Guided Lab 2: QR factorization and SVD

Due February 13, 2026 by 6pm

Please show all your work and include any code you create along with your submission. Be sure to acknowledge any substantial help you receive. This includes collaboration with others, references and online resources.

High Perfomance Computing at VCU

VCU has computing clusters housed on the second floor of Harris Hall. Visit the website to read more about it: `https://research.vcu.edu/resources/cores/hprc/`

If you don't already have an account, please request one under the "Getting Started" tab. I'll ask you to use the `athena` cluster in a future assignment.

1 Introduction

We have seen one matrix factorization, LU decomposition, that can be used to solve a linear system provided that the system is square, and that it is nonsingular, and that it is not too badly conditioned. However, if we want to handle problems with a bad condition number, or that are singular, or even rectangular, we are going to need to come up with a different approach. In this lab, we will look at two versions of the QR factorization:

$$A = QR$$

where Q is an "orthogonal" matrix, and R is upper triangular, and also the "singular value decomposition" (SVD) that factors a matrix into the form

$$A = USV^{\dagger}.$$

We will see that the QR factorization can be used to solve the same problems that the LU factorization handles, but can be extended to do several other tasks for which the LU factorization is useless. In situations where we are concerned about controlling roundoff error, the QR factorization can be more accurate than the LU factorization, and is even a little easier to use when solving linear systems, since the inverse of Q is simply Q^T . We will also see that the SVD can be used to compress data, to identify the range and nullspace of a linear operator, and even to solve non-square systems.

This guided lab is designed for Matlab, but you are free to use any programming language that you prefer. You'll just need to identify the analogous packages and commands that are described for Matlab. Please let me know if you have questions about this, I'll do my best to help.

2 Orthogonal Matrices

An **orthogonal matrix** is a real matrix whose inverse is equal to its transpose. By convention, an orthogonal matrix is usually denoted by the symbol Q. The definition of an orthogonal matrix immediately implies that

$$QQ^T = Q^TQ = I.$$

From the definition of an orthogonal matrix, and from the fact that the L^2 vector norm of x can be defined by:

$$\|x\|_2 = \sqrt{x^Tx}$$

and the fact that $(Ax)^T = x^TA^T$ you should be able to deduce that, for orthogonal matrices, $\|Qx\|_2 = \|x\|_2$. If I multiply a two-dimensional vector x by Q and its L^2 norm doesn't change, then Qx must lie on the circle whose radius is $\|x\|_2$. In other words, the only thing that has happened is that Qx has been rotated around the origin by some angle, or reflected through the origin or about a diameter. That means that a two-dimensional orthogonal matrix represents a rotation or reflection. In N dimensions, orthogonal matrices are also often called rotations. When matrices are complex, the term "unitary" is an analog to "orthogonal." A matrix is **unitary** if $UU^\dagger = I$, where the $\dagger$ superscript refers to the "Hermitian" or "conjugate-transpose" of the matrix. In Matlab, the prime operator implements the Hermitian and the dot-prime operator implements the transpose. A real matrix that is unitary is orthogonal.

3 The Gram-Schmidt Method

The "Gram Schmidt method" can be thought of as a process that analyzes a set of vectors $\mathcal{X}$, producing an "equivalent" (and possibly smaller) set of vectors $\mathcal{Q}$ that span the same space, have unit L^2 norm, and are pairwise orthogonal. Note that if we can produce such a set of vectors, then we can easily answer many questions of interest. In particular, the size of the set $\mathcal{Q}$ tells us whether the set of vectors $\mathcal{X}$ is linearly independent, and the dimension of the column space spanned and the rank of a matrix constructed from $\mathcal{X}$. And of course, the column vectors in $\mathcal{Q}$ are the orthonormal basis we were seeking.
We will be using the Gram-Schmidt process to factor a matrix, but the process itself pops up repeatedly whenever sets of vectors must be made orthogonal. You may see it, for example, in Krylov space methods for iteratively solving systems of equations and for eigenvalue problems. In words, the Gram-Schmidt process goes like this:

1. Start with no vectors in $\mathcal{Q}$;
2. Consider x, the next (possibly the first) vector in $\mathcal{X}$;
3. For every vector q_i already in $\mathcal{Q}$, compute the projection of q_i on x (i.e., $q_i \cdot x$) and subtract it from x to get a vector orthogonal to q_i;
4. Compute the norm of (what's left of) x. If the norm is zero (or too small), discard x from $\mathcal{Q}$; otherwise, divide x by its norm and move it from $\mathcal{X}$ to $\mathcal{Q}$.

5. If there are more vectors in $\mathcal{X}$, return to step 2.

6. When you get here, $\mathcal{X}$ is empty and you have an orthonormal set of vectors $\mathcal{Q}$ spanning the same space as the original set $\mathcal{X}$.

Here is a sketch of the Gram-Schmidt process as an algorithm. Assume that n_x is the number of `x` vectors:

```
nq = 0            % nq will become the number of q vectors
for k = 1 to nx
      for ℓ = 1 to nq       % if nq = 0 the loop is skipped
           r_{ℓ,k} = q_ℓ · x_k
           x_k = x_k - r_{ℓ,k} q_ℓ
      end
      r_{k,k} = (x_k · x_k)^{1/2}
      if r_{k,k} > 0
           nq = nq + 1
           q_{nq} = x_k /r_{k,k}
     end
end
```

You should be able to match this algorithm to the previous verbal description. Can you see how the L^2 norm of a vector is being computed?

Exercise 1:

(a) Implement the Gram-Schmidt process in an m-file called `gram_schmidt.m`. The first lines of your file should be:

```
function Q = gram_schmidt( X )
% Q = gram_schmidt( X )
% more comments describing how to use this function
% and what it does
% your name and the date
```

Assume the x vectors are stored as columns of the matrix X. Be sure you understand this data structure because it is a potential source of confusion. In the algorithm description above, the x vectors are members of a set $\mathcal{X}$, but here these vectors are stored as the columns of a matrix X. Similarly, the vectors in the set $\mathcal{Q}$ are stored as the columns of a matrix Q. The first column of X corresponds to the vector x_1, so that the Matlab expression `X(k,1)` refers to the k^th component of the vector x_1, and the Matlab expression `X(:,1)` refers to the Matlab column vector corresponding to x_1. Similarly for the other columns.

(b) It is always best to test your code on simple examples for which you know the answers. Test your code using the following 3×3 matrix input:

```
X = [ 1 1 1; 0 1 1 ; 0 0 1 ]
```

Explain why you should expect that the correct result is the identity matrix.

(c) Another simple test you can do in your head is the following:

```
X = [ 1 1 1; 1 1 0; 1 0 0 ]
```

The columns of Q should have L^2 norm of 1, and be pairwise orthogonal. Confirm this.

(d) Test your Gram-Schmidt code to compute a matrix Q using the following input:

```
X = [ 2 -1 0; 0 -1 2; -1 0 0; -1 2 -1 ]
```

Verify that the columns of Q have L^2 norm 1, and are pairwise orthogonal. If you like programming problems, think of a way to do this check in a single line of Matlab code.

(e) Show that the matrix Q you just computed is not an orthogonal matrix, even though its columns form an orthornormal set. Are you surprised?

4 QR factorization by Gram-Schmidt method

Recall how the process of Gaussian-elimination could actually be regarded as a process of factoriwhichzation in the case of LU decomposition. This insight enabled us to solve many other problems. In the same way, the Gram-Schmidt process is actually carrying out a different factorization that will give us the key to other problems. Just to keep our heads on straight, let me point out that we're about to stop thinking of X as a bunch of vectors, and instead regard it as a matrix. Since it is traditional for matrices to be called A, that's what we'll call our set of vectors from now on. Now, in the Gram-Schmidt algorithm, the numbers that we called $r_{\ell,k}$ and $r_{k,k}$, that we computed, used, and discarded, actually record important information. They can be regarded as the nonzero elements of an upper triangular matrix R. The Gram-Schmidt process actually produces a factorization of the matrix A of the form:

$$A = QR$$

Here, the matrix Q has the same $M \times N$ "shape" as A, so it's only square if A is. The matrix R will be square $(N \times N)$, and upper triangular. Now that we're trying to produce a factorization of a matrix, we need to modify the Gram-Schmidt algorithm of the previous section. Every time we consider a vector x_k at the beginning of a loop, we will now always produce a vector q_k at the end of the loop, instead of dropping some out. Hence, if r_k, k, the norm of vector x_k, is zero, we will simply set q_k to the zero vector.

Exercise 2:

(a) Make a copy of the m-file `gram_schmidt.m` and call it `gs_factor.m`. Modify it to compute the Q and R factors of a (possibly) rectangular matrix A. The first lines of your file should be:

```
function [Q, R] = gs_factor( A )
% [Q, R] = gs_factor( A )
% more comments describing how to use this function
% and what it does
% your name and the date
```

Recall: `[Q, R] = gs_factor` is the syntax that Matlab uses to return two matrices from the function. When calling the function, you use the same syntax:
`[Q, R]=gs_factor(A)`
When you use this syntax, it is OK to leave out the comma between the Q and the R but leaving out that comma is a syntax error in the signature line of a function m-file.

(b) Compute the QR factorization of the following matrix.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

You computed Q in an earlier exercise, and you should get the same matrix Q again. Check that $A = QR$: if so, your code is correct.

(c) Compute the QR factorization of a 100×100 matrix of random numbers (`A=rand(100,100)`). (Hint: use norms to check the equalities.)

- Is it true that $Q^TQ = I$? (Hint: Does `norm(Q'*Q-eye(100))` equal 0?)
- Is it true that $QQ^T = I$?
- Is Q orthogonal?
- Is the matrix R upper triangular? (Hint: the Matlab functions `tril` or `triu` can help, so you don't have to look at all those numbers.)
- Is it true that $A = QR$?

Note: One step in the Gram-Schmidt process avoids dividing by zero by checking that $r_{k,k} > 0$. This is, in fact, very poor practice! The reason is that $r_{k,k}$ might be of roundoff size. The "solution" to this problem is more art than science, and is beyond the scope of this project. One feasible way to fix the problem would be to replace zero with, say, `eps*max(max(R))` in the inequality.

5 QR decomposition by Householder Reflections

Matlab has a built in QR decomposition function (type `help qr` for documentation about it). This uses a technique for computing the factorization based on "Householder reflections" instead of the Gram-Schmidt process. You can read more about this approach by looking on wikipedia for "QR decomposition" or on this blog post:

https://blogs.mathworks.com/cleve/2016/10/03/householder-reflections-and-the-qr-decomposition/

A comparison of methods of QR factorization can also be found here:

https://blogs.mathworks.com/cleve/2016/07/25/compare-gram-schmidt-and-householder-orthogonalization-algorithms/

The Gram-Schmidt process relies on subtracting off projections in order to make the vectors orthonormal, and this can lead to numerical roundoff issues when vectors are nearly perpendicular. The approach based on "Householder reflections" generates orthanormal vectors by reflecting the vector about an appropriate axis, thereby removing the need to subtract off projections.

Exercise 3: Repeat Exercise 2 (b,c) using Householder QR decomposition. You may use the above blogposts to help you create a script that implements the algorithm.

6 Solve systems of linear equations the QR way

If we have computed the QR factorization of a matrix without encountering any singularities, then it is easy to solve linear systems. We use the property of the Q factor that $Q^TQ = I$:

$$\begin{aligned} Ax &= b \\ QRx &= b \\ Q^TQRx &= Q^Tb \\ Rx &= Q^Tb \end{aligned}$$

So all we have to do is form the new right hand side Q^Tb and then solve the upper triangular system. And that's easy because it's the same as the backward substitution method for solving systems using the LU decomposition method.

Exercise 4: Write a file called `qr_solve.m`. It begin with:

```
function x = qr_solve ( Q, R, b )
% x = qr_solve ( Q, R, b )
% more comments about how to use it
% and how it works
% your name and the date
```

Assume that the Q and R factors come from either your `gs_factor` routine or the built in `qr` routine. The matrix R is upper triangular, so you can use backwards substitution. Set up your code to compute Q^Tb and then solve the upper triangular system $Rx = Q^Tb$. When you think your solver is working, test it out on a system as follows:

```
n = 7
A = magic ( n )
x = [ 1 :  n ]'
```

```
b = A * x
[Q, R ]= qr( A )
x2 = qr_solve( Q, R, b )
norm(x - x2) % should be close to zero.
```

Now you know another way to solve a square linear system. It turns out that you can also use the QR algorithm to solve non-square systems, but that task is better left to the singular value decomposition.

7 Singular Value Decompostion

Another very important matrix product decomposition is the singular value decomposition of a matrix into the product of three matrices. If A is an $m \times n$ matrix, then there are three matrices U, S and V, with

$$A = USV^\dagger,$$

where U is $n \times n$ and unitary, and V is $m \times m$ and unitary. Recall that unitary matrices are like orthogonal matrices but they could have complex entries. Unitary matrices with all real entries are orthogonal matrices.
The matrix S is $n \times m$ and is of the form

$$S_{k\ell} = \begin{cases} \sigma_k & k = \ell \\ 0 & k \neq \ell \end{cases}$$

where $\sigma_k \geq \sigma_{k-1}$, $k = 2, \ldots r$. The number r is the rank of the matrix A and is necessarily no larger than `min(m, n)`. The "singular values," σ_k , are real and positive and are the eigenvalues of the Hermitian matrix $A^\dagger A$. The singular value decomposition provides much useful information about the matrix A. This information includes:

- Information about the range and nullspace of A regarded as a linear operator, including the dimensions of these spaces. The matrices U and V provide orthonormal bases of these spaces.
- A method of "solving" non-square matrix systems.
- A method of "compressing" the matrix A.

Exercise 5: Write out an expression for the solution of $Ax = b$, assuming the SVD of A is $A = USV^\dagger$.

You may find it useful to define a diagonal matrix S^+ with the same shape as S^T and values

$$S^+_{k\ell} = \begin{cases} \dfrac{1}{\sigma_k} & k = \ell \text{ and } \sigma_k > 0 \\ 0 & k \neq \ell \end{cases}$$

and consider the product S^+S.

Note: if A is close to singular, a similar definition but with diagonal entries $1/\sigma_k$ for $\sigma_k > \epsilon$ for some $\epsilon > 0$ can work very nicely. In these latter cases, the "solution" is the least-squares best fit solution and the matrix $A^+ = VS^+U^\dagger$ is called the Moore-Penrose pseudo-inverse of A.

In the final exercises you will see how the singular value decomposition can be used to "compress" a graphical figure by representing the figure as a matrix and then using the singular value decomposition to find the closest matrix of lower rank to the original. This approach can form the basis of efficient compression methods.

Exercise 6: Note that the provided m-file containes code for this exercise.

(a) The following photo of Commodore Grace Hopper was taken on January 20, 1984. She was a pioneer of computer programming and the person who popularized the term "bug" in the context of computer science.

https://en.wikipedia.org/wiki/Grace_Hopper#/media/File:Commodore_Grace_M._Hopper,_USN_(covered).jpg

(b) Matlab provides various image processing utilities. In this case, read the image in and convert it to a matrix of real ("double") numbers using the following commands.

```
rgbGH=imread('GraceHopper.jpg');

figure();
image(rgbGH), axis image;

bwGH=rgb2gray(rgbGH);
imGH=double(bwGH);
```

The variable `bwGH` will be a 3000×2400 matrix of integers between 0 and 255, and we convert it to a matrix of double-precision floating point numbers that we store as `imGH`.

(c) Matlab has the notion of a "colormap" that determines how the values in the matrix will be colored. Use the command

```
colormap(gray(256));
```

to set a grayscale colormap. Now you can show the picture with the commands

```
image(imGH)
daspect([1 1 1])
```

(d) Use the command

```
[U S V]=svd(imGH);
```

to perform the singular value decomposition.

(e) Plot the singular values: `plot(diag(S))`. You should see that the first few hundred singular values are significantly larger than the others.

(f) For various values of `ns`, construct new matrices

```
imNs=U(:,1:ns)*S(1:ns,1:ns)*V(:,1:ns)';
```

These matrices are of lower rank than the imGH matrix, and can be stored in a more efficient manner. (The imGH matrix is $3,000 \times 2,400$ real values and requires 7,200,000 numbers to store it. In contrast, imNs, with ns=100, requires subsets of (complex) U and V that are $3,000 \times 100$ and $2,400 \times 100$, and the largest 100 (real) values on the diagonal of S. Accounting for the fact that complex numbers require storing 2 real numbers, the total is reduced to 1,060,100. This is almost a factor of 7 savings in space from the original image.)

(g) Plot the "compressed" matrices as images using the following commands;

```
 figure();
colormap(gray(256));
image(ImNs);
daspect([1 1 1]);
title([num2str(ns) 'singular values']);
```

You should see only a minor difference between the original and the one with 100 singular values, while you should see major degradation of the image in the case of 10 singular values.

Exercise 7: Repeat Excercise 6 for any image you choose. You will likely need to use slightly different numbers of singular values depending on when your notice image degradation. Estimate the compression, i.e., the amount of space savings for storing a suitable truncated SVD representation vs. storing the full image.

Note: If your image is in color and have stored it to a variable image1, you can to convert it to grayscale using the command
`rgb2gray(image1)`

Alternatively, you can separate the image into 3 matrices: `image1(:,:,1)` stores red, `image1(:,:,2)` stores blue, and `image1(:,:,3)` stores green. You would then need to apply the SVD compression to each color separately.

(Modified version of lab from `http://www.math.pitt.edu/~sussmanm/2071Spring09/lab07/index.html`)