LAB 12 - GRAM-SCHMIDT AND QR DECOMPOSITIONS

1. Orthogonal Vectors

Inner products play a vital role in mathematics. One example of an inner product on $\mathbb{R}^n$ is the classical dot product. The dot product is defined as follows: for each $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n,$$

where the $u_i, v_i \in \mathbb{R}$ are the components of the vectors $\mathbf{u}$ and $\mathbf{v}$, respectively. We say vectors $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* if the dot product between them is zero, i.e. $\mathbf{u} \cdot \mathbf{v} = 0$. In $\mathbb{R}^2$ or $\mathbb{R}^3$ orthogonal vectors are perpendicular, and therefore form a right angle. You can think of orthogonality in higher dimensions as a generalization of being perpendicular, where orthogonal vectors do not "share any directions."

As an example, consider the vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^4$ where $\mathbf{u} = [1,\ 2,\ -1,\ 4]^T$ and $\mathbf{v} = [2,\ -2,\ -6,\ -1]^T$.

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ -1 \\ 4 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -2 \\ -6 \\ -1 \end{bmatrix} = (1)(2) + (2)(-2) + (-1)(-6) + (4)(-1) = 0.$$

Because the dot product is zero, these vectors are orthogonal. In the following problem, you will write a function which tests whether two vectors are orthogonal, by checking whether their dot product is equal to zero. Before doing this, however, it will be helpful to discuss what it means for two values to be equal in Python.

Recall that in Python we can check if two values `a` and `b` are equal by using the command `a==b`, which will return `True` if the value of `a` equals the value of `b`, and `False` otherwise.

```
a=15.1
b=15.1

print(a==b)
print(a==15.1)
print(b==7)
```

```
True
True
False
```

- In your practice notebook, execute the following command:

```
np.sqrt(2)**2==2
```

(recall that `np.sqrt()` is the square root function in NumPy, and `n**2` is the value of `n` squared). Does the code above produce the output you expect?

You may be surprised to observe that Python does not seem to recognize the simple fact that $(\sqrt{2})^2 = 2$! By now you probably recognize that the problem lies in the fact that whenever Python stores a decimal value, it is only able to store an approximation of the actual value. So when we compute `np.sqrt(2)` the output NumPy produces is only an approximation to $\sqrt{2}$, and hence `np.sqrt(2)**2` is only an approximation to $(\sqrt{2})^2 = 2$.

Often times when programming we need to check when two values `a` and `b` are equal. To get around this problem we instead check whether `a` and `b` are very close to each other, or in other words whether $|a - b|$ is very close to zero. We do this by picking some very small cutoff value, say 1×10^{-12}, and checking whether $|a - b|$ is smaller than this cutoff. More precisely, if $|a - b| < 1 \times 10^{-12}$ then we think of a and b as being equal, while if $|a - b| \geq 1 \times 10^{-12}$ we think of them as being different. This is illustrated in the code below:

```
def are_close(a,b):
  cutoff=1e-12                    # This is the same as 1 x 10^(-12).
  if np.abs(a-b)<cutoff:
    return True
  else:
    return False
```

- Using the code above, check whether the values of `np.sqrt(2)**2` and `2` are close.

Problem 1. Write a function called `orthogonal_check(u,v)` which checks whether two vectors are orthogonal. Your function should accept two vectors `u` and `v` of the same size (as NumPy arrays) and returns `True` if the vectors are orthogonal and `False` if they are not orthogonal. You may use the built-in NumPy function `np.dot(u,v)` for performing the dot product between `u` and `v`.

For example, if `u=np.array([1,2,-1,4])` and `v=np.array([2,-2,-6,-1])` then `orthogonal_check(u,v)` should return `True`. If `x=np.array([1,1,1,1])` and `y=np.array([1,1,1,2])` then `orthogonal_check(x,y)` should return `False`.

Hint: Due to the numerical precision issues outlined above, you will need to check whether the dot product of `u` *and* `v` *is close to zero. Use a cutoff value of* `1e-12` *as above.*

An *orthogonal set* of vectors is a collection of vectors for which any pair of distinct vectors are orthogonal to each other. In other words, $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ is an orthogonal set of vectors if $\mathbf{v}_i \cdot \mathbf{v}_j = 0$ for all $i \neq j$.

Problem 2. Write a new function called `orth_set_check(vect_set)` which uses the function in *Problem 1* to determine whether a set of vectors is an orthogonal set. Your function should accept a *list* of vectors (each vector written as a NumPy array) and return the value `True` if the vectors form an orthogonal set and `False` if they do not.

For example, if `L1=[np.array([1,-1,0]),np.array([1,1,1]),np.array([0,1,-1])]` then `orth_set_check(L1)` should return `False`.

If `L2=[np.array([1,0,0]),np.array([0,2,0]),np.array([0,0,3])]` then `orth_set_check(L2)` should return `True`.

Hint: There are several ways to design such a function. One way is to include two `for` *loops, one inside the other. Each of the* `for` *loops runs through the different vectors in the list, and the function* `orthogonal_check(u,v)` *from Problem 1 is used to check if the given vectors are orthogonal. You may want to include an* `if` *statement that immediately returns* `False` *if a pair of nonzero orthogonal vectors is encountered, while the function returns* `True` *if it reaches the end of the loops without encountering any pairs of nonorthogonal vectors.*

A vector is called a *unit vector* if the magnitude of the vector is equal to 1. (Recall that the magnitude $\|\mathbf{v}\|$ of a vector $\mathbf{v}$ is defined by $\|\mathbf{v}\| = \sqrt{\mathbf{v}\cdot\mathbf{v}}$.) For instance the vector $\mathbf{v} = [1/2,\ 1/2,\ 1/\sqrt{2}]^T$ is a unit vector since

$$\|\mathbf{v}\| = \sqrt{(1/2)^2 + (1/2)^2 + (1/\sqrt{2})^2} = 1.$$

If every vector in an orthogonal set is also a unit vector, we call the set an *orthonormal set*. Another way to express this condition is

$$\mathbf{v}_i \cdot \mathbf{v}_j = \begin{cases} 0 & i \neq j, \\ 1 & i = j. \end{cases}$$

Suppose a matrix Q has n columns which form an orthonormal set. One particularly nice property of Q is that

$$Q^T Q = I_n.$$

Try to prove that this is true for yourself using the definition of orthonormal vectors. Thus if Q is a square matrix, then $Q^T = Q^{-1}$. Such a matrix Q is called an *orthogonal matrix*.

Problem 3. Write a program `normalize(v)` which converts the input vector `v` (expressed as a NumPy array) into a unit vector in the same direction. You may assume that `v` is nonzero.

It is often convenient to work with an *orthonormal basis* (a basis consisting of orthonormal vectors is called an orthonormal basis). Although it is true that any vector $\mathbf{w} \in W$ can be uniquely written as a combination of linearly independent vectors which span W, if the set is also orthonormal, then it is easy to calculate which linear combination to use. If $\mathbf{x}_1, \ldots, \mathbf{x}_n$ is an orthonormal basis for a subspace W, then we can write $\mathbf{w} \in W$ as $\mathbf{w} = c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + \cdots + c_n\mathbf{x}_n$, where $c_1 = \mathbf{w}\cdot\mathbf{x}_1,\ c_2 = \mathbf{w}\cdot\mathbf{x}_2,\ \ldots,\ c_n = \mathbf{w}\cdot\mathbf{x}_n$.

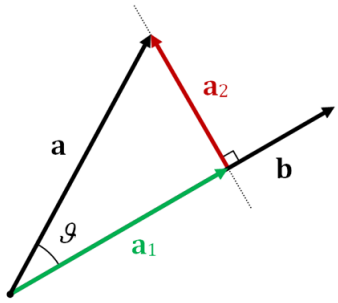

FIGURE 1. Vector $\mathbf{a}$ is being projected onto vector $\mathbf{b}$. The projection vector $\mathbf{a_1} = \text{proj}_\mathbf{b}\mathbf{a}$ is the part of $\mathbf{a}$ in the same direction as $\mathbf{b}$, while $\mathbf{a}_2 = \mathbf{a} - \text{proj}_\mathbf{b}\mathbf{a}$ is the part of $\mathbf{a}$ which is perpendicular to $\mathbf{b}$. *Source: Wikipedia.org*

2. Projections

Given a vector $\mathbf{a}$ and a vector $\mathbf{b}$, it can be useful to break $\mathbf{a}$ into the sum of two vectors, one which is parallel to $\mathbf{b}$ and one which is perpendicular to $\mathbf{b}$ (see Figure 1). We call the vector which is parallel to $\mathbf{b}$ the *projection* of $\mathbf{a}$ onto $\mathbf{b}$, which we write as $\text{proj}_\mathbf{b}\mathbf{a}$. This vector can be found by scaling the vector $\mathbf{b}$ by an appropriate amount. The formula for finding the projection of one vector onto another vector is given by

$$\text{proj}_\mathbf{b}\mathbf{a} = \left(\frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\right)\mathbf{b}.$$

The vector perpendicular to $\mathbf{b}$ can be found simply by subtracting the projection from the original vector, i.e. $\mathbf{a_2} = \mathbf{a} - \text{proj}_\mathbf{b}\mathbf{a}$.

Problem 4. Write a program `proj(a,b)` which accepts two vectors `a` and `b` (as NumPy arrays of the same size) and returns $\text{proj}_\mathbf{b}\mathbf{a}$. You can use the built in function `np.dot(a,b)` for the dot product.

For example, if `a=np.array([1,4,1])` and `b=np.array([1,1,1])` then `proj(a,b)` should return `array([2., 2., 2.])`.

Note: This function is *not* symmetric. By this we mean that `proj(a,b)` does NOT equal `proj(b,a)`. Be sure when you define your function that `proj(a,b)` calculates $\text{proj}_\mathbf{b}\mathbf{a}$.

3. Gram-Schmidt Decomposition

The span of any collection of vectors in $\mathbb{R}^n$ will define a subspace of $\mathbb{R}^n$. However, as we mentioned earlier, it is often desirable to define a basis for the subspace which is *orthonormal*. Forming such a basis is the goal of the *Gram-Schmidt algorithm*. It takes as its input a set of linearly independent vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, and outputs an orthonormal set of vectors $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ which span the same subspace as the original set. The process works by calculating an $\mathbf{x}_i$ at each step by taking $\mathbf{v}_i$ and subtracting off its projection onto the $\mathbf{x}_i$'s

found in previous steps, as shown below.

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{v}_1 \\
\mathbf{x}_2 &= \mathbf{v}_2 - \text{proj}_{\mathbf{x}_1}\mathbf{v}_2 \\
\mathbf{x}_3 &= \mathbf{v}_3 - \text{proj}_{\mathbf{x}_1}\mathbf{v}_3 - \text{proj}_{\mathbf{x}_2}\mathbf{v}_3 \\
&\vdots \\
\mathbf{x}_n &= \mathbf{v}_n - \text{proj}_{\mathbf{x}_1}\mathbf{v}_n - \cdots - \text{proj}_{\mathbf{x}_{n-1}}\mathbf{v}_n
\end{aligned}$$

Once all the $\mathbf{x}_i$'s are found, we find the $\mathbf{u}_j$ vectors by normalizing each $\mathbf{x}_j$.

$$\begin{aligned}
\mathbf{u}_1 &\leftarrow \mathbf{x}_1/\|\mathbf{x}_1\| \\
\mathbf{u}_2 &\leftarrow \mathbf{x}_2/\|\mathbf{x}_2\| \\
&\vdots \\
\mathbf{u}_n &\leftarrow \mathbf{x}_n/\|\mathbf{x}_n\|
\end{aligned}$$

Thus we are left with an orthonormal set of vectors $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$.

To implement this in a computer, we can use the following algorithm, which we have described in pseudo-code. In the code we input V which represents a list of linearly independent vectors. We begin by defining a new list X which starts out identical to the input list V. We then iterate through each vector in X starting with the second element ($i = 1$) in the outer loop. Letting V_i denote the first i vectors of V, the inner loop subtracts off the projection of V_i onto all of the previously defined elements of X. Note that the inner loop only goes from $j = 0$ to $j = i - 1$. Thus, at the conclusion of each iteration of the outer loop, the vectors of X are replaced with the resulting Gram-Schmidt vectors. Once our new list of orthogonal vectors is complete, we then loop back through the list and normalize each one (here we replace the vectors in X with unit vectors, instead of creating a new list). At the end we return the list X of orthonormal vectors.

Algorithm 1: Gram-Schmidt Process

```
Gram-Schmidt(V):
X = V
n = length(V)
for i = 1, ..., n − 1 do
    for j = 0, ..., i − 1 do
        X_i = X_i − proj_{X_j} V_i
    end for
end for
for i = 0, ..., n − 1 do
    X_i = X_i / ||X_i||
end for
return X
```

Problem 5. Define a function `gram_schmidt(V)` which accepts a list of vectors `V` (where each vector is a NumPy array), and returns a list of orthonormal vectors which span the same space as the original list. You may assume that the vectors in the list `V` are linearly independent. Your program should use the results of *Problem 3* and *Problem 4*.

For example, if

```
V=[np.array([1,3,2,4,0]),np.array([-1,0,4,5,0]), np.array([0,2,2,2,2]),
np.array([3,2,3,2,0])]
```

is a list of vectors, then `A=gram_schmidt(V)` should return

```
[array([0.18257419, 0.54772256, 0.36514837, 0.73029674, 0.  ]),
array([-0.45161355, -0.64176663, 0.52292096, 0.33276788, 0.  ]),
array([-0.18160984, 0.21187815, 0.25728061, -0.24214646, 0.89291506]),
array([ 0.72782534, -0.2079501 , 0.57186277, -0.31192515, -0.05198752])],
```

and `orth_set_check(A)` should return `True`.

Here we mention that although the Gram-Schmidt method is easy to implement and intuitive, it is actually not very numerically stable. Each new vector which is created for the output set is not exactly orthogonal to the previous ones because of round-off error. Thus if the program is used on a large set of vectors, the round off error accumulates. This can cause the final vectors to be far from an orthogonal set. There are other more stable methods which can create an orthonormal set of vectors.

4. QR decomposition

Recall that in Lab 5 we learned how to decompose a matrix into the matrix product of two matrices, L and U. Now we can introduce another important matrix decomposition technique called the QR decomposition, where we decompose a matrix A into a matrix product, $A = QR$. In this decomposition, Q is a matrix with orthonormal columns, and whose column space is the same as the column space of A. In addition, R is a square matrix with an upper triangular form.

To create the Q matrix we can perform the Gram-Schmidt process on the columns of A. Because Q has orthonormal columns, we know that $Q^TQ = I$, so to find R we can simply multiply Q^T to both sides of $A = QR$, to get

$$\begin{aligned} Q^TQR &= Q^TA \\ IR &= Q^TA \\ R &= Q^TA \end{aligned}$$

If we keep the vectors in the same order in which we found them using the Gram-Schmidt process, R will already have an upper triangular form (since at each step we only used previous columns to define the next column).

Problem 6. Define a function `QR_decomposition(A)` which accepts a matrix `A` (formatted as a 2-dimensional NumPy array), and returns the matrices Q and R.

Hint: Recall that the function `gram_schmidt` *takes in a list of NumPy vectors, not an array. Therefore, if the function* `QR_decomposition` *calls* `gram_schmidt` *then you will first need to create a list which contains the columns of* `A`*. You can do this using list comprehensions quite quickly:* `[np.array(A[:,j]) for j in range(len(A[0]))]`*. Don't forget to turn the result back into an array with the correct columns.*

For example, if

```
A=np.array([[ 1., -1., 0., 3.],[ 3., 0., 2., 2.],[ 2., 4., 2., 3.],
[ 4., 5., 2., 2.],[ 0., 0., 2., 0.]])
```

then the output of `QR_decomposition(A)` should be

```
 Q=
[[ 0.18257419 -0.45161355 -0.18160984 0.72782534]
[ 0.54772256 -0.64176663 0.21187815 -0.2079501 ]
[ 0.36514837 0.52292096 0.25728061 0.57186277]
[ 0.73029674 0.33276788 -0.24214646 -0.31192515]
[ 0.  0.  0.89291506 -0.05198752]]

R=
[[ 5.47722558e+00 4.92950302e+00 3.28633535e+00 4.19920627e+00]
[ 1.33226763e-15 4.20713679e+00 4.27844420e-01 -4.04075285e-01]
[ 1.22124533e-15 -8.32667268e-17 2.23985472e+00 1.66475688e-01]
[ 1.99840144e-15 0.00000000e+00 3.05311332e-16 2.85931385e+00]]
```

5. Applications

Performing a QR decomposition can be useful in a number of applications. For instance, it can be used to estimate eigenvalues in a slick way, it can be used to solve the least-squares problem in a more reliable way, and for many other applications it can be numerically better to work with an orthogonal set. In many cases, solving a system $A\mathbf{x} = \mathbf{b}$ can also be simplified using a QR decomposition, similar to how we found a solution to a system using the LU factorization in Lab 5.

Given an invertible matrix A, we can perform QR factorization to write $A = QR$ (because we are assuming A is invertible, the columns of A are linearly independent, and hence it is possible to find a QR factorization). Now if we want to find the unique solution to $A\mathbf{x} = \mathbf{b}$, we can solve the equivalent system $QR\mathbf{x} = \mathbf{b}$. Since Q is an orthogonal matrix, $Q^T = Q^{-1}$, so taking our original problem and left multiplying both sides of the equation by Q^T yields the following:

$$\begin{aligned} Q^TQR\mathbf{x} &= Q^T\mathbf{b} \\ Q^{-1}QR\mathbf{x} &= Q^T\mathbf{b} \\ IR\mathbf{x} &= Q^T\mathbf{b} \\ R\mathbf{x} &= Q^T\mathbf{b}. \end{aligned}$$

Since R is in an upper triangular form, determining the solution simply requires back substitution. We used this technique in Lab 5 to solve for $\mathbf{x}$ in an LU decomposition.

Problem 7. Define a function `QR_solver(A,b)` which accepts as input a square matrix `A` and a vector `b` (as NumPy arrays) and returns the solution $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$. We assume that the matrix A is invertible. Thus `QR_solver` may call the functions `QR_decomposition` (from *Problem 6*), and `back_substitution` (which is provided in the notebook).

For example, if `A=np.array([[3,1,-2],[1.5,2,-5],[2,-4,1]])`, and the given vector is `b=np.array([1.1,3,-2])` then `QR_solver(A,b)` should return the vector

```
array([-0.07032967, 0.34395604, -0.48351648]).
```

Hint: Your function should first call `QR_decomposition` *to find the matrices* `Q` *and* `R`. *Once these matrices are found, use the function* `back_substitution` *to solve the system* $R\mathbf{x} = Q^T\mathbf{b}$.

Once you have a QR decomposition of a matrix, using it to find solutions to a system of equations (or the normal equations for a system with no solutions) is usually numerically better than using an LU factorization. However, using the Gram-Schmidt method to find the QR decomposition of a matrix is time consuming and not numerically robust, meaning small round-off errors grow and can cause the resulting output vectors not to be orthogonal. There are a number of other ways to create an orthonormal basis for a set of vectors which are more stable than Gram-Schmidt, but are beyond the scope of this lab. If you use one of these other methods for finding the QR decomposition, this provides a better method for solving a linear system than either Gaussian elimination or LU decomposition.