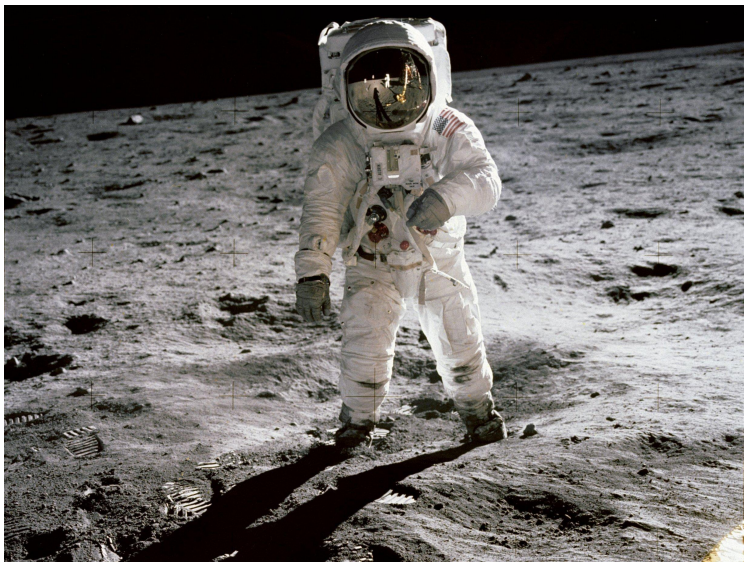
Integration Workshop

Integration Workshop Programming Projects

Jeffrey Humpherys & Tyler J. Jarvis



List of Contributors

B. Barker
Brigham Young University

E. Evans
Brigham Young University

R. Evans
Brigham Young University

J. Grout
Drake University

J. Humpherys
Brigham Young University

T. Jarvis
Brigham Young University

J. Whitehead
Brigham Young University

J. Adams
Brigham Young University

K. Baldwin
Brigham Young University

J. Bejarano
Brigham Young University

J. Bennett
Brigham Young University

A. Berry
Brigham Young University

Z. Boyd
Brigham Young University

M. Brown
Brigham Young University

A. Carr
Brigham Young University

C. Carter
Brigham Young University

S. Carter
Brigham Young University

T. Christensen
Brigham Young University

M. Cook
Brigham Young University

M. Cutler
Brigham Young University

R. Dorff
Brigham Young University

B. Ehlert
Brigham Young University

M. Fabiano
Brigham Young University

K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University

R. Flores
Brigham Young University

R. Fowers
Brigham Young University

A. Frandsen
Brigham Young University

R. Fuhriman
Brigham Young University

T. Gledhill
Brigham Young University

S. Giddens
Brigham Young University

C. Gigena
Brigham Young University

M. Graham
Brigham Young University

F. Glines
Brigham Young University

C. Glover
Brigham Young University

M. Goodwin
Brigham Young University

R. Grout
Brigham Young University

D. Grundvig
Brigham Young University

S. Halverson
Brigham Young University

E. Hannesson
Brigham Young University

K. Harmer
Brigham Young University

J. Henderson
Brigham Young University

J. Hendricks
Brigham Young University

A. Henriksen
Brigham Young University

I. Henriksen
Brigham Young University

B. Hepner
Brigham Young University

C. Hettinger
Brigham Young University

S. Horst
Brigham Young University

R. Howell
Brigham Young University

E. Ibarra-Campos
Brigham Young University

K. Jacobson
Brigham Young University

R. Jenkins
Brigham Young University

J. Larsen
Brigham Young University

J. Leete
Brigham Young University

Q. Leishman
Brigham Young University

J. Lytle
Brigham Young University

E. Manner
Brigham Young University

M. Matsushita
Brigham Young University

R. McMurray
Brigham Young University

S. McQuarrie
Brigham Young University

E. Mercer
Brigham Young University

D. Miller
Brigham Young University

J. Morrise
Brigham Young University

M. Morrise
Brigham Young University

A. Morrow
Brigham Young University

R. Murray
Brigham Young University

J. Nelson
Brigham Young University

C. Noorda
Brigham Young University

A. Oldroyd
Brigham Young University

A. Oveson
Brigham Young University

E. Parkinson
Brigham Young University

M. Probst
Brigham Young University

M. Proudfoot
Brigham Young University

D. Reber
Brigham Young University

H. Ringer
Brigham Young University

C. Robertson
Brigham Young University

M. Russell
Brigham Young University

R. Sandberg
Brigham Young University

C. Sawyer
Brigham Young University

N. Sill
Brigham Young University

D. Smith
Brigham Young University

J. Smith
Brigham Young University

P. Smith
Brigham Young University

M. Stauffer
Brigham Young University

E. Steadman
Brigham Young University

J. Stewart
Brigham Young University

S. Suggs
Brigham Young University

A. Tate
Brigham Young University

T. Thompson
Brigham Young University

B. Trendler
Brigham Young University

M. Victors
Brigham Young University

E. Walker
Brigham Young University

J. Webb
Brigham Young University

R. Webb
Brigham Young University

J. West
Brigham Young University

R. Wonnacott
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

These projects are derived from the Volume 1 Lab Manual that accompanies the textbook *Foundations of Applied Mathematics Volume 1: Mathematical Analysis*, by Humpherys, Jarvis, and Evans. The original lab manual is owned by Dr. J. Humpherys and can be found at

`https://github.com/Foundations-of-Applied-Mathematics/Labs`

Contents

1 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 project we implement three popular iterative methods for solving large systems: Jacobi, Gauss-Seidel, and Successive Over-Relaxation.*

Iterative methods are often useful to solve large systems of equations. In this project, let $\mathbf{x}^{(k)}$ denote the kth iteration of the iterative method for solving the problem $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. Furthermore, let x_i be the ith component of $\mathbf{x}$ so that $x_i^{(k)}$ is the ith component of $\mathbf{x}$ in the kth iteration. Like other iterative methods, there are two stopping parameters: a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$. Iterations continue until either

$$\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| < \varepsilon \qquad \text{or} \qquad k > N. \tag{1.1}$$

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 3×3

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3, \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3, \\ & + & x_2 & + & 3x_3 & = & -1. \end{array}$$

Solving the first equation for x_1, the second for x_2, and the third for x_3 yields

$$\begin{aligned} x_1 &= \tfrac{1}{2}(3 + x_3), \\ x_2 &= \tfrac{1}{3}(3 + x_1 - 2x_3), \\ x_3 &= \tfrac{1}{3}(-1 - x_2). \end{aligned}$$

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 equations:

$$\begin{array}{rcccccc} x_1^{(1)} &=& \tfrac{1}{2}(3 + x_3^{(0)}) &=& \tfrac{1}{2}(3 + 0) &=& \tfrac{3}{2}, \\ x_2^{(1)} &=& \tfrac{1}{3}(3 + x_1^{(0)} - 2x_3^{(0)}) &=& \tfrac{1}{3}(3 + 0 - 0) &=& 1, \\ x_3^{(1)} &=& \tfrac{1}{3}(-1 - x_2^{(0)}) &=& \tfrac{1}{3}(-1 - 0) &=& -\tfrac{1}{3}. \end{array}$$

Thus $\mathbf{x}^{(1)} = [\frac{3}{2}, 1, -\frac{1}{3}]^\mathsf{T}$. Computing $\mathbf{x}^{(2)}$ is similar:

$$\begin{array}{rcccccl} x_1^{(2)} & = & \frac{1}{2}(3 + x_3^{(1)}) & = & \frac{1}{2}(3 - \frac{1}{3}) & = & \frac{4}{3}, \\ x_2^{(2)} & = & \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(1)}) & = & \frac{1}{3}(3 + \frac{3}{2} + \frac{2}{3}) & = & \frac{31}{18}, \\ x_3^{(2)} & = & \frac{1}{3}(-1 - x_2^{(1)}) & = & \frac{1}{3}(-1 - 1) & = & -\frac{2}{3}. \end{array}$$

The process is repeated until at least one of the two stopping criteria in (1.1) is met. For this particular problem, convergence to 8 decimal places ($\varepsilon = 10^{-8}$) 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	−0.33333333
$\mathbf{x}^{(2)}$	1.33333333	1.72222222	−0.66666667
$\mathbf{x}^{(3)}$	1.16666667	1.88888889	−0.90740741
$\mathbf{x}^{(4)}$	1.04629630	1.99382716	−0.96296296
⋮	⋮	⋮	⋮
$\mathbf{x}^{(28)}$	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, $\mathbf{x}$ can be expressed 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.

$$\begin{aligned} \mathbf{x}^{(k+1)} &= D^{-1}(-(A - D)\mathbf{x}^{(k)} + \mathbf{b}) \\ &= D^{-1}(D\mathbf{x}^{(k)} - A\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \end{aligned} \tag{1.2}$$

There is a potential problem with (1.2): 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" $\mathbf{x}/\mathbf{d}$.

Problem 1. Write a function that accepts a matrix A, a vector $\mathbf{b}$, a convergence tolerance `tol` defaulting to 10^{-8}, and a maximum number of iterations `maxiter` defaulting to 100. Implement the Jacobi method using (1.2), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Run the iteration until $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty <$ `tol`, and only iterate at most `maxiter` times. Avoid using `la.inv()` to calculate D^{-1}, but use `la.norm()` to calculate the vector ∞-norm.

Your function should be robust enough to accept systems of any size. To test your function, generate a random $\mathbf{b}$ with `np.random.random()` and use the following function to generate an $n \times n$ matrix A for which the Jacobi method is guaranteed to converge. Run the iteration, then check that $A\mathbf{x}^{(k)}$ and $\mathbf{b}$ are close using `np.allclose()`. There is a file called `test_iterative_solvers.py` that contains prewritten unit tests to test your function for this problem.

```
def diag_dom(n, num_entries=None, as_sparse=False):
    """Generate a strictly diagonally dominant (n, n) matrix.
    Parameters:
        n (int): The dimension of the system.
        num_entries (int): The number of nonzero values.
            Defaults to n^(3/2)-n.
        as_sparse: If True, an equivalent sparse CSR matrix is returned.
    Returns:
        A ((n,n) ndarray): A (n, n) strictly diagonally dominant matrix.
    """
    if num_entries is None:
        num_entries = int(n**1.5) - n
    A = sparse.dok_matrix((n,n))
    rows = np.random.choice(n, size=num_entries)
    cols = np.random.choice(n, size=num_entries)
    data = np.random.randint(-4, 4, size=num_entries)
    for i in range(num_entries):
        A[rows[i], cols[i]] = data[i]
    B = A.tocsr()          # convert to row format for the next step
    for i in range(n):
        A[i,i] = abs(B[i]).sum() + 1
    return A.tocsr() if as_sparse else A.toarray()
```

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

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all} \quad i = 1, 2, \dots, n,$$

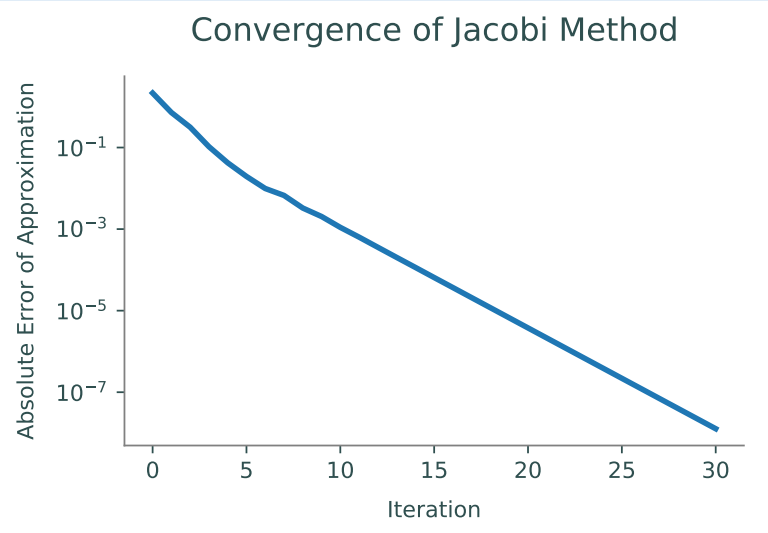
then A is called *strictly diagonally dominant* (`diag_dom()` in Problem 1 generates a strictly diagonally dominant $n \times n$ matrix). If this is the case,[1] then the Jacobi method always converges, regardless of the initial guess $\mathbf{x}_0$. This is a very different convergence result than many other iterative methods such as Newton's method where convergence is highly sensitive to 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 at each iteration.
3. If `plot` is True, produce a lin-log plot (use `plt.semilogy()`) of the error against iteration count. Remember to still return the approximate solution $\mathbf{x}$.

If the iteration converges, your plot should resemble the following figure.



[1] Although this seems like a strong requirement, most real-world linear systems can be represented by strictly diagonally dominant matrices.

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. As an example, consider again the system from the previous section,

$$\begin{array}{rcrcrcr} 2x_1 & & & - & x_3 & = & 3, \\ -x_1 & + & 3x_2 & + & 2x_3 & = & 3, \\ & + & x_2 & + & 3x_3 & = & -1. \end{array}$$

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:

$$\begin{array}{rcl} x_1 & = & \frac{1}{2}(3 + x_3), \\ x_2 & = & \frac{1}{3}(3 + x_1 - 2x_3), \\ x_3 & = & \frac{1}{3}(-1 - x_2). \end{array}$$

Using $\mathbf{x}^{(0)}$ to compute $x_1^{(1)}$ in the first equation as before,

$$x_1^{(1)} = \frac{1}{2}(3 + x_3^{(0)}) = \frac{1}{2}(3 + 0) = \frac{3}{2}.$$

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	−0.83333333
$x^{(2)}$	1.08333333	1.91666667	−0.97222222
$x^{(3)}$	1.01388889	1.98611111	−0.99537037
$x^{(4)}$	1.00231481	1.99768519	−0.99922840
⋮	⋮	⋮	⋮
$x^{(11)}$	1.00000001	1.99999999	−1
$x^{(12)}$	1	2	−1

Notice that Gauss-Seidel converges in less than half as many iterations as Jacobi does for this system.

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 equation

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right). \tag{1.3}$$

Let $\mathbf{a}_i$ be the ith **row** of A and let $\hat{\mathbf{x}}_i^{(k)} = (x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i^{(k)}, x_{i+1}^{(k)}, \ldots, x_n^{(k)})$. The two sums are the regular vector product of $\mathbf{a}_i$ and $\hat{\mathbf{x}}_i^{(k)}$ without the i^{th} term $a_{ii}x_i^{(k)}$. This suggests the simplification

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\hat{\mathbf{x}}_i^{(k)} + a_{ii}x_i^{(k)}\right) \\ &= x_i^{(k)} + \frac{1}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\hat{\mathbf{x}}_i^{(k)}\right). \end{aligned} \tag{1.4}$$

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 `tol` defaulting to 10^{-8}, a maximum number of iterations `maxiter` defaulting to 100, and a keyword argument `plot` that defaults to `False`. Implement the Gauss-Seidel method using (1.4) and details from the example above. Return 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 method converges 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 $\mathbf{x}^{(0)}$.

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 4. Write a new function that accepts a **sparse** matrix A, a vector $\mathbf{b}$, a convergence tolerance `tol`, and a maximum number of iterations `maxiter` (plotting the convergence is not required for this problem). Implement the Gauss-Seidel method using (1.4), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual default 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.

```
# Get the indices of where the i-th row of A starts and ends if the
# nonzero entries of A were flattened.
rowstart = A.indptr[i]
rowend = A.indptr[i+1]

# Multiply only the nonzero elements of the i-th row of A with the
# corresponding elements of x.
Aix = A.data[rowstart:rowend] @ x[A.indices[rowstart:rowend]]
```

To test your function, remeber to call `diag_dom()` using `as_sparse=True`

```
>>> A = diag_dom(1000, as_sparse=True)
>>> b = np.random.random(1000)
```

Unit Test

There is a file called `test_iterative_solvers.py` that contains prewritten unit tests for Problem 1. There is a place for you to add your own unit tests to test your function for Problem 4, which will be graded.

Successive Over-Relaxation

There are many systems that meet the requirements for convergence with the Gauss-Seidel method, but for which convergence is still relatively slow. A slightly altered version of the Gauss-Seidel

method, called *Successive Over-Relaxation* (SOR), can result in faster convergence. This is achieved by introducing a *relaxation factor* $\omega \geq 1$ and modifying (1.3) as

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k)}\right).$$

Simplifying the equation, we have

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \mathbf{a}_i^\mathsf{T}\mathbf{x}^{(k)}\right). \tag{1.5}$$

Note that when $\omega = 1$, SOR reduces to Gauss-Seidel. The relaxation factor ω weights the new iteration between the current best approximation and the next approximation in a way that can sometimes dramatically improve convergence.

Problem 5. Write a function that accepts a sparse matrix A, a vector $\mathbf{b}$, a relaxation factor ω, a convergence tolerance `tol`, and a maximum number of iterations `maxiter`. Implement SOR using (1.5), compute the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion. Return the approximate solution $\mathbf{x}$ as well as a boolean indicating whether the function converged and the number of iterations computed.
(Hint: this requires changing only one line of code from the sparse Gauss-Seidel function.)

A 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.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{1.6}$$

Laplace's equation can be used to model heat flow. Consider a square metal plate where the top and bottom borders 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. 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 (1.6) analytically. However, the problem can also be solved numerically using a *finite difference method*. To begin, we impose a discrete, square grid on the plate with uniform spacing. Denote the points on the grid by (x_i, y_j) and the value of u at these points (the heat) as $u(x_i, y_j) = U_{i,j}$. Using the centered difference quotient for second derivatives to approximate the partial derivatives,

$$\begin{aligned} 0 &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ &\approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h^2} \\ &= \frac{1}{h^2}\left(-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right), \end{aligned} \tag{1.7}$$

where $h = x_{i+1} - x_i = y_{j+1} - y_j$ is the distance between the grid points in either direction. This

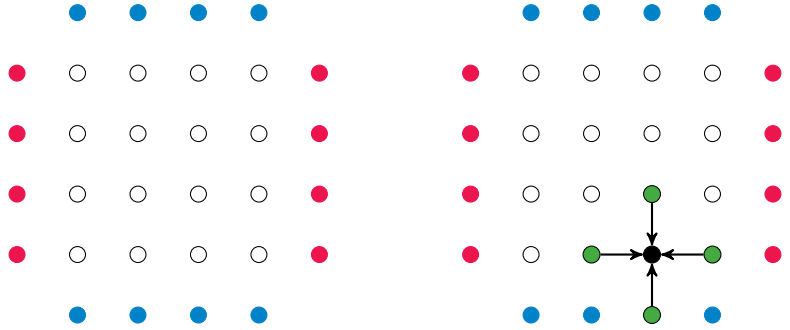


Figure 1.1: On the left, an example of a 6×6 grid ($n = 4$) where the red dots are hot boundary zones and the blue dots are cold boundary zones. On the right, the green dots are the neighbors of the interior black dot that are used to approximate the heat at the black dot.

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 (where $u(x, y)$ is unknown) 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 nth row.

$$\mathbf{u} = \begin{bmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} & U_{2,1} & U_{2,2} & \cdots & U_{2,n} & \cdots & U_{n,n} \end{bmatrix}^\mathsf{T}$$

From (1.7), for an interior point $U_{i,j}$, we have

$$-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} = 0. \tag{1.8}$$

If any of the neighbors to $U_{i,j}$ is a boundary point on the grid, its value is already determined by the boundary conditions. For example, the neighbor $U_{3,0}$ of the gridpoint for $U_{3,1}$ is fixed at $U_{3,0} = 100$. In this case, (1.8) becomes

$$-4U_{3,1} + U_{2,1} + U_{3,2} + U_{4,1} = -100.$$

The constants on the right side of (1.8) become the $n^2 \times 1$ vector $\mathbf{b}$. All nonzero entries of $\mathbf{b}$ correspond to interior points that touch the left or right boundaries.

As an example, writing (1.8) for the 16 interior points of the grid in Figure 1.1 results in the following 16×16 system $A\mathbf{u} = \mathbf{b}$. Note the block structure (empty blocks are all zeros).

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & & & & & \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & & \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & & & & & \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & & & & & \\
\hline
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & \\
\hline
 & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
 & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
 & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
\hline
 & & & & & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
 & & & & & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
 & & & & & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
 & & & & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{array}\right]
\left[\begin{array}{c} U_{1,1} \\ U_{1,2} \\ U_{1,3} \\ U_{1,4} \\ U_{2,1} \\ U_{2,2} \\ U_{2,3} \\ U_{2,4} \\ U_{3,1} \\ U_{3,2} \\ U_{3,3} \\ U_{3,4} \\ U_{4,1} \\ U_{4,2} \\ U_{4,3} \\ U_{4,4} \end{array}\right]
=
\left[\begin{array}{c} -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \\ -100 \\ 0 \\ 0 \\ -100 \end{array}\right]$$

More concisely, for any positive integer n, the matrix A can be written as

$$A = \begin{bmatrix} B & I & & & \\ I & B & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{bmatrix}, \quad \text{where } B = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix} \text{ is } n \times n.$$

Problem 6. Write a function that accepts an integer n, a relaxation factor ω, a convergence tolerance `tol` that defaults to 10^{-8}, a maximum number of iterations `maxiter` that defaults to 100, and a bool `plot` that defaults to `False`. Generate and solve the corresponding system $A\mathbf{u} = \mathbf{b}$ using Problem 5. Also return a boolean indicating whether the function converged and the number of iterations computed.
(Hint: `np.tile()` may be useful for constructing $\mathbf{b}$.)

If `plot=True`, visualize the solution $\mathbf{u}$ with a heatmap using `plt.pcolormesh()` (the colormap `"coolwarm"` is a good choice in this case). This shows the distribution of heat over the hot plate after it has reached its steady state. Note that the $\mathbf{u}$ must be reshaped as an $n \times n$ array to properly visualize the result.

> **Achtung!**
>
> Integer arrays are generally not suitable for iterative methods that need to converge. Float arrays should be used instead. To ensure that the array being passed into your iterative method is of the desired datatype use `b.astype(float)` where `b` is the array in question.

Problem 7. To demonstrate how convergence is affected by the value of the relaxation factor ω in SOR, run your function from Problem 6 with $\omega = 1,\ 1.05,\ 1.1,\ \ldots, 1.9, 1.95$ and $n = 20$. Plot the number of computed iterations as a function of ω. Return the value of ω that results in the least number of iterations.

Note that the matrix A from Problem 6 is not strictly diagonally dominant. However, A is positive definite, so the algorithm will converge. Unfortunately, convergence for these kinds of systems usually requires more iterations than for strictly diagonally dominant systems. Therefore, set `tol=1e-2` and `maxiter=1000`.

Recall that $\omega = 1$ corresponds to the Gauss-Seidel method. Choosing a more optimal relaxation factor saves a large number of iterations. This could translate to saving days or weeks of computation time while solving extremely large linear systems on a supercomputer.

2 Newton's Method

Lab Objective: *Newton's method, the classical method for finding the zeros of a function, is one of the most important algorithms of all time. In this lab we implement Newton's method in arbitrary dimensions and use it to solve a few interesting problems. We also explore in some detail the convergence (or lack of convergence) of the method under various circumstances.*

Iterative Methods

An *iterative method* is an algorithm that must be applied repeatedly to obtain a result. The general idea behind any iterative method is to make an initial guess at the solution to a problem, apply a few easy computations to better approximate the solution, use that approximation as the new initial guess, and repeat until done. More precisely, let F be some function used to approximate the solution to a problem. Starting with an initial guess of x_0, compute

$$x_{k+1} = F(x_k) \tag{2.1}$$

for successive values of k to generate a sequence $(x_k)_{k=0}^{\infty}$ that hopefully converges to the true solution. If the terms of the sequence are vectors, they are denoted by $\mathbf{x}_k$.

In the best case, the iteration converges to the true solution x, written $\lim_{k\to\infty} x_k = x$ or $x_k \to x$. In the worst case, the iteration continues forever without approaching the solution. In practice, iterative methods require carefully chosen *stopping criteria* to guarantee that the algorithm terminates at some point. The general approach is to continue iterating until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. That is, choose a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$, and update the approximation using (2.1) until either

$$|x_k - x_{k-1}| < \varepsilon \qquad \text{or} \qquad k > N. \tag{2.2}$$

The choices for ε and N are significant: a "large" ε (such as 10^{-6}) produces a less accurate result than a "small" ε (such 10^{-16}), but demands less computations; a small N (10) also potentially lowers accuracy, but detects and halts nonconvergent iterations sooner than a large N (10,000). In code, ε and N are often named `tol` and `maxiter`, respectively (or similar).

While there are many ways to structure the code for an iterative method, probably the cleanest way is to combine a `for` loop with a `break` statement. As a very simple example, let $F(x) = \frac{x}{2}$. This method converges to $x = 0$ independent of starting point.

```
>>> F = lambda x: x / 2
>>> x0, tol, maxiter = 10, 1e-9, 8
>>> for k in range(maxiter):           # Iterate at most N times.
...     print(x0, end='  ')
...     x1 = F(x0)                     # Compute the next iteration.
...     if abs(x1 - x0) < tol:         # Check for convergence.
...         break                      # Upon convergence, stop iterating.
...     x0 = x1                        # Otherwise, continue iterating.
...
10  5.0  2.5  1.25  0.625  0.3125  0.15625  0.078125
```

In this example, the algorithm terminates after $N = 8$ iterations (the maximum number of allowed iterations) because the tolerance condition $|x_k - x_{k-1}| < 10^{-9}$ is not met fast enough. If N had been larger (say 40), the iteration would have quit early due to the tolerance condition.

Newton's Method in One Dimension

Newton's method is an iterative method for finding the zeros of a function. That is, if $f : \mathbb{R} \to \mathbb{R}$, the method attempts to find a $\bar{x}$ such that $f(\bar{x}) = 0$. Beginning with an initial guess x_0, calculate successive approximations for $\bar{x}$ with the recursive sequence

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{2.3}$$

The sequence converges to the zero $\bar{x}$ of f if three conditions hold:

1. f and f' exist and are continuous,
2. $f'(\bar{x}) \neq 0$, and
3. x_0 is "sufficiently close" to $\bar{x}$.

In applications, the first two conditions usually hold. If $\bar{x}$ and x_0 are not "sufficiently close," Newton's method may converge very slowly, or it may not converge at all. However, when all three conditions hold, Newton's method converges quadratically, meaning that the maximum error is squared at every iteration. This is very quick convergence, making Newton's method as powerful as it is simple.

Problem 1. Write a function that accepts a function f, an initial guess x_0, the derivative f', a stopping tolerance defaulting to 10^{-5}, and a maximum number of iterations defaulting to 15. Use Newton's method as described in (2.3) to compute a zero $\bar{x}$ of f. Terminate the algorithm when $|x_k - x_{k-1}|$ is less than the stopping tolerance or after iterating the maximum number of allowed times. Return the last computed approximation to $\bar{x}$, a boolean value indicating whether or not the algorithm converged, and the number of iterations completed.

Test your function against functions like $f(x) = e^x - 2$ (see Figure 2.1) or $f(x) = x^4 - 3$. Check that the computed zero $\bar{x}$ satisfies $f(\bar{x}) \approx 0$. Also consider comparing your function to `scipy.optimize.newton()`, which accepts similar arguments.

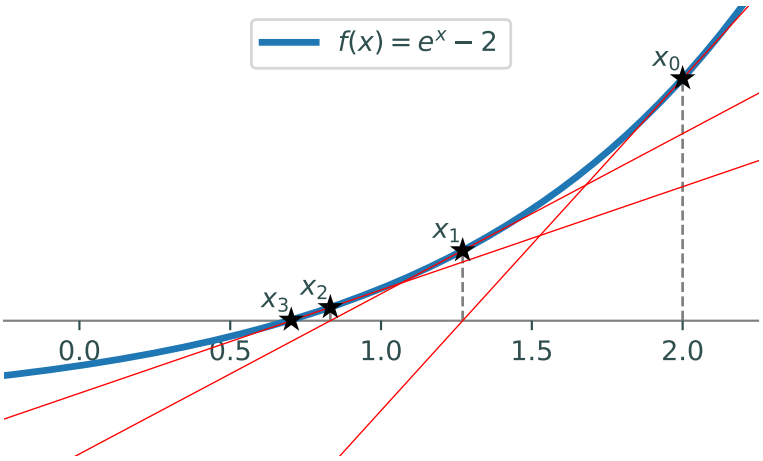


Figure 2.1: Newton's method approximates the zero of a function (blue) by choosing as the next approximation the x-intercept of the tangent line (red) that goes through the point $(x_k, f(x_k))$. In this example, $f(x) = e^x - 2$, which has a zero at $\bar{x} = \log(2)$. Setting $x_0 = 2$ and using (2.3) to iterate, we have $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{e^2-2}{e^2} \approx 1.2707$. Similarly, $x_2 \approx 0.8320$, $x_3 \approx .7024$, and $x_4 \approx 0.6932$. After only a few iterations, the zero $\log(2) \approx 0.6931$ is already computed to several digits of accuracy.

Note

Newton's method can be used to find zeros of functions that are hard to solve for analytically. For example, the function $f(x) = \frac{\sin(x)}{x} - x$ is not continuous on any interval containing 0, but it can be made continuous by defining $f(0) = 1$. Newton's method can then be used to compute the zeros of this function.

Problem 2. 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 (so $r = .05$ corresponds to 5% interest). In addition, 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

$$P_1[(1 + r)^{N_1} - 1] = P_2[1 - (1 + r)^{-N_2}].$$

Write a function that, given N_1, N_2, P_1, and P_2, uses Newton's method to determine r. For the initial guess, use $r_0 = 0.1$.
(Hint: Construct $f(r)$ such that when $f(r) = 0$, the equation is satisfied. Also compute $f'(r)$.)

To test your function, if $N_1 = 30, N_2 = 20, P_1 = 2000$, and $P_2 = 8000$, then $r \approx 0.03878$. (From Atkinson, page 118).

There is a file called `test_newtons_method.py` that contains a unit test for this problem that you can use to check your code.

Backtracking

Newton's method may not converge for a variety of reasons. One potential problem occurs when the step from x_k to x_{k+1} is so large that the zero is stepped over completely. *Backtracking* is a strategy that combats the problem of overstepping by moving only a fraction of the full step from x_k to x_{k+1}. This suggests a slight modification to (2.3),

$$x_{k+1} = x_k - \alpha \frac{f(x_k)}{f'(x_k)}, \qquad \alpha \in (0, 1]. \tag{2.4}$$

Note that setting $\alpha = 1$ results in the exact same method defined in (2.3), but for $\alpha \in (0, 1)$, only a fraction of the step is taken at each iteration.

Problem 3. Modify your function from Problem 1 so that it accepts a parameter α that defaults to 1. Incorporate (2.4) to allow for backtracking.

To test your modified function, consider $f(x) = x^{1/3}$. The command `x**(1/3.)` fails when `x` is negative, so the function and its derivative can be defined with NumPy as follows.

```
import numpy as np
f = lambda x: np.sign(x) * np.power(np.abs(x), 1./3)
Df = lambda x: np.power(np.abs(x), -2./3) / 3.
```

With $x_0 = .01$ and $\alpha = 1$, the iteration should **not** converge. However, setting $\alpha = .4$, the iteration should converge to a zero that is close to 0.

The backtracking constant α is significant, as it can result in faster convergence or convergence to a different zero (see Figure 2.2). However, it is not immediately obvious how to choose an optimal value for α.

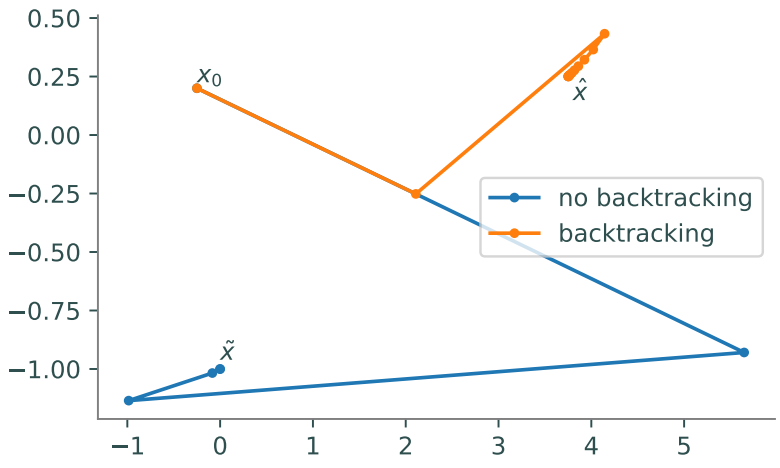


Figure 2.2: Starting at the same initial value but using different backtracking constants can result in convergence to two different solutions. The blue line converges to $\tilde{\mathbf{x}} = (0, -1)$ with $\alpha = 1$ in 5 iterations of Newton's method while the orange line converges to $\hat{\mathbf{x}} = (3.75, .25)$ with $\alpha = 0.4$ in 15 iterations. Note that the points in this example are 2-dimensional, which is discussed in the next section.

Problem 4. Write a function that accepts the same arguments as your function from Problem 3 except for α. Use Newton's method to find a zero of f using various values of α in the interval $(0, 1]$. Plot the values of α against the number of iterations performed by Newton's method. Return a value for α that results in the lowest number of iterations.

A good test case for this problem is the function $f(x) = x^{1/3}$ discussed in Problem 3. In this case, your plot should show that the optimal value for α is actually closer to .3 than to .4.

Newton's Method in Higher Dimensions

Newton's method can be generalized to work on functions with a multivariate domain and range. Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be given by $f(\mathbf{x}) = [f_1(\mathbf{x}) \ f_2(\mathbf{x}) \ \dots \ f_k(\mathbf{x})]^\mathsf{T}$, with $f_i : \mathbb{R}^n \to \mathbb{R}$ for each i. The derivative $Df : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ is the $n \times n$ Jacobian matrix of f.

$$Df = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_k} \end{bmatrix}$$

In this setting, Newton's method seeks a vector $\bar{\mathbf{x}}$ such that $f(\bar{\mathbf{x}}) = \mathbf{0}$, the vector of n zeros. With backtracking incorporated, (2.4) becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha Df(\mathbf{x}_k)^{-1} f(\mathbf{x}_k). \tag{2.5}$$

Note that if $n = 1$, (2.5) is exactly (2.4) because in that case, $Df(x)^{-1} = 1/f'(x)$.

This vector version of Newton's method terminates when the maximum number of iterations is reached or the difference between successive approximations is less than a predetermined tolerance ε with respect to a vector norm, that is, $||\mathbf{x}_k - \mathbf{x}_{k-1}|| < \varepsilon$.

Problem 5. Modify your function from Problems 1 and 3 so that it can compute a zero of a function $f : \mathbb{R}^n \to \mathbb{R}^n$ for any $n \in \mathbb{N}$. Take the following tips into consideration.

- If $n > 1$, f should be a function that accepts a 1-D NumPy array with n entries and returns another NumPy array with n entries. Similarly, Df should be a function that accepts a 1-D array with n entries and returns a $n \times n$ array. In other words, f and Df are callable functions, but $f(\mathbf{x})$ is a vector and $Df(\mathbf{x})$ is a matrix.
- `np.isscalar()` may be useful for determining whether or not $n > 1$.
- Instead of computing $Df(\mathbf{x}_k)^{-1}$ directly at each step, solve the system $Df(\mathbf{x}_k)\mathbf{y}_k = f(\mathbf{x}_k)$ and set $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{y}_k$. In other words, use `la.solve()` instead of `la.inv()`.
- The stopping criterion now requires using a norm function instead of `abs()`.

After your modifications, carefully verify that your function still works in the case that $n = 1$, and that your functions from Problems 2 and 4 also still work correctly. In addition, your function from Problem 4 should also work for any $n \in \mathbb{N}$.

Unit Test

The file `test_newtons_method.py` contains unit tests for Problem 2. There is a place to add your own unit tests to test your function for Problem 5, which will be graded.

Problem 6. Bioremediation involves the use of bacteria to consume toxic wastes. At a steady state, the bacterial density x and the nutrient concentration y satisfy the system of nonlinear equations

$$\begin{aligned} \gamma xy - x(1+y) &= 0 \\ -xy + (\delta - y)(1+y) &= 0, \end{aligned}$$

where γ and δ are parameters that depend on various physical features of the system.[a]

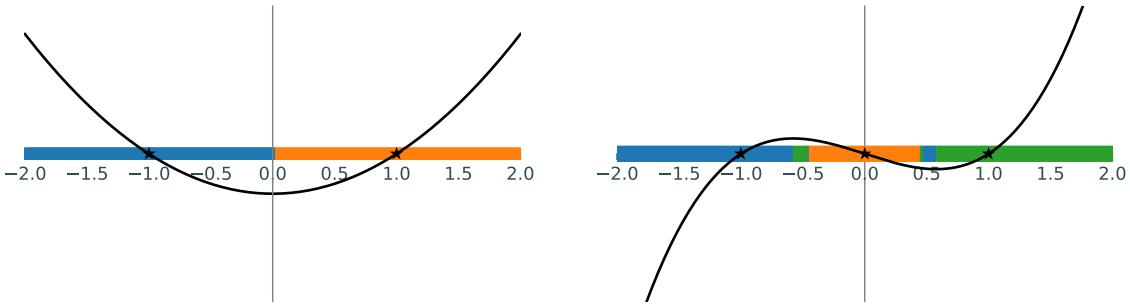
For this problem, assume the typical values $\gamma = 5$ and $\delta = 1$, for which the system has solutions at $(x, y) = (0, 1), (0, -1)$, and $(3.75, .25)$. Write a function that finds an initial point $\mathbf{x}_0 = (x_0, y_0)$ such that Newton's method converges to either $(0, 1)$ or $(0, -1)$ with $\alpha = 1$, and to $(3.75, .25)$ with $\alpha = 0.55$. As soon as a valid $\mathbf{x}_0$ is found, return it (stop searching).
(Hint: search within the rectangle $[-\frac{1}{4}, 0] \times [0, \frac{1}{4}]$.)

[a] This problem is adapted from exercise 5.19 of [Hea02] and the notes of Homer Walker).

Basins of Attraction

When a function f has many zeros, the zero that Newton's method converges to depends on the initial guess x_0. For example, the function $f(x) = x^2 - 1$ has zeros at -1 and 1. If $x_0 < 0$, then Newton's method converges to -1; if $x_0 > 0$ then it converges to 1 (see Figure 2.3a). The regions $(-\infty, 0)$ and $(0, \infty)$ are called the *basins of attraction* of f. Starting in one basin of attraction leads to finding one zero, while starting in another basin yields a different zero.

When f is a polynomial of degree greater than 2, the basins of attraction are much more interesting. For example, the basis of attraction for $f(x) = x^3 - x$ are shown in Figure 2.3b. The basin for the zero at the origin is connected, but the other two basins are disconnected and share a kind of symmetry.



(a) Basins of attraction for $f(x) = x^2 - 1$.

(b) Basins of attraction for $f(x) = x^3 - x$.

Figure 2.3: Basins of attraction with $\alpha = 1$. Since choosing a different value for α can change which zero Newton's method converges to, the basins of attraction may change for other values of α.

It can be shown that Newton's method converges in any Banach space with only slightly stronger hypotheses than those discussed previously. In particular, Newton's method can be performed over the complex plane $\mathbb{C}$ to find imaginary zeros of functions. Plotting the basins of attraction over $\mathbb{C}$ yields some interesting results.

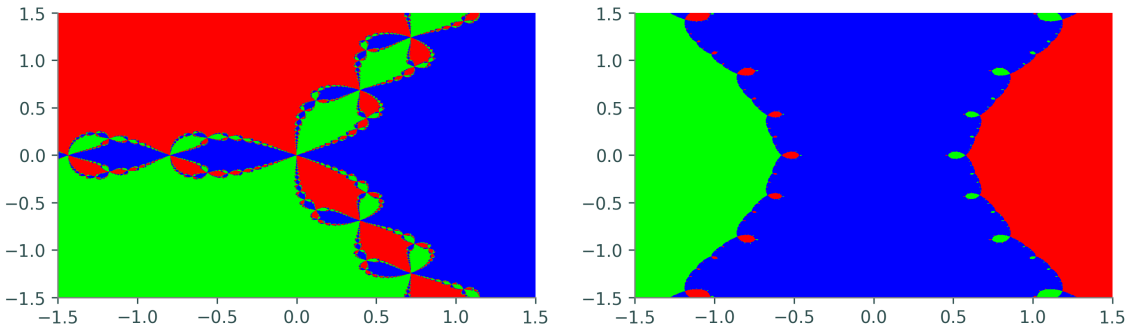
The zeros of $f(x) = x^3 - 1$ are 1, and $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$. To plot the basins of attraction for $f(x) = x^3 - 1$ on the square complex domain $X = \{a + bi \mid a \in [-\frac{3}{2}, \frac{3}{2}], b \in [-\frac{3}{2}, \frac{3}{2}]\}$, create an initial grid of complex points in this domain using `np.meshgrid()`.

```
>>> x_real = np.linspace(-1.5, 1.5, 500)    # Real parts.
>>> x_imag = np.linspace(-1.5, 1.5, 500)    # Imaginary parts.
>>> X_real, X_imag = np.meshgrid(x_real, x_imag)
>>> X_0 = X_real + 1j*X_imag                # Combine real and imaginary parts.
```

The grid X_0 is a 500×500 array of complex values to use as initial points for Newton's method. Array broadcasting makes it easy to compute an iteration of Newton's method at every grid point.

```
>>> f = lambda x: x**3 - 1
>>> Df = lambda x: 3*x**2
>>> X_1 = X_0 - f(X_0)/Df(X_0)
```

After enough iterations, the (i, j)th element of the grid X_k corresponds to the zero of f that results from using the (i, j)th element of X_0 as the initial point. For example, with $f(x) = x^3 - 1$, each entry of X_k should be close to 1, $-\frac{1}{2} + \frac{\sqrt{3}}{2}i$, or $-\frac{1}{2} - \frac{\sqrt{3}}{2}i$. Each entry of X_k can then be assigned a value indicating which zero it corresponds to. Some results of this process are displayed below.



(a) Basins of attraction for $f(x) = x^3 - 1$.

(b) Basins of attraction for $f(x) = x^3 - x$.

Figure 2.4

Note

Notice that in some portions of Figure 2.4a, whenever red and blue try to come together, a patch of green appears in between. This behavior repeats on an infinitely small scale, producing a fractal. Because it arises from Newton's method, this kind of fractal is called a *Newton fractal*.

Newton fractals show that the long-term behavior of Newton's method is **extremely** sensitive to the initial guess x_0. Changing x_0 by a small amount can change the output of Newton's method in a seemingly random way. This phenomenon is called *chaos* in mathematics.

Problem 7. Write a function that accepts a function $f : \mathbb{C} \to \mathbb{C}$, its derivative $f' : \mathbb{C} \to \mathbb{C}$, an array `zeros` of the zeros of f, bounds $[r_{\min}, r_{\max}, i_{\min}, i_{\max}]$ for the domain of the plot, an integer `res` that determines the resolution of the plot, and number of iterations `iters` to run the iteration. Compute and plot the basins of attraction of f in the complex plane over the specified domain in the following steps.

1. Construct a `res`×`res` grid X_0 over the domain $\{a + bi \mid a \in [r_{\min}, r_{\max}], b \in [i_{\min}, i_{\max}]\}$.
2. Run Newton's method (without backtracking) on X_0 `iters` times, obtaining the `res`×`res` array x_k. To avoid the additional computation of checking for convergence at each step, do not use your function from Problem 5.
3. X_k cannot be directly visualized directly because its values are complex. Solve this issue by creating another `res`×`res` array Y. To compute the (i, j)th entry $Y_{i,j}$, determine which zero of f is closest to the (i, j)th entry of X_k. Set $Y_{i,j}$ to the index of this zero in the array `zeros`. If there are R distinct zeros, each $Y_{i,j}$ should be one of $0, 1, \ldots, R - 1$. (Hint: `np.argmin()` may be useful.)
4. Use `plt.pcolormesh()` to visualize the basins. Recall that this function accepts three array arguments: the x-coordinates (in this case, the real components of the initial grid), the y-coordinates (the imaginary components of the grid), and an array indicating color values (Y). Set `cmap="brg"` to get the same color scheme as in Figure 2.4.

Test your function using $f(x) = x^3 - 1$ and $f(x) = x^3 - x$. The resulting plots should resemble Figures 2.4a and 2.4b, respectively (perhaps with the colors permuted).

3 Linear Transformations

Lab Objective: *Linear transformations are the most basic and essential operators in vector space theory. In this project we visually explore how linear transformations alter points in the Cartesian plane. We also 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 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, consider the array of points H that collectively form a picture of a horse, stored in the file `horse.npy`. The coordinate pairs $\mathbf{x}_i$ are organized by column, so the array has two rows: one for x-coordinates, and one for y-coordinates. Matrix multiplication on the left transforms each coordinate pair, resulting in another matrix H' whose columns are the transformed coordinate pairs:

$$AH = A\begin{bmatrix} x_1 & x_2 & x_3 & \dots \\ y_1 & y_2 & y_3 & \dots \end{bmatrix} = A\left[\begin{array}{c|c|c|c} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots \end{array}\right] = \left[\begin{array}{c|c|c|c} A\mathbf{x}_1 & A\mathbf{x}_2 & A\mathbf{x}_3 & \dots \end{array}\right]$$
$$= \left[\begin{array}{c|c|c|c} \mathbf{x}_1' & \mathbf{x}_2' & \mathbf{x}_3' & \dots \end{array}\right] = \begin{bmatrix} x_1' & x_2' & x_3' & \dots \\ y_1' & y_2' & y_3' & \dots \end{bmatrix} = H'.$$

To begin, use `np.load()` to extract the array from the `npy` file, then plot the unaltered points as individual pixels. See Figure 3.1 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 3.1 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 (or both simultaneously). The matrix representation is

$$\begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix}.$$

  Pure horizontal shears ($b = 0$) skew the x-coordinate of the vector while pure vertical shears ($a = 0$) skew the y-coordinate. Figure 3.1 has a horizontal shear with $a = \frac{1}{2}, b = 0$.

- **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 3.1 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 3.1.

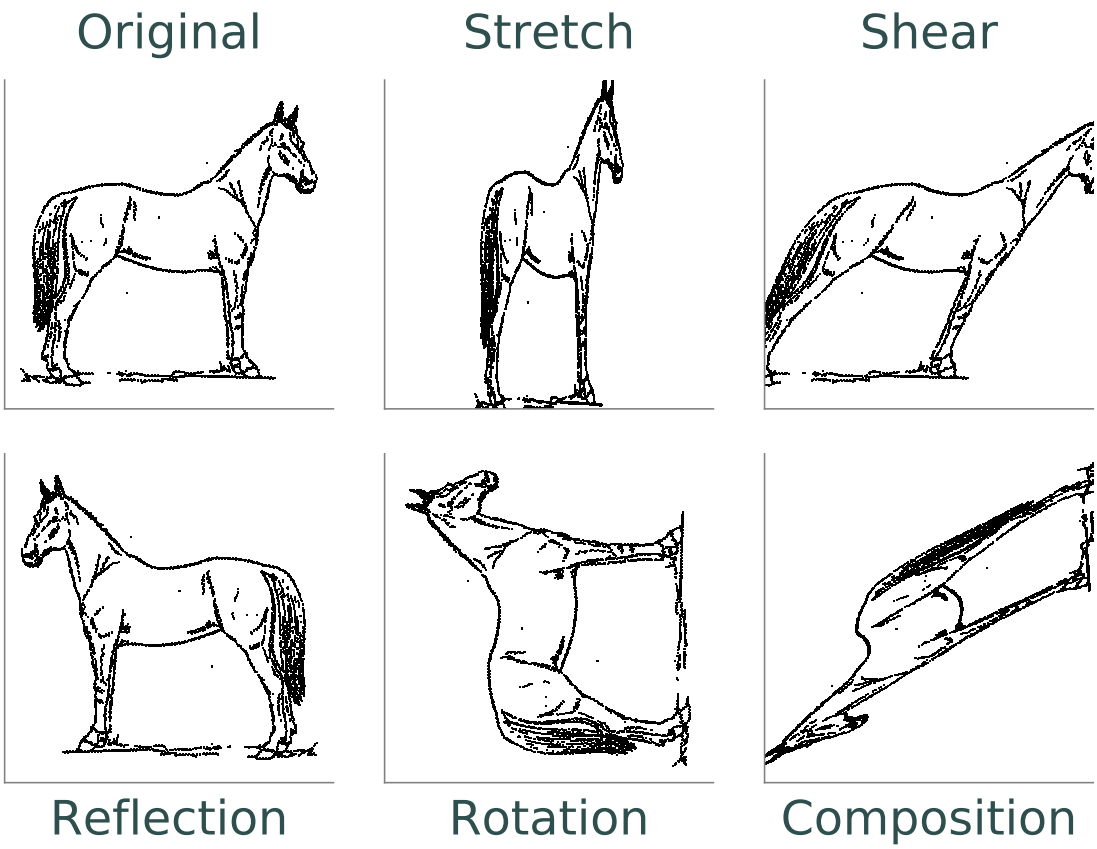


Figure 3.1: The points stored in `horse.npy` under various linear transformations.

Problem 1. Write a function for each type of linear transformation. 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, left multiply it with the input array, and return a transformation of the data.

To test these functions, write a function to plot the original points in `horse.npy` together with the transformed points in subplots for a side-by-side comparison. Compare your results to Figure 3.1.

Compositions of Linear Transformations

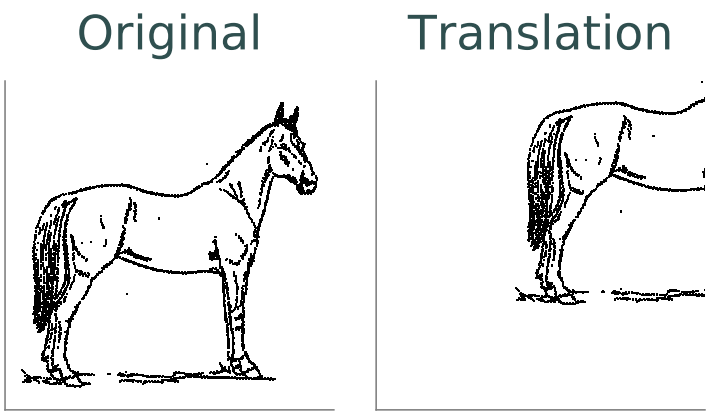
Let V, W, and Z be finite-dimensional vector spaces. If $L : V \to W$ and $K : W \to Z$ are linear transformations with matrix representations A and B, respectively, then the *composition* function $KL : V \to Z$ is also a linear transformation, and its matrix representation is the matrix product BA.

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 3.1 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, 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*.

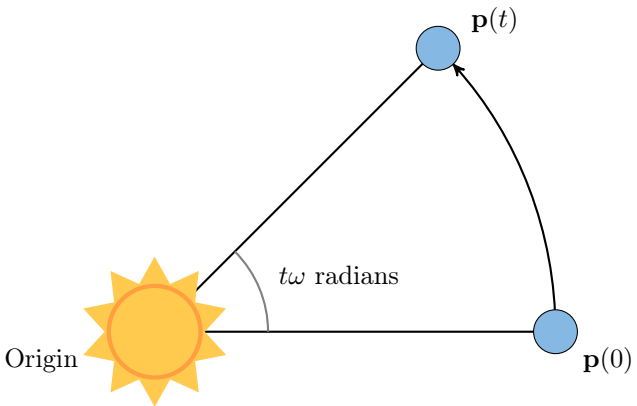
For example, if T is the translation with $\mathbf{b}=\left[\frac{3}{4},\frac{1}{2}\right]^{\mathsf{T}}$, then applying T to an image will shift it right by $\frac{3}{4}$ and up by $\frac{1}{2}$. This translation is illustrated below.



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 velocity 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.

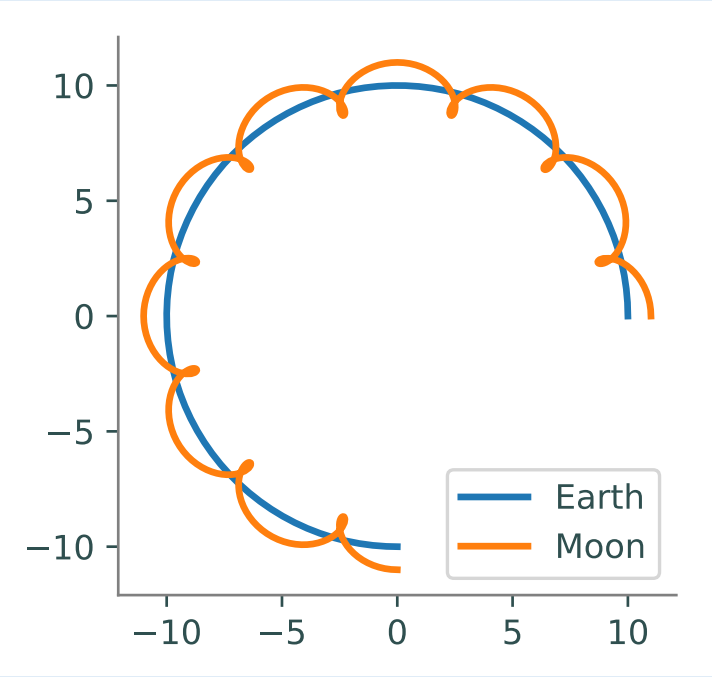
Problem 2. The moon orbits the earth while the earth orbits the sun. Assuming circular orbits, we can compute the trajectories of both the earth and the moon using only linear and affine transformations.

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 velocity. 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, initial positions x_e and x_m, and the angular momenta ω_e and ω_m. Assuming initial positions $\mathbf{p}_e(0) = (x_e, 0)$ and $\mathbf{p}_m(0) = (x_m, 0)$, plot $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ over the time interval $t \in [0, T]$.

Setting $T = \frac{3\pi}{2}$, $x_e = 10$, $x_m = 11$, $\omega_e = 1$, and $\omega_m = 13$, your plot should resemble the following figure (fix the aspect ratio with `ax.set_aspect("equal")`). Note that a more celestially accurate figure would use $x_e = 400$, $x_m = 401$ (the interested reader should see `http://www.math.nus.edu.sg/aslaksen/teaching/convex.html`).



Timing Matrix Operations

Linear transformations are easy to perform via 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

Recall that the `time` module's `time()` function measures the number of seconds since the Epoch. To measure 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. Additionally, in IPython, the quick command `%timeit` uses the `timeit` module to quickly time a single line of code.

```
In [1]: import time

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

In [3]: def time_for_loop():
   ...:     """Time for_loop() with time.time()."""
   ...:     start = time.time()             # Clock the starting time.
   ...:     for_loop()
   ...:     return time.time() - start      # Return the elapsed time.

In [4]: time_for_loop()
0.24458789825439453

In [5]: %timeit for_loop()
248 ms +- 5.35 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

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.

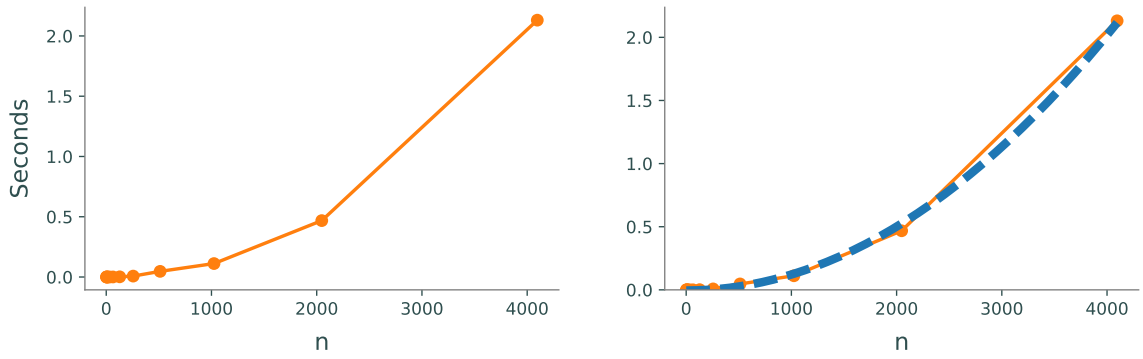
```
from random import random
def random_vector(n):       # Equivalent to np.random.random(n).tolist()
    """Generate a random vector of length n as a list."""
    return [random() for i in range(n)]

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 range(n)] for i in range(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)` $\sim n^2$.

Note

An algorithm like `random_matrix(n)` whose execution time increases quadratically with n is called $O(n^2)$, notated by `random_matrix(n)` $\in O(n^2)$. Big-oh notation is common for indicating both the *temporal complexity* of an algorithm (how the execution time grows with n) and the *spatial complexity* (how the memory usage grows with n).

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:

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

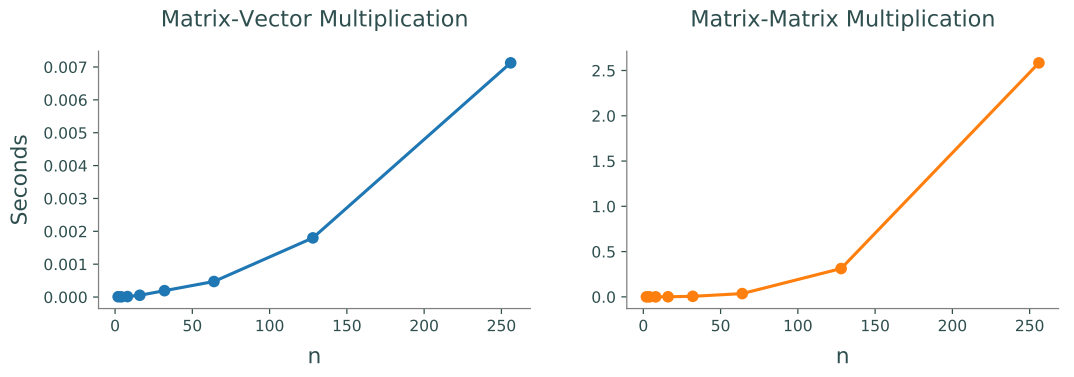
These formulas are implemented below **without** using NumPy arrays or operations.

```
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) ]
```

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

Though the two plots from Problem 3 look similar, the scales on the y-axes show that the actual execution times differ greatly. To be compared correctly, the results need to be viewed 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 `base=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. Then the following code produces the **left** subplot in Figure 3.5.

```
# Plot both curves on a base 2 log-log plot.
>>> plt.loglog(domain, vector_times, 'b.-', base=2, lw=2,label="Matrix-Vector")
>>> plt.loglog(domain, matrix_times, 'g.-', base=2, lw=2,label="Matrix-Matrix")

# Be sure to properly label your plots
>>> plt.title("log-log Plot")
>>> plt.xlabel('n')
>>> plt.ylabel("Seconds")
>>> plt.legend(loc="upper left")

>>> plt.show()
```

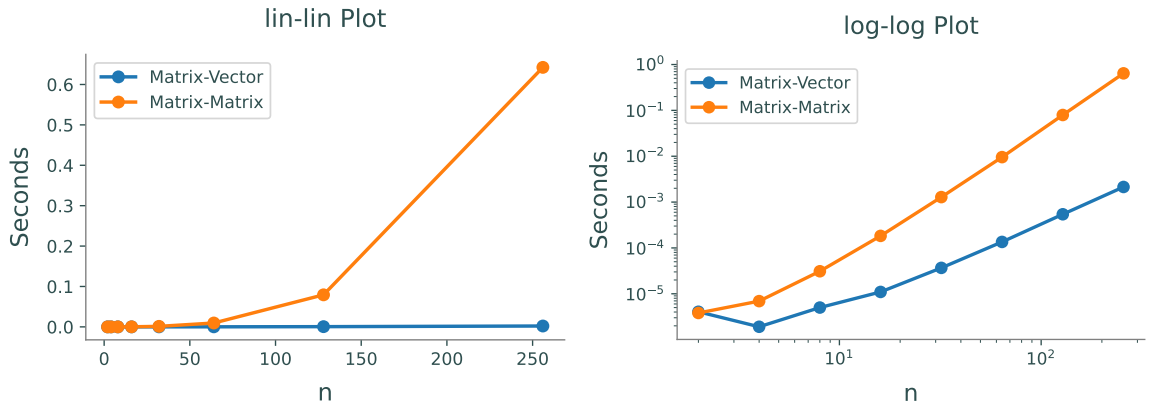


Figure 3.5

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 $O(n^3)$, while matrix-vector multiplication (which only has 2 loops) is only $O(n^2)$.

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()` or `@`.
- matrix-matrix multiplication with `np.dot()` or `@`.

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. Your results should resemble Figure 3.5 except it should have four lines on each subplot. Remember that `np.dot()` and `@` only work on NumPy arrays.

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 carefully takes 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 Function Parameters

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 an 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.
    Parameters:
        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, ax = plt.subplots(1,1)                     # Make a figure explicitly.
    plt.axis([-15,15,-15,15])                       # Set the window limits.
    ax.set_aspect("equal")                          # Make the window square.
    earth_dot,  = ax.plot([],[], 'C0o', ms=10)      # Blue dot for the earth.
    earth_path, = ax.plot([],[], 'C0-')             # Blue line for the earth.
    moon_dot,   = ax.plot([],[], 'C2o', ms=5)       # Green dot for the moon.
    moon_path,  = ax.plot([],[], 'C2-')             # Green line for the moon.
    ax.plot([0],[0],'y*', ms=20)                    # Yellow star for the sun.

    def animate(index):
        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()
```

4 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 project we explore several ways to produce the QR decomposition and implement a few immediate applications.*

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 $\{\mathbf{q}_j\}_{j=1}^{n}$ of Q 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 $\{\mathbf{q}_j\}_{j=1}^{m}$ of Q 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.

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

$\widehat{Q}\ (m \times n)$ and $\widehat{R}\ (n \times n)$

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

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

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

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 vectors in $\{\mathbf{q}_j\}_{j=1}^{k-1}$. 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 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 $\mathbf{q}_1$:

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

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 $\mathbf{q}_2$:

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

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.

Algorithm 1

```
procedure MODIFIED GRAM-SCHMIDT(A)
    m, n ← shape(A)                          ▷ Store the dimensions of A.
    Q ← copy(A)                              ▷ Make a copy of A with np.copy().
    R ← zeros(n, n)                          ▷ An n × n array of all zeros.
    for i = 0 ... n − 1 do
        R_{i,i} ← ||Q_{:,i}||
        Q_{:,i} ← Q_{:,i} / R_{i,i}          ▷ Normalize the ith column of Q.
        for j = i + 1 ... n − 1 do
            R_{i,j} ← Q_{:,j}^T Q_{:,i}
            Q_{:,j} ← Q_{:,j} − R_{i,j} Q_{:,i}   ▷ Orthogonalize the jth column of Q.
    return Q, R
```

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

Consider the following tips for implementing the algorithm.

- 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 routine, `scipy.linalg.qr()`.

```
>>> 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(Q.T @ Q, np.identity(4))
True
>>> np.allclose(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,

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

Then since $\det(AB) = \det(A)\det(B)$,

$$|\det(A)| = |\det(QR)| = |\det(Q)\det(R)| = |\det(Q)|\,|\det(R)| = \left|\prod_{i=1}^{n} r_{i,i}\right|. \tag{4.1}$$

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

Problem 2. Write a function that accepts an invertible matrix A. Use the QR decomposition of A and (4.1) 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?
(Hint: `np.diag()` and `np.prod()` may be useful.)

Check your answer against `la.det()`, which calculates the determinant.

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 $\mathbf{x}$.

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.

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

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.

[2]See the Linear Systems lab for details on back substitution.

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.

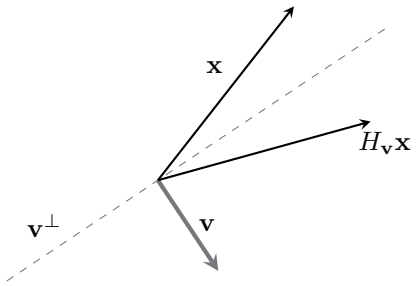


Figure 4.1: The vector $\mathbf{v}$ defines the orthogonal complement $\mathbf{v}^\perp$, which in this case is a line. Applying the Householder transformation $H_\mathbf{v}$ to $\mathbf{x}$ reflects $\mathbf{x}$ across $\mathbf{v}^\perp$.

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, i.e.,

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

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

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

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 4.2). 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 $\mathbf{x}_k$).

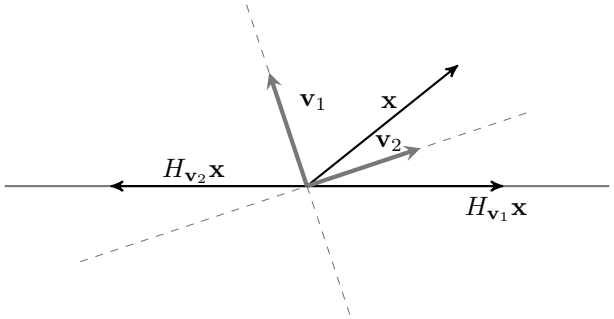


Figure 4.2: There are two reflections that map $\mathbf{x}$ into the span of $\mathbf{e}_1$, defined by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$. In this illustration, $H_{\mathbf{v}_2}$ is the more stable transformation since it reflects $\mathbf{x}$ further than $H_{\mathbf{v}_1}$.

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 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 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 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.

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 @ 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.

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

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

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

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 reflection

$$Q_k = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & I_{m-k} - 2\mathbf{u}_k\mathbf{u}_k^\mathsf{T} \end{bmatrix}, \text{ where}$$
$$\mathbf{v}_k = \mathbf{y}''_k + \operatorname{sign}(z_k)\|\mathbf{y}''_k\|\mathbf{e}_k, \qquad \mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}.$$

The complete algorithm is given below. Note how similar it is to Algorithm 2.

Algorithm 3

procedure HESSENBERG(A)
  $m, n \leftarrow \operatorname{shape}(A)$
  $H \leftarrow \operatorname{copy}(A)$
  $Q \leftarrow I_m$
  for $k = 0 \ldots n-3$ **do**
    $\mathbf{u} \leftarrow \operatorname{copy}(H_{k+1:,k})$
    $u_0 \leftarrow u_0 + \operatorname{sign}(u_0)\|\mathbf{u}\|$
    $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$
    $H_{k+1:,k:} \leftarrow H_{k+1:,k:} - 2\mathbf{u}(\mathbf{u}^\mathsf{T}H_{k+1:,k:})$ ▷ 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 to compute the upper Hessenberg H and orthogonal Q satisfying $A = QHQ^\mathsf{T}$.

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\trp  = A↩
    .
>>> np.allclose(np.triu(H, -1), H)
True
>>> np.allclose(Q @ H @ Q.T, A)
True
```

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 1 and 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 4.2. 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: x/np.abs(x) if 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. To compute the pivoted QR decomposition with `scipy.linalg.qr()`, set the keyword `pivoting` to `True`.

```
# 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 @ 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 matrix for a counterclockwise rotation of θ radians is given by

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

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 $\mathbf{e}_1$.

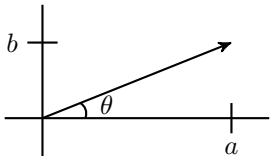


Figure 4.3: Rotating clockwise by θ sends the vector $[a, b]^\mathsf{T}$ to the span of $\mathbf{e}_1$.

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 ith and jth 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 ith and jth entries of any vector it acts on (and thus the ith and jth 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 4.4. Zero out the ijth entry with a rotation in the plane spanned by $\mathbf{e}_{i-1}$ and $\mathbf{e}_i$, represented by the Givens matrix $G(i-1, i, \theta)$.

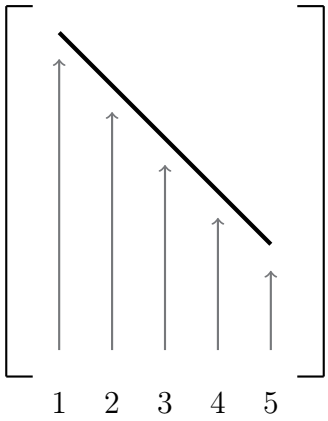


Figure 4.4: The order in which to zero out subdiagonal entries in the Givens triangularization algorithm. The heavy black line is the main diagonal of the matrix. Entries should be zeroed out from bottom to top in each column, beginning with the leftmost column.

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

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

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 ijth stage of the algorithm a_{ij} is nonzero, Algorithm 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 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

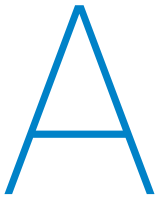
QR of a Hessenberg Matrix via Givens

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 5 details this process.

Algorithm 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

> **NOTE**
>
> When A is symmetric, its upper Hessenberg form is a *tridiagonal* matrix, meaning its only nonzero entries are on the main diagonal, the first subdiagonal, and the first superdiagonal. 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 computing the Hessenberg form of a symmetric matrix is very useful.



NumPy Visual Guide

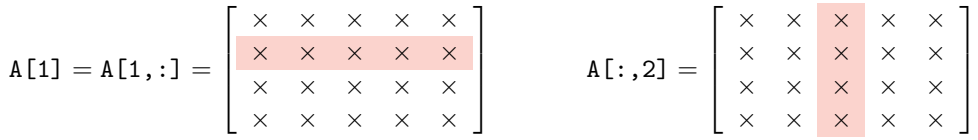
Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

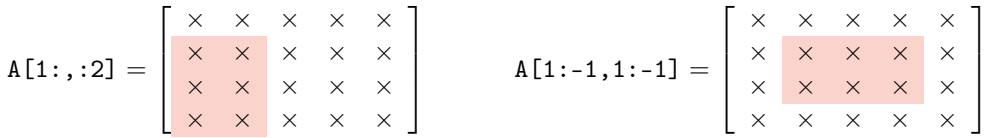
Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

Slicing

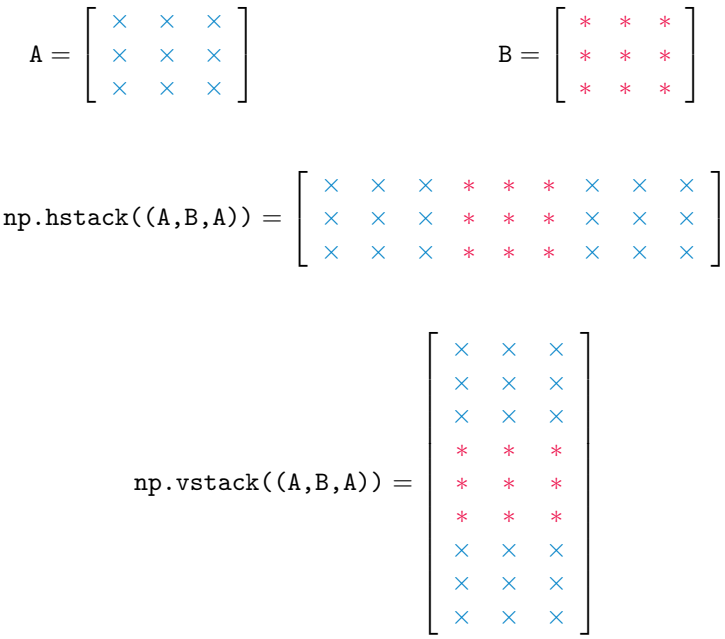
A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the ath entry up to (but not including) the bth entry." Similarly, `[a:]` means "the ath entry to the end" and `[:b]` means "everything up to (but not including) the bth entry."



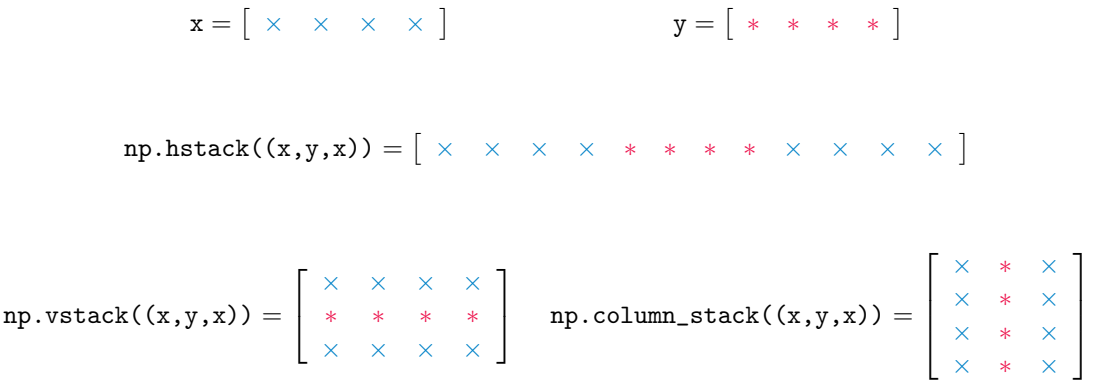


Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.



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()`.



The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

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

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

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

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

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

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

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

B Matplotlib Syntax and Customization Guide

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. It is not intended to be read all at once, but rather to be used as a reference when needed. For an interative introduction to Matplotlib, see the Introduction to Matplotlib lab in Python Essentials. For more details on any specific function, refer to the Matplotlib documentation at* `https://matplotlib.org/`.

Matplotlib Interface

Matplotlib plots are made in a `Figure` object that contains one or more `Axes`, which themselves contain the graphical plotting data. Matplotlib provides two ways to create plots:

1. Call plotting functions directly from the module, such as `plt.plot()`. This will create the plot on whichever `Axes` is currently active.

2. Call plotting functions from an `Axes` object, such as `ax.plot()`. This is particularly useful for complicated plots and for animations.

   Table B.1 contains a summary of functions that are used for managing `Figure` and `Axes` objects.

Function	Description
`add_subplot()`	Add a single subplot to the current figure
`axes()`	Add an axes to the current figure
`clf()`	Clear the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Table B.1: Basic functions for managing plots.

`Axes` objects are usually managed through the functions `plt.subplot()` and `plt.subplots()`. The function `subplot()` is used as `plt.subplot(nrows, ncols, plot_number)`. Note that if the

inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

The function `subplots()` is used as `plt.subplots(nrows, ncols)`, and returns a `Figure` object and an array of `Axes`. This array has the shape `(nrows, ncols)`, and can be accessed as any other array. Figure B.1 demonstrates the layout and indexing of subplots.

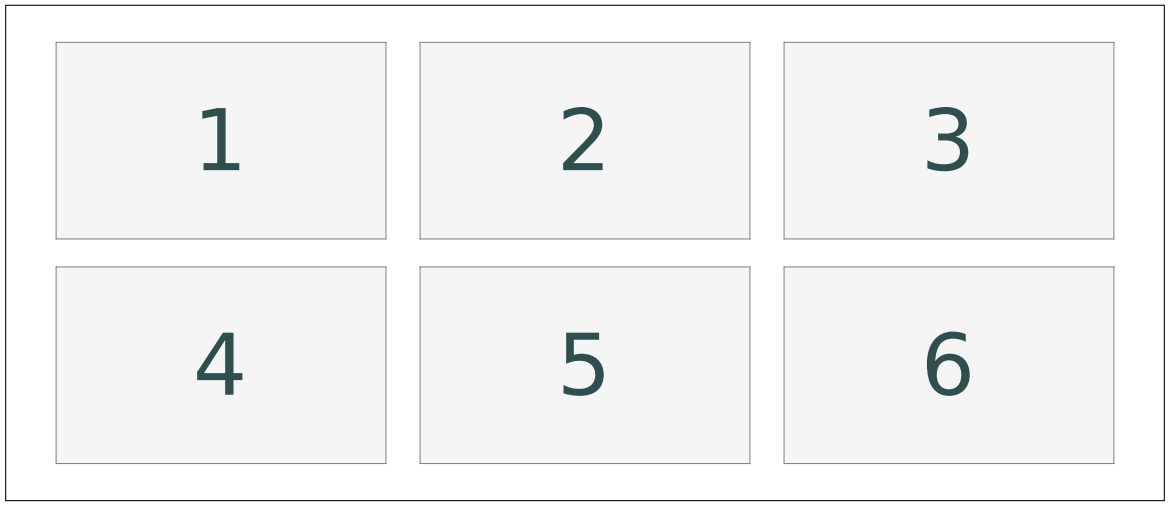


Figure B.1: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

The following example demonstrates three equivalent ways of producing a figure with two subplots, arranged next to each other in one row:

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

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

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

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

Achtung!

Be careful not to mix up the following similarly-named functions:

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

Plot Customization

Styles

Matplotlib has a number of built-in styles that can be used to set the default appearance of plots. These can be used via the function `plt.style.use()`; for instance, `plt.style.use("seaborn")` will have Matplotlib use the "seaborn" style for all plots created afterwards. A list of built-in styles can be found at `https://matplotlib.org/stable/gallery/style_sheets/style_sheets_reference.html`.

The style can also be changed only temporarily using `plt.style.context()` along with a `with` block:

```
with plt.style.context('dark_background'):
    # Any plots created here use the new style
    plt.subplot(1,2,1)
    plt.plot(x, y)
    # ...
# Plots created here are unaffected
plt.subplot(1,2,2)
plt.plot(x, y)
```

Plot layout

Axis properties

Table B.2 gives an overview of some of the functions that may be used to configure the axes of a plot.

The functions `xlim()`, `ylim()`, and `axis()` are used to set one or both of the x and y ranges of the plot. `xlim()` and `ylim()` each accept two arguments, the lower and upper bounds, or a single list of those two numbers. `axis()` accepts a single list consisting, in order, of `xmin, xmax, ymin, ymax`. Passing `None` instead of one of the numbers to any of these functions will make it not change the corresponding value from what it was. Each of these functions can also be called without any arguments, in which case it will return the current bounds. Note that `axis()` can also be called directly on an `Axes` object, while `xlim()` and `ylim()` cannot.

`axis()` also can be called with a string as its argument, which has several options. The most common is `axis('equal')`, which makes the scale of the x- and y-scales equal (i.e. makes circles circular).

Function	Description
`axis()`	set the x- and y-limits of the plot
`grid()`	add gridlines
`xlim()`	set the limits of the x-axis
`ylim()`	set the limits of the y-axis
`xticks()`	set the location of the tick marks on the x-axis
`yticks()`	set the location of the tick marks on the y-axis
`xscale()`	set the scale type to use on the x-axis
`yscale()`	set the scale type to use on the y-axis
`ax.spines[side].set_position()`	set the location of the given spine
`ax.spines[side].set_color()`	set the color of the given spine
`ax.spines[side].set_visible()`	set whether a spine is visible

Table B.2: Some functions for changing axis properties. `ax` is an `Axes` object.

To use a logarithmic scale on an axis, the functions `xscale("log")` and `yscale("log")` can be used.

The functions `xticks()` and `yticks()` accept a list of tick positions, which the ticks on the corresponding axis are set to. Generally, this works the best when used with `np.linspace()`. This function also optionally accepts a second argument of a list of labels for the ticks. If called with no arguments, the function returns a list of the current tick positions and labels instead.

The spines of a Matplotlib plot are the black border lines around the plot, with the left and bottom ones also being used as the axis lines. To access the spines of a plot, call `ax.spines[side]`, where `ax` is an `Axes` object and `side` is `'top'`, `'bottom'`, `'left'`, or `'right'`. Then, functions can be called on the `Spine` object to configure it.

The function `spine.set_position()` has several ways to specify the position. The two simplest are with the arguments `'center'` and `'zero'`, which place the spine in the center of the subplot or at an x- or y-coordinate of zero, respectively. The others are a passed as a tuple `(position_type, amount)`:

- `'data'`: place the spine at an x- or y-coordinate equal to `amount`.
- `'axes'`: place the spine at the specified `Axes` coordinate, where 0 corresponds to the bottom or left of the subplot, and 1 corresponds to the top or right edge of the subplot.
- `'outward'`: places the spine `amount` pixels outward from the edge of the plot area. A negative value can be used to move it inwards instead.

`spine.set_color()` accepts any of the color formats Matplotlib supports. Alternately, using `set_color('none')` will make the spine not be visible. `spine.set_visible()` can also be used for this purpose.

The following example adjusts the ticks and spine positions to improve the readability of a plot of $\sin(x)$. The result is shown in Figure B.2.

```
>>> x = np.linspace(0,2*np.pi,150)
>>> plt.plot(x, np.sin(x))
>>> plt.title(r"$y=\sin(x)$")

#Set the ticks to multiples of pi/2, make nice labels
>>> ticks = np.pi / 2 * np.array([0,1,2,3,4])
```

```
>>> tick_labels = ["$0$", r"$\frac{\pi}{2}$", r"$\pi$", r"$\frac{3\pi}{2}$",
...                 r"$2\pi$"]
>>> plt.xticks(ticks, tick_labels)

#Move the bottom spine to zero, remove the top and right ones
>>> ax = plt.gca()
>>> ax.spines['bottom'].set_position('zero')
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')

>>> plt.show()
```

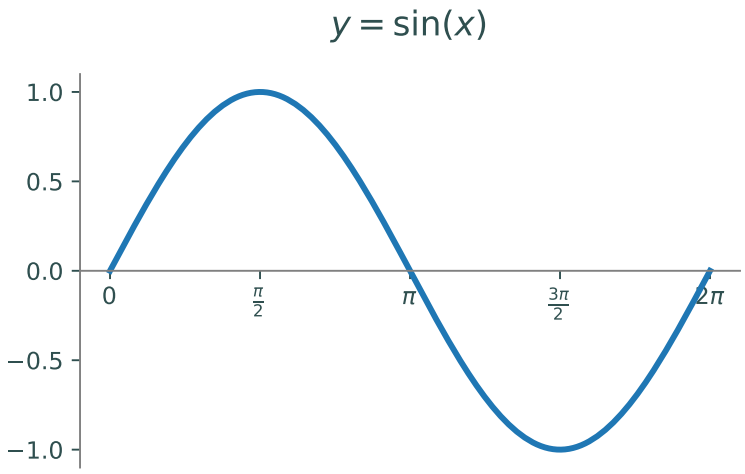


Figure B.2: Plot of $y = \sin(x)$ with axes modified for clarity

Plot Layout

The position and spacing of all subplots within a figure can be modified using the function `plt.subplots_adjust()`. This function accepts up to six keyword arguments that change different aspects of the spacing. `left`, `right`, `top`, and `bottom` are used to adjust the rectangle around all of the subplots. In the coordinates used, 0 corresponds to the bottom or left edge of the figure, and 1 corresponds to the top or right edge of the figure. `hspace` and `wspace` set the vertical and horizontal spacing, respectively, between subplots. The units for these are in fractions of the average height and width of all subplots in the figure. If more fine control is desired, the position of individual `Axes` objects can also be changed using `ax.get_position()` and `ax.set_position()`.

The size of the figure can be configured using the `figsize` argument when creating a figure:

```
>>> plt.figure(figsize=(12,8))
```

Note that many environments will scale the figure to fill the available space. Even so, changing the figure size can still be used to change the aspect ratio as well as the relative size of plot elements.

The following example uses `subplots_adjust()` to create space for a legend outside of the plotting space. The result is shown in Figure B.3.

```
#Generate data
>>> x1 = np.random.normal(-1, 1.0, size=60)
>>> y1 = np.random.normal(-1, 1.5, size=60)
>>> x2 = np.random.normal(2.0, 1.0, size=60)
>>> y2 = np.random.normal(-1.5, 1.5, size=60)
>>> x3 = np.random.normal(0.5, 1.5, size=60)
>>> y3 = np.random.normal(2.5, 1.5, size=60)

#Make the figure wider
>>> fig = plt.figure(figsize=(5,3))

#Plot the data
>>> plt.plot(x1, y1, 'r.', label="Dataset 1")
>>> plt.plot(x2, y2, 'g.', label="Dataset 2")
>>> plt.plot(x3, y3, 'b.', label="Dataset 3")

#Create a legend to the left of the plot
>>> lspace = 0.35
>>> plt.subplots_adjust(left=lspace)
#Put the legend at the left edge of the figure
>>> plt.legend(loc=(-lspace/(1-lspace),0.6))
>>> plt.show()
```

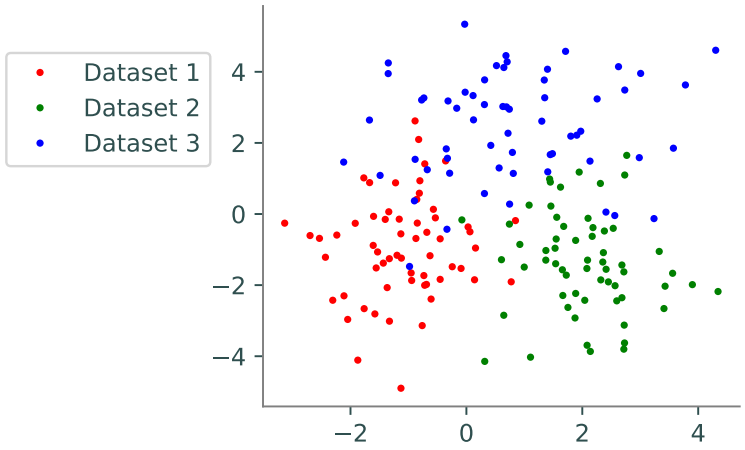


Figure B.3: Example of repositioning axes.

Colors

The color that a plotting function uses is specified by either the `c` or `color` keyword arguments; for most functions, these can be used interchangeably. There are many ways to specific colors. The most simple is to use one of the basic colors, listed in Table B.3. Colors can also be specified using an RGB tuple such as `(0.0, 0.4, 1.0)`, a hex string such as `"0000FF"`, or a CSS color name like `"DarkOliveGreen"` or `"FireBrick"`. A full list of named colors that Matplotlib supports can be found

at `https://matplotlib.org/stable/gallery/color/named_colors.html`. If no color is specified for a plot, Matplotlib automatically assigns it one from the default color cycle.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

Code	Color
`'y'`	yellow
`'k'`	black
`'w'`	white
`'C0'` - `'C9'`	Default colors

Table B.3: Basic colors available in Matplotlib

Plotting functions also accept an `alpha` keyword argument, which can be used to set the transparency. A value of `1.0` corresponds to fully opaque, and `0.0` corresponds to fully transparent.

The following example demonstrates different ways of specifying colors:

```
#Using a basic color
>>> plt.plot(x, y, 'r')
#Using a hexadecimal string
>>> plt.plot(x, y, color='FF0080')
#Using an RGB tuple
>>> plt.plot(x, y, color=(1, 0.5, 0))
#Using a named color
>>> plt.plot(x, y, color='navy')
```

Colormaps

Certain plotting functions, such as heatmaps and contour plots, accept a colormap rather than a single color. A full list of colormaps available in Matplotlib can be found at `https://matplotlib.org/stable/gallery/color/colormap_reference.html`. Some of the more commonly used ones are `"viridis"`, `"magma"`, and `"coolwarm"`. A colorbar can be added by calling `plt.colorbar()` after creating the plot.

Sometimes, using a logarithmic scale for the coloring is more informative. To do this, pass a `matplotlib.colors.LogNorm` object as the `norm` keyword argument:

```
# Create a heatmap with logarithmic color scaling
>>> from matplotlib.colors import LogNorm
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', norm=LogNorm())
```

Text and Annotations

Matplotlib has several ways to add text and other annotations to a plot, some of which are listed in Table B.4. The color and size of the text in most of these functions can be adjusted with the `color` and `fontsize` keyword arguments.

Matplotlib also supports formatting text with LaTeX, a system for creating technical documents.[1] To do so, use an `r` before the string quotation mark and surround the text with dollar

[1] See `http://www.latex-project.org/` for more information.

Function	Description	Usage
`annotate()`	adds a commentary at a given point on the plot	annotate('text',(x,y))
`arrow()`	draws an arrow from a given point on the plot	arrow(x,y,dx,dy)
`colorbar()`	Create a colorbar	colorbar()
`legend()`	Place a legend in the plot	legend(loc='best')
`text()`	Add text at a given position on the plot	text(x,y,'text')
`title()`	Add a title to the plot	title('text')
`suptitle()`	Add a title to the figure	suptitle('text')
`xlabel()`	Add a label to the x-axis	xlabel('text')
`ylabel()`	Add a label to the y-axis	ylabel('text')

Table B.4: Text and annotation functions in Matplotlib

signs. This is particularly useful when the text contains a mathematical expression. For example, the following line of code will make the title of the plot be $\frac{1}{2}\sin(x^2)$:

```
>>> plt.title(r"$\frac{1}{2}\sin(x^2)$")
```

The function `legend()` can be used to add a legend to a plot. Its optional `loc` keyword argument specifies where to place the legend within the subplot. It defaults to `'best'`, which will cause Matplotlib to place it in whichever location overlaps with the fewest drawn objects. The other locations this function accepts are `'upper right'`, `'upper left'`, `'lower left'`, `'lower right'`, `'center left'`, `'center right'`, `'lower center'`, `'upper center'`, and `'center'`. Alternately, a tuple of `(x,y)` can be passed as this argument, and the bottom-left corner of the legend will be placed at that location. The point (0,0) corresponds to the bottom-left of the current subplot, and (1,1) corresponds to the top-right. This can be used to place the legend outside of the subplot, although care should be taken that it does not go outside the figure, which may require manually repositioning the subplots.

The labels the legend uses for each curve or scatterplot are specified with the `label` keyword argument when plotting the object. Note that `legend()` can also be called with non-keyword arguments to set the labels, although it is less confusing to set them when plotting.

The following example demonstrates creating a legend:

```
>>> x = np.linspace(0,2*np.pi,250)

# Plot sin(x), cos(x), and -sin(x)
# The label argument will be used as its label in the legend.
>>> plt.plot(x, np.sin(x), 'r', label=r'$\sin(x)$')
>>> plt.plot(x, np.cos(x), 'g', label=r'$\cos(x)$')
>>> plt.plot(x, -np.sin(x), 'b', label=r'$-\sin(x)$')

# Create the legend
>>> plt.legend()
```

Line and marker styles

Matplotlib supports a large number of line and marker styles for line and scatter plots, which are listed in Table B.5.

character	description	character	description
-	solid line style	3	tri_left marker
--	dashed line style	4	tri_right marker
-.	dash-dot line style	s	square marker
:	dotted line style	p	pentagon marker
.	point marker	*	star marker
,	pixel marker	h	hexagon1 marker
o	circle marker	H	hexagon2 marker
v	triangle_down marker	+	plus marker
^	triangle_up marker	x	x marker
<	triangle_left marker	D	diamond marker
>	triangle_right marker	d	thin_diamond marker
1	tri_down marker	|	vline marker
2	tri_up marker	_	hline marker

Table B.5: Available line and marker styles in Maplotlib.

The function `plot()` has several ways to specify this argument; the simplest is to pass it as the third positional argument. The `marker` and `linestyle` keyword arguments can also be used. The size of these can be modified using `markersize` and `linewidth`. Note that by specifying a marker style but no line style, `plot()` can be used to make a scatter plot. It is also possible to use both a marker style and a line style. To set the marker using `scatter()`, use the `marker` keyword argument, with `s` being used to change the size.

The following code demonstrates specifying marker and line styles. The results are shown in Figure B.4.

```
#Use dashed lines:
>>> plt.plot(x, y, '--')
#Use only dots:
>>> plt.plot(x, y, '.')
#Use dots with a normal line:
>>> plt.plot(x, y, '.-')
#scatter() uses the marker keyword:
>>> plt.scatter(x, y, marker='+')

#With plot(), the color to use can also be specified in the same string.
#Order usually doesn't matter.
#Use red dots:
>>> plt.plot(x, y, '.r')
#Equivalent:
>>> plt.plot(x, y, 'r.')

#To change the size:
>>> plt.plot(x, y, 'v-', linewidth=1, markersize=15)
>>> plt.scatter(x, y, marker='+', s=12)
```

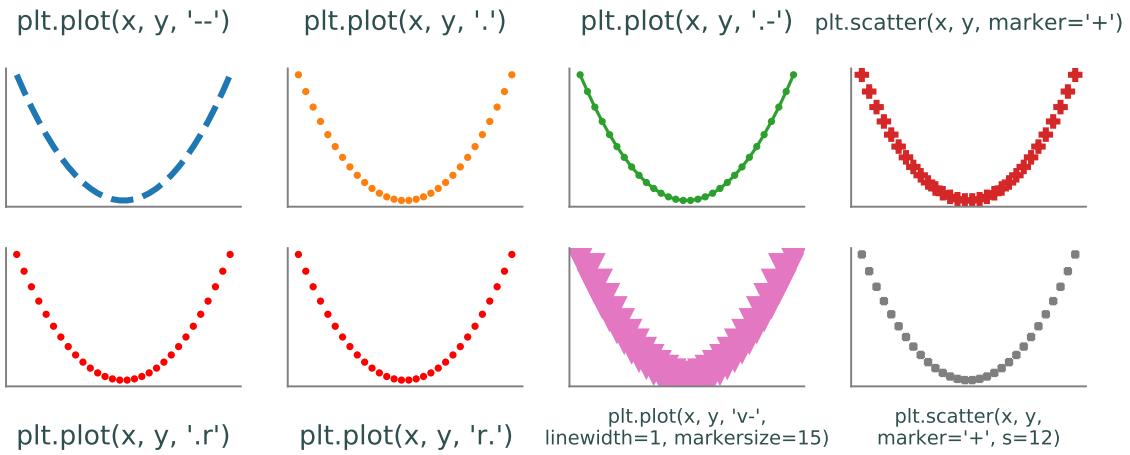


Figure B.4: Examples of setting line and marker styles.

Plot Types

Matplotlib has functions for creaing many different types of plots, many of which are listed in Table B.6. This section gives details on using certain groups of these functions.

Function	Description	Usage
`bar`	makes a bar graph	bar(x,height)
`barh`	makes a horizontal bar graph	barh(y,width)
`boxplots`	makes one or more boxplots	boxplots(data)
`contour`	makes a contour plot	contour(X,Y,Z)
`contourf`	makes a filled contour plot	contourf(X,Y,Z)
`imshow`	shows an image	imshow(image)
`fill`	plots lines with shading under the curve	fill(x,y)
`fill_between`	plots lines with shading between two given y values	fill_between(x,y1, y2=0)
`hexbin`	creates a hexbin plot	hexbin(x,y)
`hist`	plots a histogram from data	hist(data)
`pcolormesh`	makes a heatmap	pcolormesh(X,Y,Z)
`pie`	makes a pie chart	pie(x)
`plot`	plots lines and data on standard axes	plot(x,y)
`plot_surface`	plot a surface in 3-D space	plot_surface(X,Y,Z)
`polar`	plots lines and data on polar axes	polar(theta,r)
`loglog`	plots lines and data on logarithmic x and y axes	loglog(x,y)
`scatter`	plots data in a scatterplot	scatter(x,y)
`semilogx`	plots lines and data with a log scaled x axis	semilogx(x,y)
`semilogy`	plots lines and data with a log scaled y axis	semilogy(x,y)
`specgram`	makes a spectogram from data	specgram(x)
`spy`	plots the sparsity pattern of a 2D array	spy(Z)
`triplot`	plots triangulation between given points	triplot(x,y)

Table B.6: Some basic plotting functions in Matplotlib.

Line plots

Line plots, the most basic type of plot, are created with the `plot()` function. It accepts two lists of x- and y-values to plot, and optionally a third argument of a string of any combination of the color, line style, and marker style. Note that this method only works with the single-character color codes; to use other colors, use the `color` argument. By specifying only a marker style, this function can also be used to create scatterplots.

There are a number of functions that do essentially the same thing as `plot()` but also change the axis scaling, including `loglog()`, `semilogx()`, `semilogy()`, and `polar`. Each of these functions is used in the same manner as `plot()`, and has identical syntax.

Bar Plots

Bar plots are a way to graph categorical data in an effective way. They are made using the `bar()` function. The most important arguments are the first two that provide the data, `x` and `height`. The first argument is a list of values for each bar, either categorical or numerical; the second argument is a list of numerical values corresponding to the height of each bar. There are other parameters that may be included as well. The `width` argument adjusts the bar widths; this can be done by choosing a single value for all of the bars, or an array to give each bar a unique width. Further, the argument `bottom` allows one to specify where each bar begins on the y-axis. Lastly, the `align` argument can be set to 'center' or 'edge' to align as desired on the x-axis. As with all plots, you can use the `color` keyword to specify any color of your choice. If you desire to make a horizontal bar graph, the syntax follows similarly using the function `barh()`, but with argument names `y`, `width`, `height` and `align`.

Box Plots

A box plot is a way to visualize some simple statistics of a dataset. It plots the minimum, maximum, and median along with the first and third quartiles of the data. This is done by using `boxplot()` with an array of data as the argument. Matplotlib allows you to enter either a one dimensional array for a single box plot, or a 2-dimensional array where it will plot a box plot for each column of the data in the array. Box plots default to having a vertical orientation but can be easily laid out horizontally by setting `vert=False`.

Scatter and hexbin plots

Scatterplots can be created using either `plot()` or `scatter()`. Generally, it is simpler to use `plot()`, although there are some cases where `scatter()` is better. In particular, `scatter()` allows changing the color and size of individual points within a single call to the function. This is done by passing a list of colors or sizes to the `c` or `s` arguments, respectively.

Hexbin plots are an alternative to scatterplots that show the concentration of data in regions rather than the individual points. They can be created with the function `hexbin()`. Like `plot()` and `scatter()`, this function accepts two lists of x- and y-coordinates.

Heatmaps and contour plots

Heatmaps and contour plots are used to visualize 3-D surfaces and complex-valued functions on a flat space. Heatmaps are created using the `pcolormesh()` function. Contour plots are created using `contour()` or `contourf()`, with the latter creating a filled contour plot.

Each of these functions accepts the x-, y-, and z-coordinates as a mesh grid, or 2-D array. To create these, use the function `np.meshgrid()`:

```
>>> x = np.linspace(0,1,100)
>>> y = np.linspace(0,1,80)
>>> X, Y = np.meshgrid(x, y)
```

The z-coordinate can then be computed using the x and y mesh grids.

Note that each of these functions can accept a colormap, using the `cmap` parameter. These plots are sometimes more informative with a logarithmic color scale, which can be used by passing a `matplotlib.colors.LogNorm` object in the `norm` parameter of these functions.

With `pcolormesh()`, it is also necessary to pass `shading='auto'` or `shading='nearest'` to avoid a deprecation error.

The following example demonstrates creating heatmaps and contour plots, using a graph of $z = (x^2 + y)\sin(y)$. The results is shown in Figure B.5

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-3,3,100)
>>> y = np.linspace(-3,3,100)
>>> X, Y = np.meshgrid(x, y)
>>> Z = (X**2+Y)*np.sin(Y)

#Heatmap
>>> plt.subplot(1,3,1)
>>> plt.pcolormesh(X, Y, Z, cmap='viridis', shading='nearest')
>>> plt.title("Heatmap")

#Contour
>>> plt.subplot(1,3,2)
>>> plt.contour(X, Y, Z, cmap='magma')
>>> plt.title("Contour plot")

#Filled contour
>>> plt.subplot(1,3,3)
>>> plt.contourf(X, Y, Z, cmap='coolwarm')
>>> plt.title("Filled contour plot")
>>> plt.colorbar()

>>> plt.show()
```

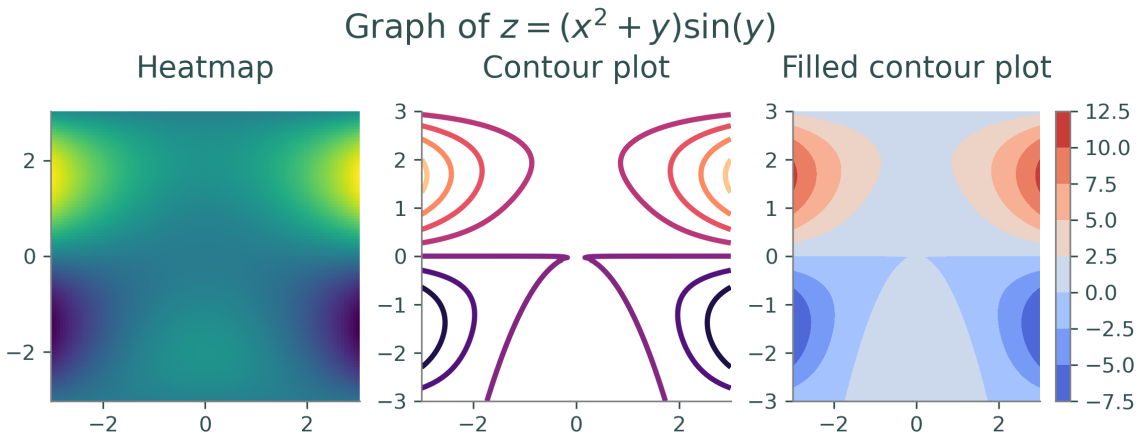


Figure B.5: Example of heatmaps and contour plots.

Showing images

The function `imshow()` is used for showing an image in a plot, and can be used on either grayscale or color images. This function accepts a 2-D $n \times m$ array for a grayscale image, or a 3-D $n \times m \times 3$ array for a color image. If using a grayscale image, you also need to specify `cmap='gray'`, or it will be colored incorrectly.

It is best to also use `axis('equal')` alongside `imshow()`, or the image will most likely be stretched. This function also works best if the images values are in the range $[0, 1]$. Some ways to load images will format their values as integers from 0 to 255, in which case the values in the image array should be scaled before using `imshow()`.

3-D Plotting

Matplotlib can be used to plot curves and surfaces in 3-D space. In order to use 3-D plotting, you need to run the following line:

```
>>> from mpl_toolkits.plot3d import Axes3D
```

The argument `projection='3d'` also must be specified when creating the subplot for the 3-D object:

```
>>> plt.subplot(1,1,1, projection='3d')
```

Curves can be plotted in 3-D space using `plot()`, by passing in three lists of x-, y-, and z-coordinates. Surfaces can be plotted using `ax.plot_surface()`. This function can be used similar to creating contour plots and heatmaps, by obtaining meshes of x- and y- coordinates from `np.meshgrid()` and using those to produce the z-axis. More generally, any three 2-D arrays of meshes corresponding to x-, y-, and z-coordinates can be used. Note that it is necessary to call this function from an Axes object.

The following example demonstrates creating 3-D plots. The results are shown in Figure B.6.

```
#Create a plot of a parametric curve
ax = plt.subplot(1,3,1, projection='3d')
```

```
t = np.linspace(0, 4*np.pi, 160)
x = np.cos(t)
y = np.sin(t)
z = t / np.pi
plt.plot(x, y, z, color='b')
plt.title("Helix curve")

#Create a surface plot from np.meshgrid
ax = plt.subplot(1,3,2, projection='3d')
x = np.linspace(-1,1,80)
y = np.linspace(-1,1,80)
X, Y = np.meshgrid(x, y)
Z = X**2 - Y**2
ax.plot_surface(X, Y, Z, color='g')
plt.title(r"Hyperboloid")

#Create a surface plot less directly
ax = plt.subplot(1,3,3, projection='3d')
theta = np.linspace(-np.pi,np.pi,80)
rho = np.linspace(-np.pi/2,np.pi/2,40)
Theta, Rho = np.meshgrid(theta, rho)
X = np.cos(Theta) * np.cos(Rho)
Y = np.sin(Theta) * np.cos(Rho)
Z = np.sin(Rho)
ax.plot_surface(X, Y, Z, color='r')
plt.title(r"Sphere")

plt.show()
```

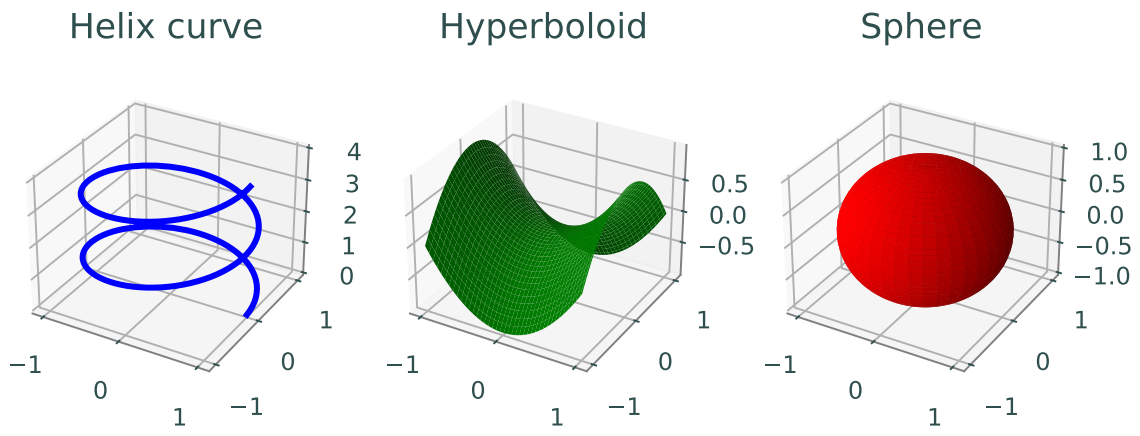


Figure B.6: Examples of 3-D plotting.

Additional Resources

rcParams

The default plotting parameters of Matplotlib can be set individually and with more fine control than styles by using `rcParams`. `rcParams` is a dictionary that can be accessed as either `plt.rcParams` or `matplotlib.rcParams`.

For instance, the resolution of plots can be changed via the `"figure.dpi"` parameter:

```
>>> plt.rcParams["figure.dpi"] = 600
```

A list of parameters that can set via `rcParams` can be found at `https://matplotlib.org/stable/api/matplotlib_configuration_api.html#matplotlib.RcParams`.

Animations

Matplotlib has capabilities for creating animated plots. The Animations lab in Volume 4 has detailed instructions on how to do so.

Matplotlib gallery and tutorials

The Matplotlib documentation has a number of tutorials, found at `https://matplotlib.org/stable/tutorials/index.html`. It also has a large gallery of examples, found at `https://matplotlib.org/stable/gallery/index.html`. Both of these are excellent sources of additional information about ways to use and customize Matplotlib.

Bibliography

[Hea02] Michael T Heath. *Scientific computing*. McGraw-Hill New York, 2002.