*exp(y)*sin(2*x)/2
```

To eliminate higher powers of a variable in an expression, use something like:

```
>>> expr.subs(x**3, 0)
y**4*exp(y)*sin(x)*cos(x)
```

which will eliminate all terms of the expression involving x^3. At present time, this will not eliminate terms involving x^4 or higher powers of x that are not divisible by 3.

Integrals in SymPy

SymPy can integrate two different variables along two different bounds.

```
sy.integrate(y**2*x**2, (x, -1, 1), (y, -1, 1))
```

It can also integrate with respect to multiple variables.

```
sy.integrate(y**2 * x**2, x, y)
```

Integrate a difficult expression like $e^x \sin(x) \sinh(x)$ can be done with one line in SymPy.

```
sy.Integral(sy.sin(x) * sy.exp(x) * sy.sinh(x), x).doit()
```

Differential Equations in SymPy

SymPy can be used to solve some sorts of basic ordinary differential equations. This will solve the equation $y_{xx} - 2 * y_x + y = \sin(x)$.

```
x = sy.symbols('x')
f = sy.Function('f')
eq = sy.Eq(f(x).diff(x, 2) - 2*f(x).diff(x) + f(x), sy.sin(x))
sy.dsolve(eq)
```

or, equivalently,

```
x = sy.symbols('x')
f = sy.Function('f')
expr = f(x).diff(x, 2) - 2*f(x).diff(x) + f(x) - sy.sin(x)
sy.dsolve(expr)
```

10 Newton's Method

Lab Objective: *Use Newton's Method to find zeros of a function. Determine where an initial point will converge to based on basins of attraction.*

Newton's method finds the roots of functions; that is, it finds $\overline{x}$ such that $f(\overline{x}) = 0$. This method can be used in optimization to determine where the maxima and minima occur. For example, it can be used to find the zeros of the first derivative.

Newton's Method

Newton's method begins with an initial guess x_0. Successive approximations of the root are found with the following recursive sequence:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

In other words, Newton's method approximates the root of a function by finding the x-intercept of the tangent line at $(x_n, f(x_n))$ (see Figure **??**).

The sequence $\{x_n\}$ will converge to the zero $\overline{x}$ of f if

1. f, f', and f'' exist and are continuous,
2. $f'(\overline{x}) \neq 0$, and
3. x_0 is "sufficiently close" to $\overline{x}$.

In applications, the first two conditions usually hold. However, if $\overline{x}$ and x_0 are not "sufficiently close," Newton's method may converge very slowly, or it may not converge at all.

Newton's method is powerful because given the three conditions above, it converges quickly. In these cases, the sequence $\{x_n\}$ converges to the actual root quadratically, meaning that the maximum error is squared at every iteration.

Let us do an example with $f(x) = x^2 - 1$. We define $f(x)$ and $f'(x)$ in Python as follows.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> f = lambda x : x**2 - 1
>>> Df = lambda x : 2*x
```

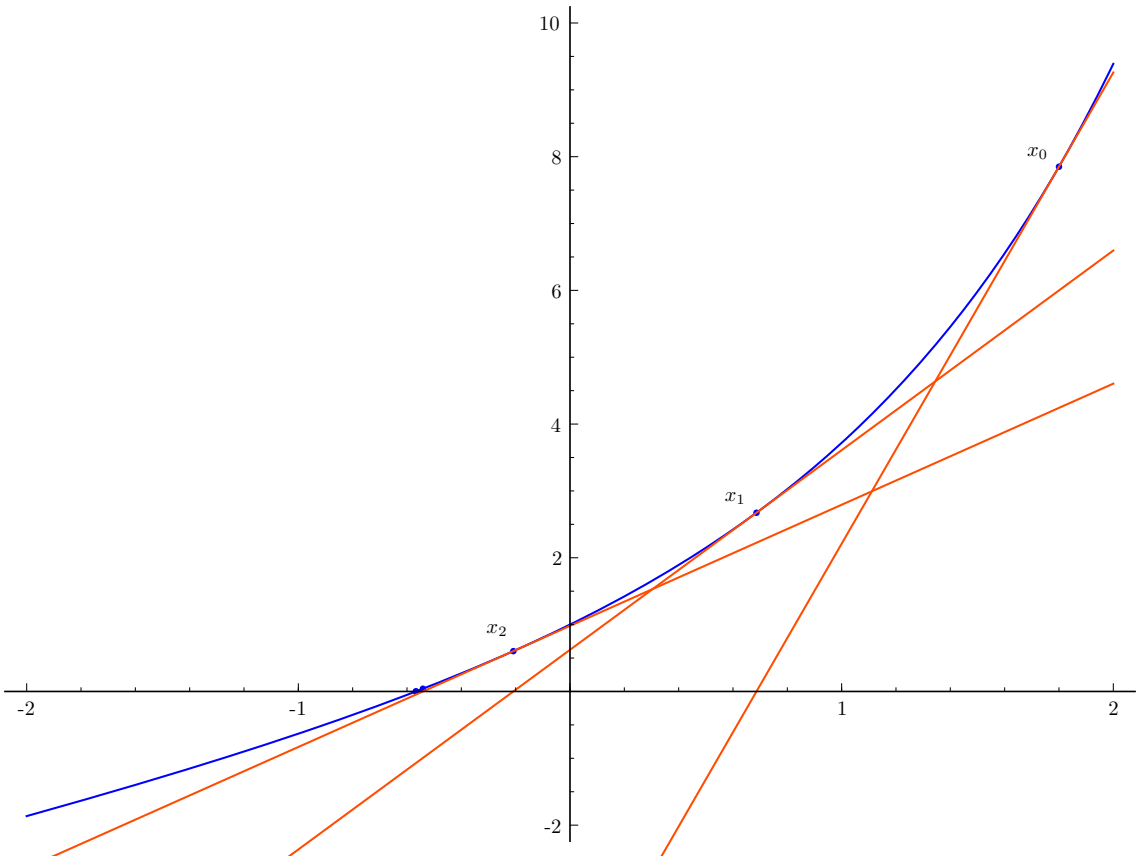


Figure 10.1: An illustration of how two iterations of Newton's method work.

Now we set $x_0 = 1.5$ and iterate.

```
>>> xold = 1.5
>>> xnew = xold - f(xold)/Df(xold)
>>> xnew
1.0833333333333333
```

We can repeat this as many times as we desire.

```
>>> xold = xnew
>>> xnew = xold - f(xold)/Df(xold)
>>> xnew
1.0032051282051282
```

We have already computed the root 1 to two digits of accuracy.

Problem 1. Implement Newton's method with a function that accepts the following parameters: a function f, an initial x-value, the derivative of the function f, the number of iterations of Newton's method to perform that defaults to 15, and a tolerance that defaults to 10^{-6}. The function returns when the difference between successive approximations is less than the tolerance or the max number of iterations has been reached.

Problem 2.

1. Newton's method can be used to find zeros of functions that are hard to solve for analytically. Plot $f(x) = \frac{sin(x)}{x} - x$ on $[-4, 4]$. Note that this function can be made continuous on this domain by defining $f(0) = 1$. Use your function `Newtons_method()` to compute the zero of this function to seven digits of accuracy.

2. Run `Newtons_method()` on $f(x) = x^{1/3}$ with $x_0 = .01$. What happens and why? Hint: The command `x**(1/3)` will not work when `x` is negative. Here is one way to define the function $f(x) = x^{1/3}$ in NumPy.

```
f = lambda x: np.sign(x)*np.power(np.abs(x), 1./3)
```

Problem 3. Suppose that an amount of P_1 dollars is put into an account at the beginning of years $1, 2, ..., N_1$ and that the account accumulates interest at a fractional rate r. (For example, $r = .05$ corresponds to 5% interest.) Suppose also that, at the beginning of years $N_1 + 1, N_1 + 2, ..., N_1 + N_2$, an amount of P_2 dollars is withdrawn from the account and that the account balance is exactly zero after the withdrawal at year $N_1 + N_2$. Then the variables satisfy the following equation:

$$P_1[(1 + r)^{N_1} - 1] = P_2[1 - (1 + r)^{-N_2}].$$

If $N_1 = 30, N_2 = 20, P_1 = 2000$, and $P_2 = 8000$, use Newton's method to determine r. (From Atkinson Page 118)

Backtracking

There are times when Newton's method may not converge due to the fact that the step from x_n to x_{n+1} was too large and the zero was stepped over completely. This was seen in Problem 2 when using $x_0 = .01$ to find the zero of $f(x) = x^{1/3}$. In that example, Newton's method did not converge since it stepped over the zero of the function, produced $x_1 = -.02$ and each iteration got increasingly more negative. To combat this problem of overstepping, backtracking is a useful tool. Backtracking is simply taking a fraction of the full step from x_n to x_{n+1}. Define Newton's Method with the recursive sequence:

$$x_{n+1} = x_n - \alpha \frac{f(x_n)}{f'(x_n)}$$

and the vector version of Newton's Method as:

$$x_{n+1} = x_n - \alpha Df(x_n)^{-1} f(x_n).$$

Previously, we have used $\alpha = 1$ in Newton's method. Backtracking uses $\alpha < 1$ in the above sequences and allows us to take a fraction of the step when the step size is too big.

Problem 4. 1. Modify your `Newtons_method()` function so that it accepts a parameter α that defaults to 1 to allow backtracking.

2. Find an $\alpha < 1$ so that running `Newtons_method()` on $f(x) = x^{1/3}$ with $x_0 = .01$ converges. (See Problem 2). Return the results of `Newtons_method()`.

Problem 5. 1. Create a `Newtons_vector()` function that performs Newton's method on vectors.

2. Bioremediation involves the use of bacteria to consume toxic wastes. At steady state, the bacterial density x and the nutrient concentration y satisfy the system of nonlinear equations

$$\gamma xy - x(1+y) = 0,$$
$$-xy + (\delta - y)(1+y) = 0,$$

where γ and δ are parameters that depend on various physical features of the system. For this problem, assume the typical values $\gamma = 5$ and $\delta = 1$, for which the system has solutions at $(x, y) = (0, 1), (0, -1)$, and $(3.75, .25)$. Solve the system using Newton's method and Newton's method with backtracking. (Find an initial point where using $\alpha = 1$ converges to either $(0, 1)$ or $(0, -1)$ and using $\alpha < 1$ converges to $(3.75, .25)$). Use `matplotlib` to demonstrate the tracks used to find the solution. (See Figure 10.2) Hint: use starting values within the rectangle

$$(x, y) : -.25 \leqslant x \leqslant .25, -.25 \leqslant y \leqslant .25.$$

(Adapted from problem 5.19 of M. T. Heath, Scientific Computing, an Introductory Survey, 2nd edition, McGraw?Hill, 2002 and the Notes of Homer Walker)

Basins of Attraction: Newton Fractals

When $f(x)$ has many roots, the root that Newton's method converges to depends on the initial guess x_0. For example, the function $f(x) = x^2 - 1$ has roots 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 10.3). We call the regions $(-\infty, 0)$ and $(0, \infty)$ *basins of attraction*.

When f is a polynomial of degree greater than 2, the basins of attraction are much more interesting. For example, if $f(x) = x^3 - x$, the basins are depicted in Figure 10.4.

We can extend these examples to the complex plane. Newton's method works in arbitrary Banach spaces with slightly stronger hypotheses (see Chapter 7 of Volume 1), and in particular it holds over $\mathbb{C}$.

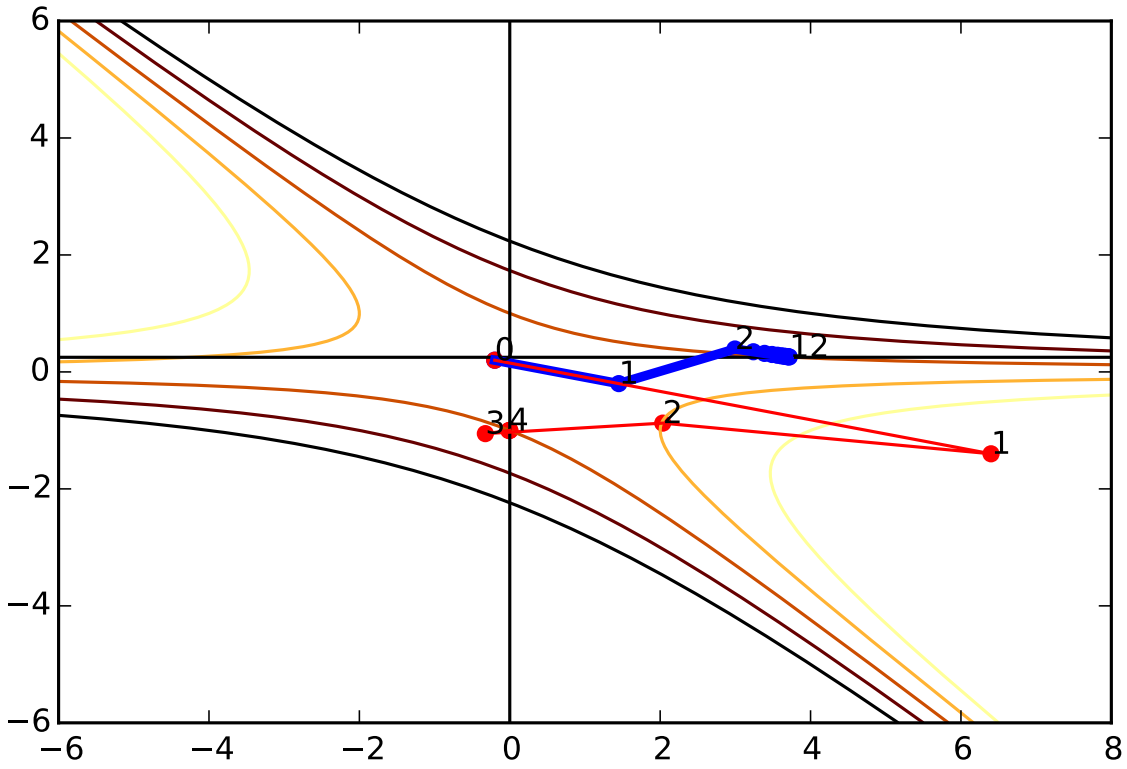


Figure 10.2: Starting at the same initial value results in convergence to two different solutions. The red line converges to $(0, -1)$ with $\alpha = 1$ in 4 iterations of Newton's method while the blue line converges to $(3.75, .25)$ with $\alpha < 1$ in 12 iterations .

Let us plot the basins of attraction for $f(x) = x^3 - x$ on the domain $\{a+bi \mid (a,b) \in [-1.5, 1.5] \times [-1.5, 1.5]\}$ in the complex plane. We begin by creating a 700×700 grid of points in this domain. We create the real and imaginary parts of the points separately, and then use `np.meshgrid()` to turn them into a single grid of complex numbers.

```
>>> xreal = np.linspace(-1.5, 1.5, 700)
>>> ximag = np.linspace(-1.5, 1.5, 700)
>>> Xreal, Ximag = np.meshgrid(xreal, ximag)
>>> Xold = Xreal+1j*Ximag
```

Recall that `1j` is the complex number i in NumPy. The array `Xold` contains 700^2 complex points evenly spaced in the domain.

We may now perform Newton's method on the points in `Xold`.

```
>>> f = lambda x : x**3-x
>>> Df = lambda x : 3*x**2 - 1
>>> Xnew = Xold - f(Xold)/Df(Xold)
```

After iterating the desired number of times, we have an array `Xnew` whose entries are various roots of $x^3 - x$.

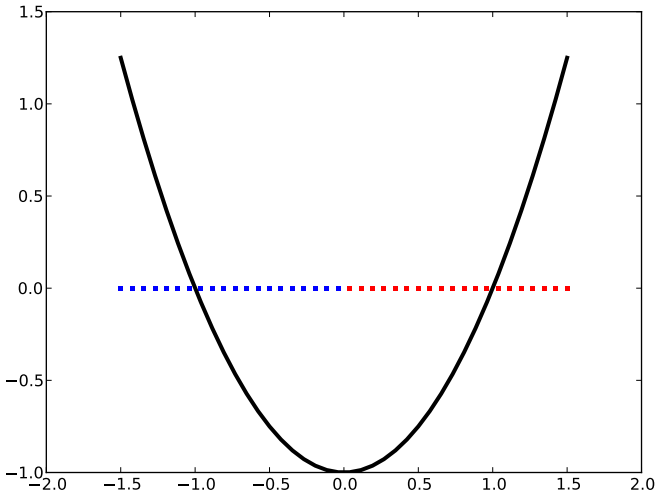


Figure 10.3: The plot of $f(x) = x^2 - 1$ along with some values for x_0. When Newton's method is initialized with a blue value for x_0 it converges to -1; when it is initialized with a red value it converges to 1.

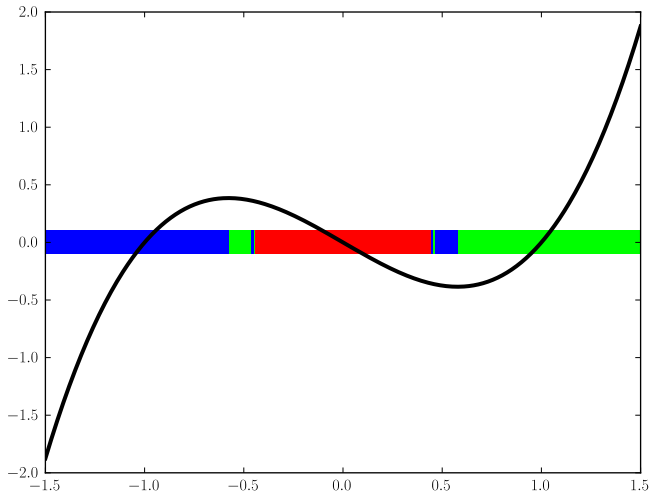


Figure 10.4: The plot of $f(x) = x^3 - x$ along with some values for x_0. Blue values converge to -1, red converge to 0, and green converge to 1.

Finally, we plot the array `Xnew`. The result is similar to Figure 10.5.

```
>>> plt.pcolormesh(Xreal, Ximag, Xnew)
```

Notice that in Figure 10.5, 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 fractal is called a *Newton fractal*.

Newton fractals tell us that the long-term behavior of the Newton 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 is an example of *chaos*.

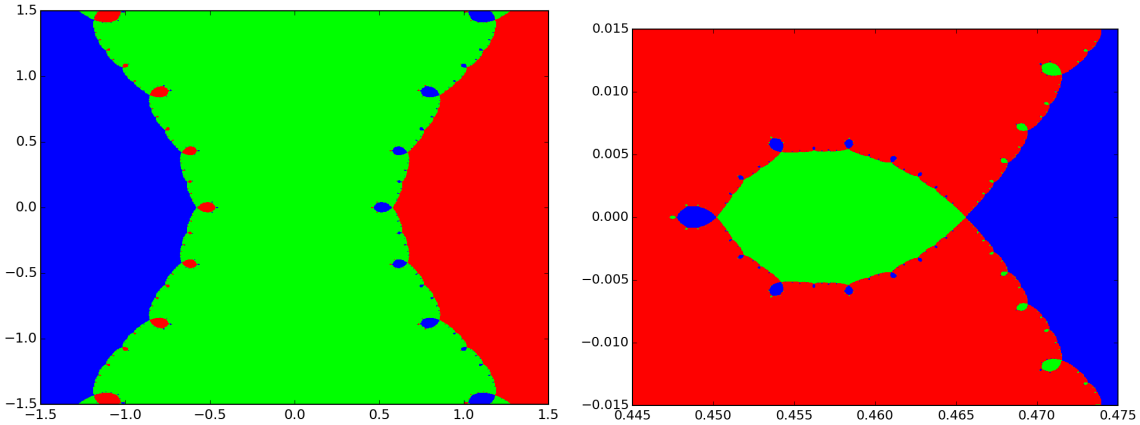


Figure 10.5: Basins of attraction for $x^3 - x$ in the complex plane. The picture on the right is a close-up of the figure on the left.

Problem 6. Complete the following function to plot the basins of attraction of a function.

```
def plot_basins(f, Df, roots, xmin, xmax, ymin, ymax, numpoints=100, iters=15, colormap='brg'):
    """Plot the basins of attraction of f.

    INPUTS:
    f       - A function handle. Should represent a function
              from C to C.
    Df      - A function handle. Should be the derivative of f.
    roots   - An array of the zeros of f.
    xmin, xmax, ymin, ymax - Scalars that define the domain
              for the plot.
    numpoints - A scalar that determines the resolution of
              the plot. Defaults to 100.
    iters   - Number of times to iterate Newton's method.
              Defaults to 15.
    colormap - A colormap to use in the plot. Defaults to 'brg'.
    """
```

You can test your function on the example $f(x) = x^3 - x$ above.

When the function `plt.pcolormesh()` is called on a complex array, it evaluates only on the real part of the complex numbers. This means that if two roots of `f` have the same real part, their basins will be the same color if you plot directly using `plt.pcolormesh()`.

One way to fix this problem is to compute `Xnew` as usual. Then iterate through the entries of `Xnew` and identify which root each entry is closest to using the input `roots`. Finally, create a new array whose entries are integers corresponding to the indices of these roots. Plot the array of integers to view the basins of attraction.
(Hint: The roots of $f(x) = x^3 - x$ are 0, 1, and -1.)

Problem 7. Run `plot_basins()` on the function $f(x) = x^3 - 1$ on the domain $\{a+bi \mid (a,b) \in [-1.5, 1.5] \times [-1.5, 1.5]\}$. The resulting plot should look like Figure 10.6.
(Hint: the roots of $f(x) = x^3 - 1$ are the third roots of unity: 1, $-\frac{1}{2} + \frac{\sqrt{3}}{2}i$, and $-\frac{1}{2} - \frac{\sqrt{3}}{2}i$.)

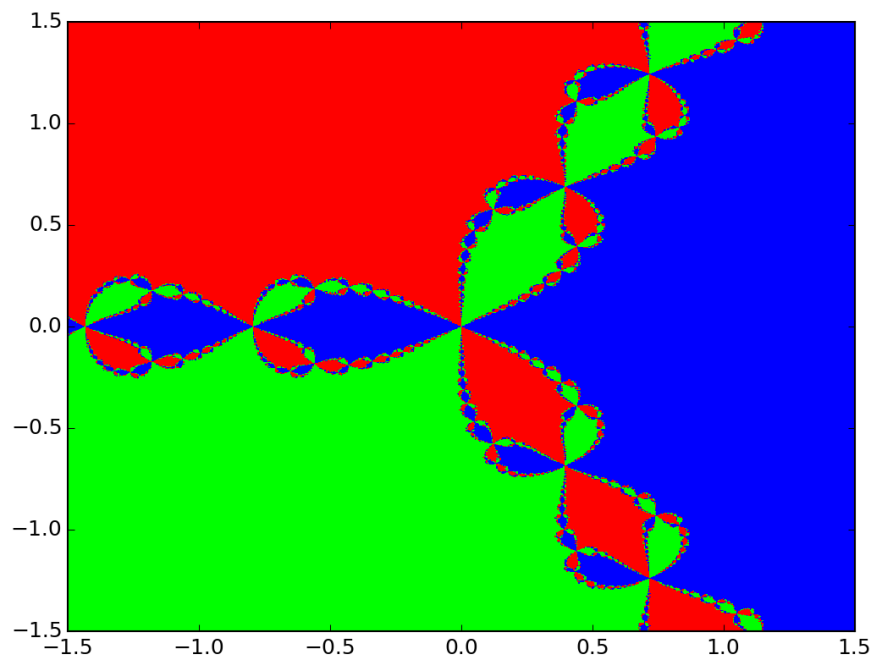


Figure 10.6: Basins of attraction for $x^3 - 1$.

11 Conditioning and Stability

Lab Objective: *The* condition number *of a function measures how sensitive that function is to changes in the input. On the other hand, the* stability *of an algorithm measures how accurately that algorithm computes the value of a function from exact input. In this lab, we examine the conditioning of common linear algebra problems, including computing polynomial roots and matrix eigenvalues. We also study several least squares algorithms to show that two algorithms for the same problem may not have the same level of stability.*

Condition Numbers

The *absolute condition number* of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ at a point $\mathbf{x} \in \mathbb{R}^m$ is defined by

$$\hat{\kappa}(\mathbf{x}) = \lim_{\delta \to 0^+} \sup_{\|\mathbf{h}\| < \delta} \frac{\|f(\mathbf{x}+\mathbf{h}) - f(\mathbf{x})\|}{\|\mathbf{h}\|}. \tag{11.1}$$

In other words, the absolute condition number of f is the limit of the change in output over the change of input. Similarly, the *relative condition number* of f is the limit of the *relative* change in output over the *relative* change in input:

$$\kappa(\mathbf{x}) = \lim_{\delta \to 0^+} \sup_{\|\mathbf{h}\| < \delta} \left(\frac{\|f(\mathbf{x}+\mathbf{h}) - f(\mathbf{x})\|}{\|f(\mathbf{x})\|} \Big/ \frac{\|\mathbf{h}\|}{\|\mathbf{x}\|} \right) = \frac{\|\mathbf{x}\|}{\|f(\mathbf{x})\|} \hat{\kappa}(\mathbf{x}). \tag{11.2}$$

When a function's condition number is large, it is called *ill-conditioned*. Small changes to the input of an ill-conditioned function may produce large changes in output. In applications, it is important to know if a function is ill-conditioned because floating point representation almost always introduces input error.

The *condition number of a matrix* $\kappa(A) = \|A\|\|A^{-1}\|$ is an upper bound on the condition number for many of the common problems associated with the matrix, such as solving the system $A\mathbf{x} = \mathbf{b}$. If A is square but not invertible, then $\kappa(A) = \infty$ by convention. To compute $\kappa(A)$, we often use the matrix 2-norm, which is the largest singular value $\sigma_{\max}$ of A. Then since if σ is a singular value of A, $\frac{1}{\sigma}$ is a singular value of A^{-1}, we have

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}, \tag{11.3}$$

which is also a valid equation for non-square matrices.

Problem 1. Write a function that accepts a matrix A and computes its condition number using (11.3). Use `scipy.linalg.svd()`, `scipy.linalg.svdvals()`, or `np.linalg.svd()` to compute the singular values of A, and avoid computing A^{-1}. If the smallest singular value is 0, return infinity (`np.inf`).

Test your function against `np.linalg.cond()`; you should expect the values to be within 10^{-7} of one another.

For large matrices where taking the SVD is difficult, the exact condition number of a matrix cannot always be computed and therefore must be estimated. Although not covered here, there exist many algorithms that can efficiently and accurately estimate the condition number of a matrix.

Example: The Wilkinson Polynomial

Let $f : \mathbb{C}^{n+1} \to \mathbb{C}^n$ be the function that maps the sequence of coefficients $(a_1, \dots, a_{n+1})$ to the roots of the polynomial $a_1x^n + a_2x^{n-1} + \dots + a_nx + a_{n+1}$. Finding the roots of polynomials is extremely ill-conditioned in general, so the condition number of f is likely very large.

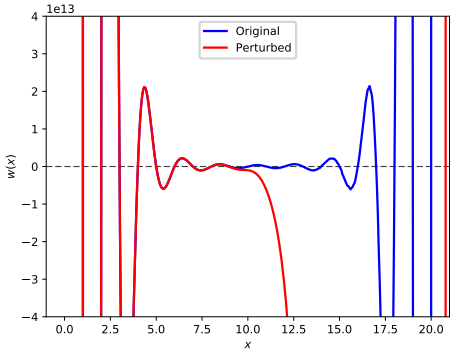
For instance, consider the Wilkinson polynomial.

$$w(x) = \prod_{r=1}^{20}(x - r) = x^{20} - 210x^{19} + 20615x^{18} - \dots$$

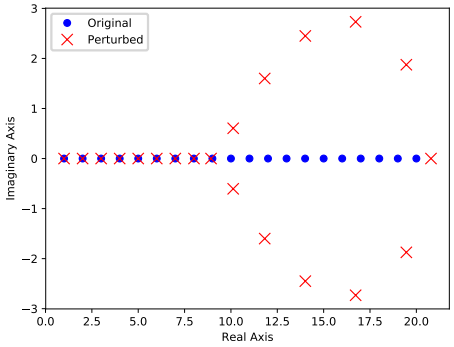
Let $\tilde{w}(x)$ be $w(x)$ where the coefficient on x^{19} is very slightly perturbed from -210 to -210.0000001. Below, we compute and compare the roots of $\tilde{w}(x)$ and $w(x)$ using NumPy and SymPy.

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

# The roots of w are 1, 2, ..., 20.
>>> w_roots = np.arange(1, 21)

# Get the exact Wilkinson polynomial coefficients using SymPy.
>>> x, i = sy.symbols('x i')
>>> w, _ = sy.poly_from_expr(sy.product(x-i, (i, 1, 20)))
>>> w_coeffs = np.array(w.all_coeffs())

# Perturb one of the coefficients very slightly.
>>> perturb = np.zeros(21)
>>> perturb[1]=1e-7
>>> perturbed_coeffs = w_coeffs - perturb

# Use NumPy to compute the roots of the perturbed polynomial.
>>> perturbed_roots = np.roots(np.poly1d(perturbed_coeffs))
```

Below we plot the polynomials $w(x)$ and $\tilde{w}(x)$ and compare their roots in the complex plane.



(a) The original and perturbed Wilkinson polynomials. They match for only about half of the domain.



(b) Roots of the original and pertubed Wilkinson polynomials. About half of the perturbed roots are imaginary.

From the figure, it's clear that a perturbation drastically changes the nature of the polynomial and its root. To quantify the difference, we estimate the condition numbers in the L^∞ norm.

```
# Sort the roots to ensure that they are in the same order.
>>> w_roots = np.sort(w_roots)
>>> perturbed_roots = np.sort(perturbed_roots)

# Estimate the absolute condition number in the infinity norm.
>>> k = la.norm(perturbed_roots - w_roots, np.inf) / la.norm(perturb, np.inf)
>>> print(k)
28262391.3304

# Estimate the relative condition number in the infinity norm.
>>> k*la.norm(w_coeffs, np.inf) / la.norm(w_roots, np.inf)
1.95063629993970+25                    # This is huge!!
```

There are some caveats to this example. First, when we compute the quotients in (11.1) and (11.2) for a fixed $\mathbf{h}$, we are only *approximating* the condition number. The actual condition number is the limit of such quotients. We hope that when $||\mathbf{h}||$ is small, a random quotient is at least the same order of magnitude as the limit, but we have no way to be sure.

Second, this example assumes that NumPy's root-finding algorithm is *stable*, so that the difference between the roots of `w_coeffs` and `perturbed_coeffs` is due to the difference in coefficients, and not the difference in roots. We will return to this issue in the next section.

Problem 2. Write a function that carries out the following experiment 100 times.

1. Randomly perturb the true coefficients by replacing each coefficient a_i with $a_i * r_i$, where r_i is drawn from a normal distribution centered at 1 with standard deviation 10^{-10} (use `np.random.normal()`).

2. Plot the perturbed roots as small points in the complex plane. That is, plot the real part of the coefficients on the x-axis and the imaginary part on the y-axis. Plot on the same figure in each experiment.

3. Compute the absolute and relative condition numbers with the L^∞ norm.

Plot the roots of the unperturbed Wilkinson polynomial with the perturbed roots. Your final plot should resemble Figure 11.2. Finally, return the average computed absolute and relative condition numbers.

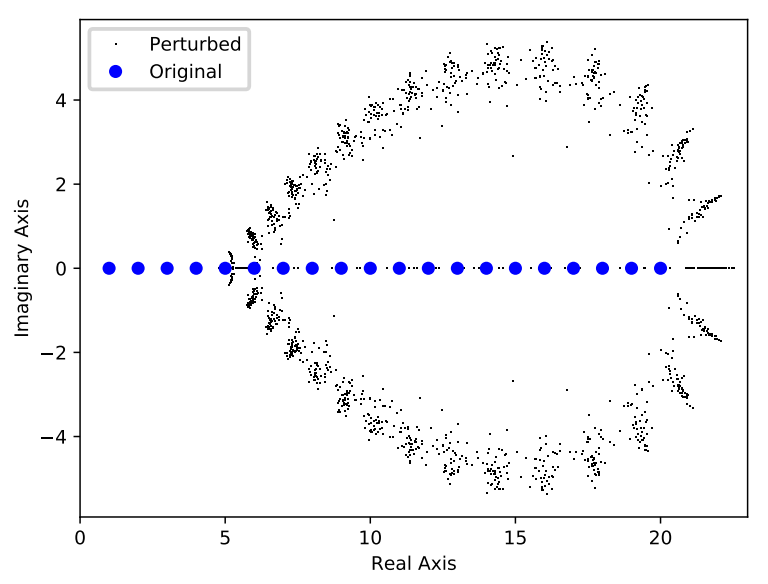


Figure 11.2: This figure replicates Figure 12.1 on p. 93 of *Numerical Linear Algebra* by Lloyd N. Trefethen and David Bau III.

Example: Calculating Eigenvalues

Let $f : \mathbb{C}^{n^2} \to \mathbb{C}^n$ be the function that maps an $n \times n$ matrix to its n eigenvalues. This problem is well-conditioned for symmetric matrices, but can be extremely ill-conditioned for non-symmetric matrices. Let A be an $n \times n$ matrix and let $\boldsymbol{\lambda}$ be the vector of the n eigenvalues of A. If $\tilde{A} = A + H$ and $\tilde{\boldsymbol{\lambda}}$ are a perturbation of A and its eigenvalues, then the condition numbers of f are

$$\hat{\kappa}(A) = \frac{\|\boldsymbol{\lambda} - \tilde{\boldsymbol{\lambda}}\|}{\|H\|}, \qquad \kappa(A) = \frac{\|A\|}{\|\boldsymbol{\lambda}\|}\hat{\kappa}(A). \tag{11.4}$$

Problem 3. Write a function that accepts a matrix A and estimates the condition number of the eigenvalue problem using (11.4). For the perturbation H, use arandom complex matrices with entries drawn from a normal distribution.

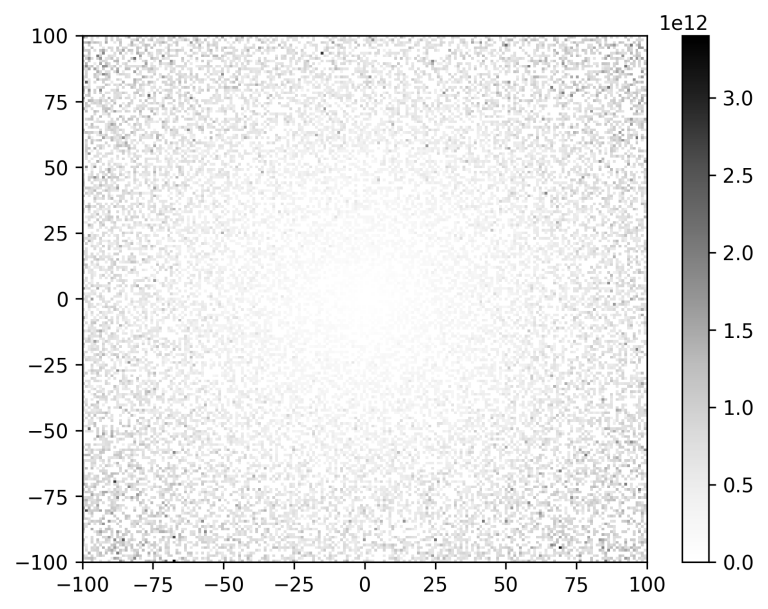
```
H = np.random.normal(0, 1e-10, A.shape) + np.random.normal(
                                                0, 1e-10, A.shape)*1j
```

Use the 2-norm for both the vector and matrix norms. Return the absolute and relative condition numbers.

Problem 4. Write a function that accepts minimal and maximal values for x and y as well as a resolution parameter `res`. Use your function from Problem 3 to compute the relative condition number of the eigenvalue for the 2x2 matrix

$$\begin{bmatrix} 1 & x \\ y & 1 \end{bmatrix}$$

at every point in the domain $[x_{min}\ x_{max}] \times [y_{min}\ y_{max}]$, where each axis is partitioned into `res` points. Plots these estimated relative condition numbers using `plt.pcolormesh()`. With `res`=200, your plot should similar to the following figure.



Stability of an Algorithm

The stability of an algorithm is measured by the error in its output. Suppose we have some algorithm to compute $f : \mathbb{R}^m \to \mathbb{R}^n$. Let $\tilde{f}(\mathbf{x})$ represent the value computed by the algorithm given an input $\mathbf{x}$. Then the *forward error* of f at $\mathbf{x}$ is $||f(\mathbf{x}) - \tilde{f}(\mathbf{x})||$, and the *relative forward error* of f at $\mathbf{x}$ is

$$\frac{||f(\mathbf{x}) - \tilde{f}(\mathbf{x})||}{||f(\mathbf{x})||}.$$

An algorithm is *stable* if its relative forward error is small.

As an example, let us examine the stability of NumPy's root finding algorithm that we used to investigate the Wilkinson polynomial. We know the exact roots of $w(x)$, and we can also compute these roots using NumPy's `np.roots()` function.

```
>>> roots = np.arange(1,21)
 # w_coeffs is an array with the coefficients of the Wilkinson polynomial.
>>> computed_roots = np.roots(np.poly1d(w_coeffs))
```

```
# Sort the roots for comparison.
>>> roots = np.sort(roots)
>>> computed_roots = np.sort(computed_roots)

# Compute the forward error.
>>> forward_error = la.norm(roots-computed_roots)
>>> print(forward_error)
0.020612653126379665

# Compute the relative forward error.
>>> forward_error / la.norm(roots)
0.00038476268486104599                              # Nice and small.
```

This analysis suggests that questions of stability did not interfere too much with our experiments in Problem 2.

Example: Least Squares

The least squares problem is to find the $\mathbf{x}$ that minimizes $\|A\mathbf{x}-\mathbf{b}\|_2$ for fixed A and $\mathbf{b}$. It can be shown that an equivalent problem is finding the solution of $A^\mathsf{T}A\mathbf{x} = A^\mathsf{T}\mathbf{b}$, called the *normal equations*. We will consider two different algorithms that solve this problem.

1. Invert the matrix $A^\mathsf{T}A$ and then right multiply by $A^\mathsf{T}\mathbf{b}$ to solve the normal equations. Although this approach seems intuitive, it is actually highly unstable and can return an answer with a very large forward error.

2. Use the QR-decomposition, factoring the $m\times n$ matrix A of rank $n \geq m$, where Q has orthonormal columns and R is upper triangular. It can also be shown that the solution of $R\mathbf{x} = Q\mathbf{b}$ is equivalent to solving the least squares problem. This algorithm has the advantage of being stable.

A common application of least squares is polynomial approximation. Given a set of data points $\{(x_k, y_k)\}_{k=1}^m$ we would like to find the set of coefficients $\{c_k\}_{k=1}^n$ such that

$$y_k = c_n x_k^n + c_{n-1}x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all k. Since this system will typically be over-determined, there will not be a set of coefficients that exactly satisfies this equation for all k. A common approach is to use least squares to pick the set of coefficients that minimizes the L^2 error of the system of equations.

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

# Use least squares to approximate sin(x) with a five-degree polynomial.
>>> x = np.linspace(0, 6, 10)           # The x-values of the data.
>>> b = np.sin(x) + .2*np.random.randn(10) # The y-values of the data (noisy).
>>> A = np.vander(x, 6)                 # Set up the matrix of data values.
>>> coeffs = la.lstsq(A, b)[0]          # Get the polynomial coefficients.

>>> domain = np.linspace(0, 6, 100)     # Define a finer domain for plotting.
>>> plt.plot(x, b, 'k*')
```

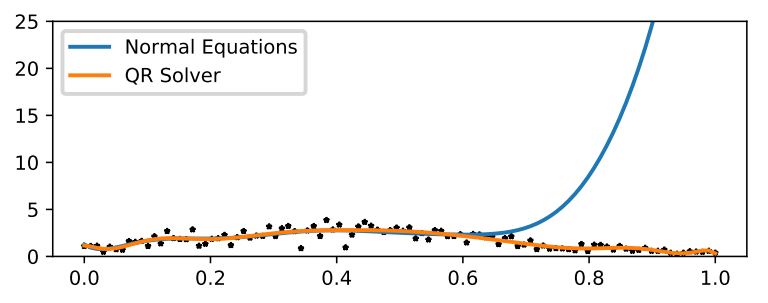
```
>>> plt.plot(domain, np.sin(domain))
>>> plt.plot(domain, np.poly1d(coeffs)(domain))
```

Problem 5. Write two functions that solve the least squares problem using the normal equation. One of your functions should solve the problem directly using `np.linalg.inv()` while the other should use a more stable algorithm, such as the QR-decomposition. The functions `la.qr()` and `la.solve_triangular()` may be helpful when implementing the QR-decomposition. Both of your functions should accept a matrix A and a vector $\mathbf{b}$.

Approximate the data from `stability_data.npy` on the interval $[0,\ 1]$ with a degree-fourteen polynomial using two approaches to get the least squares solution:

1. Use `la.inv()` and the normal equations: $\mathbf{x} = (A^\mathsf{T}A)^{-1}A^\mathsf{T}\mathbf{b}$.
2. Use `la.qr()` and `la.solve_triangular()` to solve the system $R\mathbf{x} = Q\mathbf{b}$.

Plot both of your solutions with the data points for comparision. Return the error $\|A\mathbf{x} - \mathbf{b}\|_2$ of both approximations. Your plot should look similar to the following figure.



Catastrophic Cancellation

A common cause of instability in algorithms is catastrophic cancellation. *Catastrophic cancellation* is the term for when a computer takes the difference of two very similar numbers, and the result is stored with a small number of significant digits. Because of the way computers store and perform arithmetic on numbers, future computations can amplify a catastrophic cancellation into a large error.

You are at risk for catastrophic cancellation whenever you subtract floats or large integers that are very close to each other. You can avoid this problem by either rewriting your program to not use subtraction, or by increasing the number of significant digits that your computer tracks.

Here is an example of catastrophic cancellation. Suppose we wish to compute $\sqrt{a} - \sqrt{b}$. We can either do this subtraction directly or perform the equivalent division

$$\sqrt{a} - \sqrt{b} = (\sqrt{a} - \sqrt{b})\frac{\sqrt{a} + \sqrt{b}}{\sqrt{a} + \sqrt{b}} = \frac{a - b}{\sqrt{a} + \sqrt{b}}.$$

We will perform this computation both ways in NumPy with $a = 10^{20} + 1$ and $b = 10^{20}$.

```
>>> from math import sqrt          # np.sqrt() fails for very large numbers
```

```
>>> a = 10**20+1
>>> b = 10**20
>>> sqrt(a)- sqrt(b)
0.0
>>> (a-b) / (np.sqrt(a)+np.sqrt(b))
5e-11
```

Since $a \neq b$, $\sqrt{a} - \sqrt{b}$ should clearly be nonzero.

Problem 6. Let $I(n) = \int_0^1 x^n e^{x-1} dx$.

1. Prove that $0 \leq I(n) \leq 1$ for all n.
2. It can be shown that for $n > 1$,

$$I(n) = (-1)^n !n + (-1)^{n+1} \frac{n!}{e}$$

   where $!n$ is the *subfactorial* of n. Use this formula to write a function that computes the integral.
   (Hint: Use SymPy's `subfactorial()` function.)
3. The actual values of $I(n)$ for many values of n are listed in the table below. Use your function `integral()` to compute $I(n)$ for these same values of n, and create a table comparing the data. How can you explain what is happening?

n	Actual value of $I(n)$
1	0.367879441171
5	0.145532940573
10	0.0838770701034
15	0.0590175408793
20	0.0455448840758
25	0.0370862144237
30	0.0312796739322
35	0.0270462894091
40	0.023822728669
45	0.0212860390856
50	0.0192377544343

12 Monte Carlo Integration

Lab Objective: *Implement Monte Carlo integration to estimate integrals. Use Monte Carlo Integration to calculate the integral of the joint normal distribution.*

Some multivariable integrals which are critical in applications are impossible to evaluate symbolically. For example, the integral of the joint normal distribution

$$\int_\Omega \frac{1}{\sqrt{(2\pi)^k}} e^{-\frac{\mathbf{x}^T\mathbf{x}}{2}}$$

is ubiquitous in statistics. However, the integrand does not have a symbolic antiderivative. This means we must use numerical methods to evaluate this integral. The standard technique for numerically evaluating multivariable integrals is *Monte Carlo Integration*. In the next lab, we will approximate this integral using a modified version of Monte Carlo Integration. In this lab, we address the basics of Monte Carlo Integration.

Monte Carlo integration is radically different from techniques like Simpson's rule. Whereas Simpson's rule is purely computational and deterministic, Monte Carlo integration uses randomly chosen points in the domain to calculate the integral. Although it converges slowly, Monte Carlo integration is frequently used to evaluate multivariable integrals because the higher-dimensional analogs of methods like Simpson's rule are extremely inefficient.

A Motivating Example

Suppose we want to numerically compute the area of a circle of radius 1. From analytic methods, we know the answer is π. Empirically, we can estimate the area by randomly choosing points in a 2×2 square. The percentage of points that land in the inscribed circle, times the area of the square, should approximately equal the area of the circle (see Figure 12.1).

We do this in NumPy as follows. First generate 500 random points in the square $[0, 1] \times [0, 1]$.

```
>>> N = 500 # Number of sample points
>>> points = np.random.rand(2, N)
```

We rescale and shift these points to be uniformly distributed in $[-1, 1] \times [-1, 1]$.

```
>>> points = points*2-1
```

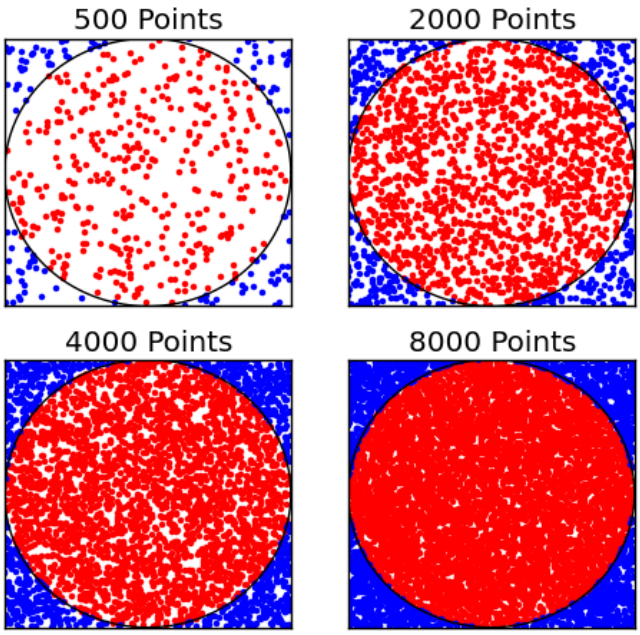


Figure 12.1: Finding the area of a circle using random points

Next we determine the number of points in the unit circle. We compute the Euclidean distance from the origin for each point, then count the points that are within a distance of 1 from the origin.

```
>>> # Compute the distance from the origin for each point
>>> pointsDistances = np.linalg.norm(points,axis=0)
>>> # Count how many are less than 1
>>> numInCircle = np.count_nonzero(pointsDistances < 1)
```

The fraction of points inside the circle is `numInCircle` divided by `N`. By multiplying this fraction by the square's area, we can estimate the area of the circle.

```
>>> circleArea = 4.*(numInCircle/N)
>>> circleArea
3.024
```

This differs from π by about 0.117.

Problem 1. Write a function that estimates the volume of the unit sphere. Your function should have a keyword argument `N` that defaults to 10^5. Your function should draw `N` points uniformly from $[-1,1]\times[-1,1]\times[-1,1]$ to make your estimate. The true volume is $\frac{4}{3}\pi \approx 4.189$.

Monte Carlo Integration

In the examples above, we drew a bounding box around a volume, then used random points drawn from the box to estimate that volume. This is easy and intuitive when the volume is a circle or sphere. But it's hard to generalize this to any arbitrary integral - in order to draw the box, we have to already know something about the volume we are estimating. Instead, given an arbitrary function $f(x) : \mathbb{R}^n \mapsto \mathbb{R}$ and a region $\Omega \subset \mathbb{R}^n$ in the domain of f, we would like to use random points drawn from Ω to estimate the integral $\int_\Omega f(x)\, dV$, *without* having to specify a bounding box around the volume of integration.

We can estimate this integral using the approximation.

$$\int_\Omega f(x)\, dV \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{12.1}$$

where x_i are uniformly distributed random points in Ω and $V(\Omega)$ is the volume of Ω. This is the generalized formula for Monte Carlo integration.

The intuition behind (12.1) is that $\frac{1}{N}\sum_{i=1}^{N} f(x_i)$ approximates the average value of f on Ω. We multiply this (approximate) average value by the volume of Ω to get the (approximate) integral of f on Ω.

For further intuition, compare (12.1) to the Average Value Theorem from single-variable calculus. By the Average Value Theorem, the average value of $f(x) : \mathbb{R} \mapsto \mathbb{R}$ on $[a, b]$ is given by

$$f_{avg} = \frac{1}{b-a}\int_a^b f(x)\, dx. \tag{12.2}$$

If we let $\Omega = [a, b]$ in (12.2) (noting that $V(\Omega) = b-a$) and replace f_{avg} with the approximation $\frac{1}{N}\sum_{i=1}^{N} f(x_i)$, then we get precisely the Monte Carlo integration formula in Equation (12.1)!

As it turns out, we can refactor the circle-area problem slightly so that it uses Equation (12.1). Let f be defined by

$$f(x, y) = \begin{cases} 1 & \text{if } (x,y) \text{ is in the unit circle} \\ 0 & \text{otherwise} \end{cases}$$

and let $\Omega = [-1, 1] \times [-1, 1]$. The area of the circle is given by $\int_\Omega f(x)\, dV$, which we can estimate with the Monte Carlo integration formula:

$$\text{Area of unit circle } \approx V(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(x_i) = \frac{4}{N}\sum_{i=1}^{N} f(x_i).$$

To summarize, we have the following steps to estimate the integral of any function f over a region Ω in the domain of f:

1. Draw N random points uniformly distributed in Ω.
2. Find the image of each point under f, and take the average of these images.
3. Multiply by the volume of Ω.

Problem 2. Write a function that performs 1-dimensional Monte Carlo integration. Given a function $f : \mathbb{R} \mapsto \mathbb{R}$, an interval $[a, b]$, and the number of random points to use, your function should return an approximation of the integral $\int_a^b f(x)\,dx$. Let the number of sample points default to 10^5. Test your function by estimating integrals that you can calculate by hand.

Problem 3. Generalize Problem 2 to multiple dimensions. Write a function that accepts a function handle f to integrate, the bounds of the interval to integrate over, and the number of points to use. Let the number of sample points default to 10^5. Your implementation should be robust enough to integrate any function $f : \mathbb{R}^n \mapsto \mathbb{R}$ over any interval in $\mathbb{R}^n$.

Hints:

1. To draw a random array of points from the given interval, first create a random array of points in $[0, 1] \times \ldots \times [0, 1]$. Multiply this array by the dimensions of the interval to rescale it, then add the lower bounds of integration to shift it. Think about using array broadcasting.

2. You can use `np.apply_along_axis()` to apply a function to each column of an array. Here is an example of applying a function to points in $\mathbb{R}^2$:

```
>>> points = np.random.rand(2,4)
>>> points
array([[ 0.33144631,  0.52558001,  0.67766158,  0.45570083],
       [ 0.70935864,  0.20985475,  0.25917177,  0.19431292]])
# Apply the norm function to each point
>>> np.apply_along_axis(np.linalg.norm,0,points)
array([ 0.78297275,  0.565927  ,  0.72553099,  0.49539959])
```

   This is especially useful for functions that don't work nicely with array inputs. For example, the code below uses a simple thresholding function `f`; calling `f(points)` would throw an error, but using `np.apply_along_axis` gives the expected result. (Note that `points` must have at least 2 dimensions for this to work.) Refer to the NumPy docs for more information.

```
# Simple function that returns a 0 if x is less than 0.5
>>> f = lambda x: x if x > 0.5 else 0.
# Get 4 random points. The 1 forces the array to be 2-D.
>>> points = np.random.rand(1,4)
>>> points
array([[ 0.2144746 ,  0.02490517,  0.86593995,  0.86401139]])
# Evaluate f at each element of points
>>> np.apply_along_axis(f,0,points)
array([ 0.        ,  0.        ,  0.86593995,  0.86401139])
```

One application of Monte Carlo integration is integrating probability density functions that do not have closed form solutions.

Problem 4. The joint normal distribution of N independent random variables with mean 0 and variance 1 is

$$f(\mathbf{x}) = \frac{1}{\sqrt{2\pi}^N} e^{-\frac{\mathbf{x}^T\mathbf{x}}{2}}.$$

The integral of $f(\mathbf{x})$ over a box is the probability that a draw from the distribution will be in the box. This is an important distribution in statistics. However, $f(\mathbf{x})$ does not have a symbolic antiderivative.

1. Let $\Omega = [-1.5, 0.75] \times [0, 1] \times [0, 0.5] \times [0, 1] \subset \mathbb{R}^4$. Use the function you wrote in Problem 1 to integrate $f(\mathbf{x})$ on Ω. Use 50000 sample points.

2. SciPy has a built in function specifically for integrating the joint normal distribution. The integral of $f(\mathbf{x})$ on $B = [-1, 1] \times [-1, 1] \times [-1, 1] \subset \mathbb{R}^3$ can be computed in SciPy with the following code.

```
>>> from scipy import stats

# Define the bounds of the box to integrate over
>>> mins = np.array([-1, -1, -1])
>>> maxs = np.array([1, 1, 1])

# Each variable has mean 0
>>> means = np.zeros(3)

# The covariance matrix of N independent random variables
#    is the NxN identity matrix.
>>> covs = np.eye(3)

# Compute the integral
>>> value, inform = stats.mvn.mvnun(mins, maxs, means, covs)
```

   Then `value` is the integral of $f(\mathbf{x})$ on B.

   Use SciPy to integrate $f(\mathbf{x})$ on Ω.

3. Return your Monte Carlo estimate, SciPy's answer, and (assuming SciPy is correct) the relative error of your Monte Carlo estimate.

Convergence

The error of the Monte Carlo method is proportional to $1/\sqrt{N}$, where N is the number of points used in the estimation. This means that to divide the error by 10, we must sample *100 times* more points.

This is a slow convergence rate, but it is independent of the number of dimensions of the problem. The error converges at the same rate whether integrating a 2-dimensional or a 20-dimensional function. This gives Monte Carlo integration an advantage over other methods, and makes it especially useful for estimating integrals in high dimensions.

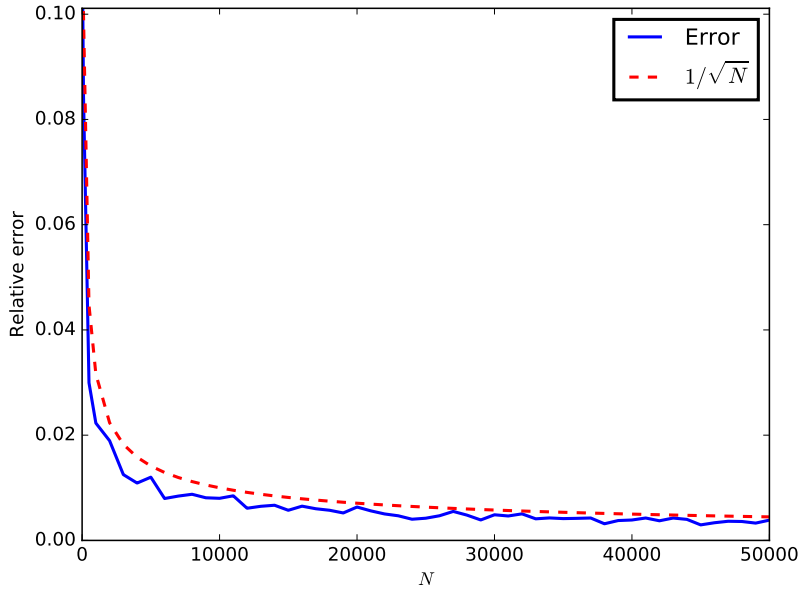


Figure 12.2: The Monte Carlo integration method was used to compute the volume of the unit sphere. The blue line plots the average error in 50 runs of the Monte Carlo method on N sample points. The red line is a plot of $1/\sqrt{N}$.

Problem 5. In this problem we will visualize how the error in Monte Carlo integration depends on the number of sample points.

Run Problem 1 with `N` equal to 50, 100 and 500, as well as 1000, 2000, 3000, ..., 50000; having some additional small values of `N` will help make the visualization better.

For each value of `N`:

1. Estimate the volume of the unit sphere using Problem 1, and use the true volume to calculate the relative error of the estimate.
2. Repeat this multiple times to get an average estimate of the relative error. Your function should accept a keyword argument `numEstimates` that defaults to 50.
3. Calculate and store the mean of the errors.

Plot the mean relative error as a function of `N`. For comparison, plot the function $1/\sqrt{N}$ on the same graph. Your plot should resemble Figure 12.2).

A Caution

You can run into trouble if you try to use Monte Carlo integration on an integral that does not converge. For example, we may attempt to evaluate

$$\int_0^1 \frac{1}{x}\,dx$$

with Monte Carlo integragtion using the following code.

```
>>> k = 5000
>>> np.mean(1/np.random.rand(k,1))
21.237332864358656
```

Since this code returns a finite value, we could assume that this integral has a finite value. In fact, the integral is infinite. We could discover this empirically by using larger and larger values of k, and noting that Monte Carlo integration fails to converge.

13 Importance Sampling

Lab Objective: *Use importance sampling to reduce the error and variance of Monte Carlo Simulations.*

Introduction

The traditional methods of Monte Carlo integration as discussed in the previous lab are not always the most efficient means to estimate an integral. For example, assume we were trying to find the probability that a randomly chosen variable X from the standard normal distribution is greater than 3. We know that one way to solve this is by solving the following integral:

$$P(X > 3) = \int_3^\infty f_X(t)\,dt = \frac{1}{\sqrt{2\pi}} \int_3^\infty e^{-t^2/2}\,dt \tag{13.1}$$

If we define the function $h : \mathbb{R} \to \mathbb{R}$ as

$$h(t) = \begin{cases} 1 & \text{if } t > 3 \\ 0 & \text{if } t \leq 3 \end{cases},$$

we can rewrite this integral as

$$\int_3^\infty f_X(t)\,dt = \int_{-\infty}^\infty h(t) f_X(t)\,dt.$$

By the Law of the Unconscious Statistician (see Volume 2 §3.5), we can restate the integral above as

$$\int_{-\infty}^\infty h(t) f_X(t)\,dt = E[h(X)].$$

Being able to write integrals as expected values is an essential tool in this lab.

Monte Carlo Simulation

In the last section, we expressed the probability of drawing a number greater than 3 from the normal distribution as an expected value problem. We can now easily estimate this same probabilty using Monte Carlo simulation. Given a random i.i.d. sample $x_1, x_2, \cdots, x_N$ generated by f_X, we can estimate $E[h(X)]$ using

$$\widehat{E}_n[h(X)] = \frac{1}{N}\sum_{i=1}^{N} h(x_i) \tag{13.2}$$

Now that we have defined the estimator, it is now quite manageable to approximate Equation 13.1. By the Weak Law of Large Numbers (see Volume 2 §3.6), the estimate will get closer and closer to the actual value as we use more and more sample points.

Problem 1. Write a function in Python that estimates the probability that a random draw from the standard normal distribution is greater than 3 using Equation 13.2. Your function should accept a parameter `n` for the number of samples to use in your approximation. Your answer should approach 0.0013499 for sufficiently large samples.

Though this approach gets the job done, it turns out that this isn't very efficient. Since the probability of drawing a number greater than 3 from the standard normal distribution is so unlikely, it turns out we need many sample points to get a good approximation.

Importance Sampling

Importance sampling is one way to make Monte Carlo simulations converge much faster. We choose a different distribution to sample our points to generate more *important* points. With our example, we want to choose a distribution that would generate more numbers around 3 to get a more reliable estimate. The theory behind importance sampling boils down to the following result. In these equations, the random variable X is generated by f_X and the random variable Y is generated by g_Y. We will refer to X and Y in this way for the remainder of the lab.

$$\begin{aligned} E[h(X)] &= \int_{-\infty}^{\infty} h(t) f_X(t)\, dt \\ &= \int_{-\infty}^{\infty} h(t) f_X(t) \left(\frac{g_Y(t)}{g_Y(t)}\right) dt \\ &= \int_{-\infty}^{\infty} \left(\frac{h(t) f_X(t)}{g_Y(t)}\right) g_Y(t)\, dt \\ &= E\left[\frac{h(Y) f_X(Y)}{g_Y(Y)}\right] \end{aligned} \tag{13.3}$$

The corresponding estimator is

$$\begin{aligned} \widehat{E}[h(X)] &= \widehat{E}\left[\frac{h(Y) f_X(Y)}{g_Y(Y)}\right] \\ &= \frac{1}{N}\sum_{i=1}^{N} \frac{h(y_i) f_X(y_i)}{g_Y(y_i)} \end{aligned} \tag{13.4}$$

The function f_X is the p.d.f. of the *target distribution*. The function g_Y is the p.d.f. of the *importance distribution*. The fraction $\frac{f_X(X)}{g_Y(X)}$ is called the *importance weight*. This allows us to draw a sample from any distribution with p.d.f. g_Y as long as we multiply $h(X)$ by the importance weight.

Choosing the Importance Distribution

There is no correct choice for the importance distribution. It may be possible to find the distribution that allows the simulation to converge the fastest, but oftentimes, we don't need a perfect answer. Close to perfect is good enough.

We will solve the same problem as in Problem 1 using importance sampling. We will choose g_Y to be the normal distribution with $\mu = 4$ and $\sigma = 1$. We have chosen this distribution for g_Y because it will give us more points closer to and greater than 3. Note that it is not necessary to choose an importance distribution of the same type.

Figure 13.1: In our problem, we choose an importance distribution that will generate more samples that are greater than 3. Though not a perfect choice, choosing a normal distribution with $\mu = 4$ and $\sigma = 1$ will suffice.

```
>>> from scipy import stats
>>> h = lambda x : x > 3
>>> f = lambda x : stats.norm().pdf(x)
>>> g = lambda x : stats.norm(loc=4,scale=1).pdf(x)

# Sample from the N(4,1).
>>> N = 10**7
>>> X = np.random.normal(4,scale=1,size=N)

# Calculate estimate.
>>> 1./N * np.sum(h(X)*f(X)/g(X))
0.00134921134631
```

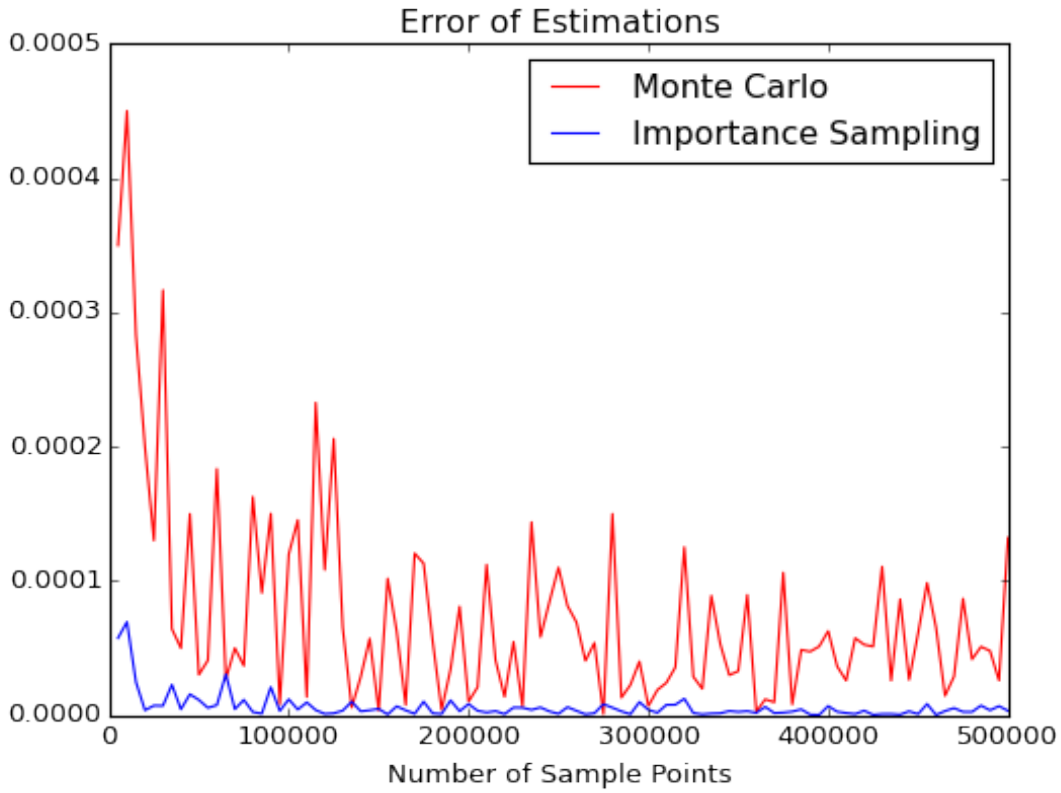


Figure 13.2: Comparison of error between standard method Monte Carlo and Importance Sampling method of Monte Carlo.

Problem 2. A tech support hotline receives an average of 2 calls per minute. What is the probability that they will have to wait at least 10 minutes to receive 9 calls? Implement your estimator using importance sampling. Calculate estimates using 5000, 10000, 15000, $\cdots$, 500000 sample points. Return an array of estimates. Your answers should approach 0.00208726.

Hint: In Volume 2 §3.5, the gamma distribution is defined as,

$$f_X(x) = \frac{b^a x^{a-1} e^{-xb}}{\Gamma(a)}.$$

The version of the gamma distribution in `scipy.stats` is determined by the shape (a) and the scale (θ) of the distribution.

$$f_X(x) = \frac{x^{a-1} e^{-x/\theta}}{\Gamma(a)\theta^a}$$

You can switch between these representations this with the fact that $\theta = 1/b$.

Problem 3. In this problem, we will visualize the benefits of importance sampling. Create a plot of the error of the traditional methods of Monte Carlo integration and the importance sampling methods of Monte Carlo for Problem 2. What do you observe? Your plot should resemble Figure 13.2.

Hint: The following code solves Problem 2 using traditional methods of Monte Carlo integration:

```
h = lambda x : x > 10
MC_estimates = []
for N in xrange(5000,505000,5000):
    X = np.random.gamma(9,scale=0.5,size=N)
    MC = 1./N*np.sum(h(X))
    MC_estimates.append(MC)
MC_estimates = np.array(MC_estimates)
```

Hint: To determine the error of your approximations, the following code returns the actual value of the probability:

```
1 - stats.gamma(a=9,scale=0.5).cdf(10)
```

Now that we have visualized the benefits of importance sampling, note that we can achieve the same results as traditional Monte Carlo with a fraction of the samples.

Generalizing the Principles of Importance Sampling

The examples we have explored to this point in the lab were merely educational. Since we have a simple means of calculating the correct answer to Problem 2, it doesn't make much sense to use methods of Monte Carlo in this situation. However, as discussed in the previous lab, there are not always closed-form solutions to the integrals we want to compute.

We can extend the same principles we have discussed thus far to solve many types of problems. For a more general problem, we can implement importance sampling by doing the following:

1. Define a function h where, $h(t) = \begin{cases} 1 & \text{if condition is met} \\ 0 & \text{otherwise} \end{cases}$.
2. Define a function f_X which is the p.d.f. of the target distribution.
3. Define a function g_Y which is the p.d.f. of the importance distribution.
4. Use these functions in conjunction with Equation (13.4).

Problem 4. The joint normal distribution of N independent random variables with mean 0 and variance 1 is

$$f_X(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^N}} e^{-(\mathbf{x}^T\mathbf{x})/2}.$$

The integral of $f_X(\mathbf{x})$ over a box is the probability that a draw from the distribution will be in the box. However, $f_X(\mathbf{x})$ does not have a symbolic antiderivative.

Use what you have learned about importance sampling to estimate the probability that a given random variable in $\mathbb{R}^2$ generated by f_X will be less than -1 in the x-direction and greater than 1 in the y-direction.

Treat f_X as the p.d.f. of your target distribution. Use the function `stats.multivariate_normal` to create a multivariate normal distribution to serve as your importance distribution. For more information on how to use this function, consult the documentation for `stats.multivariate_normal`.

Unnormalized Target Densities

The methods discussed so far are only applicable if the target density is normalized, or in other words, has an integral of 1. If the target density is not normalized, Equation 13.3 becomes

$$\begin{aligned} E[h(X)] &= \frac{\int h(t)f(t)\,dt}{\int f(t)\,dt} \\ &= \frac{\int h(t)f(t)\left(\frac{g_Y(t)}{g_Y(t)}\right)\,dt}{\int f(t)\left(\frac{g_Y(t)}{g_Y(t)}\right)\,dt} \\ &= \frac{\int \left(\frac{h(t)f(t)}{g_Y(t)}\right) g_Y(t)\,dt}{\int \left(\frac{f(t)}{g_Y(t)}\right) g_Y(t)\,dt} \\ &= \frac{E\left[\frac{h(Y)f(Y)}{g_Y(Y)}\right]}{E\left[\frac{f(Y)}{g_Y(Y)}\right]} \end{aligned}$$

The corresponding estimator becomes

$$\begin{aligned} \widehat{E}_n[h(X)] &= \frac{\widehat{E}\left[\frac{h(Y)f(Y)}{g_Y(Y)}\right]}{\widehat{E}\left[\frac{f(Y)}{g_Y(Y)}\right]} \\ &= \frac{\frac{1}{N}\sum_{i=1}^{N}\frac{h(y_i)f(y_i)}{g_Y(y_i)}}{\frac{1}{N}\sum_{i=1}^{N}\frac{f(y_i)}{g_Y(y_i)}} \end{aligned}$$

14 Visualizing Complex-valued Functions

Lab Objective: *Create visualizations of complex functions. Visually estimate their zeros and poles, and gain intuition about their behavior in the complex plane.*

Representations of Complex Numbers

A complex number $z = x + iy$ can be written in *polar coordinates* as $re^{i\theta}$ where

- $r = \sqrt{x^2 + y^2}$ is the magnitude of z, and
- $\theta = \arctan(y/x)$ is the angle between z and 0, as in Figure 14.1.

Conversely, Euler's formula implies $re^{i\theta} = r\cos(\theta) + ir\sin(\theta)$. Then if we set $re^{i\theta} = x + iy$ and equate real and imaginary parts, we find $x = r\cos(\theta)$ and $y = r\sin(\theta)$.

NumPy makes it easy to work with complex numbers and convert between coordinate systems. The function `np.angle()` returns the angle of a complex number (between $-\pi$ and π) and the function `np.absolute()` returns the magnitude. Use these to compute θ and r, respectively. These functions also operate elementwise on NumPy arrays.

Note that in Python, `1j` is used for the complex number $i = \sqrt{-1}$. See the code below for an example.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
# Set z = 2 - 2i
>>> z = 2 - 2*1j
>>> theta = np.angle(z)
>>> r = np.absolute(z)
# np.angle() returns a value between -pi and pi.
>>> print r, theta
(2.8284271247461903, -0.78539816339744828)
# Check that z=re^(i*theta)
>>> np.allclose(z, r*np.exp(1j*theta))
True
```

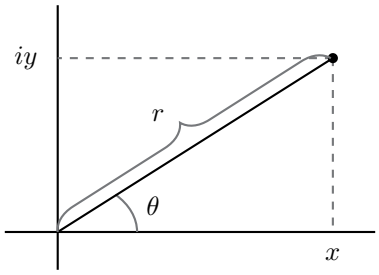


Figure 14.1: The complex number represented by the black dot equals both $x + iy$ and $re^{i\theta}$, when θ is written in radians.

Complex Functions

Suppose we wish to graph a function $f(z) : \mathbb{C} \to \mathbb{C}$. The difficulty is that $\mathbb{C}$ has 2 real dimensions, so the graph of f should use 4 real dimensions. Since we already have ways to visualize 3 dimensions, we should choose one dimension to ignore. We will ignore the magnitude $r = |f(z)|$ of the output.

To visualize f, we will assign a color to each point $z \in \mathbb{C}$. The color will correspond to the angle θ of the output $f(z)$. As an example, we have plotted the identity function $f(z) = z$ in Figure 14.2. As θ goes from 0 to 2π, the colors cycle smoothly counterclockwise from red to green to purple and back to red.

This kind of plot uses rectangular coordinates in the domain and polar coordinates (or rather, just the θ-coordinate) in the codomain. Note that this kind of plot tells us nothing about $|f(z)|$.

You can create the plot in Figure 14.2 as follows. Begin by creating a grid of complex numbers. We create the real and imaginary parts separately, and then use `np.meshgrid()` to turn them into a single array of complex numbers.

```
>>> x = np.linspace(-1, 1, 401)
>>> y = np.linspace(-1, 1, 401)
>>> X, Y = np.meshgrid(x, y)
>>> Z = X + 1j*Y
```

Now we compute the angles of the points in `Z` and plot them using `plt.pcolormesh()`. We use the colormap `'hsv'`, which is red at both ends, so that 0 and 2π will map to the same color.

```
>>> plt.pcolormesh(X, Y, np.angle(Z), cmap='hsv')
>>> plt.show()
```

Problem 1. Write the following function to plot any function from $\mathbb{C}$ to $\mathbb{C}$. Plot the angle only, as above, ignoring the magnitude.

```
def plot_complex(f, xbounds, ybounds, res=401):
    '''Plot the complex function f.

    INPUTS:
    f        - A function handle. Should represent a function
```

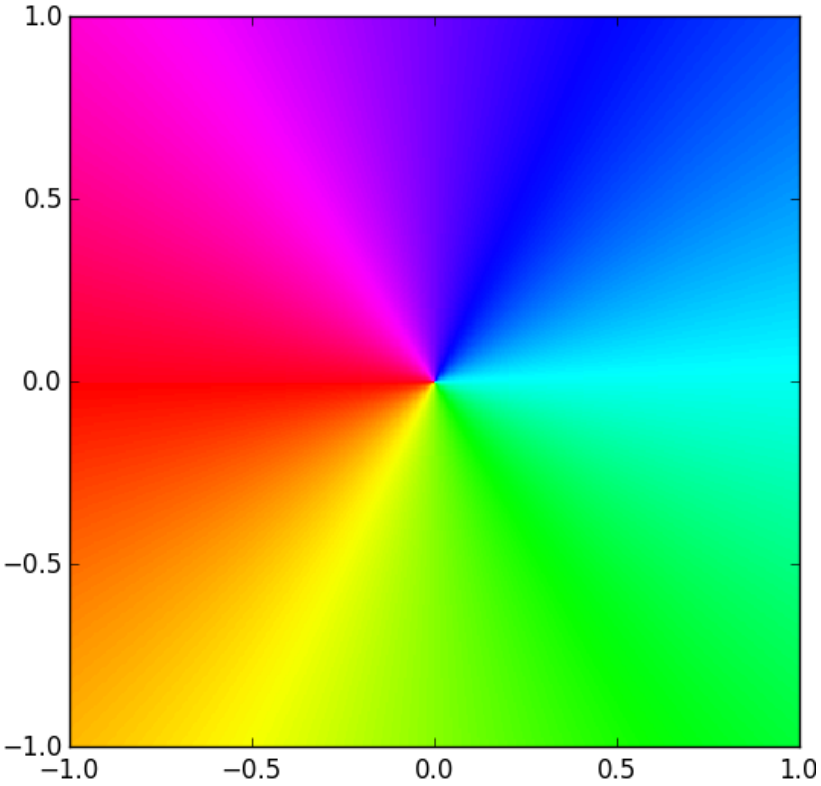


Figure 14.2: Plot of $f : \mathbb{C} \to \mathbb{C}$ defined by $f(z) = z$. The color at each point z represents the argument of $f(z)$.

```
            from C to C.
xbounds  - A tuple (xmin, xmax) describing the bounds on the real part
            of the domain.
ybounds  - A tuple (ymin, ymax) describing the bounds on the imaginary
            part of the domain.
res      - A scalar that determines the resolution of the plot.
            Defaults to 401.
'''
```

Check your function on $f(z) = z$ (graphed in Figure 14.2) and on the function $f(z) = \sqrt{z^2 + 1}$, which is graphed in Figure 14.3.

Hint: When you call `plt.pcolormesh()`, specify the keyword arguments `vmin` and `vmax`. These define which values should map to each end of the color scale. We want $-\pi$ to map to the low end of the color scale, and π to map to the high end. If not specified, matplotlib will scale the colormap to fit your data exactly.

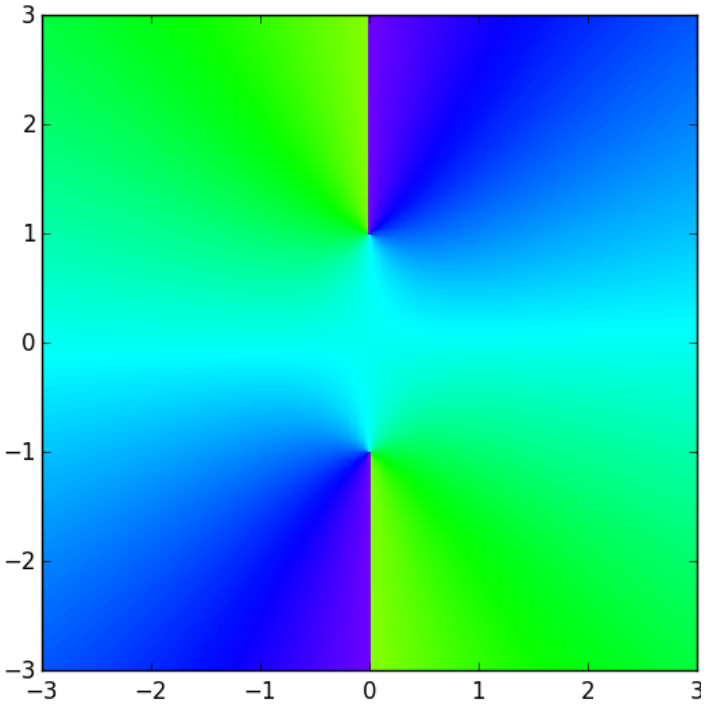


Figure 14.3: Plot of the angle of $\sqrt{z^2+1}$ on the domain $\{x+iy \mid x \in [-3,3],\ y \in [-3,3]\}$ created by `plot_complex()`.

The choice to ignore the magnitude may seem arbitrary. We can also write a complex plotting function to ignore the angle and only plot the magnitude. This will give us some different intuition about the function, while losing some information that we would get from the angle plot.

Problem 2. Write a new complex plotting function called `plot_complex_magnitude` which ignores the angle and plots only the magnitude. This should resemble your answer to Problem 1, with small modifications. Leave `vmin` and `vmax` unspecified when plotting.

Check your function on $f(z) = \sqrt{z^2+1}$. Your plot should look like the right subplot in Figure 14.4. Note the difference between this plot and the one from the previous problem.

Hint: A wraparound colormap like `'hsv'` doesn't work well here. Use any sequential colormap that makes it easy to distinguish between high and low values. See the matplotlib documentation for a list of colormaps.

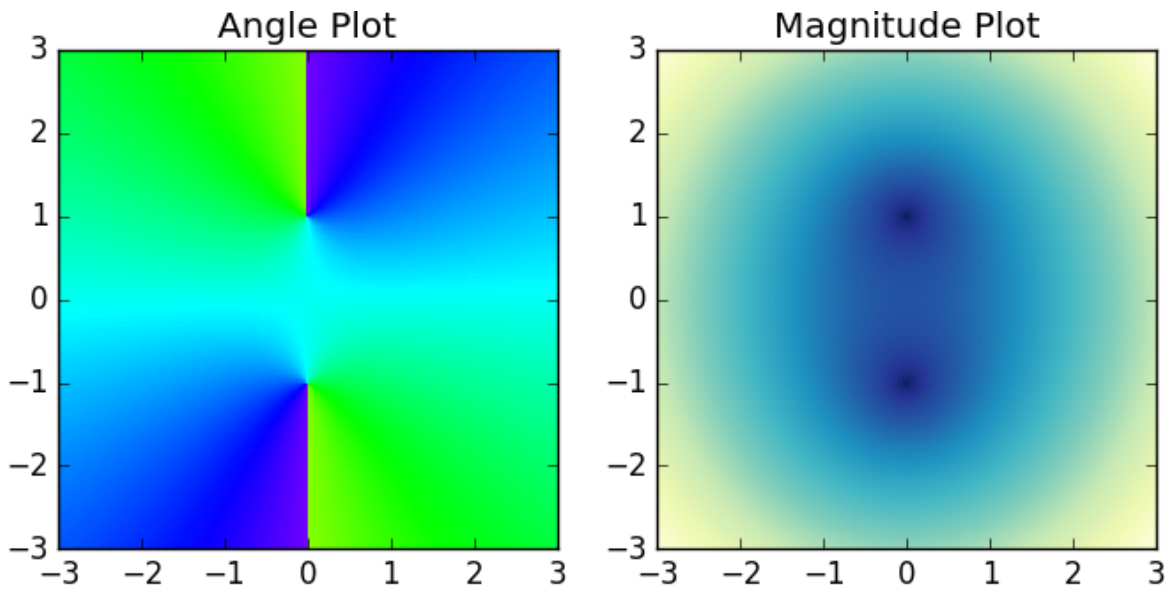


Figure 14.4: Plots of $\sqrt{z^2+1}$ on $\{x+iy \mid x \in [-3,3],\ y \in [-3,3]\}$, visualizing the angle and the magnitude of the function. Notice how a discontinuity is clearly visible on the left, but disappears from the plot on the right.

Analyzing Complex Plots

The angle plot is generally more useful than the magnitude plot for visualizing function behavior, zeros, and poles. Throughout the rest of the lab, use `plot_complex` to plot only the angle, and ignore the magnitude.

Zeros

Complex plots can be surprisingly informative. From an angle plot we can estimate not only a function's zeros, but also their multiplicities.

Problem 3.

1. Use `plot_complex()` to plot the functions z^2, z^3, and z^4.
2. Plot $z^3 - iz^4 - 3z^6$ on the domain $\{x+iy \mid x \in [-1,1],\ y \in [-1,1]\}$ (this plot is Figure 14.5). Compare it to your plot of z^3, especially near the origin. Based on these plots, what can you learn about the zeros of a function from its graph?

In Problem 3 you should have noticed that in a plot z^n, the colors cycle n times counterclockwise around 0. (Note: For the remainder of this lab we will define red $\to$ yellow $\to$ green $\to$ blue $\to$ red to be the "forward" direction, such that the colors are circling counterclockwise in Figure 14.2.)

This is explained by looking at z^n in polar coordinates:

$$z^n = (re^{i\theta})^n = r^n e^{i(n\theta)}.$$

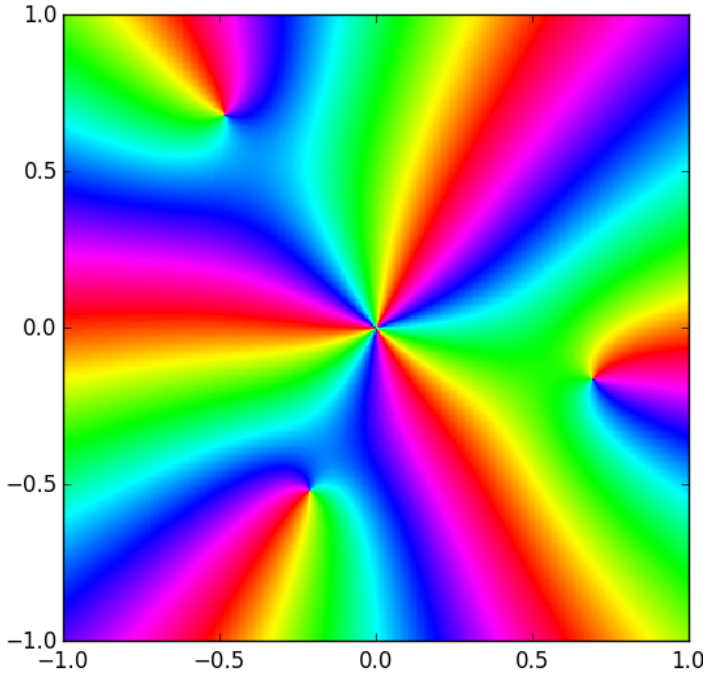


Figure 14.5: Plot of $f(z) = z^3 - iz^4 - 3z^6$ on the domain $\{x + iy \mid x \in [-1, 1],\ y \in [-1, 1]\}$. From this plot we see that $f(z)$ has a zero of order 3 at the origin, and 3 zeros of order 1 scattered around it. This accounts for the 6 roots of $f(z)$ that are guaranteed to exist by the Fundamental Theorem of Algebra.

Multiplying θ by a number greater than 1 compresses the graph along the "θ-axis" by a factor of n. In other words, the output angle repeats itself n times in one cycle of θ. Compare this to replacing $f(x)$ with $f(nx)$ when f is a function from $\mathbb{R}$ to $\mathbb{R}$.

From Problem 3 you should also have noticed that the plot of $z^3 - iz^4 - 3z^6$ looks a lot like the plot of z^3 near the origin. This is because when z is very small, z^4 and z^6 are much smaller than z^3, and so the behavior of z^3 dominates the function.

In general, $f(z)$ has a *zero of order n at z_0* if the Taylor series of $f(z)$ centered at z_0 can be written as

$$f(z) = \sum_{k=n}^{\infty} a_k (z - z_0)^k \qquad \text{with } a_n \neq 0.$$

In other words, $f(z) = a_n(z - z_0)^n + a_{n+1}(z - z_0)^{n+1} + \ldots$. In a small neighborhood of z_0, the quantity $|z - z_0|^{n+k}$ is much smaller than $|z - z_0|^n$, and so the function behaves like $a_n(z - z_0)^n$. This explains why we can estimate the order of a zero by counting the number of times the colors circle a point (see Figure 14.5).

Poles

The plots created by `plot_complex()` also contain information about the poles of the function plotted.

Problem 4.

1. Use `plot_complex()` to plot the function $f(z) = 1/z$. Compare this to the plot of $f(z) = z$ in Figure 14.2.
2. Plot z^{-2}, z^{-3}, and $z^2 + iz^{-1} + z^{-3}$ on the domain $\{x + iy \mid x \in [-1, 1],\ y \in [-1, 1]\}$. Compare the plots of the last two functions near the origin. Based on these plots, what can you learn about the poles of a function from its graph?

In Problem 4 you should have noticed that in the graph of $1/z^n$, the colors cycle n times *clockwise* around 0. Again this can be explained by looking at the polar representation:

$$z^{-n} = (re^{i\theta})^{-n} = r^{-n}e^{i(-n\theta)}.$$

The minus-sign on the θ reverses the direction of the colors, and the n makes them cycle n times.

In general, a function has a *pole of order* n at z_0 if its Laurent series on a punctured neighborhood of z_0 is

$$f(z) = \sum_{k=-n}^{\infty} a_k(z - z_0)^k \qquad \text{with } a_{-n} \neq 0.$$

In other words, $f(z) = a_{-n}(z - z_0)^{-n} + a_{-n+1}(z - z_0)^{-n+1} + \ldots$. Since $|z - z_0|^{-n+k}$ is much smaller than $|z - z_0|^{-n}$ when $|z - z_0|$ is small, near z_0 the function behaves like $a_{-n}(z - z_0)^{-n}$. This explains why we can estimate the order of a pole by counting the number of times the colors circle a point in the clockwise direction.

Finally, a function has an *essential pole* at z_0 if its Laurent series in a punctured neighborhood of z_0 requires infinitely many terms with negative exponents. For example,

$$e^{1/z} = \sum_{k=0}^{\infty} \frac{1}{n!z^n} = 1 + \frac{1}{z} + \frac{1}{2}\frac{1}{z^2} + \frac{1}{6}\frac{1}{z^3} + \ldots.$$

The plot of $f(z) = e^{1/z}$ is in Figure 14.6. The colors cycle infinitely many times around an essential singularity.

Using Plots to Estimate Poles and Zeros

To summarize, poles and zeros can be estimated from a complex plot with the following rules.

- Colors circle counterclockwise around zeros.
- Colors circle clockwise around poles.
- The number of times the colors cycle equals the order of the zero or pole.

Problem 5. Plot these functions on the domains given. Estimate the number and order of their poles and zeros.

- $f(z) = e^z$ on $\{x + iy \mid x \in [-8, 8],\ y \in [-8, 8]\}$
- $f(z) = \tan(z)$ on $\{x + iy \mid x \in [-8, 8],\ y \in [-8, 8]\}$

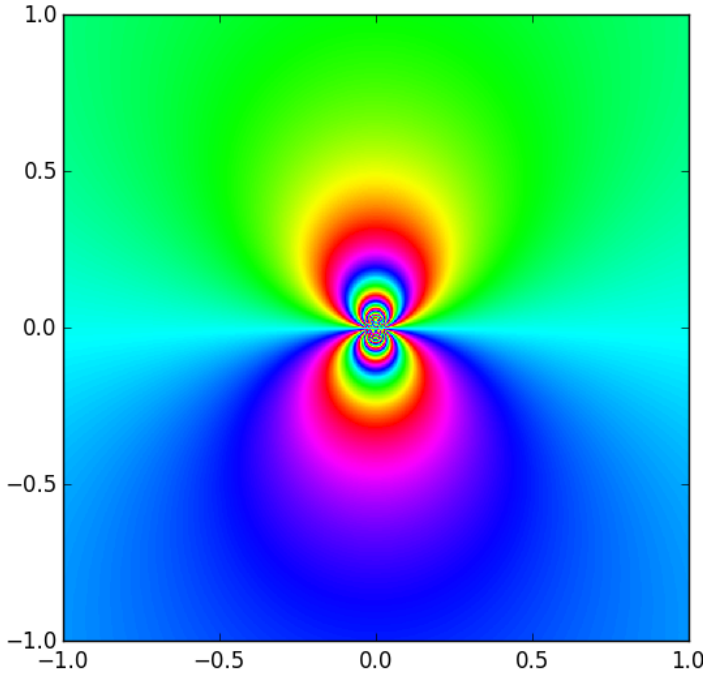


Figure 14.6: Plot of $e^{1/z}$ on the domain $\{x+iy \mid x \in [-1,1],\ y \in [-1,1]\}$. The colors circle clockwise around the origin because it is a singularity, not a zero. Because the singularity is essential, the colors repeat infinitely many times.

- $f(z) = \frac{16z^4+32z^3+32z^2+16z+4}{16z^4-16z^3+5z^2}$ on $\{x+iy \mid x \in [-1,1],\ y \in [-1,1]\}$

One useful application of complex plots is to estimate the zeros of polynomials and their multiplicity.

Problem 6. Use complex plots to determine the multiplicity of the zeros of each of the following polynomials. Use the Fundamental Theorem of Algebra to ensure that you have found them all.

1. $-4z^5 + 2z^4 - 2z^3 - 4z^2 + 4z - 4$
2. $z^7 + 6z^6 - 131z^5 - 419z^4 + 4906z^3 - 131z^2 - 420z + 4900$

Plotting functions is not a substitute for rigorous mathematics. Often, plots can be deceptive.

Problem 7.

1. This example shows that sometimes you have to "zoom in" to see all the information about a pole.
   (a) Plot the function $f(z) = \sin(\frac{1}{100z})$ on the domain $\{x+iy \mid x \in [-1,1],\ y \in [-1,1]\}$. What might you conclude about this function?

(b) Now plot $f(z)$ on $\{x+iy \mid x \in [-.01,.01],\ y \in [-.01,.01]\}$. Now what do you conclude about the function?

2. This example shows that from far away, two distinct zeros (or poles) can appear to be a single zero (or pole) of higher order.

   (a) Plot the function $f(z) = z + 1000z^2$ on the domain $\{x+iy \mid x \in [-1,1],\ y \in [-1,1]\}$. What does this plot imply about the zeros of this function?

   (b) Calculate the true zeros of $f(z)$.

   (c) Plot $f(z)$ on a domain that allows you to see the true nature of its zeros.

Multi-Valued Functions

Every complex number has two complex square roots, since if $w^2 = z$, then also $(-w)^2 = z$. If z is not zero, these roots are distinct.

Over the nonnegative real numbers, it is possible to define a continuous square root function. However, it is not possible to define a continuous square root function over any open set of the complex numbers that contains 0. This is intuitive after graphing $\sqrt{z}$ on the complex plane.

Problem 8. 1. Use `plot_complex` to graph $f(z) = \sqrt{z}$. Use `np.sqrt()` to take the square root.

2. Now plot $f(z) = -\sqrt{z}$ to see the "other square root" of z. Describe why these two plots look the way they do.

Just as raising z to a positive integer "compresses the θ-axis", making the color wheel repeat itself n times around 0, raising z to a negative power *stretches* the θ-axis, so that only one n^{th} of the color wheel appears around 0. The colors at the ends of this n^{th}-slice are not the same, but they appear next to each other in the plot of z^{-n}. This discontinuity will appear in every neighborhood of the origin.

If your domain does not contain the origin, it is possible to define a continuous root function by picking one of the roots.

Appendix

It is possible to visualize the argument and the modulus of the output of a complex function $f(z)$. One way to do so is to assign the modulus to a *lightness* of color. For example, suppose we have a complex number with argument 0, so it will map to red in the color plots described above. If its modulus is very small, then we can map it to a blackish red, and if its modulus is large, we can map it to a whitish red. With this extra rule, our complex plots will still be very much the same, except that zeros will look like black dots and poles will look like white dots (see Figure 14.7 for an example).

The code below implements the map we just described. Be warned that this implementation does not scale well. For example, if you try to plot a complex function whose outputs are all very small in modulus, the entire plot will appear black.

```
import numpy as np
import matplotlib.pyplot as plt
from colorsys import hls_to_rgb

def colorize(z):
    '''
    Map a complex number to a color (or hue) and lightness.

    INPUT:
    z - an array of complex numbers in rectangular coordinates

    OUTPUT:
    If z is an n x m array, return an n x m x 3 array whose third axis encodes
    (hue, lightness, saturation) tuples for each entry in z. This new array can
    be plotted by plt.imshow().
    '''

    zy=np.flipud(z)
    r = np.abs(zy)
    arg = np.angle(zy)

    # Define hue (h), lightness (l), and saturation (s)
    # Saturation is constant in our visualizations
    h = (arg + np.pi)  / (2 * np.pi) + 0.5
    l = 1.0 - 1.0/(1.0 + r**0.3)
    s = 0.8

    # Convert the HLS values to RGB values.
    # This operation returns a tuple of shape (3,n,m).
    c = np.vectorize(hls_to_rgb) (h,l,s)

    # Convert c to an array and change the shape to (n,m,3)
    c = np.array(c)
    c = c.swapaxes(0,2)
    c = c.swapaxes(0,1)
    return c
```

The following code uses the `colorize()` function to plot $\frac{z^2-1}{z}$. The output is Figure 14.7.

```
>>> f = lambda z :  (z**2-1)/z
>>> x = np.linspace(-.5, 1.5, 401)
>>> y = np.linspace(-1, 1, 401)
>>> X,Y = np.meshgrid(x,y)
>>> Z=f(X+Y*1j)
>>> Zc=colorize(Z)
>>> plt.imshow(Zc, extent=(-.5, 1.5, -1, 1))
>>> plt.show()
```

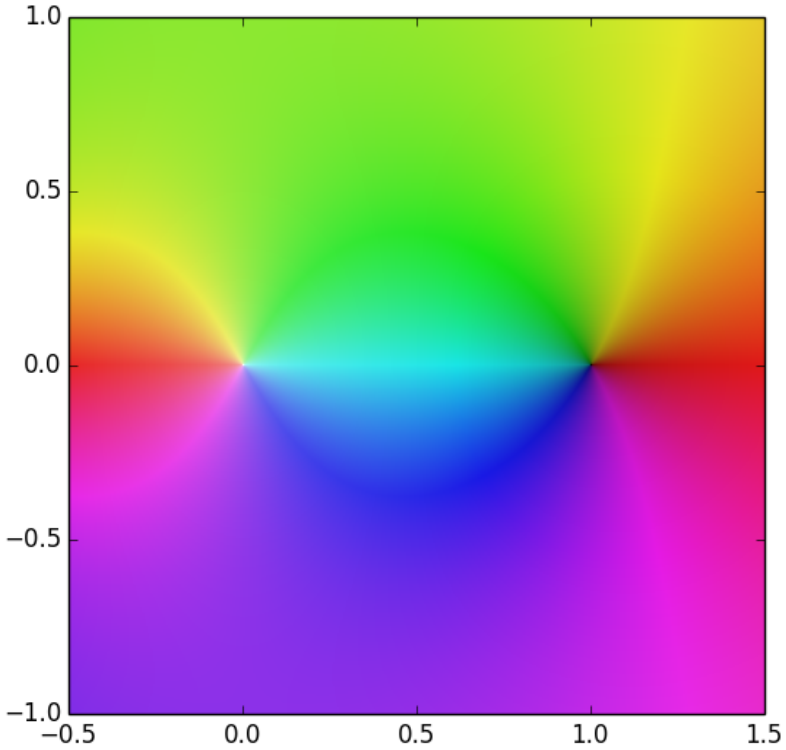


Figure 14.7: Plot of the function $\frac{z^2-1}{z}$ created with `colorize()`. Notice that the zero at 1 is a black dot and the pole at 0 is a white dot.

15 The PageRank Algorithm

Lab Objective: *Model a network as a graph and implement the PageRank algorithm based on this model. Use PageRank to predict the rankings of sports teams.*

As of 2013, the PageRank algorithm is one of over 200 algorithms that Google uses to determine the *rank*, or relative importance, of a webpage. Named for Larry Page, cofounder of Google, this algorithm ranks pages based on how many other pages link to them.

The Internet as a Graph

The PageRank algorithm models the internet with a directed graph. Each webpage is a node, and there is an edge from node i to node j if page i links to page j. Let $\text{In}(i)$ be the websites linking to page i and let $\text{Out}(i)$ be the websites that page i links to. That is, $\text{In}(i)$ is the set of nodes with an arrow to node i, and $\text{Out}(i)$ is the set of nodes with an arrow from node i. An example is illustrated in Figure 15.1.

The PageRank algorithm ranks pages by how many others link to them. A link from a more important page counts more than one from a less important page. For example, in Figure 15.1 we would expect node 0 to have a very high rank because every other node links to it. Consequently, we would expect node 7 to have a fairly high rank because node 0 links to it, even though node 0 is the only node to do so.

The PageRank Algorithm

The PageRank algorithm assumes that a surfer chooses a starting webpage randomly. Then, if the surfer is at page i, they randomly select a page from $\text{Out}(i)$ to visit next. This means that the surfer's chance of being on page i at time t is determined by where they were at time $t-1$.

Suppose the internet has N webpages, and let $p_i(t)$ be the likelihood that the surfer is on page i at time t. Then the probabilities $p_i(t)$ are given by

$$p_i(0) = \frac{1}{N} \qquad p_i(t+1) = \sum_{j \in \text{In}(i)} \frac{p_j(t)}{|\text{Out}(j)|}. \tag{15.1}$$

For example, in Figure 15.1 we have $N = 8$, and

$$p_6(t+1) = \frac{p_3(t)}{3} + \frac{p_4(t)}{3} + \frac{p_5(t)}{2}.$$

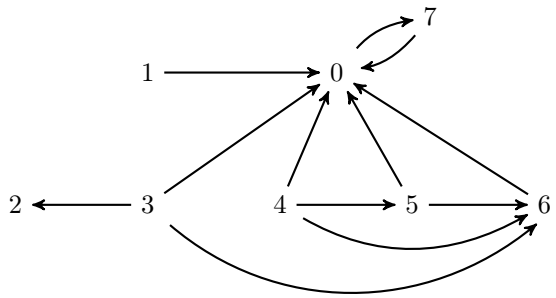


Figure 15.1: This directed graph describes the links between 8 webpages. In this example, $\text{In}(0) = \{1, 3, 4, 5, 6, 7\}$ and $\text{Out}(0) = \{7\}$.

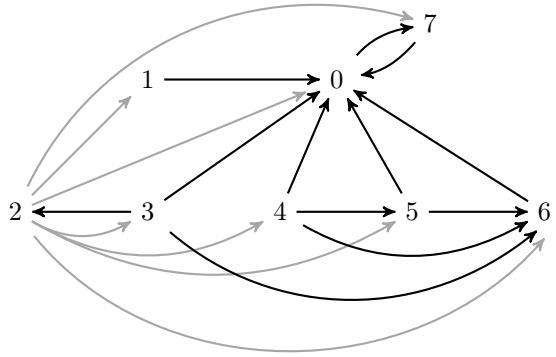


Figure 15.2: Here Figure 15.1 has been modified to guarantee that page 2 is no longer a sink. A new link has been added from page 2 to every other page (the added links are grey).

Refining the Model: Pages with No Outbound Links

A node with no outbound links, such as node 2 in Figure 15.1, is called a *sink*. According to our model, if the surfer ever visits a sink, they will stay there forever.

This is not very realistic; in this situation, a person would likely select another webpage at random and begin surfing again. Hence, in our model we replace sinks with nodes linking to every other page. This means we modify Figure 15.1 (where node 2 is a sink) to look like Figure 15.2.

Refining the Model: Adding Boredom

The equations in (15.1) assume that the current page must link to the next page. However, the model is more realistic if we assume that the surfer sometimes gets bored and randomly picks a new starting page. We will denote the probability that a surfer stays interested at step t by a constant d, called the *damping factor*. Then the probability that the surfer gets bored at time t is $1 - d$. The formulas in (15.1) then become

$$p_i(0) = \frac{1}{N} \qquad p_i(t+1) = d \sum_{j \in \text{In}(i)} \frac{p_j(t)}{|\text{Out}(j)|} + \frac{1-d}{N}. \tag{15.2}$$

Matrix Form of the PageRank Algorithm

We can rewrite (15.2) as the matrix equation

$$\mathbf{p}(0) = \frac{1}{N}\mathbf{1} \qquad \mathbf{p}(t+1) = dK\mathbf{p}(t) + \frac{1-d}{N}\mathbf{1} \tag{15.3}$$

where $\mathbf{p}(t) = (p_1(t), p_2(t), \ldots, p_N(t))^T$, $\mathbf{1}$ is a vector of N ones, and K is defined by

$$K_{ij} = \begin{cases} \frac{1}{|\mathrm{Out}(j)|} & \text{if j links to i} \\ 0 & \text{otherwise.} \end{cases}$$

Defining Page Rank

As given by the PageRank algorithm, the *rank* of page i is

$$p_i = \lim_{t\to\infty} p_i(t).$$

For those familiar with Markov Chains, Equation 15.3 defines a Markov chain. Page ranks are simply the steady state of this Markov chain.

Implementation in Python

The adjacency matrix A of a directed graph has $A_{ij} = 1$ if there is an edge from node i to node j, and $A_{ij} = 0$ otherwise. The adjacency matrix of the graph in Figure 15.1 is defined below. We use a code environment to describe A so you can easily use this example to debug the problems in this lab.

```
A = np.array([[ 0,  0,  0,  0,  0,  0,  0,  1],
              [ 1,  0,  0,  0,  0,  0,  0,  0],
              [ 0,  0,  0,  0,  0,  0,  0,  0],
              [ 1,  0,  1,  0,  0,  0,  1,  0],
              [ 1,  0,  0,  0,  0,  1,  1,  0],
              [ 1,  0,  0,  0,  0,  0,  1,  0],
              [ 1,  0,  0,  0,  0,  0,  0,  0],
              [ 1,  0,  0,  0,  0,  0,  0,  0]])
```

Problem 1. Write a function that creates an adjacency matrix from a file. The function should accept a filename, and an integer N that represents the number of nodes in the graph described by the datafile. Return the adjacency matrix as a `SciPy` sparse `dok_matrix`.
Hints:

1. The file `matrix.txt` included with this lab describes the matrix in Figure 15.1 and has the adjacency matrix `A` given above. You may use it to test your function.
2. You can open a file in Python using the `with` syntax. Then, you can iterate through the lines using a `for` loop. Here is an example.

```
# Open `matrix.txt' for read-only
with open('./matrix.txt', 'r') as myfile:
```

```
    for line in myfile:
        print line
```

3. Here is an example of how to process a line of the form in `datafile`.

```
>>> line = '0\t4\n'
# strip() removes trailing whitespace from a line.
# split() returns a list of the space-separated pieces of the line.
>>> line.strip().split()
['0', '4']
```

4. Rather than testing for lines of `matrix.txt` that contain comments, put all your string operations in a `try` block with an `except` block following.

Note

It makes sense to initialize A as a sparse matrix, since A is mostly zeros. To make the coding easier, throughout the rest of the lab the algorithms will be coded using non-sparse matrices. This means when using an adjacency matrix created in Problem 1, cast it `todense()` when inputing it to test the other functions. In other words, you may say `test_calculateK(A.todense(), N)`. Don't forget, however, that in a real-world sparse adjacency matrices are generally much more time-efficient than dense matrices.

The next step is to compute K from (15.3). A good strategy for computing K comes from writing

$$K = (D^{-1}A)^T$$

where A is the adjacency matrix of the directed graph representing the internet and D is a diagonal matrix with $D_{jj} = |\text{Out}(j)|$. Modify A so that rows corresponding to sinks have all ones instead of all zeros. For Figure 15.2, the modified adjacency matrix is defined below.

```
Am = np.array([[ 0,  0,  0,  0,  0,  0,  0,  1],
               [ 1,  0,  0,  0,  0,  0,  0,  0],
               [ 1,  1,  1,  1,  1,  1,  1,  1],
               [ 1,  0,  1,  0,  0,  0,  1,  0],
               [ 1,  0,  0,  0,  0,  1,  1,  0],
               [ 1,  0,  0,  0,  0,  0,  1,  0],
               [ 1,  0,  0,  0,  0,  0,  0,  0],
               [ 1,  0,  0,  0,  0,  0,  0,  0]])
```

The matrix D is easily obtained by summing the rows of A. Although $K = (D^{-1}A)^T$, it is better practice to only store the diagonal entries of D as a vector, and then use array broadcasting to divide A by D.

Notice that we need to transpose $D^{-1}A$ to get K. This is because $D^{-1}A$ is *row stochastic* (meaning that the rows sum to 1), but we need to multiply *column stochastic* matrices (where the columns sum to 1) To make K be column stochastic, we have to take a transpose.

For Figure 15.2, the matrix K is as follows.

```
K = np.array([[ 0   ,  1   ,  1./8,  1./3,  1./3,  1./2,  1   ,  1   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  1./3,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  0   ,  1./3,  0   ,  0   ,  0   ],
              [ 0   ,  0   ,  1./8,  1./3,  1./3,  1./2,  0   ,  0   ],
              [ 1   ,  0   ,  1./8,  0   ,  0   ,  0   ,  0   ,  0   ]])
```

Problem 2. Write a function that computes and returns the K matrix given an adjacency matrix.

1. Compute the diagonal matrix D.
2. Compute the modified adjacency matrix where the rows corresponding to sinks all have ones instead of zeros.
3. Compute K using array broadcasting.

Solving for the Page Ranks

There are several ways to solve for $\lim_{t\to\infty}\mathbf{p}(t)$.

Algebraic Method

Again, for those familiar with Markov chains, one possibility is to assume the modified Markov chain has a steady state $\mathbf{p}$ and solve for it algebraically:

$$(I - dK)\mathbf{p} = \frac{1-d}{N}\mathbf{1}. \tag{15.4}$$

We can use SciPy's solver to find the page ranks of the network in Figure 15.2.

```
>>> from scipy import linalg as la
>>> I = np.eye(8)
>>> d = .85
>>> la.solve(I-d*K, ((1-d)/8)*np.ones(8))
array([ 0.43869288,  0.02171029,  0.02786154,  0.02171029,  0.02171029,
        0.02786154,  0.04585394,  0.39459924])
```

As expected, node 0 has the highest rank, approximately equal to .44. Node 7 has a higher rank than node 6, even though $\text{In}(7) = 1$ and $\text{In}(6) = 3$. This is because node 7's single in-edge comes from a node that has a very high rank (node 0).

Iterative Method

Solving the system in (15.4) is feasible for our small working example, but this is not an efficient strategy for very large systems.

One option for large systems is an iterative method. Starting with a guess for $\mathbf{p}(0)$, we iterate on Equation (15.3) until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|$ is sufficiently small. At this point we assume we have reached the steady state.

Problem 3. Implement a function that uses the iterative method to find the steady state of the PageRank algorithm. You function should accept an adjacency matrix `A`, an integer `N` that defaults to `None`, the damping factor `d` that defaults to `0.85`, and a tolerance `tol` that defaults to `1E-5`. Return the approximation to the steady state as a float. When the argument `N` is not `None`, work with only the upper $N \times N$ portion of the array `adj`. Test your function against the example datafile that accompanies this lab.
Hints:

1. Try making your initial guess for $\mathbf{p}(0)$ a random vector.
2. NumPy can do unexpected things with the dimensions when performing matrix-vector multiplication. When debugging, check at each iteration that all arrays have the dimensions you expect.

Eigenvalue Method

Another way to solve this problem is to make it into an eigenvalue problem. Let E be an $N \times N$ matrix of ones; then $E\mathbf{p}(t) = \mathbf{1}$. Hence, the matrix equation (15.3) for $\mathbf{p}(t+1)$ becomes

$$\mathbf{p}(t+1) = \Big(dK + \frac{1-d}{N}E\Big)\mathbf{p}(t).$$

If we write $B = dK + \frac{1-d}{N}E$, this simplifies to $\mathbf{p}(t+1) = B\mathbf{p}(t)$. Thus, the steady state $\mathbf{p}(t)$ is an eigenvector of B corresponding to the eigenvalue 1.

The columns of B sum to 1, and the entries of B are strictly positive (because the entries of E are all positive). With these hypotheses, the Perron-Frobenius theorem says that 1 is the unique eigenvalue of B of largest magnitude, and the corresponding eigenvector is unique. In this case, the "iterative method" described above is just the power method for finding the eigenvector corresponding to a dominant eigenvalue, introduced in the lab on eigensolvers.

We can also compute $\mathbf{p}$ using eigenvalue solvers in SciPy.

Problem 4. Implement a function that uses the eigenvalue method to find the steady state of the PageRank algorithm. Your function should accept an adjacency matrix `A`, an integer `N` that defaults to `None`, and the damping factor `d` that defaults to `0.85`. Return the approximation to the steady state as a float.

Application: Ranking Sports Teams

This ranking algorithm can be applied not only to webpages, but to any problem with a directed graph structure. One such application is ranking sports teams.

Suppose we have data about a collection of sports teams, including which teams played each other and who won each match. We can model this as a directed graph. Each node in the graph represents a team. An edge between two nodes points from the losing team to the winning team. If two teams never played each other, there is no edge between them. Wins and losses do not cancel out; if BYU and Boise played twice, and each team won once, then there is an edge from BYU to Boise and another edge from Boise to BYU.

To simplify our model, edges are not weighted. So if Duke ever beat Harvard, no matter whether they beat them once or 5 times, there is only one edge pointing from Harvard to Duke.

The key here is that edges tend to lead from worse teams to better teams. So by starting with some team and randomly following edges, we should end up visiting better teams more often. This is reminiscent of the PageRank algorithm! Given an appropriate dataset, we can use PageRank to estimate team rankings.

Note that in this scenario, the parameter d no longer represents boredom. It allows us to jump randomly from one team to another, so it could represent a surprise upset, or the random outcome of a game between two teams who have never played each other.

Problem 5. By applying the PageRank algorithm to win-loss data from the 2013 NCAA basketball season, produce a comparative ranking of the teams.

1. The file `ncaa2013.csv` contains data on over 5000 basketball games. The first line is a header. After the header, each line represents a game and has the winning team followed by the losing team (there are no ties in basketball).

   Load this file and use it to create the adjacency matrix A, where $A_{ij} = 1$ if team j beat team i. Make sure to ignore the header line. You will need some way of mapping from team names to the integers and vice versa.

2. Use the iterative method from Problem 3 with $d = 0.7$ to find the steady state. The steady-state solution is your vector of ranks.

3. Return the ranks sorted from largest to smallest, and the corresponding list of teams sorted from "best" to "worst".

Hints:

1. The code below may be helpful for processing the .csv file:

```
>>> with open('./ncaa2013.csv', 'r') as ncaafile:
>>>     ncaafile.readline() #reads and ignores the header line
>>>     for line in ncaafile:
>>>         teams = line.strip().split(',') #split on commas
>>>         print teams
>>> ['Middle Tenn St', 'Alabama St']
>>> ...
>>> ['Mississippi', 'Florida']
```

2. Before creating the adjacency matrix, you can get all the unique teams by running through all the matches once and adding every team to a set. Next, count the number of unique teams and initialize A to be the right size. Try using dictionaries, lists, or both to map numbers to teams and teams to numbers and fill in A. There is more than one right way to do this.
3. The function `np.argsort()` will be useful for sorting the ranks and teams.
4. There should be 347 teams. PageRank should predict that the top five ranked teams are Duke, Butler, Louisville, Illinois, and Indiana (in that order). Use this to check your results.

NetworkX: Python package for networks

The purpose of this section is not to give an introduction to NetworkX, but rather, to introduce you to just enough to be able to analyze the basic properties of a network and apply NetworkX's implementation of PageRank. NetworkX takes advantage of the sparse nature of these networks. Therefore, they are more efficient than the ones we have coded in this lab.

We will first run through the simple steps to initialize the graph defined in Figure 15.1. We will initialize this graph using the edges defined in `matrix.txt`. If we create an $n \times 2$ matrix of the edges of this graph, we would get,

```
>>> edges = array([[ 0,  7],
...                [ 1,  0],
...                [ 3,  0],
...                [ 3,  6],
...                [ 4,  0],
...                [ 4,  5],
...                [ 4,  6],
...                [ 5,  0],
...                [ 5,  6],
...                [ 6,  0],
...                [ 7,  0]])
```

.

We can now initialize a NetworkX graph using this array of edges.

```
>>> import networkx as nx
>>> G = nx.from_edgelist(edges, create_using=nx.DiGraph())
```

Now that we have initialized the `DiGraph` object, we can use all the analysis tools that come with NetworkX to gain further insight into the structure of the graph. Verify the following characteristics match the graph in Figure 15.1.

```
>>> G.in_degree()
{0: 6, 1: 0, 3: 0, 4: 0, 5: 1, 6: 3, 7: 1}

>>> G.out_degree()
{0: 1, 1: 1, 3: 2, 4: 3, 5: 2, 6: 1, 7: 1}
```

```
>>> G.in_edges(0)
[(1, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0)]

>>> G.out_edges(0)
[(0, 7)]
```

NetworkX also comes with a `pagerank()` function that can be used simply by passing the function your `DiGraph` object. Compare the results to the values calculated using our methods.

```
>>> nx.pagerank(G, alpha=0.85) # alpha is the dampening factor.
{0: 0.45323691210120065,
 1: 0.021428571428571432,
 3: 0.021428571428571432,
 4: 0.021428571428571432,
 5: 0.027500000000000004,
 6: 0.04829464285714287,
 7: 0.406682730755942}
```

Application: Twitter Datasets

The SNAP graph library, located at `http://snap.stanford.edu/data/index.html`, provides a variety of medium sized data sets for public use. These datasets have to do with networks, including road systems, social networks, and online communities. There are some interesting resources here for those wanting to experiment further with the PageRank algorithm on different datasets.

Problem 6. The `twitter_combined.txt` file contains a list of edges that represent a Twitter network. To protect the privacy of users, the data has been anonumized. Each number is an ID for a user. The users in the first column represent Twitter users that follow the users in the second column.

Using these edges,

1. Create a `DiGraph` object using all the edges described in `twitter_combined.txt`.
2. Calcualte the page ranks for this graph. The page ranks create a ranking of which users are the most “influential". Even though the results will just be numbers, remember that they represent actual Twitter users.
3. Analyze the in-degree and out-degree of the top 10 ranked users. What do you notice about the second-highest ranked user? Why would this user be ranked so high? HINT: Use `G.in_edges()` and `G.out_edges()` to gain further insight into this result.
4. Return the top 10 most influential users and their scores

16 Iterative Solvers

Lab Objective: *Many real-world problems of the form $A\mathbf{x} = \mathbf{b}$ have tens of thousands of parameters. Solving such systems with Gaussian elimination or matrix factorizations could require trillions of floating point operations (FLOPs), which is of course infeasible. Solutions of large systems must therefore be approximated iteratively. In this lab, we implement three popular iterative methods for solving large systems: Jacobi, Gauss-Seidel, and Successive Over-Relaxation.*

Iterative Methods

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. Throughout this lab, we use the notation $\mathbf{x}^{(k)}$ to denote the kth approximation for the solution vector $\mathbf{x}$ and $x_i^{(k)}$ to denote the i^{th} component of $\mathbf{x}^{(k)}$. With this notation, every iterative method can be summarized as

$$\mathbf{x}^{(k+1)} = f(\mathbf{x}^{(k)}), \tag{16.1}$$

where f is some function used to approximate the true solution $\mathbf{x}$.

In the best case, the iteration converges to the true solution ($\mathbf{x}^{(k)} \to \mathbf{x}$). In the worst case, the iteration continues forever without approaching the solution. Iterative methods therefore require carefully chosen *stopping criteria* to prevent iterating forever. The general approach is to continue until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. More precisely, choose a very small $\epsilon > 0$ and an integer $N \in \mathbb{N}$, and update the approximation using Equation 16.1 until either

$$\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| < \epsilon \qquad \text{or} \qquad k > N. \tag{16.2}$$

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 non-convergent iterations sooner than with a large N (10,000).

The Jacobi Method

The *Jacobi Method* is a simple but powerful method used for solving certain kinds of large linear systems. The main idea is simple: solve for each variable in terms of the others, then use the previous values to update each approximation. As a (very small) example, consider the following 3×3 system.

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3 \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3 \\ & + & x_2 & + & 3x_3 & = & -1 \end{array}$$

Solving the first equation for x_1, the second for x_2, and the third for x_3 yields the following.

$$\begin{array}{rcl} x_1 & = & \frac{1}{2}(3 + x_3) \\ x_2 & = & \frac{1}{3}(3 + x_1 - 2x_3) \\ x_3 & = & \frac{1}{3}(-1 - x_2) \end{array}$$

Now begin with an initial guess $\mathbf{x}^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}]^\mathsf{T} = [0, 0, 0]^\mathsf{T}$. To compute the first approximation $\mathbf{x}^{(1)}$, use the entries of $\mathbf{x}^{(0)}$ as the variables on the right side of the previous equation.

$$\begin{array}{rcccccc} x_1^{(1)} & = & \frac{1}{2}(3 + x_3^{(0)}) & = & \frac{1}{2}(3 + 0) & = & \frac{3}{2} \\ x_2^{(1)} & = & \frac{1}{3}(3 + x_1^{(0)} - 2x_3^{(0)}) & = & \frac{1}{3}(3 + 0 - 0) & = & 1 \\ x_3^{(1)} & = & \frac{1}{3}(-1 - x_2^{(0)}) & = & \frac{1}{3}(-1 - 0) & = & -\frac{1}{3} \end{array}$$

So $\mathbf{x}^{(1)} = [\frac{3}{2}, 1, -\frac{1}{3}]^\mathsf{T}$. Computing $\mathbf{x}^{(2)}$ is similar.

$$\begin{array}{rcccccc} x_1^{(2)} & = & \frac{1}{2}(3 + x_3^{(1)}) & = & \frac{1}{2}(3 - \frac{1}{3}) & = & \frac{4}{3} \\ x_2^{(2)} & = & \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(1)}) & = & \frac{1}{3}(3 + \frac{3}{2} + \frac{2}{3}) & = & \frac{31}{18} \\ x_3^{(2)} & = & \frac{1}{3}(-1 - x_2^{(1)}) & = & \frac{1}{3}(-1 - 1) & = & -\frac{2}{3} \end{array}$$

The process is repeated until at least one of the two stopping criteria in Equation 16.2 is met. For this particular problem, convergence to 8 decimal places ($\epsilon = 10^{-8}$) is reached in 29 iterations.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$\mathbf{x}^{(0)}$	0	0	0
$\mathbf{x}^{(1)}$	1.5	1	-0.33333
$\mathbf{x}^{(2)}$	1.33333333	1.72222222	-0.66666667
$\mathbf{x}^{(3)}$	1.16666667	1.88888889	-0.90740741
$\mathbf{x}^{(4)}$	1.04629630	1.99382716	-0.96296296
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{x}^{(28)}$	0.99999999	2.00000001	-0.99999999
$\mathbf{x}^{(29)}$	1	2	-1

Matrix Representation

The iterative steps performed above can be expressed in matrix form. First, decompose A into its diagonal entries, its entries below the diagonal, and its entries above the diagonal, as $A = D + L + U$.

$$\underset{D}{\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix}} \quad \underset{L}{\begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \dots & a_{n,n-1} & 0 \end{bmatrix}} \quad \underset{U}{\begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & \dots & 0 \end{bmatrix}}$$

With this decomposition, we solve for $\mathbf{x}$ in the following way.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D+L+U)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= -(L+U)\mathbf{x} + \mathbf{b} \\ \mathbf{x} &= D^{-1}(-(L+U)\mathbf{x} + \mathbf{b}) \end{aligned}$$

Now using $\mathbf{x}^{(k)}$ as the variables on the right side of the equation to produce $\mathbf{x}^{(k+1)}$ on the left, and noting that $L+U = A-D$, we have the following.

$$\begin{aligned} \mathbf{x}^{(k+1)} &= D^{-1}(-(A-D)\mathbf{x}^{(k)} + \mathbf{b}) \\ &= D^{-1}(D\mathbf{x}^{(k)} - A\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \end{aligned} \tag{16.3}$$

There is a potential problem with Equation 16.3: calculating a matrix inverse is the cardinal sin of numerical linear algebra, yet the equation contains D^{-1}. However, since D is a diagonal matrix, D^{-1} is also diagonal, and is easy to compute.

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{bmatrix}$$

Because of this, the Jacobi method requires that A have nonzero diagonal entries.

The diagonal D can be represented by the 1-dimensional array $\mathbf{d}$ of the diagonal entries. Then the matrix multiplication $D\mathbf{x}$ is equivalent to the component-wise vector multiplication $\mathbf{d} * \mathbf{x} = \mathbf{x} * \mathbf{d}$. Likewise, the matrix multiplication $D^{-1}\mathbf{x}$ is equivalent to the component-wise "vector division" $\mathbf{x}/\mathbf{d}$.

```
>>> import numpy as np

>>> D = np.array([[2,0],[0,16]])    # Let D be a diagonal matrix.
>>> d = np.diag(D)                  # Extract the diagonal as a 1-D array.
>>> x = np.random.random(2)
>>> np.allclose(D.dot(x), d*x)
True
```

Problem 1. Write a function that accepts a matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N. Implement the Jacobi method using Equation 16.3, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Run the iteration until $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty < \epsilon$, and only iterate at most N times. Avoid using `la.inv()` to calculate D^{-1}, but use `la.norm()` to calculate the vector ∞-norm $\|\mathbf{x}\|_\infty = \sup |x_i|$.

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

>>> x = np.random.random(10)
>>> la.norm(x, ord=np.inf)          # Use la.norm() for ||x||.
0.74623726404168045
```

```
>>> np.max(np.abs(x))              # Use pure NumPy for ||x||.
0.74623726404168045
```

Your function should be robust enough to accept systems of any size. To test your function, use the following function to generate an $n \times n$ matrix A for which the Jacobi method is guaranteed to converge.

```
def diag_dom(n, num_entries=None):
    """Generate a strictly diagonally dominant nxn matrix.

    Inputs:
        n (int): the dimension of the system.
        num_entries (int): the number of nonzero values
            Defaults to n^(3/2)-n.

    Returns:
        A ((n,n) ndarray): An nxn strictly diagonally dominant matrix.
    """
    if num_entries is None:
        num_entries = int(n**1.5) - n
    A = np.zeros((n,n))
    rows = np.random.choice(np.arange(0,n), size=num_entries)
    cols = np.random.choice(np.arange(0,n), size=num_entries)
    data = np.random.randint(-4, 4, size=num_entries)
    for i in xrange(num_entries):
        A[rows[i], cols[i]] = data[i]
    for i in xrange(n):
        A[i,i] = np.sum(np.abs(A[i])) + 1
    return A
```

Generate a random $\mathbf{b}$ with `np.random.random()`. Run the iteration, then check that $A\mathbf{x}^{(k)}$ and $\mathbf{b}$ are close using `np.allclose()`.

Also test your function on random $n \times n$ matrices. If the iteration is non-convergent, the successive approximations will have increasingly large entries.

Convergence

Most iterative methods only converge under certain conditions. For the Jacobi method, convergence mostly depends on the nature of the matrix A. If the entries a_{ij} of A satisfy the property

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \text{ for all } i = 1, 2, \ldots, n,$$

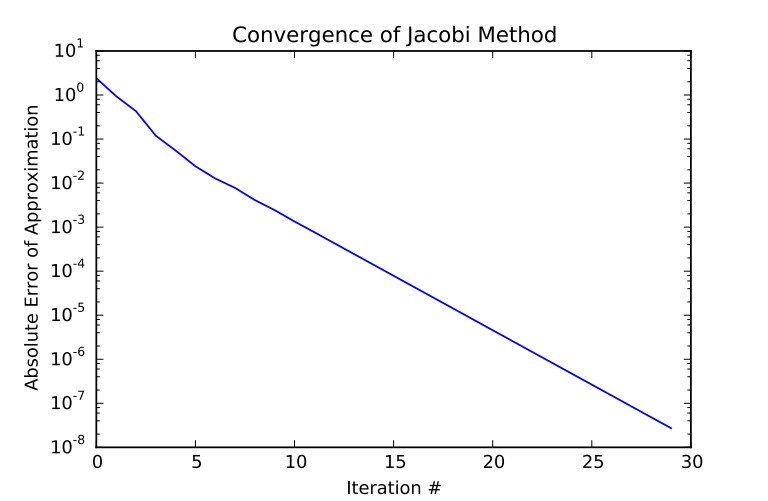
then A is called *strictly diagonally dominant* (for example, `diag_dom()` in Problem 1 generates a strictly diagonally dominant $n \times n$ matrix). If this is the case, then the Jacobi method always converges, regardless of the initial guess $\mathbf{x}_0$.[1] Other iterative methods, such as Newton's method, depend mostly on the initial guess.

There are a few ways to determine whether or not an iterative method is converging. For example, since the approximation $\mathbf{x}^{(k)}$ should satisfy $A\mathbf{x}^{(k)} \approx \mathbf{b}$, the normed difference $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ should be small. This value is called the *absolute error* of the approximation. If the iterative method converges, the absolute error should decrease to ϵ.

Problem 2. Modify your Jacobi method function in the following ways:

1. Add a keyword argument called `plot`, defaulting to `False`.
2. Keep track of the absolute error $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ of the approximation for each value of k.
3. If `plot` is `True`, produce a lin-log plot the error against iteration count (use `plt.semilogy()` instead of `plt.plot()`). Return the approximate solution `x` even if `plot` is `True`.

If the iteration converges, your plot should resemble the following figure.



The Gauss-Seidel Method

The Gauss-Seidel method is essentially a slight modification of the Jacobi method. The main difference is that in Gauss-Seidel, new information is used immediately. Consider the same system as in the previous section.

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3 \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3 \\ & + & x_2 & + & 3x_3 & = & -1 \end{array}$$

[1] Although this seems like a strong requirement, most real-world linear systems can be represented by strictly diagonally dominant matrices.

As with the Jacobi method, solve for x_1 in the first equation, x_2 in the second equation, and x_3 in the third equation.

$$\begin{aligned} x_1 &= \tfrac{1}{2}(3 + x_3) \\ x_2 &= \tfrac{1}{3}(3 + x_1 - 2x_3) \\ x_3 &= \tfrac{1}{3}(-1 - x_2) \end{aligned}$$

Use $\mathbf{x}^{(0)}$ to compute $x_1^{(1)}$ in the first equation.

$$x_1^{(1)} = \frac{1}{2}(3 + x_3^{(0)}) = \frac{1}{2}(3 + 0) = \frac{3}{2}$$

Now, however, use the updated value of $x_1^{(1)}$ in the calculation of $x_2^{(1)}$.

$$x_2^{(1)} = \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(0)}) = \frac{1}{3}(3 + \frac{3}{2} - 0) = \frac{3}{2}$$

Likewise, use the updated values of $x_1^{(1)}$ and $x_2^{(1)}$ to calculate $x_3^{(1)}$.

$$x_3^{(1)} = \frac{1}{3}(-1 - x_2^{(1)}) = \frac{1}{3}(-1 - \frac{3}{2}) = -\frac{5}{6}$$

This process of using calculated information immediately is called *forward substitution*, and causes the algorithm to (generally) converge much faster.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$x^{(0)}$	0	0	0
$x^{(1)}$	1.5	1.5	-0.833333
$x^{(2)}$	1.08333333	1.91666667	-0.97222222
$x^{(3)}$	1.01388889	1.98611111	-0.99537037
$x^{(4)}$	1.00231481	1.99768519	-0.9992284
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x^{(11)}$	1.00000001	1.99999999	-1
$x^{(12)}$	1	2	-1

Notice that Gauss-Seidel converged in less than half as many iterations.

Implementation

Because Gauss-Seidel updates only one element of the solution vector at a time, the iteration cannot be summarized by a single matrix equation. Instead, the process is most generally described by the following equation.

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k)}\right) \tag{16.4}$$

Let A_i be the ith row of A. The two sums closely resemble the regular vector product of A_i and $\mathbf{x}^{(k)}$ without the i^{th} term $a_{ii}x_i^{(k)}$. This gives a simplification.

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}}\left(b_i - A_i^\mathsf{T}\mathbf{x}^{(k)} + a_{ii}x_i^{(k)}\right) \\ &= x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - A_i^\mathsf{T}\mathbf{x}^{(k)}\right) \end{aligned} \tag{16.5}$$

One sweep through all the entries of $\mathbf{x}$ completes one iteration.

Problem 3. Write a function that accepts a matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, a maximum number of iterations N, and a keyword argument `plot` that defaults to `False`. Implement the Gauss-Seidel method using Equation 16.5, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Use the same stopping criterion as in Problem 1. Also keep track of the absolute errors of the iteration, as in Problem 2. If `plot` is `True`, plot the error against iteration count. Use `diag_dom()` to generate test cases.

ACHTUNG!

Since the Gauss-Seidel algorithm operates on the approximation vector in place (modifying it one entry at a time), the previous approximation $\mathbf{x}^{(k-1)}$ must be stored at the beginning of the kth iteration in order to calculate $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty$. Additionally, since NumPy arrays are mutable, the past iteration must be stored as a **copy**.

```
>>> x0 = np.random.random(5)        # Generate a random vector.
>>> x1 = x0                         # Attempt to make a copy.
>>> x1[3] = 1000                    # Modify the "copy" in place.
>>> np.allclose(x0, x1)             # But x0 was also changed!
True
# Instead, make a copy of x0 when creating x1.
>>> x0 = np.copy(x1)                # Make a copy.
>>> x1[3] = -1000
>>> np.allclose(x0, x1)
False
```

Convergence

Whether or not the Gauss-Seidel converges or not also depends on the nature of A. If all of the eigenvalues of A are positive, A is called *positive definite*. If A is positive definite *or* if it is strictly diagonally dominant, then the Gauss-Seidel method converges regardless of the initial guess $\mathbf{x}^{(0)}$.

Problem 4. The Gauss-Seidel method is faster than the standard system solver used by `la.solve()` if the system is sufficiently large and sufficiently sparse. For each vale of $n = 5, 6, \ldots, 11$, generate a random $2^n \times 2^n$ matrix A using `diag_dom()` and a random 2^n vector $\mathbf{b}$. Time how long it takes to solve $A\mathbf{x} = \mathbf{b}$ using your Gauss-Seidel function from Problem 3, and how long it takes to solve using `la.solve()`.

Plot the times against the system size. Use log scales if appropriate.

Solving Sparse Systems Iteratively

Iterative solvers are best suited for solving very large sparse systems. However, using the Gauss-Seidel method on sparse matrices requires translating code from NumPy to `scipy.sparse`. The algorithm is the same, but there are some functions that are named differently between these two packages.

Problem 5. Write a new function that accepts a **sparse** matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N (plotting the convergence is not required for this problem). Implement the Gauss-Seidel method using Equation 16.5, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion.

The Gauss-Seidel method requires extracting the rows A_i from the matrix A and computing $A_i^\mathsf{T}\mathbf{x}$. There are many ways to do this that cause some fairly serious runtime issues, so we provide the code for this specific portion of the algorithm.

```
# Slice the i-th row of A and dot product the vector x.
rowstart = A.indptr[i]
rowend = A.indptr[i+1]
Aix = np.dot(A.data[rowstart:rowend], x[A.indices[rowstart:rowend]])
```

To test your function, cast the result of `diag_dom()` as a sparse matrix.

```
from scipy import sparse

>>> A = sparse.csr_matrix(diag_dom(50000))
>>> b = np.random.random(50000)
```

Successive Over-Relaxation (SOR)

Some systems meet the requirements for convergence with the Gauss-Seidel method, but that do not converge very quickly. A slightly altered version of the Gauss-Seidel method, called *Successive Over-Relaxation*, can result in faster convergence. This is achieved by introducing a *relaxation factor*, ω. The iterative equation for Gauss-Seidel, Equation 16.4 becomes the following.

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k)}\right)$$

Simplifying the equation results in the following.

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - A_i^\mathsf{T}\mathbf{x}^{(k)}\right) \tag{16.6}$$

Note that when $\omega = 1$, Successive Over-Relaxation reduces to Gauss-Seidel.

Problem 6. Write a function that accepts a *sparse* matrix A, a vector $\mathbf{b}$, a convergence tolerance ϵ, and a maximum number of iterations N. Implement Successive Over-Relaxation using Equation 16.6, returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion.
(Hint: this requires changing only one line of code from the sparse Gauss-Seidel function.)

Finite Difference Method

Laplace's equation is an important partial differential equation that arises often in both pure and applied mathematics. In two dimensions, the equation has the following form.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{16.7}$$

Laplace's equation can be used to model heat flow. Consider a square metal plate where the top and bottom sides are fixed at 0° Celsius and the left and right sides are fixed at 100° Celsius. Given these boundary conditions, we want to describe how heat diffuses through the rest of the plate. If $f(x, y) = 0$, then the solution to Laplace's equation describes the plate when it is in a *steady state*, meaning that the heat at a given part of the plate no longer changes with time.

It is possible to solve Equation 16.7 analytically when $f(x, y) = 0$. Instead, however, we use a *finite difference method* to solve the problem numerically. To begin, we impose a discrete, square grid on the plate (see Figure 16.1). Denote the points on the grid by $u_{i,j}$. Then the interior points of the grid can be numerically approximated as follows:

$$\frac{\partial^2 u_{i,j}}{\partial x^2} + \frac{\partial^2 u_{i,j}}{\partial y^2} \approx \frac{1}{h^2}\left(-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right), \tag{16.8}$$

where h is the distance between $u_{i,j}$ and $u_{i+1,j}$ (and between $u_{i,j}$ and $u_{i,j+1}$).[2]

[2]The derivation of Equation 16.8 will be studied in the lab on numerical differentiation.

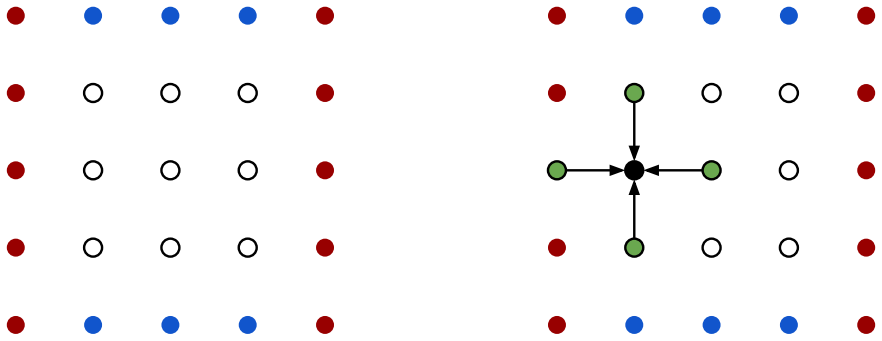


Figure 16.1: On the left, an example of a 6×6 grid where the red dots are hot boundary zones and the blue dots are cold boundary zones. On the right, the green dots are the neighbors of the interior black dot that are used to approximate the heat at the black dot.

This problem can be formulated as a linear system. Suppose the grid has exactly $(n+2)\times(n+2)$ entries. Then the interior of the grid is $n \times n$, and can be flattened into an $n^2 \times 1$ vector $\mathbf{u}$. The entire first row goes first, then the second row, proceeding to the n^{th} row.

$$\mathbf{u} = [u_{1,1},\ u_{1,2},\ \cdots,\ u_{1,n},\ u_{2,1},\ u_{2,2},\ \cdots,\ u_{2,n}\ \ldots,\ u_{n,n}]^\mathsf{T}$$

From Equations 16.7 and 16.8 with $f(x, y) = 0$, we have the following for an interior point $u_{i,j}$:

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = 0. \tag{16.9}$$

If any of the neighbors to $u_{i,j}$ is a boundary point on the grid, its value is already determined. For example, for $u_{3,1}$, the neighbor $u_{3,0} = 100$, so

$$-4u_{3,1} + u_{4,1} + u_{2,1} + u_{3,2} = -100.$$

The constants on the right side of the equation become the $n^2 \times 1$ vector $\mathbf{b}$.

For example, writing Equation 16.9 for the 9 interior points of the grid in Figure 16.1 result the a 9×9 system, $A\mathbf{u} = \mathbf{b}$.

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{bmatrix} = \begin{bmatrix} -100 \\ 0 \\ -100 \\ -100 \\ 0 \\ -100 \\ -100 \\ 0 \\ -100 \end{bmatrix}$$

More generally, for any positive integer n, the corresponding system $A\mathbf{u} = \mathbf{b}$ can be expressed as follows.

$$A = \begin{bmatrix} B_1 & I & & & \\ I & B_2 & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B_n \end{bmatrix}, \quad B_i = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix},$$

where each B_i is $n \times n$. All nonzero entries of $\mathbf{b}$ correspond to interior points that touch the left or right boundaries (since the top and bottom boundaries are held constant at 0).

Problem 7. Write a function that accepts an integer n for the number of interior grid points. Return the corresponding sparse matrix A and NumPy array b.
(Hint: Consider using `scipy.sparse.block_diag` and the `setdiag()` method of scipy sparse matrices for dynamically creating the matrix A.)

Problem 8. To demonstrate how convergence is affected by the value of ω in SOR, time your function from Problem 6 with $\omega = 1,\ 1.05,\ 1.1,\ \ldots, 1.9, 1.95$ using the A and $\mathbf{b}$ generated by problem 7 with $n = 20$. Plot the times as a function of ω.

Note that the matrix A is not strictly diagonally dominant. However, A is positive definite, so the algorithm will converge. Unfortunately, convergence for these kinds of systems usually require more iterations than for strictly diagonally dominant systems. Therefore, set `tol=1e-2` and `maxiters = 1000` on the SOR function.

Problem 9. Write a function that accepts an integer n. Use Problem 7 to generate the corresponding system $A\mathbf{u} = \mathbf{b}$, then solve the system using SciPy's sparse system solver, `scipy.sparse.linalg.spsolve()` (`spla.spsolve()` from the previous lab). Visualize the solution using a heatmap using `np.meshgrid()` and `plt.pcolormesh()` (`"seismic"` is a good color map in this case). This shows the distribution of heat over the hot plate after it has reached its steady state. Note that the solution vector $\mathbf{u}$ must be reshaped to properly visualize the result.

17 GMRES

Lab Objective: *In this lab we will learn how to use the GMRES algorithm.*

The GMRES ("Generalized Minimal Residuals") algorithm is an efficient way to solve large linear systems. It is an iterative method that uses Krylov subspaces to reduce a high-dimensional problem to a sequence of smaller dimensional problems.

The GMRES Algorithm

Let A be an invertible $m \times m$ matrix and let $\mathbf{b}$ be an m-vector. Let $\mathcal{K}_n(A, \mathbf{b})$ be the order-n Krylov subspace generated by A and $\mathbf{b}$. The idea of the GMRES algorithm is that instead of solving $A\mathbf{x} = \mathbf{b}$ directly, we use least squares to find $\mathbf{x}_n \in \mathcal{K}_n$ that minimizes the residual $r_n = \|\mathbf{b} - A\mathbf{x}_n\|_2$. The algorithm returns when this residual is sufficiently small. In good circumstances, this will happen when n is still much less than m.

The GMRES algorithm is implemented with the Arnoldi iteration for numerical stability. The Arnoldi iteration produces H_n, an $(n+1)\times n$ upper Hessenberg matrix, and Q_n, the matrix containing the basis vectors of $\mathcal{K}_n(A, \mathbf{b})$, such that $AQ_n = Q_{n+1}H_n$. We are looking for $\mathbf{x}_n = Q_n\mathbf{y}_n + \mathbf{x}_0$ for some $\mathbf{y}_n \in \mathbb{R}^n$ which minimizes the norm of $\mathbf{b} - A\mathbf{x}_n$. Since the columns of Q are orthonormal, we can compute the residual equivalently as

$$\|\mathbf{b} - A\mathbf{x}_n\|_2 = \|Q_{n+1}(\beta e_1 - H_n\mathbf{y}_n)\|_2 = \|H_n\mathbf{y}_n - \beta e_1\|_2. \tag{17.1}$$

Here $\mathbf{e}_1$ is the vector $(1, 0, \dots, 0)$ of length $n + 1$. β is the Euclidean norm of $\mathbf{b} - A\mathbf{x}_0$, where $\mathbf{x}_0$ is an initial arbitrary guess of the solution. (Ordinarily this guess is zero, and then the $A\mathbf{x}_0$ could be left out; however, a modified version of the algorithm will be discussed at the end of the lab, in which other nonzero guesses will be made.) Thus to minimize the left side of 17.1, we can minimize the right, and $\mathbf{x}_n$ can be computed as $Q_n\mathbf{y}_n + \mathbf{x}_0$.

This algorithm is outlined in Algorithm 17.1. For a complete derivation see [TODO: ref textbook].

Problem 1. Use Algorithm 17.1 to complete the following Python function implementing the GMRES algorithm.

```
def gmres(A, b, x0, k=100, tol=1e-8):
```

Algorithm 17.1 The GMRES algorithm. This algorithm operates on a vector $\mathbf{b}$ and matrix A. It iterates k times or until the residual is less than tol, returning an approximate solution to $A\mathbf{x} = \mathbf{b}$ and the error in this approximation.

procedure GMRES($A, \mathbf{b}, \mathbf{x}_0, k, tol$)
  $Q \leftarrow \text{empty}(\text{size}(\mathbf{b}), k+1)$ ▷ Initialize
  $H \leftarrow \text{zeros}(k+1, k)$
  $r_0 \leftarrow \mathbf{b} - A\mathbf{x}_0$
  $Q[:, 0] = r_0 / \|r_0\|_2$
  for $n = 1 \dots k$ **do**
    Set entries of Q and H as in Arnoldi iteration.
    Compute the residual res and the least squares solution $\mathbf{y}_n$ for the part of H so far created (equation 17.1).
    if $res < tol$ **then**
      return $Q[:, :n+1]\mathbf{y} + \mathbf{x}_0,\ res$
  return $Q[:, :n+1]\mathbf{y} + \mathbf{x}_0,\ res$

```
'''Calculate approximate solution of Ax=b using GMRES algorithm.

INPUTS:
A    - Callable function that calculates Ax for any input vector x.
b    - A NumPy array of length m.
x0   - An arbitrary initial guess.
k    - Maximum number of iterations of the GMRES algorithm. Defaults to 100.
tol  - Stop iterating if the residual is less than 'tol'. Defaults to 1e-8.

RETURN:
Return (y, res) where 'y' is an approximate solution to Ax=b and 'res'
is the residual.

Examples:
>>> a = np.array([[1,0,0],[0,2,0],[0,0,3]])
>>> A = lambda x: a.dot(x)
>>> b = np.array([1, 4, 6])
>>> x0 = np.zeros(b.size)
>>> gmres(A, b, x0)
(array([ 1.,  2.,  2.]), 1.09808907533e-16)
'''
```

You may assume that the input `b` is a real array and the function `A()` always outputs real arrays.

Hint: Use `numpy.linalg.lstsq()` to solve the least squares problem. Be sure to read the documentation so you know what the function returns to you.

Convergence of GMRES

At the n-th iteration, GMRES computes the best approximate solution $\mathbf{x} \in \mathcal{K}_n$ to $A\mathbf{x} = \mathbf{b}$. If A is full rank, then $\mathcal{K}_m = \mathbb{F}^m$, so the m^{th} iteration will always return an exact answer. However, we say the algorithm converges after n steps if the n^{th} residual is sufficiently small.

The rate of convergence of GMRES depends on the eigenvalues of A.

Problem 2. Implement the following Python function by modifying your solution to Problem 1.

```
def plot_gmres(A, b, x0, tol=1e-8):
    '''Use the GMRES algorithm to approximate the solution to Ax=b. Plot ↩
        the eigenvalues of A and the convergence of the algorithm.

    INPUTS:
    A   - A 2-D NumPy array of shape mxm.
    b   - A 1-D NumPy array of length m.
    x0  - An arbitrary initial guess.
    tol - Stop iterating and create the desired plots when the residual is
          less than 'tol'. Defaults to 1e-8.

    OUTPUT:
    Follow the GMRES algorithm until the residual is less than tol, for a
    maximum of m iterations. Then create the two following plots (subplots
    of a single figure):

    1. Plot the eigenvalues of A in the complex plane.

    2. Plot the convergence of the GMRES algorithm by plotting the
    iteration number on the x-axis and the residual on the y-axis.
    Use a log scale on the y-axis.
    '''
```

Use this function to investigate the convergence of GMRES as follows. Define an $m \times m$ matrix

$$A_n = nI + P,$$

where I is the $m \times m$ identity matrix and P is a $m \times m$ matrix of numbers from a random normal distribution with mean 0 and standard deviation $1/(2\sqrt{m})$. Write a function that calls `plot_gmres` on A_n for $n = -4, -2, 0, 2, 4$. Use $m = 200$, let `b` be an array of ones, and let `x0` be the zero vector or anything else that suits you. How does the convergence of the GMRES algorithm relate to the eigenvalues?

Hints:

1. Create a plot with a log scale on the y-axis with `plt.yscale('log')`.
2. Create a matrix with entries from a random normal distribution with `np.random.normal()`. Read the documentation for more information.

3. Note that the parameter A required here is not a callable function but a matrix; this is to allow the finding of the eigenvalues.

4. Output for $n = 2$, $m = 200$ is in Figure 17.1 below.

Ideas for this problem were taken from Example 35.1 on p. 271 of [**Trefethen1997**].

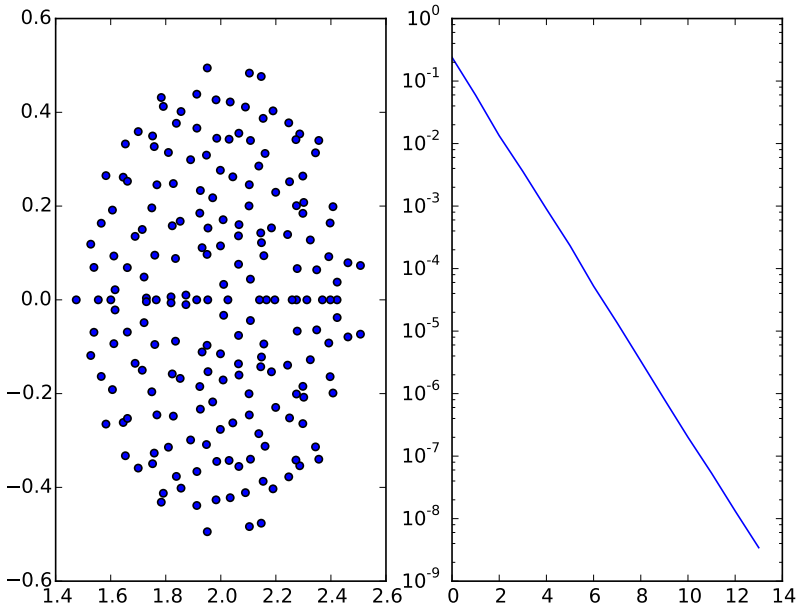


Figure 17.1: The left plot is the eigenvalues of the matrix A_2, which is defined in Problem 2. The right plot is the convergence of the GMRES algorithm on A_2 with starting vector $\mathbf{b} = (1, 1, \ldots, 1)$. This figure is one possible output of the function `plot_gmres()`.

Improving GMRES

There are many ways to make the GMRES algorithm more robust and efficient.

Breakdowns in GMRES

One of the selling points of GMRES is that it can't break down unless it reaches an exact solution. In other words, the only way GMRES could break down is if a vector found by the Arnoldi iteration is 0. That is, suppose, we have already computed

$$\mathcal{K}_n(A, \mathbf{b}) = \text{span}\{\mathbf{b}, A\mathbf{b}, \ldots, A^{n-1}\mathbf{b}\} = \text{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_n\}.$$

We next compute $A^n\mathbf{b}$ and orthogonalize it against $\mathcal{K}_n(A, \mathbf{b})$, yielding $\mathbf{q}_{n+1}$. But if $A^n\mathbf{b} \in \mathcal{K}_n(A, \mathbf{b})$ then $\mathbf{q}_{n+1}$ will be 0, and our algorithm will break when we try to normalize $\mathbf{q}_{n+1}$.

In this situation, the least squares solution to (17.1) is an *exact* solution to $A\mathbf{x} = \mathbf{b}$. In other words, $\mathbf{b}$ is in the $\text{span}\{\mathbf{q}_1, \ldots, \mathbf{q}_n\}$. Fortunately, precautions against this have been taken in our implementation of the Arnoldi algorithm.

GMRES with Restarts

The first few iterations of GMRES have low spatial and temporal complexity. However, as k increases, the k^{th} iteration of GMRES becomes more expensive in both time and memory. In fact, computing the k^{th} iteration of GMRES for very large k can be prohibitively complex.

This issue is addressed by using GMRES(k), or GMRES with restarts. When k becomes large, this algorithm restarts GMRES but with an improved initial guess. GMRES with restarts is outlined in Algorithm 17.2.

Algorithm 17.2 The GMRES(k) algorithm. This algorithm performs GMRES on a vector $\mathbf{b}$ and matrix A. It iterates k times before restarting. It terminates after *restarts* restarts or when the residual is less than *tol*, returning an approximate solution to $A\mathbf{x} = \mathbf{b}$ and the error in this approximation.

procedure GMRES(K)$(A, \mathbf{b}, \mathbf{x}_0, k, tol, restarts)$
  $n \leftarrow 0$ ▷ Initialize
  while $n \leq restarts$ **do**
    Perform the GMRES algorithm, obtaining a least squares solution $\mathbf{y}$.
    If the desired tolerance was reached, return. Otherwise, continue.
    $\mathbf{x}_0 \leftarrow \mathbf{y}$
    $n \leftarrow n + 1$
  return $\mathbf{y}$, *res* ▷ Return the approximate solution and the residual

The algorithm GMRES(k) will always have manageable spatial and temporal complexity, but it is less reliable than GMRES. If the true solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$ is nearly orthogonal to the Krylov subspaces $\mathcal{K}_n(A, \mathbf{b})$ for $n \leq k$, then GMRES(k) could converge very slowly or not at all.

Problem 3. Implement Algorithm 17.2 with the following function.

```
def gmres_k(A, b, x0, k=5, tol=1E-8, restarts=50):
    '''Use the GMRES(k) algorithm to approximate the solution to Ax=b.

    INPUTS:
    A        - A callable function that calculates Ax for any vector x.
    b        - A NumPy array.
    x0       - An arbitrary initial guess.
    k        - Maximum number of iterations of the GMRES algorithm before
               restarting. Defaults to 5.
    tol      - Stop iterating if the residual is less than 'tol'. Defaults
               to 1E-8.
    restarts - Maximum number of restarts. Defaults to 50.

    RETURN:
    Return (y, res) where 'y' is an approximate solution to Ax=b and 'res'
    is the residual.
    '''
```

Compare the speed of `gmres()` from Problem 1 and `gmres_k()` on the matrices in Problem 2.

GMRES in SciPy

The GMRES algorithm is implemented in SciPy as the function `scipy.sparse.linalg.gmres()`. Here we use this function to solve $A\mathbf{x} = \mathbf{b}$ where A is a random 300×300 matrix and $\mathbf{b}$ is a random vector.

```
>>> import numpy as np
>>> from scipy import sparse as spar
>>> from scipy import linalg as la
>>>
>>> A = np.random.rand(300, 300)
>>> b = np.random(300)
>>> x, info = spar.linalg.gmres(A, b)
>>> info
3000
```

The function outputs two objects: the approximate solution `x` and a constant `info` telling if the function converged. If `info=0` then convergence occured; if `info` is positive then it equals the number of iterations performed. In this case, the function performed 3000 iterations of GMRES before returning the approximate solution `x`. We can check how close the solution is.

```
>>> la.norm(A.dot(x)-b)
4.744196381683801
```

We can get a better approximation using GMRES with restarts.

```
>>> # Restart after 1000 iterations
>>> x, info = spar.linalg.gmres(A, b, restart=1000)
>>> info
0
>>> la.norm(A.dot(x)-b)
1.0280404494143551e-12
```

This time, the returned approximation `x` is about as close to a true solution as we could hope for.

18 The Arnoldi Iteration

Lab Objective: *Use Krylov subspaces to find eigenvalues of extremely large matrices.*

One of the biggest difficulties in computational linear algebra is the amount of memory needed to store a large matrix and the amount of time needed to read its entries. Methods using Krylov subspaces avoid this difficulty by studying how a matrix acts on vectors, making it unnecessary in many cases to create the matrix itself.

The *Arnoldi iteration* is an algorithm for finding an orthonormal basis of a Krylov subspace. One of its strengths is it can run on any linear operator without knowing the operator's underlying matrix representation. The outputs of the Arnoldi algorithm can be used to approximate the eigenvalues of the matrix of the linear operator.

Krylov Subspaces

The order-N Krylov subspace of A generated by $\mathbf{x}$ is

$$\mathcal{K}_n(A, \mathbf{x}) = \text{span}\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}.$$

If the vectors $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$ are linearly independent, then they form a basis for $\mathcal{K}_n(A, \mathbf{x})$. However, this basis is usually far from orthogonal, and hence computations using this basis will likely be ill-conditioned.

The Arnoldi Iteration Algorithm

One way to find an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$ is to use the modified Gram-Schmidt algorithm from Lab 3 on the set $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$. The Arnold iteration does this more efficiently by integrating the creation of $\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{n-1}\mathbf{x}\}$ with the modified Gram-Schmidt algorithm. It returns an orthonormal basis for $\mathcal{K}_n(A, \mathbf{x})$. This algorithm is described in Algorithm 18.1.

In Algorithm 18.1, k is the number of times we multiply by A. This will result in an order-$k+1$ Krylov subspace.

Something perhaps unexpected happens in the Arnoldi iteration if the starting vector $\mathbf{x}$ is an eigenvector of A. If the corresponding eigenvalue is λ, then by definition $\mathcal{K}_k(A, \mathbf{x}) = \text{span}\{\mathbf{x}, \lambda\mathbf{x}, \lambda^2\mathbf{x}, \dots, \lambda^k\mathbf{x}\}$, which is equal to the span of $\mathbf{x}$. Let us trace through Algorithm 18.1 in this case. We will use $\mathbf{q}_i$ to denote the i^{th} column of Q.

Algorithm 18.1 The Arnoldi Iteration. This algorithm accepts a square matrix A and starting vector $\mathbf{b}$. It iterates k times or until the norm of the next vector in the iteration is less than tol. The algorithm returns upper Hessenberg H and orthonormal Q such that $H = Q^{\mathsf{H}}AQ$.

```
1:  procedure ARNOLDI(b, A, k, tol)
2:      Q ← empty(size(b), k + 1)                    ▷ Some initialization steps
3:      H ← zeros(k + 1, k)
4:      Q[:, 0] ← b/‖b‖_2
5:      for j = 0 ... k − 1 do                       ▷ Perform the actual iteration.
6:          Q[:, j + 1] ← AQ[:, j]
7:          for i = 0 ... j do                       ▷ Modified Gram-Schmidt.
8:              H[i, j] ← Q[:, i]^T Q[:, j + 1]
9:              Q[:, j + 1] ← Q[:, j + 1] − H[i, j]Q[:, i]
10:         H[j + 1, j] ← ‖Q[:, j + 1]‖_2            ▷ Set subdiagonal element of H.
11:         if |H[j + 1, j]| < tol then              ▷ Stop if ‖Q[:, j + 1]‖_2 is too small.
12:             return H[: j + 1, : j + 1], Q[:, : j + 1]
13:         Q[:, j + 1] ← Q[:, j + 1]/H[j + 1, j]    ▷ Normalize q_{j+1}.
14:     return H[: −1, :], Q                         ▷ Return H_k.
```

In line 4 we normalize $\mathbf{x}$, setting $\mathbf{q}_1 = \mathbf{x}/\|\mathbf{x}\|$. In line 6 we set $\mathbf{q}_2 = A\mathbf{q}_1 = \lambda\mathbf{q}_1$. Then in line 8

$$H_{1,1} = \langle \mathbf{q}_1, \mathbf{q}_2 \rangle = \langle \mathbf{q}_1, \lambda\mathbf{q}_1 \rangle = \lambda\langle \mathbf{q}_1, \mathbf{q}_1 \rangle = \lambda,$$

so in line 9 we subtract $\lambda\mathbf{q}_1$ from $\mathbf{q}_2$, ending with $\mathbf{q}_2 = 0$.

The vector $\mathbf{q}_2$ is supposed to be the next vector in the orthonormal basis for $\mathcal{K}_k(A, \mathbf{x})$, but since it is 0, it is not linearly independent of $\mathbf{q}_1$. In fact, $\mathbf{q}_1$ already spans $\mathcal{K}_k(A, \mathbf{x})$. Hence, when in line 11 we find that the norm of $\mathbf{q}_2$ is zero (or close to it, allowing for numerical error), we terminate the algorithm early, returning the 1×1 matrix $H = H_{1,1} = \lambda$ and the $n \times 1$ matrix $Q = \mathbf{q}_1$.

A similar phenomenon may occur if the starting vector $\mathbf{x}$ is contained in a proper invariant subspace of A.

Arnoldi Iteration on Linear Operators

A major strength of the Arnoldi Iteration is that it can run on a linear operator, even without knowing the matrix representation of the operator. If A_{mul} is some linear function, then we can modify the pseudocode above by replacing $AQ[:, j]$ with $A_{mul}(Q[:, j])$. This will make it possible to find the eigenvalues of an arbitrary linear transformation. We will use this method in the problem below.

Problem 1. Using Algorithm 18.1, complete the following Python function that performs the Arnoldi iteration. Write this function so that it can run on complex arrays.

```
def arnoldi(b, Amul, k, tol=1e-8):
    """Perform `k' steps of the Arnoldi iteration on the linear operator
    defined by `Amul', starting with the vector 'b'.

    Inputs:
```

```
        b (ndarray): The starting vector for the iteration.
        Amul (function): A function handle that describes a linear ↩
            operator.
        k (int): The number of times to perform the iteration.
        tol (float): Stop iterating if the next vector in the iteration ↩
            has
            norm less than `tol'. Defaults to 1e-8.

    Returns:
        H_n (ndarray)
        Q_n (ndarray)
            The number n will equal k, unless the algorithm terminated ↩
                early,
            in which case n will be less than k.

    Examples:
        >>> A = np.array([[1,0,0],[0,2,0],[0,0,3]])
        >>> Amul = lambda x: A.dot(x)
        >>> H, Q = arnoldi(np.array([1,1,1]), Amul, 3)
        >>> np.allclose(H, np.conjugate(Q.T).dot(A).dot(Q) )
        True

        >>> H, Q = arnoldi(np.array([1,0,0]), Amul, 3)
        >>> H
        array([[ 1.+0.j]])
        >>> np.conjugate(Q.T).dot(A).dot(Q)
        array([[ 1.+0.j]])
    """
```

Hints:

1. Since `H` and `Q` will eventually hold complex numbers, initialize them as complex arrays (e.g., `A = np.empty((3,3), dtype=np.complex128)`).
2. Remember to use complex inner products.
3. This function can be tested on a matrix A by passing in `A.dot` for `Amul`.

Finding Eigenvalues Using Arnoldi Iteration

Let A be an $n \times n$ matrix. Let Q_k be the matrix whose columns $\mathbf{q}_1, \ldots, \mathbf{q}_k$ are the orthonormal basis for $\mathcal{K}_m(A, \mathbf{x})$ generated by the Arnoldi algorithm, and let H_k be the $k \times k$ upper Hessenburg matrix defined at the k^{th} stage of the algorithm. Then these matrices satisfy

$$H_k = Q_k^{\mathsf{H}} A Q_k. \tag{18.1}$$

If $k < n$, then H_k is a low-rank approximation to A. We may use its eigenvalues as approximations to the eigenvalues of A. The eigenvalues of H_k are called *Ritz values*, and in fact they converge quickly to the largest eigenvalues of A.

Problem 2. Finish the following function that computes the Ritz values of a matrix.

```
def ritz(Amul, dim, k, iters):
    """Find `k' Ritz values of the linear operator defined by `Amul'.

    Inputs:
        Amul (function): A function describing a linear operator on R^(dim↩
            ).
        dim (int): The dimension of the space on which `Amul' acts.
        k (int): The number of Ritz values to return.
        iters (int): The number of times to perform the Arnoldi iteration.
            Must be between `k' and `dim'.

    Returns:
        ((k,) ndarray): `k' Ritz values of the operator defined by `Amul.'
    """
```

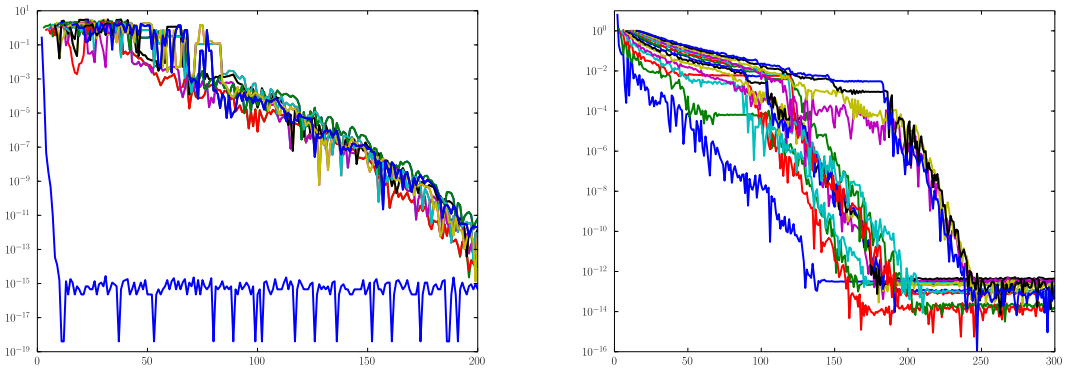
One application of the Arnoldi iteration is to find the eigenvalues of linear operators that are too large to store in memory. For example, if an operator acts on $\mathbb{C}^{2^{20}}$, then its matrix representation contains 2^{40} complex values. Storing such a matrix would require 64 terabytes of memory!

An example of such an operator is the Fast Fourier Transform, cited by SIAM as one of the top algorithms of the century [**Cipra2000**]. The Fast Fourier Transform is used ubiquitously in the field of signal processing.

Problem 3. The four largest eigenvalues of the Fast Fourier Transform are known to be $\{-\sqrt{N}, \sqrt{N}, -i\sqrt{N}, i\sqrt{N}\}$ where N is the dimension of the space on which the transform acts. Use your function `ritz()` from Problem 2 to approximate the eigenvalues of the Fast Fourier Transform. Set `k` to be 10 and set `dim` to be 2^{20}. For the argument `Amul`, use the `fft` function from `scipy.fftpack`.

The Arnoldi iteration for finding eigenvalues is implemented in a Fortran library called ARPACK. SciPy interfaces with the Arnoldi iteration in this library via the function `scipy.sparse.linalg.eigs()`. This function has many more options than the implementation we wrote in Problem 2. In this example, the keyword argument `k=5` specifies that we want five Ritz values. Note that even though this function comes from the `sparse` library in SciPy, we can still call it on regular NumPy arrays.

```
>>> B = np.random.rand(10000).reshape(100, 100)
>>> sp.sparse.linalg.eigs(B, k=5, return_eigenvectors=False)
array([ -1.15577072-2.59438308j,  -2.63675878-1.09571889j,
        -2.63675878+1.09571889j,  -3.00915592+0.j        ,  50.14472893+0.j ])
```



(a) The blue line plots the error of the Ritz value of largest magnitude. This eigenvalue converges after fewer than 20 iterations

(b) All Ritz values have roughly equivalent magnitude. They take from 150 to 250 iterations to converge.

Figure 18.1: These plots show the relative error of the Ritz values as approximations to the eigenvalues of a matrix. The figure at left plots the largest 15 Ritz values for a 500×500 matrix with random entries. The figure at right plots the largest 15 Ritz values for a 500×500 matrix with uniformly distributed eigenvalues.

Convergence

The Arnoldi method for finding eigenvalues quickly converges to eigenvalues whose magnitude is distinctly larger than the rest. For example, matrices with random entries tend to have one eigenvalue of distinctly greatest magnitude. Convergence of the Ritz values for such a matrix is plotted in Figure 18.1a.

However, Ritz values converge more slowly for matrices with random eigenvalues. Figure 18.1b plots convergence of the Ritz values for a matrix with eigenvalues uniformly distributed in $[0, 1)$.

Problem 4. Finish the following function to visualize the convergence of the Ritz values.

```
def plot_ritz(A, n, iters):
    """Plot the relative error of the Ritz values of `A'. Use the number ↩
        of
    iterations as the x-axis and the relative error of the Ritz values of ↩
        H_k
    a approximations to the eigenvalues of A as the y-axis.

    Inputs:
        A (ndarray)
        n (int): The number of Ritz values to plot.
        iters (int): The number of times to perform the Arnoldi iteration.
    """
```

If $\tilde{\mathbf{x}}$ is an an approximation to $\mathbf{x}$, then the *absolute error* in the approximation is

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|}.$$

Hint: The most difficult part of this problem is to identify which Ritz values correspond to which eigenvalues. After finding the Ritz values (or eigenvalues) of largest magnitude, use `np.sort()` to put them in order. Make sure that this order is preserved throughout your program.

It may help to use the following algorithm.

1. Find n eigenvalues of A of largest magnitude. Store these in order.
2. Create an empty array to store the relative errors. For every $k \in$`[1, iters)`,
   (a) Compute H_k with the Arnoldi iteration.
   (b) Find n eigenvalues of A of largest magnitude. Note that for small k, the matrix H_k may not have this many eigenvalues.
   (c) Store the absolute error. Make sure that the errors are stored in the correct order. For small k, some entries in the row or column may not be used.
3. Use array broadcasting to compute the absolute error.
4. Iteratively plot the errors. Lines for distinct eigenvalues should start at different places on the x-axis.

Run your function on these examples. The plots should be fairly similar to Figures 18.1b and 18.1a.

```
>>> # A matrix with random entries
>>> A = np.random.rand(300, 300)
>>> plot_ritz(A, 10, 175)
>>>
>>> # A matrix with uniformly distributed eigenvalues
>>> D = np.diag(np.random.rand(300))
>>> B = A.dot( D.dot(la.inv(A)) )
>>> plot_ritz(B, 10, 175)
```

If your code takes too long to run, consider integrating your solutions to Problems 1 and 2 with the body of this function.

Lanczos Iteration (Optional)

The Lanczos iteration is a version of the Arnoldi iteration that is optimized to operate on symmetric matrices. If A is symmetric, then (18.1) shows that H_k is symmetric and hence tridiagonal. This leads to two simplifications of the Arnoldi algorithm.

First, we have $0 = H_{k,n} = \langle \mathbf{q}_k, A\mathbf{q}_n \rangle$ for $k \leq n-2$; i.e., $A\mathbf{q}_n$ is orthogonal to $\mathbf{q}_1, \ldots, \mathbf{q}_{n-2}$. Thus, if the goal is only to compute H_k (say to find the Ritz values), then we only need to store the two most recently computed columns of Q. Second, the data of H_k can also be stored in two vectors, one containing the main diagonal and one containing the first subdiagonal of H_k. (By symmetry, the first superdiagonal equals the first subdiagonal of H_k.)

The Lanczos iteration is found in Algorithm 18.2.

Algorithm 18.2 The Lanczos Iteration. This algorithm operates on a vector $\mathbf{b}$ of length n and an $n \times n$ symmetric matrix A. It iterates k times or until the norm of the next vector in the iteration is less than *tol*. It returns two vectors $\mathbf{x}$ and $\mathbf{y}$ that respectively contain the main diagonal and first subdiagonal of the current Hessenberg approximation.

procedure LANCZOS($\mathbf{b}, A, k, tol$)
  $\mathbf{q}_0 \leftarrow$ zeros(size($\mathbf{b}$)) ▷ Some initialization
  $\mathbf{q}_1 \leftarrow \mathbf{b}/\|\mathbf{b}\|_2$
  $\mathbf{x} \leftarrow$ empty(k)
  $\mathbf{y} \leftarrow$ empty(k)
  for $i = 0 \ldots k-1$ **do** ▷ Perform the iteration.
    $\mathbf{z} \leftarrow A\mathbf{q}_1$ ▷ $\mathbf{z}$ is a temporary vector to store $\mathbf{q}_{i+1}$.
    $\mathbf{x}[i] \leftarrow \mathbf{q}_1^\mathsf{T}\mathbf{z}$ ▷ $\mathbf{q}_1$ is used to store the previous $\mathbf{q}_i$.
    $\mathbf{z} \leftarrow \mathbf{z} - \mathbf{x}[i]\mathbf{q}_1 + \mathbf{y}[i-1]\mathbf{q}_0$ ▷ $\mathbf{q}_0$ is used to store $\mathbf{q}_{i-1}$.
    $\mathbf{y}[i] = \|\mathbf{z}\|_2$ ▷ Initialize $\mathbf{y}[i]$.
    if $\mathbf{y}[i] < tol$ **then** ▷ Stop if $\|\mathbf{q}_{i+1}\|_2$ is too small.
      return $\mathbf{x}[:i+1]$, $\mathbf{y}[:i]$
    $\mathbf{z} = \mathbf{z}/\mathbf{y}[i]$
    $\mathbf{q}_0, \mathbf{q}_1 = \mathbf{q}_1, \mathbf{z}$ ▷ Store new $\mathbf{q}_{i+1}$ and $\mathbf{q}_i$ on top of $\mathbf{q}_1$ and $\mathbf{q}_0$.
  return $\mathbf{x}$, $\mathbf{y}[:-1]$

Problem 5. Implement Algorithm 18.2 by completing the following function. Write it so that it can operate on complex arrays.

```
def lanczos(b, Amul, k, tol=1E-8):
    '''Perform `k' steps of the Lanczos iteration on the symmetric linear
    operator defined by `Amul', starting with the vector 'b'.

    INPUTS:
    b    - A NumPy array. The starting vector for the Lanczos iteration.
    Amul - A function handle. Should describe a symmetric linear operator.
    k    - Number of times to perform the Lanczos iteration.
    tol  - Stop iterating if the next vector in the Lanczos iteration has
           norm less than `tol'. Defaults to 1E-8.

    RETURN:
    Return (alpha, beta) where alpha and beta are the main diagonal and
    first subdiagonal of the tridiagonal matrix computed by the Lanczos
    iteration.
    '''
```

As it is described in Algorithm 18.2, the Lanczos iteration is not stable. Roundoff error may cause the $\mathbf{q}_i$ to be far from orthogonal. In fact, it is possible for the $\mathbf{q}_i$ to be so adulterated by roundoff error that they are no longer linearly independent.

Problem 6. The following code performs multiplication by a tridiagonal symmetric matrix.

```
def tri_mul(a, b, u):
    ''' Return Au where A is the tridiagonal symmetric matrix with main
    diagonal a and subdiagonal b.
    '''
     v = a * u
     v[:-1] += b * u[1:]
     v[1:] += b * u[:-1]
     return v
```

Let A be a 1000×1000 symmetric tridiagonal matrix with random values in its nonzero diagonals. Use the function `lanczos()` from Problem 5 with 100 iterations to estimate the 5 eigenvalues of A of largest norm. Compare these to the 5 largest true eigenvalues of A

If you do this problem for different vectors a and b, you may notice that occasionally the largest Ritz value is repeated. This happens because the vectors used in the Lanczos iteration may not be orthogonal. These erroneous eigenvalues are called “ghost eigenvalues."

There are modified versions of the Lanczos iteration that are numerically stable. One of these, the Implicitly Restarted Lanczos Method, is found in SciPy as the function `scipy.sparse.linalg.eigsh()`.

Part II

Appendices

NumPy Visual Guide

Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the a^{th} entry up to (but not including) the b^{th} entry." Similarly, `[a:]` means "the a^{th} entry to the end" and `[:b]` means "everything up to (but not including) the b^{th} entry."

$$\texttt{A[1]} = \texttt{A[1,:]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[:,2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[1:,:2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \texttt{A[1:-1,1:-1]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

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

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

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad y = \begin{bmatrix} * & * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & * & * & * & * & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} \qquad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

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

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

$$\texttt{A + np.vstack(x)} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

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

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

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

B Plot Customization and Matplotlib Syntax 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. For an introduction to Matplotlib, see lab* **??**.

Colors

By default, every plot is assigned a different color specified by the "color cycle". It can be overwritten by specifying what color is desired in a few different ways.

-

Matplotlib recognizes some basic built-in colors.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta
`'y'`	yellow
`'k'`	black
`'w'`	white

The following displays how these colors can be implemented. The result is displayed in Figure B.1.

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

colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
x = np.linspace(0, 5, 1000)
y = np.ones(1000)
```

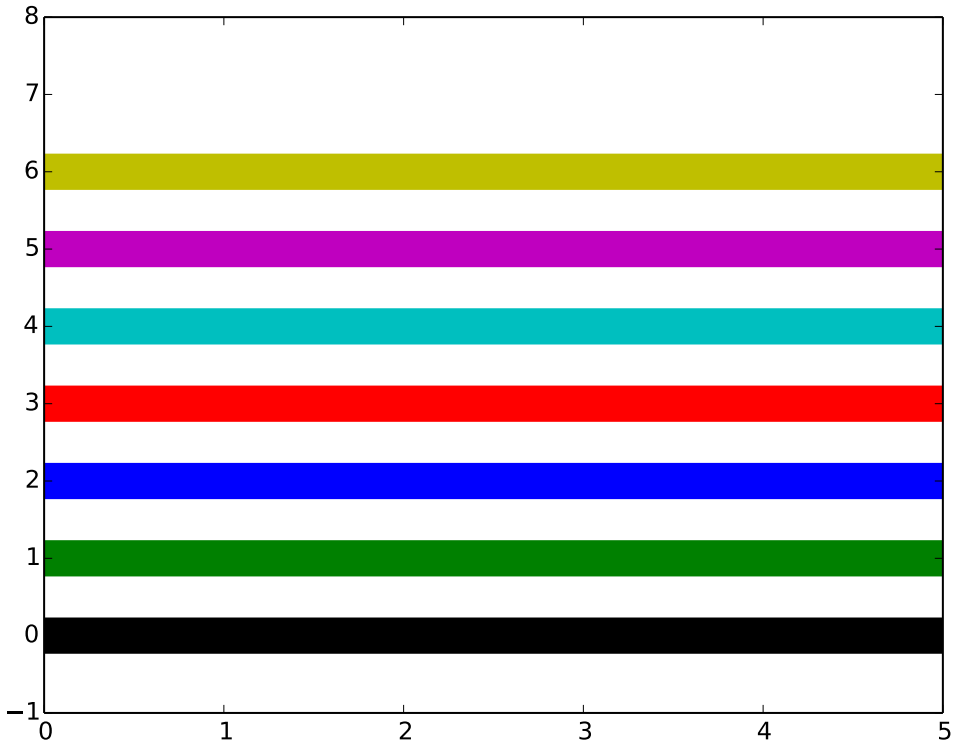


Figure B.1: A display of all the built-in colors.

```
for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=18)

plt.ylim([-1, 8])
plt.savefig("colors.pdf", format='pdf')
plt.clf()
```

colors.py

There are many other ways to specific colors. A popular method to access colors that are not built-in is to use a `RGB` tuple. Colors can also be specified using an html hex string or its associated html color name like `"DarkOliveGreen"`, `"FireBrick"`, `"LemonChiffon"`, `"MidnightBlue"`, `"PapayaWhip"`, or `"SeaGreen"`.

Window Limits

You may have noticed the use of `plt.ylim([ymin, ymax])` in the previous code. This explicitly sets the boundary of the y-axis. Similarily, `plt.xlim([xmin, xmax])` can be used to set the boundary of the x-axis. Doing both commands simultaneously is possible with the `plt.axis([xmin, xmax, ymin, ymax])`. Remember that these commands must be executed after the plot.

Lines

Thickness

You may have noticed that the width of the lines above seemed thin considering we wanted to inspect the line color. `linewidth` is a keyword argument that is defaulted to be `None` but can be given any real number to adjust the line width.

The following displays how `linewidth` is implemented. It is displayed in Figure B.2.

```
lw = np.linspace(.5, 15, 8)

for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=lw[i])

plt.ylim([-1, 8])
plt.show()
```

linewidth.py

Style

By default, plots are drawn with a solid line. The following are accepted format string characters to indicate line style.

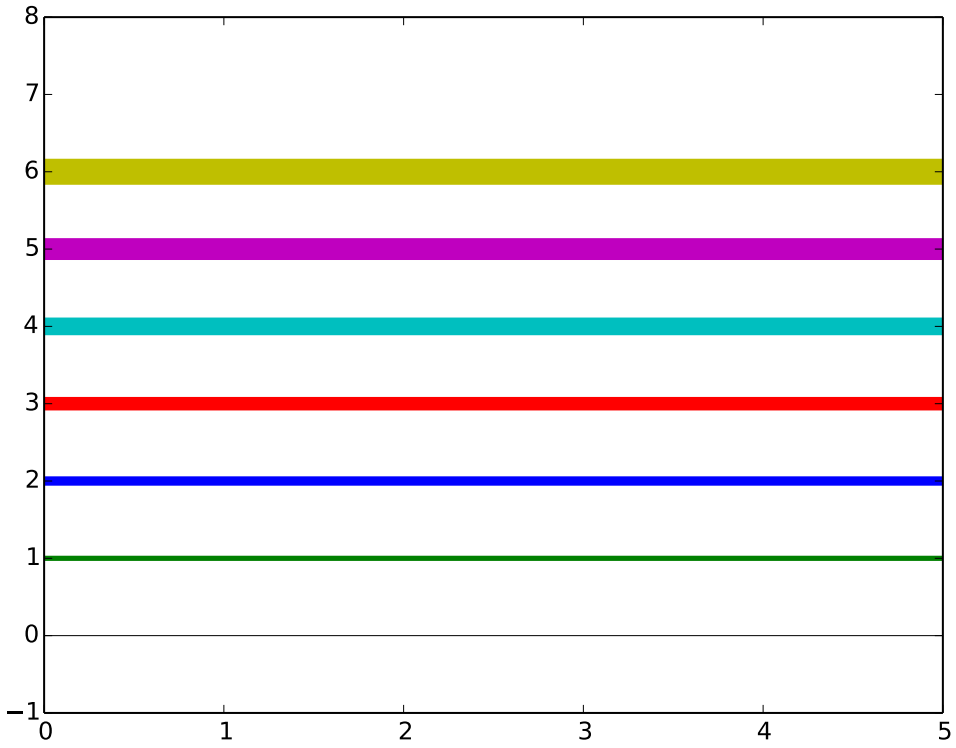


Figure B.2: plot of varying linewidths.

character	description
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

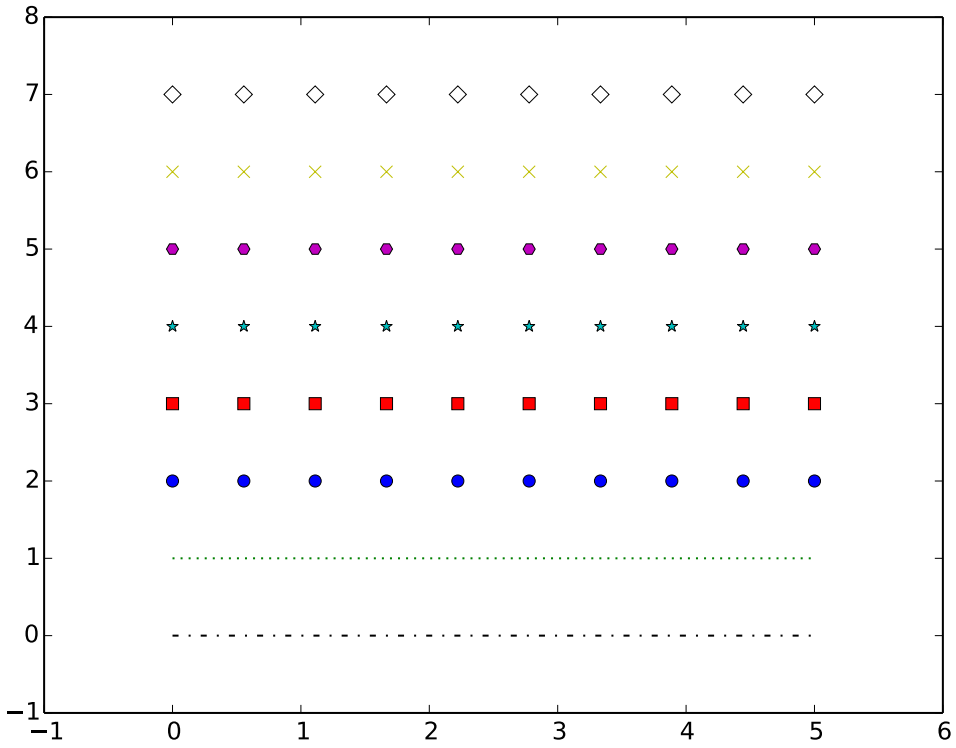


Figure B.3: plot of varying linestyles.

The following displays how `linestyle` can be implemented. It is displayed in Figure B.3.

```
x = np.linspace(0, 5, 10)
y = np.ones(10)
ls = np.array(['-.', ':', 'o', 's', '*', 'H', 'x', 'D'])

for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.axis([-1, 6, -1, 8])
plt.show()
```

linestyle.py

Text

It is also possible to add text to your plots. To label your axes, the `plt.xlabel()` and the `plt.ylabel()` can both be used. The function `plt.title()` will add a title to a plot. If you are working with subplots, this command will add a title to the subplot you are currently modifying. To add a title above the entire figure, use `plt.suptitle()`.

All of the `text()` commands can be customized with `fontsize` and `color` keyword arguments.

We can add these elements to our previous example. It is displayed in Figure B.4.

```
for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")
plt.xlabel("x-axis", fontsize = 10, color = "darkcyan")
plt.ylabel("y-axis", fontsize = 10, color = "darkcyan")

plt.axis([-1, 6, -1, 8])
plt.show()
```

text.py

See `http://matplotlib.org` for Matplotlib documentation.

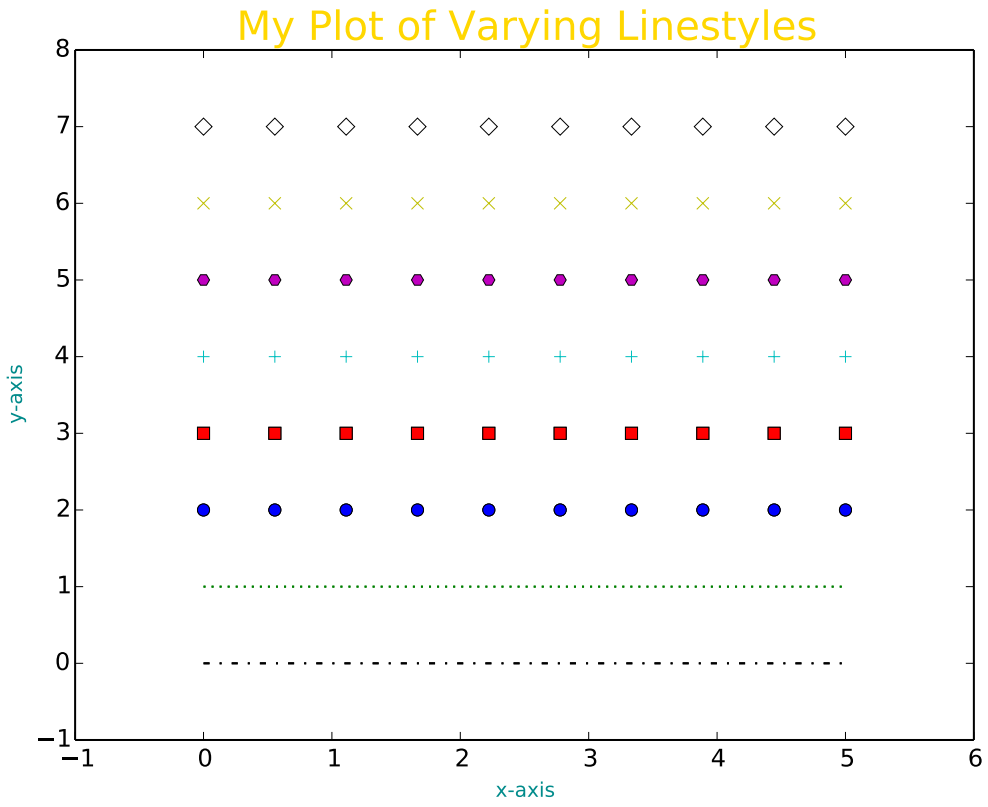


Figure B.4: plot of varying linestyles using text labels.