C. Linear Algebra

You cannot learn too much linear algebra.
– Every mathematician

C.1. Intro to Numerical Linear Algebra

The preceding comment says it all – linear algebra is the most important of all of the mathematical tools that you can learn as a practitioner of the mathematical sciences. The theorems, proofs, conjectures, and big ideas in almost every other mathematical field find their roots in linear algebra. Numerical Linear Algebra is the study of algorithms for solving problems in linear algebra. This subject has a somewhat different flavour than the numerical analysis we are studying in the main text and hence we present it as an optional appendix.

Our goal in this appendix is to explore numerical algorithms for the primary questions of linear algebra:

- solving systems of equations,
- finding eigenvalue-eigenvector pairs for a matrix.

Take careful note that in our current digital age, numerical linear algebra and its fast algorithms are behind the scenes for wide varieties of computing applications. Applications of numerical linear algebra include:

- building neural networks and AI algorithms,
- determining the most important web page in a Google search,
- modelling realistic 3D environments in video games,
- digital image processing,
- and many many more.

What's more, researchers have found provably optimal ways to perform most of the typical tasks of linear algebra so most scientific software works very well and very quickly with linear algebra.

C.2. Notation

Throughout this chapter we will use the following notation conventions.

- A bold mathematical symbol such as x or u will represent a vector.
- If u is a vector then u_j will be the jth entry of the vector.
- Vectors will typically be written vertically with parenthesis as delimiters such as

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}. \tag{C.1}$$

- Two bold symbols separated by a centred dot such as $u \cdot v$ will represent the dot product of two vectors.
- A capital mathematical symbol such as A or X will represent a matrix
- If A is a matrix then A_{ij} will be the element in the ith row and j^{th} column of the matrix.
- A matrix will typically be written with parenthesis as delimiters such as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & \pi \end{pmatrix}. \tag{C.2}$$

- The juxtaposition of a capital symbol and a bold symbol such as Ax will represent matrix-vector multiplication.
- A lower case or Greek mathematical symbol such as x, c, or λ will represent a scalar.
- The scalar field of real numbers is given as $\mathbb{R}$ and the scalar field of complex numbers is given as $\mathbb{C}$.
- The symbol $\mathbb{R}^n$ represents the collection of all n-dimensional vectors where the elements are drawn from the real numbers.
- The symbol $\mathbb{C}^n$ represents the collection of all n-dimensional vectors where the elements are drawn from the complex numbers.

It is an important part of learning to read and write linear algebra to give special attention to the symbolic language so you can communicate your work easily and efficiently.

C.3. Vectors and Matrices in Python

We first need to understand how Python's `numpy` library builds and stores vectors and matrices. The following exercises will give you some experience building and working with these data structures and will point out some common pitfalls that mathematicians fall into when using Python for linear algebra.

Example C.1 (numpy Arrays)**.** In Python you can build a list using square brackets such as `[1,2,3]`. This is called a "Python list" and is NOT a vector in the way that we think about it mathematically. It is simply an ordered collection of objects. To build mathematical vectors in Python we need to use `numpy` arrays with `np.array()`. For example, the vector

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{C.3}$$

would be built with the following code.

```
import numpy as np
u = np.array([1,2,3])
print(u)
```

Notice that Python defines the vector `u` as a matrix with only one dimension. You can see that in the following code.

```
import numpy as np
u = np.array([1,2,3])
print("The length of the u vector is \n",len(u))
print("The shape of the u vector is \n",u.shape)
```

Example C.2 (numpy Matrices)**.** In `numpy`, a matrix is validly built as a list of lists. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{C.4}$$

is defined using `np.array()` where each row is an individual list, and the matrix is a collection of these lists.

```
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

Moreover, we can extract the shape, the number of rows, and the number of columns of A using the `A.shape` command. To be a bit more clear on this one we will use the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{C.5}$$

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
print("The shape of the A matrix is \n",A.shape)
print("Number of rows in A is \n",A.shape[0])
print("Number of columns in A is \n",A.shape[1])
```

Example C.3 (Row and Column Vectors in Python)**.** You can more specifically build row or column vectors in Python using the `np.array()` command and then only specifying one row or column. But take careful note that `numpy` treats a 1D array (like `np.array([1,2,3])`) differently from a 2D array (like `np.array([[1,2,3]])`). For example, if you want the vectors

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \tag{C.6}$$

then we would use the following Python code.

```
import numpy as np
u = np.array([[1],[2],[3]])
print("The column vector u is \n",u)
v = np.array([[1,2,3]])
print("The row vector v is \n",v)
```

Alternatively, if you want to define a column vector you can define a row vector (since there are far fewer brackets to keep track of) and then transpose the matrix to turn it into a column. Note that the `.transpose()` method (or `.T`) only swaps dimensions; it won't turn a 1D array into a 2D column vector unless it was already 2D.

```
import numpy as np
u = np.array([[1,2,3]])
u = u.transpose()
print("The column vector u is \n",u)
```

Example C.4 (Matrix Indexing)**.** Python indexes all arrays, vectors, lists, and matrices starting from index 0. Let us get used to this fact.

Consider the matrix A defined in the previous problem. Mathematically we know that the entry in row 1 column 1 is a 1, the entry in row 1 column 2 is a 2, and so on. However, with Python we need to shift the way that we enumerate the rows and columns of a matrix. Hence we would say that the entry in row 0 column 0 is a 1, the entry in row 0 column 1 is a 2, and so on.

Mathematically we can view all Python matrices as follows. If A is an $n \times n$ matrix then

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & A_{n-1,2} & \cdots & A_{n-1,n-1} \end{pmatrix} \tag{C.7}$$

Similarly, we can view all vectors as follows. If u is an $n \times 1$ vector then

$$u = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} \tag{C.8}$$

The following code should help to illustrate this indexing convention.

```
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("Entry in row 0 column 0 is",A[0,0])
print("Entry in row 0 column 1 is",A[0,1])
print("Entry in the bottom right corner",A[2,2])
```

Exercise C.1. Build your own matrix in Python and practice choosing individual entries from the matrix.

Example C.5 (Matrix Slicing)**.** The last thing that we need to be familiar with is *slicing* a matrix. The term "slicing" generally refers to pulling out individual rows, columns, entries, or blocks from a list, array, or matrix in Python. Examine the code below to see how to slice parts out of a `numpy` matrix.

```
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A)
print("The first column of A is \n",A[:,0])
print("The second row of A is \n",A[1,:])
print("The top left 2x2 sub matrix of A is \n",A[:-1,:-1])
print("The bottom right 2x2 sub matrix of A is \n",A[1:,1:])
u = np.array([1,2,3,4,5,6])
print("The first 3 entries of the vector u are \n",u[:3])
print("The last entry of the vector u is \n",u[-1])
print("The last two entries of the vector u are \n",u[-2:])
```

Exercise C.2. Define the matrix A and the vector u in Python. Then perform all of the tasks below.

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ -3 & -2 & -1 & 0 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix} \tag{C.9}$$

1. Print the matrix A, the vector u, the shape of A, and the shape of u.
2. Print the first column of A.
3. Print the first two rows of A.
4. Print the first two entries of u.
5. Print the last two entries of u.
6. Print the bottom left 2×2 submatrix of A.
7. Print the middle two elements of the middle row of A.

C.4. Matrix and Vector Operations

Now let us start doing some numerical linear algebra. We start our discussion with the basics: the dot product and matrix multiplication. The numerical routines in Python's `numpy` packages are designed to do these tasks in very efficient ways but it is a good coding exercise to build your own dot product and matrix multiplication routines just to further cement the way that Python deals with these data structures and to remind you of the mathematical algorithms. What you will find in numerical linear algebra is that the indexing and the housekeeping in the codes is the hardest part. So why do not we start "easy."

C.4.1. The Dot Product

Exercise C.3. This problem is meant to jog your memory about dot products, how to compute them, and what you might use them for. If your linear algebra is a bit rusty then read ahead a bit and then come back to this problem.

Consider two vectors u and v defined as

$$u = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 3 \\ 4 \end{pmatrix}. \tag{C.10}$$

1. Draw a picture showing both u and v.
2. What is $u \cdot v$?
3. What is $\|u\|$?
4. What is $\|v\|$?
5. What is the angle between u and v?
6. Give two reasons why we know that u is not perpendicular to v.
7. What is the scalar projection of u onto v? Draw this scalar projections on your picture from part (1).
8. What is the scalar projection of v onto u? Draw this scalar projections on your picture from part (1).

Now let us get the formal definitions of the dot product on the table.

Definition C.1 (Dot product)**.** The **dot product** of two vectors $u, v \in \mathbb{R}^n$ is

$$u \cdot v = \sum_{j=1}^{n} u_j v_j. \tag{C.11}$$

Without summation notation the dot product of two vectors is ,

$$u \cdot v = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n. \tag{C.12}$$

You may also recall that the dot product of two vectors is given geometrically as

$$u \cdot v = \|u\| \|v\| \cos\theta \tag{C.13}$$

where $\|u\|$ and $\|v\|$ are the magnitudes (or lengths) of u and v respectively, and θ is the angle between the two vectors. In physical applications the dot product is often used to find the angle between two vectors (e.g. between two forces). Hence, the last form of the dot product is often rewritten as

$$\theta = \cos^{-1}\left(\frac{u \cdot v}{\|u\| \|v\|} \right). \tag{C.14}$$

Definition C.2 (Magnitude of a Vector)**.** The **magnitude** of a vector $u \in \mathbb{R}^n$ is defined as [1]

$$\|u\| = \sqrt{u \cdot u}. \tag{C.15}$$

Exercise C.4. Write a Python function that accepts two vectors (defined as `numpy` arrays) and returns the dot product. Write this code without the use any loops.

```
import numpy as np
def myDotProduct(u,v):
    return # the dot product formula uses a product inside a sum.
```

---

[1] You should also note that $\|u\| = \sqrt{u \cdot u}$ is not the only definition of distance. More generally, if you let $\langle u, v \rangle$ be an inner product for u and v in some vector space $\mathcal{V}$ then $\|u\| = \sqrt{\langle u, u \rangle}$. In most cases in this text we will be using the dot product as our preferred inner product so we will not have to worry much about this particular natural extension of the definition of the length of a vector.

Exercise C.5. Test your `myDotProduct()` function on several dot products to make sure that it works. Example code to find the dot product between

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \tag{C.16}$$

is given below. Test your code on other vectors. Then implement an error catch into your code to catch the case where the two input vectors are not the same size. You will want to use the `len()` command to find the length of the vectors.

```
u = np.array([1,2,3])
v = np.array([4,5,6])
myDotProduct(u,v)
```

Exercise C.6. Try sending Python lists instead of `numpy` arrays into your `myDotProduct` function. What happens? Why does it happen? What is the cautionary tale here? Modify your `myDotProduct()` function one more time so that it starts by converting the input vectors into `numpy` arrays.

```
u = [1,2,3]
v = [4,5,6]
myDotProduct(u,v)
```

Exercise C.7. The `numpy` library in Python has a built-in command for doing the dot product: `np.dot()`. Test the `np.dot()` command and be sure that it does the same thing as your `myDotProduct()` function.

C.4.2. Matrix Multiplication

Next we will blow the dust off of your matrix multiplication skills.

Exercise C.8. Verify that the product of A and B is indeed what we show below. Work out all of the details by hand.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \tag{C.17}$$

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix} \tag{C.18}$$

Now that you have practised the algorithm for matrix multiplication we can formalize the definition and then turn the algorithm into a Python function.

Definition C.3 (Matrix Multiplication)**.** If A and B are matrices with $A \in \mathbb{R}^{n\times p}$ and $B \in \mathbb{R}^{p\times m}$ then the product AB is defined as

$$(AB)_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj}. \tag{C.19}$$

A moment's reflection reveals that each entry in the matrix product is actually a dot product,

$$(\text{Entry in row } i \text{ column } j \text{ of } AB) = (\text{Row } i \text{ of matrix } A) \cdot (\text{Column } j \text{ of matrix } B)\,. \tag{C.20}$$

Exercise C.9. The definition of matrix multiplication above contains the cryptic phrase *a moment's reflection reveals that each entry in the matrix product is actually a dot product.* Let us go back to the matrices A and B defined in Exercise C.8 above and re-evaluate the matrix multiplication algorithm to make sure that you see each entry as the end result of a dot product.

We want to find the product of matrices A and B using dot products.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \tag{C.21}$$

1. Why will the product AB clear be a 3×3 matrix?

2. When we do matrix multiplication we take the product of a row from the first matrix times a column from the second matrix … at least that's how many people think of it when they perform the operation by hand.

    a. The rows of A can be written as the vectors

$$a_0 = \begin{pmatrix} 1 & 2 \end{pmatrix} \tag{C.22}$$

$$a_1 = (______ \quad ______) \tag{C.23}$$

$$a_2 = (______ \quad ______) \tag{C.24}$$

    b. The columns of B can be written as the vectors

$$b_0 = \begin{pmatrix} 7 \\ 10 \end{pmatrix} \tag{C.25}$$

$$b_1 = \begin{pmatrix} ______ \\ ______ \end{pmatrix} \tag{C.26}$$

$$b_2 = \begin{pmatrix} ______ \\ ______ \end{pmatrix} \tag{C.27}$$

3. Now let us write each entry in the product AB as a dot product.

$$AB = \begin{pmatrix} a_0 \cdot b_0 & ___ \cdot ___ & ___ \cdot ___ \\ ___ \cdot ___ & ___ \cdot ___ & ___ \cdot ___ \\ ___ \cdot ___ & ___ \cdot ___ & ___ \cdot ___ \end{pmatrix} \tag{C.28}$$

4. Verify that you get

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix} \tag{C.29}$$

    when you perform all of the dot products from part (3).

Exercise C.10. The observation that matrix multiplication is just a bunch of dot products is what makes the code for doing matrix multiplication very fast and very streamlined. We want to write a Python function that accepts two `numpy` matrices and returns the product of the two matrices. Inside the code we will leverage the `np.dot()` command to do the appropriate dot products.

Partial code is given below. Fill in all of the details and give ample comments showing what each line does.

```python
import numpy as np
def myMatrixMult(A,B):
    # Get the shapes of the matrices A and B.
    # Then write an if statement that catches size mismatches
    # in the matrices.  Next build a zeros matrix that is the
    # correct size for the product of A and B.
    AB = ???
    # AB is a zeros matix that will be filled with the values
    # from the product
    #
    # Next we do a double for-loop that loops through all of
    # the indices of the product
    for i in range(n): # loop over the rows of AB
        for j in range(m): # loop over the columns of AB
            # use the np.dot() command to take the dot product
            AB[i,j] = ???
    return AB
```

Use the following test code to determine if you actually get the correct matrix product out of your code.

```
``` python
A = np.array([[1,2],[3,4],[5,6]])
B = np.array([[7,8,9],[10,11,12]])
AB = myMatrixMult(A,B)
print(AB)
```

Exercise C.11. Try your `myMatrixMult()` function on several other matrix multiplication problems.

```

Exercise C.12. Build in an error catch so that your `myMatrixMult()` function catches when the input matrices do not have compatible sizes for multiplication. Write your code so that it returns an appropriate error message in this special case.

Now that you have been through the exercise of building a matrix multiplication function we will admit that using it inside larger coding problems would be a bit cumbersome (and perhaps annoying). It would be nice to just type `@` and have Python just *know* that you mean to do matrix multiplication. This is where `numpy` arrays come in quite handy.

Exercise C.13 (Matrix Multiplication with Python)**.** Python will handle matrix multiplication easily so long as the matrices are defined as `numpy` arrays. For example, with the matrices A and B from above if you can just type `A @ B` (or `np.dot(A, B)`) in Python and you will get the correct result. Pretty nice!! Let us take another moment to notice, though, that regular Python lists do not behave in the same way. Can you guess what happens if you run the following Python code? (Note: `*` does strictly element-wise multiplication for numpy arrays, which we will see in a moment).

```
A = [[1,2],[3,4],[5,6]] # a Python list of lists
B = [[7,8,9],[10,11,12]] # a Python list of lists
A @ B
```

Example C.6 (Element-by-Element Multiplication)**.** Sometimes it is convenient to do naive multiplication of matrices when you code. That is, if you have two matrices that are the same size, "naive multiplication" would just line up the matrices on top of each other and multiply the corresponding entries.[2] In Python you can do this with the `*` operator (or `np.multiply()`). The code below demonstrates this tool with the matrices

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}. \tag{C.30}$$

[2]You might have thought that *naive multiplication* was a much more natural way to do matrix multiplication when you first saw it. Hopefully now you see the power in the definition of matrix multiplication that we actually use. If not, then I give you this moment to ponder that (a) matrix multiplication is just a bunch of dot products, and (b) dot products can be seen as projections. Hence, matrix multiplication is really just a projection of the rows of A onto the columns of B. This has much more rich geometric flavour than *naive multiplication*.

(Note that the product AB does not make sense under the mathematical definition of matrix multiplication, but it does make sense in terms of element-by-element ("naive") multiplication.)

```
import numpy as np
A = np.array([[1,2],[3,4],[5,6]])
B = np.array([[7,8],[9,10],[11,12]])
print(A * B)
```

The key takeaways for doing matrix multiplication in Python are as follows:

- If you are doing linear algebra in Python then you should define vectors and matrices with `np.array()`.
- If your matrices are defined with `np.array()` then `@` (or `np.dot()`) does regular matrix multiplication and `*` does element-by-element multiplication.

C.5. The LU Factorization

One of the many classic problems of linear algebra is to solve the linear system $Ax = b$ where A is a matrix of coefficients and b is a vector of right-hand sides. You likely recall your go-to technique for solving systems was row reduction (or Gaussian Elimination). Furthermore, you likely rarely actually did row reduction by hand, and instead you relied on a computer to do most of the computations for you. Just what was the computer doing, exactly? Do you think that it was actually following the same algorithm that you did by hand?

C.5.1. A Recap of Row Reduction

Let us blow the dust off your row reduction skills before we look at something better.

Exercise C.14. Solve the following system of equations by hand.

$$\begin{aligned} x_0 + 2x_1 + 3x_2 &= 1 \\ 4x_0 + 5x_1 + 6x_2 &= 0 \\ 7x_0 + 8x_1 &= 2 \end{aligned} \tag{C.31}$$

Note that the system of equations can also be written in the matrix form

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.32}$$

If you need a nudge to get started then jump ahead to the next problem.

Exercise C.15. We want to solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.33}$$

Row Reduction Process:

Note: Throughout this discussion we use Python-type indexing so the rows and columns are enumerated starting at 0. That is to say, we will talk about row 0, row 1, and row 2 of a matrix instead of rows 1, 2, and 3.

1. Augment the coefficient matrix and the vector on the right-hand side to get

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array} \right) \tag{C.34}$$

2. The goal of row reduction is to perform elementary row operations until our augmented matrix gets to (or at least gets as close as possible to)

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \star \\ 0 & 1 & 0 & \star \\ 0 & 0 & 1 & \star \end{array} \right) \tag{C.35}$$

   The allowed elementary row operations are:

   a. We are allowed to scale any row.

   b. We can add two rows.

c. We can interchange two rows.

3. We are going to start with column 0. We already have the "1" in the top left corner so we can use it to eliminate all of the other values in the first column of the matrix.

   a. For example, if we multiply the 0th row by -4 and add it to the first row we get
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 7 & 8 & 0 & 2 \end{array}\right). \tag{C.36}$$

   b. Multiply row 0 by a scalar and add it to row 2. Your end result should be
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 0 & -6 & -21 & -5 \end{array}\right). \tag{C.37}$$
   What did you multiply by? Why?

4. Now we should deal with column 1.

   a. We want to get a 1 in row 1 column 1. We can do this by scaling row 1. What did you scale by? Why? Your end result should be
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & -6 & -21 & -5 \end{array}\right). \tag{C.38}$$

   b. Now scale row 1 by something and add it to row 0 so that the entry in row 0 column 1 becomes a 0.

   c. Next scale row 1 by something and add it to row 2 so that the entry in row 2 column 1 becomes a 0.

   d. At this point you should have the augmented system
   $$\left(\begin{array}{ccc|c} 1 & 0 & -1 & -\frac{5}{3} \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & 0 & -9 & 3 \end{array}\right). \tag{C.39}$$

5. Finally we need to work with column 2.

   a. Make the value in row 2 column 2 a 1 by scaling row 2. What did you scale by? Why?

   b. Scale row 2 by something and add it to row 1 so that the entry in row 1 column 2 becomes a 0. What did you scale by? Why?

c. Scale row 2 by something and add it to row 0 so that the entry in row 0 column 2 becomes a 0. What did you scale by? Why?

d. By the time you have made it this far you should have the system

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -\frac{1}{3} \end{array}\right) \tag{C.40}$$

and you should be able to read off the solution to the system.

6. You should verify your answer in two different ways:

   a. If you substitute your values into the original system then all of the equal signs should be true. Verify this.

   b. If you substitute your values into the matrix equation and perform the matrix-vector multiplication on the left-hand side of the equation you should get the right-hand side of the equation. Verify this.

Exercise C.16. Summarize the process for doing Gaussian Elimination to solve a square system of linear equations.

C.5.2. The LU Decomposition

You may have used the `rref()` command either on a calculator in other software to perform row reduction in the past. You will be surprised to learn that there is no `rref()` command in Python's `numpy` library! That's because there are far more efficient and stable ways to solve a linear system on a computer. There is an `rref` command in Python's `sympy` (symbolic Python) library, but given that it works with symbolic algebra it is quite slow.

In solving systems of equations we are interested in equations of the form $Ax = b$. Notice that the b vector is just along for the ride, so to speak, in the row reduction process since none of the values in b actually cause you to make different decisions in the row reduction algorithm. Hence, we only really need to focus on the matrix A. Furthermore, let us change our awfully restrictive view of always seeking a matrix of the form

$$\left(\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \star \\ 0 & 1 & \cdots & 0 & \star \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \star \end{array}\right) \tag{C.41}$$

and instead say:

What if we just row reduce until the system is simple enough to solve by hand?

That's what the next several exercises are going to lead you to. Our goal here is to develop an algorithm that is fast to implement on a computer and simultaneously performs the same basic operations as row reduction for solving systems of linear equations.

Exercise C.17. Let A be defined as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}. \tag{C.42}$$

1. The first step in row reducing A would be to multiply row 0 by -4 and add it to row 1. Do this operation by hand so that you know what the result is supposed to be. Check out the following amazing observation. Define the matrix L_1 as follows:

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{C.43}$$

   Now multiply L_1 and A.

$$L_1 A = \begin{pmatrix} ___ & ___ & ___ \\ ___ & ___ & ___ \\ ___ & ___ & ___ \end{pmatrix} \tag{C.44}$$

   What just happened?!

2. Let us do it again. The next step in the row reduction of your result from part (1) would be to multiply row 0 by -7 and add to row 2. Again, do this by hand so you know what the result should be. Then define the matrix L_2 as

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix} \tag{C.45}$$

   and find the product $L_2 (L_1 A)$.

$$L_2 (L_1 A) = \begin{pmatrix} ___ & ___ & ___ \\ ___ & ___ & ___ \\ ___ & ___ & ___ \end{pmatrix} \tag{C.46}$$

   Pure insanity!!

3. Now let us say that you want to make the entry in row 2 column 1 into a 0 by scaling row 1 by something and then adding to row 2. Determine what the scalar would be and then determine which matrix, call it L_3, would do the trick so that $L_3(L_2L_1A)$ would be the next row reduced step.

$$L_3 = \begin{pmatrix} 1 & ___ & ___ \\ ___ & 1 & ___ \\ ___ & ___ & 1 \end{pmatrix} \tag{C.47}$$

$$L_3\left(L_2L_1A\right) = \begin{pmatrix} ___ & ___ & ___ \\ ___ & ___ & ___ \\ ___ & ___ & ___ \end{pmatrix} \tag{C.48}$$

Exercise C.18. Apply the same idea from the previous problem to do the first three steps of row reduction to the matrix

$$A = \begin{pmatrix} 2 & 6 & 9 \\ -6 & 8 & 1 \\ 2 & 2 & 10 \end{pmatrix} \tag{C.49}$$

Exercise C.19. Now let us make a few observations about the two previous problems.

1. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{C.50}$$

   do?

2. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ c & 0 & 1 \end{pmatrix} \tag{C.51}$$

   do?

3. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & c & 1 \end{pmatrix} \tag{C.52}$$

   do?

4. More generally: If you wanted to multiply row j of an $n \times n$ matrix by c and add it to row k, that is the same as multiplying by what matrix?

Exercise C.20. After doing all of the matrix products, $L_3 L_2 L_1 A$, the resulting matrix will have zeros in the entire lower triangle. That is, all of the non-zero entries of the resulting matrix will be on the main diagonal or above. We call this matrix U, for upper-triangular. Hence, we have formed a matrix

$$L_3 L_2 L_1 A = U \tag{C.53}$$

and if we want to solve for A we would get

$$A = (______)^{-1}(______)^{-1}(______)^{-1}U \tag{C.54}$$

(Take care that everything is in the right order in your answer.)

Exercise C.21. It would be nice, now, if the inverses of the L matrices were easy to find. Use `np.linalg.inv()` to directly compute the inverse of L_1, L_2, and L_3 for each of the example matrices. Then complete the statement: If L_k is an identity matrix with some non-zero c in row i and column j then L_k^{-1} is what matrix?

Exercise C.22. We started this discussion with A as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \tag{C.55}$$

and we defined

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix}, \quad \text{and} \quad L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}. \tag{C.56}$$

Based on your answer to the previous exercises we know that

$$A = L_1^{-1} L_2^{-1} L_3^{-1} U. \tag{C.57}$$

Explicitly write down the matrices L_1^{-1}, L_2^{-1}, and L_3^{-1}.

Now explicitly find the product $L_1^{-1} L_2^{-1} L_3^{-1}$ and call this product L. Verify that L itself is also a lower-triangular matrix with ones on the main diagonal. Moreover, take note of exactly the form of the matrix. The answer should be super surprising to you!!

Throughout all of the preceding exercises, our final result is that we have factored the matrix A into the product of a lower-triangular matrix and an upper-triangular matrix. Stop and think about that for a minute ... we just factored a matrix!

Let us return now to our discussion of solving the system of equations $Ax = b$. If A can be factored into $A = LU$ then the system of equations can be rewritten as $LUx = b$. As we will see in the next subsection, solving systems of equations with triangular matrices is super fast and relatively simple! Hence, we have partially achieved our modified goal of reducing the row reduction into some simpler case.[3]

It remains to implement the LU decomposition (also called the LU factorization) in Python.

Example C.7 (The LU Factorization)**.** The following Python function takes a square matrix A and outputs the matrices L and U such that $A = LU$. The entire code is given to you. It will be up to you in the next exercise to pick apart every step of the function.

```
def myLU(A):
    n = A.shape[0] # get the dimension of the matrix A
    L = np.eye(n) # Build the identity part of L
    U = np.copy(A) # start the U matrix as a copy of A
    for j in range(0,n-1):
        for i in range(j+1,n):
            mult = U[i,j] / U[j,j]
            U[i, :] = U[i, :] - mult * U[j,:]
            L[i,j] = mult
    return L,U
```

---

[3]Take careful note here. We have actually just built a special case of the LU decomposition. Remember that in row reduction you are allowed to swap the order of the rows, but in our LU algorithm we do not have any row swaps. The version of LU with row swaps is called LU with partial pivoting. We will not built the full partial pivoting algorithm in this text but feel free to look it up. The wikipedia page is a decent place to start. What you will find is that there are indeed many different versions of the LU decomposition.

Exercise C.23. Go to Example C.7 and go through every iteration of every loop **by hand** starting with the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}. \tag{C.58}$$

Give details of what happens at every step of the algorithm. I will get you started.

- `n=3`, `L` starts as an identity matrix of the correct size, and `U` starts as a copy of `A`.
- Start the outer loop: `j=0`: (`j` is the counter for the column)
  - Start the inner loop: `i=1`: (`i` is the counter for the row)
    * `mult = A[1,0] / A[0,0]` so `mult`$= 4/1$.
    * `A[1, 1:3] = A[1, 1:3] - 4 * A[0,1:3]`. Translated, this states that columns 1 and 2 of matrix A took their original value minus 4 times the corresponding values in row 0.
    * `U[1, 1:3] = A[1, 1:3]`. Now we replace the locations in U with the updated information from our first step of row reduction.
    * `L[1,0]=4`. We now fill the L matrix with the proper value.
    * `U[1,0]=0`. Finally, we zero out the lower triangle piece of the U matrix which we have now taken care of.
  - `i=2`:
    * ... keep going from here ...

Exercise C.24. Apply your new `myLU` code to other square matrices and verify that indeed A is the product of the resulting L and U matrices. You can produce a random matrix with `np.random.randn(n,n)` where `n` is the number of rows and columns of the matrix. For example, `np.random.randn(10,10)` will produce a random 10×10 matrix with entries chosen from the normal distribution with centre 0 and standard deviation 1. Random matrices are just as good as any other when testing your algorithm.

C.5.3. Solving Triangular Systems

We now know that row reduction is just a collection of sneaky matrix multiplications. In the previous exercises we saw that we can often turn our system of equations $Ax = b$ into the system $LUx = b$ where L is lower-triangular (with ones on the main diagonal) and U is upper-triangular. But why was this important?

Well, if $LUx = b$ then we can rewrite our system of equations as two systems:

$$\text{An upper-triangular system: } Ux = y \tag{C.59}$$

and

$$\text{A lower-triangular system: } Ly = b. \tag{C.60}$$

In the following exercises we will devise algorithms for solving triangular systems. After we know how to work with triangular systems we will put all of the pieces together and show how to leverage the LU decomposition and the solution techniques for triangular systems to quickly and efficiently solve linear systems.

Exercise C.25. Outline a fast algorithm (without formal row reduction) for solving the lower-triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}. \tag{C.61}$$

Exercise C.26. As a convention we will always write our lower-triangular matrices with ones on the main diagonal. Generalize your steps from the previous exercise so that you have an algorithm for solving any lower-triangular system. The most natural algorithm that most people devise here is called **forward substitution**.

Definition C.4 (Forward Substutition Algorithm (`lsolve`))**.** The general statement of the Forward Substitution Algorithm is:

Solve $Ly = b$ for y, where the matrix L is assumed to be lower-triangular with ones on the main diagonal.

The code below gives a full implementation of the **Forward Substitution** algorithm (also called the `lsolve` algorithm).

```
def lsolve(L, b):
    # L is assumed to be a lower-triangular np.array
    n = b.size # what does this do?
    y = np.zeros(n) # what does this do?
    for i in range(n):
        # start the loop by assigning y to the value on the right
        y[i] = b[i]
        for j in range(i): # now adjust y
            y[i] = y[i] - L[i,j] * y[j]
    return(y)
```

Exercise C.27. Work with your partner(s) to apply the `lsolve()` code to the lower-triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.62}$$

by hand. It is incredibly important to implement numerical linear algebra routines by hand a few times so that you truly understand how everything is being tracked and calculated.

I will get you started.

- Start: `i=0`:
  - `y[0]=1` since `b[0]=1`.
  - The next `for` loop does not start since `range(0)` has no elements (stop and think about why this is).
- Next step in the loop: `i=1`:
  - `y[1]` is initialized as 0 since `b[1]=0`.
  - Now we enter the inner loop at `j=0`:
    - What does `y[1]` become when `j=0`?
  - Does `j` increment to anything larger?
- Finally we increment `i` to `i=2`:
  - What does `y[2]` get initialized to?

- Enter the inner loop at `j=0`:
  - What does `y[2]` become when `j=0`?
- Increment the inner loop to `j=1`:
  - What does `y[2]` become when `j=1`?

- Stop

Exercise C.28. Copy the code from Definition C.4 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates a lower-triangular matrix of the correct form and a right-hand side b and solve for y. Test your code by giving it a large lower-triangular system.

Now that we have a method for solving lower-triangular systems, let us build a similar method for solving upper-triangular systems. The merging of lower and upper-triangular systems will play an important role in solving systems of equations.

Exercise C.29. Outline a fast algorithm (without formal row reduction) for solving the upper-triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix} \tag{C.63}$$

The most natural algorithm that most people devise here is called **backward substitution**. Notice that in our upper-triangular matrix we do not have a diagonal containing all ones.

Exercise C.30. Generalize your backward substitution algorithm from the previous problem so that it could be applied to any upper-triangular system.

Definition C.5 (Backward Substitution Algorithm)**.** The following code solves the problem $Ux = y$ using backward substitution. The matrix U is assumed to be upper-triangular. You will notice that most of this code is incomplete. It is your job to complete this code, and the next exercise should help.

```
def usolve(U, y):
    # U is assumed to be an upper-triangular np.array
    n = y.size
    x = np.zeros(n)
    for i in range( ??? ):     # what should we be looping over?
        x[i] = y[i] / ???      # what should we be dividing by?
        for j in range( ??? ): # what should we be looping over:
            x[i] = x[i] - U[i,j] * x[j] / ??? # complete this line
            # ... what does the previous line do?
    return(x)
```

Exercise C.31. Now we will work through the backward substitution algorithm to help fill in the blanks in the code. Consider the upper-triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix} \tag{C.64}$$

Work the code from Definition C.5 to solve the system. Keep track of all of the indices as you work through the code. You may want to work this problem in conjunction with the previous two problems to unpack all of the parts of the *backward substitution* algorithm.

I will get you started.

- In your backward substitution algorithm you should have started with the last row, therefore the outer loop starts at `n-1` and reads backward to 0. (Why are we starting at `n-1` and not `n`?)
- Outer loop: `i=2`:
  - We want to solve the equation $-9x_2 = 3$ so the clear solution is to divide by -9. In code this means that `x[2]=y[2]/U[2,2]`.
  - There is nothing else to do for row 3 of the matrix, so we should not enter the inner loop. How can we keep from entering the inner loop?
- Outer loop: `i=1`:

  - Now we are solving the algebraic equation $-3x_1 - 6x_2 = -4$. If we follow the high school algebra we see that $x_1 = \frac{-4-(-6)x_2}{-3}$ but this can be rearranged to

$$x_1 = \frac{-4}{-3} - \frac{-6x_2}{-3}. \tag{C.65}$$

    So we can initialize x_1 with $x_1 = \frac{-4}{-3}$. In code, this means that we initialize with `x[1] = y[1] / U[1,1]`.

  - Now we need to enter the inner loop at `j=2`: (why are we entering the loop at `j=2`?)
    * To complete the algebra we need to take our initialized value of `x[1]` and subtract off $\frac{-6x_2}{-3}$. In code this is `x[1] = x[1] - U[1,2] * x[2] / U[1,1]`
  - There is nothing else to do so the inner loop should end.

- Outer loop: `i=0`:
  - Finally, we are solving the algebraic equation $x_0 + 2x_1 + 3x_2 = 1$ for x_0. The clear and obvious solution is $x_0 = \frac{1-2x_1-3x_2}{1}$ (why am I explicitly showing the division by 1 here?).
  - Initialize x_0 at `x[0] = ???`
  - Enter the inner loop at `j=2`:
    * Adjust the value of `x[0]` by subtracting off $\frac{3x_2}{1}$. In code we have `x[0] = x[0] - ??? * ??? / ???`
  - Increment `j` to `j=1`:
    * Adjust the value of `x[0]` by subtracting off $\frac{2x_1}{1}$. In code we have `x[0] = x[0] - ??? * ??? / ???`
- Stop.
- You should now have a solution to the equation $Ux = y$. Substitute your solution in and verify that your solution is correct.

Exercise C.32. Copy the code from Definition C.5 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates an upper-triangular matrix of the correct form and a right-hand side y and solve for x. Your code needs to work on systems of arbitrarily large size.

C.5.4. Solving Systems with LU Decomposition

We are finally ready for the punch line of this whole LU and triangular systems business!

Exercise C.33. If we want to solve $Ax = b$ then

1. If we can, write the system of equations as $LUx = b$.
2. Solve $Ly = b$ for y using forward substitution.
3. Solve $Ux = y$ for x using backward substitution.

Pick a matrix A and a right-hand side b and solve the system using this process.

Exercise C.34. Try the process again on the 3×3 system of equations

$$\begin{pmatrix} 3 & 6 & 8 \\ 2 & 7 & -1 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -13 \\ 4 \\ 1 \end{pmatrix} \tag{C.66}$$

That is: Find matrices L and U such that $Ax = b$ can be written as $LUx = b$. Then do two triangular solves to determine x.

Let us take stock of what we have done so far.

- Solving lower-triangular systems is super fast and easy!
- Solving upper-triangular systems is super fast and easy (so long as we never divide by zero).
- It is often possible to rewrite the matrix A as the product of a lower-triangular matrix L and an upper-triangular matrix U so $A = LU$.
- Now we can re-frame the equation $Ax = b$ as $LUx = b$.
- Substitute $y = Ux$ so the system becomes $Ly = b$. Solve for y with forward substitution.
- Now solve $Ux = y$ using backward substitution.

We have successfully take row reduction and turned into some fast matrix multiplications and then two very quick triangular solves. Ultimately this will be a faster algorithm for solving a system of linear equations.

Definition C.6 (Solving Linear Systems with the LU Decomposition)**.** Let A be a square matrix in $\mathbb{R}^{n\times n}$ and let $x, b \in \mathbb{R}^n$. To solve the problem $Ax = b$,

1. Factor A into lower and upper-triangular matrices $A = LU$.
   `L, U = myLU(A)`
2. The system can now be written as $LUx = b$. Substitute $Ux = y$ and solve the system $Ly = b$ with forward substitution. `y = lsolve(L,b)`
3. Finally, solve the system $Ux = y$ with backward substitution.
   `x = usolve(U,y)`

Exercise C.35. The LU decomposition is not perfect. Discuss where the algorithm will fail.

Exercise C.36. What happens when you try to solve the system of equations

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 9 \\ -3 \end{pmatrix} \tag{C.67}$$

with the LU decomposition algorithm? Discuss.

C.6. The QR Factorization

In this section we will try to find an improvement on the LU factorization scheme from the previous section. What we will do here is leverage the geometry of the column space of the A matrix instead of leveraging the row reduction process.

Exercise C.37. We want to solve the system of equations

$$\begin{pmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & 1/3 & -2/3 \\ -2/3 & 2/3 & -1/3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ -9 \end{pmatrix}. \tag{C.68}$$

1. We could do row reduction by hand … yuck … do not do this.
2. We could apply our new-found skills with the LU decomposition to solve the system, so go ahead and do that with your Python code.
3. What do you get if you compute the product $A^T A$?
   a. Why do you get what you get? In other words, what was special about A that gave such an nice result?
   b. What does this mean about the matrices A and A^T?
4. Now let us leverage what we found in part (3) to solve the system of equations $Ax = b$ much faster. Multiply both sides of the matrix equation by A^T, and now you should be able to just read off the solution. This seems amazing!!
5. What was it about this particular problem that made part (4) so elegant and easy?

The previous exercise tells us something amazing:

Theorem C.1 (Orthonomal Matrices)**.** *If A is an orthonormal matrix where the columns are mutually orthogonal and every column is a unit vector, then $A^T = A^{-1}$ and to solve the system of equation $Ax = b$ we simply need to multiply both sides of the equation by A^T. Hence, the solution to $Ax = b$ is just $x = A^T b$ in this special case.*

Theorem C.1 begs an obvious question: *Is there a way to turn any matrix A* into an orthogonal matrix so that we can solve $Ax = b$ in this same very efficient and fast way?

The answer: Yes. Kind of.

In essence, if we can factor our coefficient matrix into an orthonormal matrix and some other nicely formatted matrix (like a triangular matrix, perhaps) then the job of solving the linear system of equations comes down to matrix multiplication and a quick triangular solve – both of which are extremely fast!

What we will study in this section is a new matrix factorization called the QR factorization whose goal is to convert the matrix A into a product of two matrices, Q and R, where Q is orthonormal and R is upper-triangular.

Exercise C.38. Let us say that we have a matrix A and we know that it can be factored into $A = QR$ where Q is an orthonormal matrix and R is an upper-triangular matrix. How would we then leverage this factorization to solve the system of equation $Ax = b$ for x?

Before proceeding to the algorithm for the QR factorization let us pause for a moment and review scalar and vector projections from Linear Algebra. In Figure C.1 we see a graphical depiction of the vector u projected onto vector v. Notice that the projection is indeed the perpendicular projection as this is what seems natural geometrically.

The **vector projection** of u onto v is the vector cv. That is, the vector projection of u onto v is a scalar multiple of the vector v. The value of the scalar c is called the **scalar projection** of u onto v.

Figure 4.1: Projection of one vector onto another.

We can arrive at a formula for the scalar projection rather easily if we consider that the vector w in Figure C.1 must be perpendicular to cv. Hence

$$w \cdot (cv) = 0. \tag{C.69}$$

From vector geometry we also know that $w = u - cv$. Therefore

$$(u - cv) \cdot (cv) = 0. \tag{C.70}$$

If we distribute we can see that

$$cu \cdot v - c^2 v \cdot v = 0 \tag{C.71}$$

and therefore either $c = 0$, which is only true if $u \perp v$, or

$$c = \frac{u \cdot v}{v \cdot v} = \frac{u \cdot v}{\|v\|^2}. \tag{C.72}$$

Therefore,

- the **scalar projection of** u onto v is

$$c = \frac{u \cdot v}{\|v\|^2} \tag{C.73}$$

- the **vector projection of** u onto v is

$$cv = \left(\frac{u \cdot v}{\|v\|^2}\right) v \tag{C.74}$$

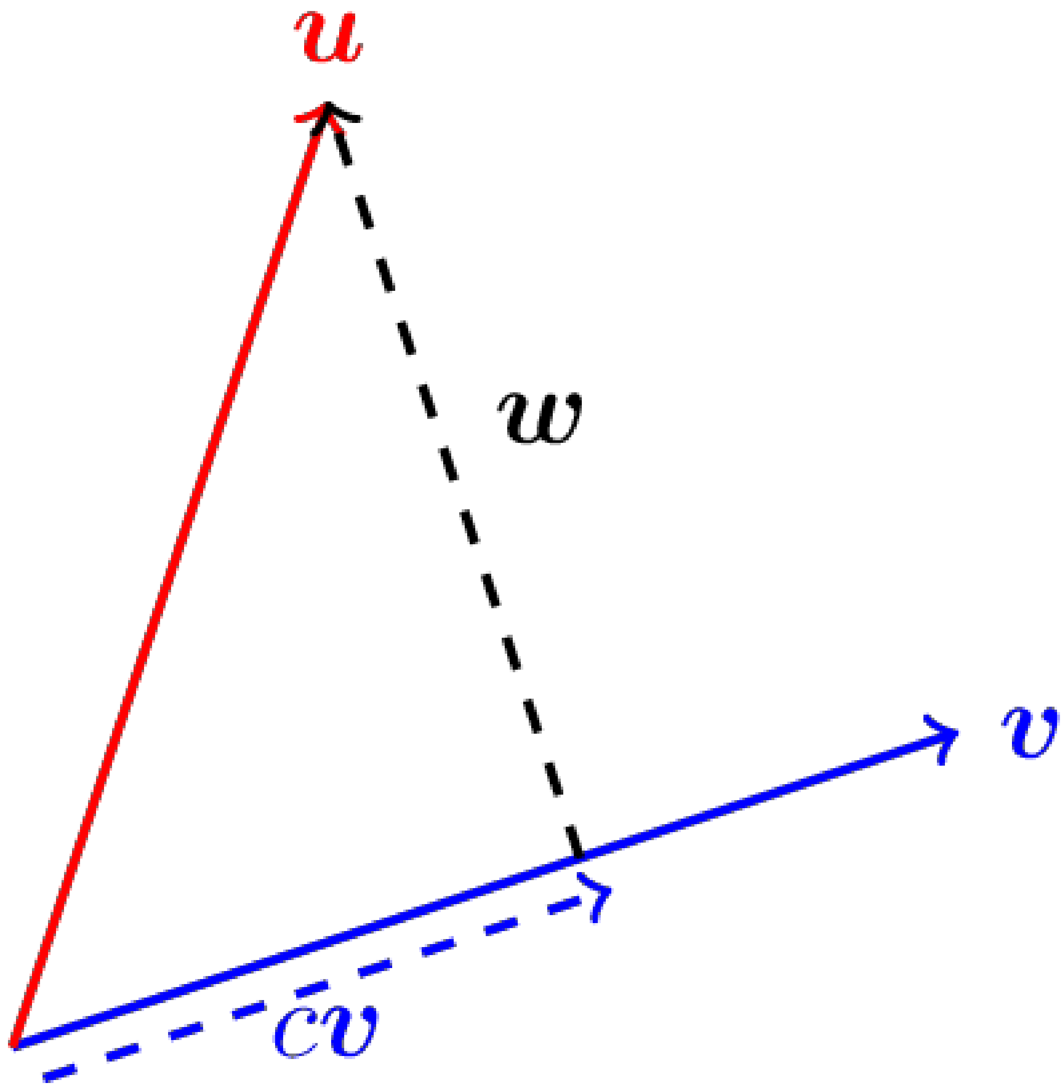


Figure C.1.: Projection of one vector onto another.

Another problem related to scalar and vector projections is to take a basis for the column space of a matrix and transform that basis into an orthogonal (or orthonormal) basis. Indeed, in Figure C.1 if we have the matrix

$$A = \begin{pmatrix} | & | \\ u & v \\ | & | \end{pmatrix} \tag{C.75}$$

it should be clear from the picture that the columns of this matrix are not perpendicular. However, if we take the vector v and the vector w we do arrive at two orthogonal vector that form a basis for the same space. Moreover, if we normalize these vectors (by dividing by their respective lengths) then we can easily transform the original basis for the column space of A into an orthonormal basis. This process is called the Gram-Schmidt process, and you have encountered it in your Linear Algebra module.

Now we return to our goal of finding a way to factor a matrix A into an orthonormal matrix Q and an upper-triangular matrix R. The algorithm that we are about to build depends greatly on the ideas of scalar and vector projections.

Exercise C.39. We want to build a QR factorization of the matrix A in the matrix equation $Ax = b$ so that we can leverage the fact that solving the equation $QRx = b$ is easy. Consider the matrix A defined as

$$A = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix}. \tag{C.76}$$

Notice that the columns of A are NOT orthonormal (they are not unit vectors and they are not perpendicular to each other).

1. Draw a picture of the two column vectors of A in $\mathbb{R}^2$. we will use this picture to build geometric intuition for the rest of the QR factorization process.
2. Define a_0 as the first column of A and a_1 as the second column of A. That is

$$a_0 = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \quad \text{and} \quad a_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{C.77}$$

   Turn a_0 into a unit vector and call this unit vector q_0

$$q_0 = \frac{a_0}{\|a_0\|} = \begin{pmatrix} \underline{\qquad} \\ \underline{\qquad} \end{pmatrix}. \tag{C.78}$$

   This vector q_0 will be the first column of the 2×2 matrix Q. Why is this a nice place to start building the Q matrix (think about the desired structure of Q)?

3. In your picture of a_0 and a_1 mark where q_0 is. Then draw the orthogonal projection from a_1 onto q_0. In your picture you should now see a right triangle with a_1 on the hypotenuse, the projection of a_1 onto q_0 on one leg, and the second leg is the vector difference of the hypotenuse and the first leg. Simplify the projection formula for leg 1 and write the formula for leg 2.

$$\text{hypotenuse } = a_1 \tag{C.79}$$

$$\text{leg 1 } = \left(\frac{a_1 \cdot q_0}{q_0 \cdot q_0}\right) q_0 = \underline{\hspace{3cm}} \tag{C.80}$$

$$\text{leg 2 } = \underline{\hspace{3cm}} - \underline{\hspace{3cm}}. \tag{C.81}$$

4. Compute the vector for leg 2 and then normalize it to turn it into a unit vector. Call this vector q_1 and put it in the second column of Q.

5. Verify that the columns of Q are now orthogonal and are both unit vectors.

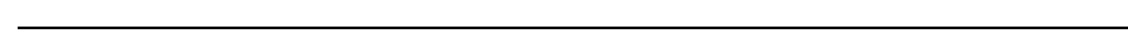
6. The matrix R is supposed to complete the matrix factorization $A = QR$. We have built Q as an orthonormal matrix. How can we use this fact to solve for the matrix R?

7. You should now have an orthonormal matrix Q and an upper-triangular matrix R. Verify that $A = QR$.

8. An alternate way to build the R matrix is to observe that

$$R = \begin{pmatrix} a_0 \cdot q_0 & a_1 \cdot q_0 \\ 0 & a_1 \cdot q_1 \end{pmatrix}. \tag{C.82}$$

Show that this is indeed true for the matrix A from this problem.



Exercise C.40. Keeping track of all of the arithmetic in the QR factorization process is quite challenging, so let us leverage Python to do some of the work for us. The following block of code walks through the previous exercise without any looping (that way we can see every step transparently). Some of the code is missing so you will need to fill it in.

```
import numpy as np
# Define the matrix $A$
A = np.array([[3,1],[4,1]])
n = A.shape[0]
# Build the vectors a0 and a1
a0 = A[??? , ???] # ... write code to get column 0 from A
a1 = A[??? , ???] # ... write code to get column 1 from A
# Set up storage for Q
Q = np.zeros( (n,n) )
# build the vector q0 by normalizing a0
q0 = a0 / np.linalg.norm(a0)
# Put q0 as the first column of Q
Q[:,0] = q0
# Calculate the lengths of the two legs of the triangle
leg1 = # write code to get the vector for leg 1 of the triangle
leg2 = # write code to get the vector for leg 2 of the triangle
# normalize leg2 and call it q1
q1 = # write code to normalize leg2
q1 = q1 / np.linalg.norm(q1) # Just to be safe with normalization if not implied
Q[:,1] = q1 # What does this line do?
R = # ... build the R matrix out of A and Q

print("The Q matrix is \n",Q,"\n")
print("The R matrix is \n",R,"\n")
print("The A matrix is \n",A,"\n")
print("The product QR is\n",Q @ R)
```

Exercise C.41. You should notice that the code in the previous exercise does not depend on the specific matrix A that we used? Put in a different 2×2 matrix and verify that the process still works. That is, verify that Q is orthonormal, R is upper-triangular, and $A = QR$. Be sure, however, that your matrix A is full rank.

Exercise C.42. Draw two generic vectors in $\mathbb{R}^2$ and demonstrate the process outlined in the previous problem to build the vectors for the Q matrix starting from your generic vectors.

Exercise C.43. Now we will extend the process from the previous exercises to three dimensions. This time we will seek a matrix Q that has three orthonormal vectors starting from the three original columns of a 3×3 matrix A. Perform each of the following steps **by hand** on the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{C.83}$$

In the end you should end up with an orthonormal matrix Q and an upper-triangular matrix R.

- **Step 1**: Pick column a_0 from the matrix A and normalize it. Call this new vector q_0 and make that the first column of the matrix Q.
- **Step 2**: Project column a_1 of A onto q_0. This forms a right triangle with a_1 as the hypotenuse, the projection of a_1 onto q_0 as one of the legs, and the vector difference between these two as the second leg. Notice that the second leg of the newly formed right triangle is perpendicular to q_0 by design. If we normalize this vector then we have the second column of Q, q_1.
- **Step 3:** Now we need a vector that is perpendicular to both q_0 AND q_1. To achieve this we are going to project column a_2 from A onto the plane formed by q_0 and q_1. we will do this in two steps:
  - **Step 3a**: We first project a_2 down onto both q_0 and q_1.
  - **Step 3b**: The vector that is perpendicular to both q_0 and q_1 will be the difference between a_2 the projection of a_2 onto q_0 and the projection of a_2 onto q_1. That is, we form the vector $w = a_2 - (a_2 \cdot q_0)q_0 - (a_2 \cdot q_1)q_1$. Normalizing this vector will give us q_2. (Stop now and prove that q_2 is indeed perpendicular to both q_1 and q_0.)

The result should be the matrix Q which contains orthonormal columns. To build the matrix R we simply recall that $A = QR$ and $Q^{-1} = Q^T$ so $R = Q^T A$.

Exercise C.44. Repeat the previous exercise but write code for each step so that Python can handle all of the computations. Again use the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{C.84}$$

Example C.8. (**QR for** $n = 3$) For the sake of clarity let us now write down the full QR factorization for a 3×3 matrix.

If the columns of A are a_0, a_1, and a_2 then

$$q_0 = \frac{a_0}{\|a_0\|} \tag{C.85}$$

$$q_1 = \frac{a_1 - (a_1 \cdot q_0) q_0}{\|a_1 - (a_1 \cdot q_0) q_0\|} \tag{C.86}$$

$$q_2 = \frac{a_2 - (a_2 \cdot q_0) q_0 - (a_2 \cdot q_1) q_1}{\|a_2 - (a_2 \cdot q_0) q_0 - (a_2 \cdot q_1) q_1\|} \tag{C.87}$$

and

$$R = \begin{pmatrix} a_0 \cdot q_0 & a_1 \cdot q_0 & a_2 \cdot q_0 \\ 0 & a_1 \cdot q_1 & a_2 \cdot q_1 \\ 0 & 0 & a_2 \cdot q_2 \end{pmatrix} \tag{C.88}$$

Exercise C.45 (The QR Factorization)**.** Now we are ready to build general code for the QR factorization. The following Python function definition is partially complete. Fill in the missing pieces of code and then test your code on square matrices of many different sizes. The easiest way to check if you have an error is to find the normed difference between A and QR with `np.linalg.norm(A - Q*R)`.

```
import numpy as np
def myQR(A):
    n = A.shape[0]
    Q = np.zeros( (n,n) )
    for j in range( ??? ): # The outer loop goes over the columns
        q = A[:,j]
        # The next loop is meant to do all of the projections.
        # When do you start the inner loop and how far do you go?
        # Hint: You do not need to enter this loop the first time
        for i in range( ??? ):
            length_of_leg = np.dot(A[:,j], Q[:,i])
            q = q -  ??? * ??? # This is where we do projections
        Q[:,j] = q / np.linalg.norm(q)
    R = # finally build the R matrix
    return Q, R
```

```
# Test Code
A = np.array( ... )
# or you can build A with use np.random.randn()
# Often time random matrices are good test cases
Q, R = myQR(A)
error = np.linalg.norm(A - Q @ R)
print(error)
```

We now have a robust algorithm for doing QR factorization of square matrices we can finally return to solving systems of equations.

Theorem C.2. ***(Solving Systems with*** QR***)*** *Remember that we want to solve $Ax = b$ and since $A = QR$ we can rewrite it with $QRx = b$. Since we know that Q is orthonormal by design we can multiply both sides of the equation by Q^T to get $Rx = Q^T b$. Finally, since R is upper-triangular we can use our* `usolve` *code from the previous section to solve the resulting triangular system.*

Exercise C.46. Solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.89}$$

by first computing the QR factorization of A and then solving the resulting upper-triangular system.

Exercise C.47. Write code that builds a random $n \times n$ matrix and a random $n \times 1$ vector. Solve the equation $Ax = b$ using the QR factorization and compare the answer to what we find from `np.linalg.solve()`. Do this many times for various values of n and create a plot with n on the horizontal axis and the normed error between Python's answer and your answer from the QR algorithm on the vertical axis. It would be wise to use a `plt.semilogy()` plot. To find the normed difference you should use `np.linalg.norm()`. What do you notice?

C.7. Over-determined Systems and Curve Fitting

Exercise C.48. Consider the problem of finding the quadratic function $f(x) = ax^2+bx+c$ that *best fits* the points

$$(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8). \tag{C.90}$$

We do not know the values of a, b, or c but we do have four different (x, y) ordered pairs. Hence, we have four equations:

$$1.07 = a(0)^2 + b(0) + c \tag{C.91}$$

$$3.9 = a(1)^2 + b(1) + c \tag{C.92}$$

$$14.8 = a(2)^2 + b(2) + c \tag{C.93}$$

$$26.8 = a(3)^2 + b(3) + c. \tag{C.94}$$

There are four equations and only three unknowns. This is what is called an **over determined systems** – when there are more equations than unknowns. Let us play with this problem.

1. First turn the system of equations into a matrix equation.

$$\begin{pmatrix} 0 & 0 & 1 \\ ___ & ___ & ___ \\ ___ & ___ & ___ \\ ___ & ___ & ___ \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1.07 \\ 3.9 \\ 14.8 \\ 26.8 \end{pmatrix}. \tag{C.95}$$

2. None of our techniques for solving systems will likely work here since it is highly unlikely that the vector on the right-hand side of the equation is in the column space of the coefficient matrix. Discuss this.

3. One solution to the unfortunate fact from part (b) is that we can project the vector on the right-hand side into the subspace spanned by the columns of the coefficient matrix. Think of this as casting the shadow of the right-hand vector down onto the space spanned by the columns. If we do this projection we will be able to solve the equation for the values of a, b, and c that will create the projection exactly – and hence be as close as we can get to the actual right-hand side. Draw a picture of what we have said here.

4. Now we need to project the right-hand side, call it b, onto the column space of the the coefficient matrix A. Recall the following facts:

- Projections are dot products
- Matrix multiplication is nothing but a bunch of dot products.
- The projections of b onto the columns of A are the dot products of b with each of the columns of A.
- What matrix can we multiply both sides of the equation $Ax = b$ by in order for the right-hand side to become the projection that we want? (Now do the projection in Python)

5. If you have done part (d) correctly then you should now have a square system (i.e. the matrix on the left-hand side should now be square). Solve this system for a, b, and c.

Theorem C.3 (Solving Overdetermined Systems)**.** *If $Ax = b$ is an overdetermined system (i.e. A has more rows than columns) then we first multiply both sides of the equation by A^T (why do we do this?) and then solve the square system of equations $(A^T A)x = A^T b$ using a system solving like LU or QR. The answer to this new system is interpreted as the vector x which solves exactly for the projection of b onto the column space of A.*

*The equation $(A^T A)x = A^T b$ is called **the normal equations** and arises often in Statistics and Machine Learning.*

Exercise C.49. Fit a linear function to the following data. Solve for the slope and intercept using the technique outlined in Theorem C.3. Make a plot of the points along with your best fit curve.

x	y
0	4.6
1	11
2	12
3	19.1
4	18.8
5	39.5
6	31.1
7	43.4
8	40.3
9	41.5

x	y
10	41.6

Exercise C.50. Fit a quadratic function to the following data using the technique outlined in Theorem C.3. Make a plot of the points along with your best fit curve.

x	y
0	-6.8
1	11.8
2	50.6
3	94
4	224.3
5	301.7
6	499.2
7	454.7
8	578.5
9	1102
10	1203.2

Code to download the data directly is given below.

```
import numpy as np
import pandas as pd
URL1 = 'https://raw.githubusercontent.com/NumericalMethodsSullivan'
URL2 = '/NumericalMethodsSullivan.github.io/master/data/'
URL = URL1+URL2
data = np.array( pd.read_csv(URL+'Exercise4_52.csv') )
# Exercise4_52.csv
```

Exercise C.51. The Statistical technique of curve fitting is often called "linear regression." This even holds when we are fitting quadratic functions, cubic functions, etc to the data ... we still call that linear regression! Why?

This section of the text on solving over determined systems is just a bit of a teaser for a bit of higher-level statistics, data science, and machine learning. The normal equations and solving systems via projections is the starting point of many modern machine learning algorithms.

C.8. The Eigenvalue-Eigenvector Problem

We finally turn our attention to another major topic in numerical linear algebra.[4]

Definition C.7 (The Eigenvalue Problem)**.** Recall that the eigenvectors, x, and the eigenvalues, λ of a square matrix satisfy the equation $Ax = \lambda x$. Geometrically, the eigen-problem is the task of finding the special vectors x such that multiplication by the matrix A only produces a scalar multiple of x.

Thinking about matrix multiplication, the geometric notion of the eigenvalue problem is rather peculiar since matrix-vector multiplication usually results in a scaling and a rotation of the vector x. Therefore, in some sense the eigenvectors are the only special vectors which avoid geometric rotation under matrix multiplication. For a graphical exploration of this idea see:
https://www.geogebra.org/m/JP2XZpzV.

Recall that to solve the eigen-problem for a square matrix A we complete the following steps:

1. First rearrange the definition of the eigenvalue-eigenvector pair to

$$(Ax - \lambda x) = 0. \tag{C.96}$$

2. Next, factor the x on the right to get

$$(A - \lambda I)x = 0. \tag{C.97}$$

[4]Numerical Linear Algebra is a huge field and there is way more to say … but alas, this is an introductory course in Numerical Analysis so we cannot do everything. Sigh.

3. Now observe that since $x \neq 0$ the matrix $A - \lambda I$ must NOT have an inverse. Therefore,

$$\det(A - \lambda I) = 0. \tag{C.98}$$

4. Solve the equation $\det(A - \lambda I) = 0$ for all of the values of λ.

5. For each λ, find a solution to the equation $(A - \lambda I)x = 0$. Note that there will be infinitely many solutions so you will need to make wise choices for the free variables.

Exercise C.52. Find the eigenvalues and eigenvectors of

$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}. \tag{C.99}$$

Exercise C.53. In the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{C.100}$$

one of the eigenvalues is $\lambda_1 = 0$.

1. What does that tell us about the matrix A?

2. What is the eigenvector v_1 associated with $\lambda_1 = 0$?

3. What is the null space of the matrix A?

OK. Now that you have recalled some of the basics, let us play with a little limit problem. The following exercises are going to work us toward the **power method** for finding certain eigen-structures of a matrix.

Exercise C.54. Consider the matrix

$$A = \begin{pmatrix} 8 & 5 & -6 \\ -12 & -9 & 12 \\ -3 & -3 & 5 \end{pmatrix}. \tag{C.101}$$

This matrix has the following eigen-structure:

$$v_1 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \quad \text{with} \quad \lambda_1 = 3 \tag{C.102}$$

$$v_2 = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} \quad \text{with} \quad \lambda_2 = 2 \tag{C.103}$$

$$v_3 = \begin{pmatrix} -1 \\ 3 \\ 1 \end{pmatrix} \quad \text{with} \quad \lambda_3 = -1 \tag{C.104}$$

If we have

$$x = -2v_1 + 1v_2 - 3v_3 = \begin{pmatrix} 3 \\ -7 \\ -1 \end{pmatrix} \tag{C.105}$$

then we want to do a bit of an experiment. What happens when we iteratively multiply x by A but at the same time divide by the largest eigenvalue. Let us see:

- What is $A^1 x/3^1$?
- What is $A^2 x/3^2$?
- What is $A^3 x/3^3$?
- What is $A^4 x/3^4$?
- ...

It might be nice now to go to some Python code to do the computations (if you have not already). Use your code to conjecture about the following limit.

$$\lim_{k\to\infty} \frac{A^k x}{\lambda_{max}^k} = ???. \tag{C.106}$$

In this limit we are really interested in the direction of the resulting vector, not the magnitude. Therefore, in the code below you will see that we normalize the resulting vector so that it is a unit vector.

Note: be careful, computers do not do infinity, so for powers that are too large you will not get any results.

```
import numpy as np
A = np.array([[8,5,-6],[-12,-9,12],[-3,-3,5]])
x = np.array([[3],[-7],[-1]])
eigval_max = 3

k = 4
# Note: For numpy arrays, A**k is element-wise power.
# We must use np.linalg.matrix_power for matrix power.
result = np.linalg.matrix_power(A, k) @ x / eigval_max**k
print(result / np.linalg.norm(result) )
```

Exercise C.55. If a matrix A has eigenvectors v_1, v_2, v_3, $\cdots$, v_n with eigenvalues $\lambda_1, \lambda_2, \lambda_3, \dots, \lambda_n$ and x is in the column space of A then what will we get, approximately, if we evaluate $A^k x / \max_j(\lambda_j)^k$ for very large values of k?

Discuss your conjecture with your peers. Then try to verify it with several numerical examples.

Exercise C.56. Explain your result from the previous exercise geometrically.

Exercise C.57. The algorithm that we have been toying with will find the dominant eigenvector of a matrix fairly quickly. Why might you be only interested in the dominant eigenvector of a matrix? Discuss.

Exercise C.58. In this problem we will formally prove the conjecture that you just made. This conjecture will lead us to the **power method** for finding the dominant eigenvector and eigenvalue of a matrix.

1. Assume that A has n linearly independent eigenvectors $v_1, v_2, \dots, v_n$ and choose $x = \sum_{j=1}^n c_j v_j$. You have proved in the past that

$$A^k x = c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots c_n \lambda_n^k v_n. \tag{C.107}$$

   Stop and sketch out the details of this proof now.

2. If we factor λ_1^k out of the right-hand side we get

$$A^k x = \lambda_1^k \left(c_1 ??? + c_2 \left(\frac{???}{???} \right)^k v_2 + c_3 \left(\frac{???}{???} \right)^k v_3 + \cdots + c_n \left(\frac{???}{???} \right)^k v_n \right) \quad \text{(C.108)}$$

   (fill in the question marks)

3. If $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ then what happens to each of the $(\lambda_j/\lambda_1)^k$ terms as $k \to \infty$?

4. Using your answer to part (c), what is $\lim_{k\to\infty} A^k x / \lambda_1^k$?

Theorem C.4 (The Power Method)**.** *The following algorithm, called* ***the power method*** *will quickly find the eigenvalue of largest absolute value for a square matrix $A \in \mathbb{R}^{n\times n}$ as well as the associated (normalized) eigenvector. We are assuming that there are n linearly independent eigenvectors of A.*

Step 1: *Given a non-zero vector x, set $v^{(1)} = x/\|x\|$. (Here the superscript indicates the iteration number) Note that the initial vector x is pretty irrelevant to the process so it can just be a random vector of the correct size..*

Step 2: *For $k = 2, 3, \ldots$*

Step 2a: *Compute $\tilde{v}^{(k)} = Av^{(k-1)}$ (this gives a non-normalized version of the next estimate of the dominant eigenvector.)*

Step 2b: *Set $\lambda^{(k)} = \tilde{v}^{(k)} \cdot v^{(k-1)}$. (this gives an approximation of the eigenvalue since if $v^{(k-1)}$ was the actual eigenvector we would have $\lambda = Av^{(k-1)} \cdot v^{(k-1)}$. Stop now and explain this.)*

Step 2c: *Normalize $\tilde{v}^{(k)}$ by computing $v^{(k)} = \tilde{v}^{(k)}/\|\tilde{v}^{(k)}\|$. (This guarantees that you will be sending a unit vector into the next iteration of the loop)*

Exercise C.59. Go through Theorem C.4 carefully and describe what we need to do in each step and why we are doing it. Then complete all of the missing pieces of the following Python function.

```
import numpy as np
def myPower(A, tol = 1e-8):
    n = A.shape[0]
    x = np.random.randn(n)
    x = # turn x into a unit vector
    # we do not actually need to keep track of the old iterates
    L = 1 # initialize the dominant eigenvalue
    counter = 0 # keep track of how many steps we have taken
    # You can build a stopping rule from the definition
    # Ax = lambda x ...
    while (???) > tol and counter < 10000:
        x = A @ x # update the dominant eigenvector
        x = ??? # normalize
        L = ??? # approximate the eignevalue
        counter += 1 # increment the counter
    return x, L
```

Exercise C.60. Test your `myPower()` function on several matrices where you know the eigenstructure. Then try the `myPower()` function on larger random matrices. You can check that it is working using `np.linalg.eig()` (be sure to normalize the vectors in the same way so you can compare them.)

Exercise C.61. In the Power Method iteration you may end up getting a different sign on your eigenvector as compared to `np.linalg.eig()`. Why might this happen? Generate a few examples so you can see this. You can avoid this issue if you use a `while` loop in your Power Method code and the logical check takes advantage of the fact that we are trying to solve the equation $Ax = \lambda x$. Hint: $Ax = \lambda x$ is equivalent to $Ax - \lambda x = 0$.

Exercise C.62. What happens in the power method iterations when λ_1 is complex. The maximum eigenvalue can certainly be complex if $|\lambda_1|$ (the modulus of the complex number) is larger than all of the other eigenvalues. It may be helpful to build a matrix specifically with complex eigenvalues.[5]

[5] To build a matrix with specific eigenvalues it may be helpful to recall the matrix factorization $A = PDP^{-1}$ where the columns of P are the eigenvectors of A and the diagonal entries of D are the eigenvalues. If you choose P and D then you can build A with your specific eigen-structure. If you are looking for complex eigenvalues then remember that the eigenvectors may well be complex too.

Exercise C.63 (Convergence Rate of the Power Method)**.** The proof that the power method will work hinges on the fact that $|\lambda_1| > |\lambda_2| \ge |\lambda_3| \ge \cdots \ge |\lambda_n|$. In Exercise C.58 we proved that the limit

$$\lim_{k\to\infty} \frac{A^k x}{\lambda_1^k} \tag{C.109}$$

converges to the dominant eigenvector, but how fast is the convergence? What does the speed of the convergence depend on?

Take note that since we are assuming that the eigenvalues are ordered, the ratio λ_2/λ_1 will be larger than λ_j/λ_1 for all $j > 2$. Hence, the speed at which the power method converges depends mostly on the ratio λ_2/λ_1. Let us build a numerical experiment to see how sensitive the power method is to this ratio.

Build a 4×4 matrix A with dominant eigenvalue $\lambda_1 = 1$ and all other eigenvalues less than 1 in absolute value. Then choose several values of λ_2 and build an experiment to determine the number of iterations that it takes for the power method to converge to within a pre-determined tolerance to the dominant eigenvector. In the end you should produce a plot with the ratio λ_2/λ_1 on the horizontal axis and the number of iterations to converge to a fixed tolerance on the vertical axis. Discuss what you see in your plot.

Hint: To build a matrix with specific eigen-structure use the matrix factorization $A = PDP^{-1}$ where the columns of P contain the eigenvectors of A and the diagonal of D contains the eigenvalues. In this case the P matrix can be random but you need to control the D matrix. Moreover, remember that λ_3 and λ_4 should be smaller than λ_2.

C.9. Algorithm Summaries

Exercise C.64. Explain in clear language how to efficiently solve an upper-triangular system of linear equations.

Exercise C.65. Explain in clear language how to efficiently solve a lower-triangular system of linear equations.

Exercise C.66. Explain in clear language how to solve the equation $Ax = b$ using an LU decomposition.

Exercise C.67. Explain in clear language how to solve an overdetermined system of linear equations (more equations than unknowns) numerically.

Exercise C.68. Explain in clear language the algorithm for finding the columns of the Q matrix in the QR factorization. Give all of the mathematical details.

Exercise C.69. Explain in clear language how to find the upper-triangular matrix R in the QR factorization. Give all of the mathematical details.

Exercise C.70. Explain in clear language how to solve the equation $Ax = b$ using a QR decomposition.

Exercise C.71. Explain in clear language how the power method works to find the dominant eigenvalue and eigenvector of a square matrix. Give all of the mathematical details.