MATH2071: LAB 7: Factorizations

1 Introduction

We have seen that the PLU factorization 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. In a later lab, we will look at the "singular value decomposition" (SVD) that factors a matrix into the form $A = USV^T$.

The QR factorization will form the basis of a method for finding eigenvalues of matrices in a later lab. It can also be used to solve linear systems, especially poorly-conditioned ones that the PLU factorization can have trouble with. In addition, topics such as the Gram Schmidt method and Householder matrices have application in many other contexts.

You may find it convenient to print the pdf version of this lab rather than the web page itself. This lab will take two sessions.

2 Orthogonal Matrices

Definition: 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

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

One way to interpret this equation is that the columns of the matrix Q, when regarded as vectors, form an orthonormal set of vectors, and similarly for the rows. From the definition of an orthogonal matrix, and from the fact that the L^2 vector norm of x can be defined by:

$$||\mathbf{x}||_2 = \sqrt{(\mathbf{x}^T\mathbf{x})}$$

and the fact that

$$(A\mathbf{x})^T = \mathbf{x}^T A^T$$

you should be able to deduce that, for orthogonal matrices,

$$||Q\mathbf{x}||_2 = ||\mathbf{x}||_2$$

If I multiply a *two-dimensional* vector $\mathbf{x}$ by Q then its L^2 norm doesn't change, and so $Q\mathbf{x}$ must lie on the circle whose radius is $\|\mathbf{x}\|$. In other words, $Q\mathbf{x}$ is $\mathbf{x}$ 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.* Even in N dimensions, orthogonal matrices are often called rotations.

When matrices are complex, the term “unitary” is an analog to “orthogonal.” A matrix is unitary if $UU^H = U^HU = I$, where the H 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 a (possibly smaller) set of vectors $\mathcal{Q}$ that: (a) span the same space; (b) have unit L^2 norm; and, (c) are pairwise orthogonal. Note that if we can produce such a set of vectors, then we can easily answer several important questions. For example, the size of the set $\mathcal{Q}$ tells us whether the set $\mathcal{X}$ was linearly independent, and the dimension of the space spanned and the rank of a matrix constructed from the vectors. And of course, the vectors in $\mathcal{Q}$ are the orthonormal basis we were seeking. So you should believe that being able to compute the vectors $\mathcal{Q}$ is a valuable ability.

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 $\mathbf{x}$, the next (possibly the first) vector in $\mathcal{X}$;
3. For each vector $\mathbf{q}_i$ already in $\mathcal{Q}$, compute the projection of $\mathbf{q}_i$ on $\mathbf{x}$ (*i.e.,* $\mathbf{q}_i \cdot \mathbf{x}$). Subtract all of these projections from $\mathbf{x}$ to get a vector orthogonal to $\mathbf{q}_i$;
4. Compute the norm of (what's left of) $\mathbf{x}$. If the norm is zero (or too small), discard $\mathbf{x}$ from $\mathcal{Q}$; otherwise, divide $\mathbf{x}$ 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 $\mathcal{X}$ is finally empty, 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 $\mathbf{x}$ vectors:

Classical Gram-Schmidt algorithm

```
$n_q$ = 0 % $n_q$ will become the number of $\mathbf{q}$ vectors
for k = 1 to $n_x$
  $\mathbf{y} = \mathbf{x}_k$
  for $\ell$ = 1 to $n_q$      % if $n_q = 0$ the loop is skipped
    $r_{\ell k}$ = $\mathbf{q}_\ell \cdot \mathbf{x}_k$
    $\mathbf{y}$ = $\mathbf{y}$ - $r_{\ell k}\mathbf{q}_\ell$
  end
  $r_{kk}$ = $\sqrt{\mathbf{y} \cdot \mathbf{y}}$
  if $r_{kk}$ > 0
    $n_q$ = $n_q$ + 1
    $\mathbf{q}_{n_q}$ = $\mathbf{y}/r_{kk}$
  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? Note that the auxilliary vector $\mathbf{y}$ is needed here because without it the vector $\mathbf{x}_k$ would be changed *during* the loop on ℓ instead of only *after* the loop is complete.

The Gram-Schmidt algorithm as described above can be subject to errors due to roundoff when implemented on a computer. Nonetheless, in the first exercise you will implement it and confirm that it works well in some cases. In the subsequent exercise, you will examine an alternative version that has been modifed to reduce its sensitivity to roundoff error.

Exercise 1:

(a) Implement the Gram-Schmidt process in an m-file called `unstable_gs.m`. Your function should have the signature:

```
function Q = unstable_gs ( X )
% Q = unstable_gs( X )
% more comments

% 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 $\mathbf{x}_1$, so that the Matlab expression `X(k,1)` refers to the k^{th} component of the vector $\mathbf{x}_1$, and the Matlab expression `X(:,1)` refers to the Matlab column vector corresponding to $\mathbf{x}_1$. Similarly for the other columns. Include your code with your summary report.

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

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

You should be able to see that the correct result is the identity matrix. If you do not get the identity matrix, find your bug before continuing.

(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 1, and be pairwise orthogonal, and you should be able to confirm these facts "by inspection."

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

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

If your code is working correctly, you should compute approximately:

```
[  0.8944   0.3586   0.1952
  -0.4472   0.7171   0.3904
   0.0000  -0.5976   0.5855
   0.0000   0.0000  -0.6831 ]
```

You should 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?

(f) Consider a Hilbert matrix `X=hilb(10)`, and compute the result `Q1=unstable_gs(X)`. Is `Q1` an orthogonal matrix? Is it close to an orthogonal matrix? You do not need to include `Q1` in your summary, but please describe the results of the tests your performed.

(g) To examine `Q1` further, compute `B1=Q1'*Q1`, where `Q1` is the matrix you just computed from a Hilbert matrix. If `Q1` were orthogonal, `B1` would be the identity. Look at the 3×3 submatrix at the top left of `B1` (print the 3×3 submatrix `B1(1:3,1:3)`). Similarly, look at the 3×3 submatrix at the lower right of `B1`, and choose a 3×3 submatrix including the diagonal near the middle of `B1`. You should be able see how roundoff has grown as the algorithm progressed. Include these submatrices in your summary.

It would be foolish to use the unstable form of the Gram-Schmidt factorization when the modified Gram-Schmidt algorithm is only slightly more complicated. The modified Gram-Schmidt algorithm can be described in the following way.

Modified Gram-Schmidt algorithm

```
$n_q$ = 0 % $n_q$ will become the number of $\mathbf{q}$ vectors
for k = 1 to $n_x$
  $\mathbf{y} = \mathbf{x}_k$
  for $\ell$ = 1 to $n_q$      % if $n_q = 0$ the loop is skipped
    $r_{\ell k}$ = $\mathbf{q}_\ell \cdot \mathbf{y}$     % modification
    $\mathbf{y}$ = $\mathbf{y}$ - $r_{\ell k}\mathbf{q}_\ell$
  end
  $r_{kk}$ = $\sqrt{\mathbf{y} \cdot \mathbf{y}}$
  if $r_{kk}$ > 0
    $n_q$ = $n_q$ + 1
    $\mathbf{q}_{n_q}$ = $\mathbf{y}/r_{kk}$
  end
end
```

The modification seems pretty minor. In exact arithmetic, it makes no difference at all. Suppose, however, that because of roundoff errors a "little bit" of $\mathbf{q}_1$ slips into $\mathbf{q}_3$. In the original algorithm the "little bit" of $\mathbf{q}_1$ will cause $r_{31} = \mathbf{q}_3 \cdot \mathbf{x}_1$ to be slightly less accurate than $r_{31} = \mathbf{q}_3 \cdot \mathbf{y}$ because there is a "more of" $\mathbf{q}_1$ inside $\mathbf{x}_3$ than there is inside $\mathbf{y}$. In the next exercise you will see that the modified algorithm can be less affected by roundoff than the original one.

Exercise 2:

(a) Implement the modified Gram-Schmidt process in an m-file named `modified_gs.m`. Your function should have the signature:

```
function Q = modified_gs( X )
% Q = modified_gs( X )
% more comments

% your name and the date
```

Your code should be something like `unstable_gs.m`.

(b) Choose at least three different 3×3 matrices and compare the results from `modified_gs.m` with those from `unstable_gs.m`. The results should agree, up to roundoff. If they do not, fix `modified_gs.m`. Describe your tests completely in the summary in a manner that your tests can be replicated.

(c) Consider again the Hilbert matrix `X1=hilb(10)` and compute `Q2=modified_gs(X1)` and `B2=Q2'*Q2`. Show that `B2` is "closer" to the identity matrix than `B1` (from Exercise 1) by computing the Frobenius norm of the differences `norm(B1-eye(10),'fro')` and `norm(B2-eye(10),'fro')`. Include your results in your summary.

4 Gram-Schmidt Factorization

We need to take a closer look at the Gram-Schmidt process. Recall how the process of Gauss elimination could actually be regarded as a process of factorization. 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_{kk}, 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 $\mathbf{x}_k$ at the beginning of a loop, we *must* produce a vector q_k at the end of the loop, instead of dropping some out. Otherwise, the product QR will be missing some columns. The choice of column is not important because the corresponding row of R is zero, but the additional column of Q must result in Q being orthogonal. It is not hard to make such a choice, but it adds complexity. For the purpose of this exercise, if r_{kk}, the norm of vector $\mathbf{x}_k$, is zero, call `error()` with an appropriate error message.

Exercise 3:

(a) Make a copy of the m-file `modified_gs.m` and call it `gs_factor.m`. Modify it to compute the Q and R factors of a (possibly) rectangular matrix A. It should have the signature

```
function [ Q, R ] = gs_factor ( A )
% [ Q, R ] = gs_factor ( A )
% more comments

% your name and the date
```

Include a copy of your code with your summary.

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(...
```

When you *use* this syntax, it is OK (but not a great idea) 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. I recommend you use the comma all the time.

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

```
A = [ 1  1  1
      1  1  0
      1  0  0 ]
```

Compare the matrix `Q` computed using `gs_factor` with the matrix computed using `modified_gs`. They should be the same. Check that $A = QR$ and R is upper triangular. If not, fix your code now.

(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),'fro') 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 $A = QR$?

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

```
A = [ 0   0   1
      1   2   3
      5   8  13
     21  34  55 ]
```

- Is it true that $Q^TQ = I$?
- Is it true that $QQ^T = I$?
- Is Q orthogonal?
- Is the matrix R upper triangular?
- Is it true that $A = QR$?

5 Householder Matrices

It turns out that it is possible to take a given vector $\mathbf{d}$ and a given integer k and find a matrix H so that the product $H\mathbf{d}$ is proportional to the vector $\mathbf{e}_k = (0, \dots, 0, \underbrace{1}_{k}, 0, \dots, 0)^T$. That is, it is possible to find a matrix H that "zeros out" all but the k^{th} entry of $\mathbf{d}$. (In this section, we will be concerned primarily with the case $k = 1$.) The matrix is given by

$$\begin{aligned} \mathbf{v} &= \frac{\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k}{||(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)||} \\ H &= I - 2\mathbf{v}\mathbf{v}^T \end{aligned} \tag{1}$$

Note that $\mathbf{v}^T\mathbf{v}$ is a scalar (the inner or dot product) but $\mathbf{v}\mathbf{v}^T$ is a matrix (the outer or exterior product). To see why Equation (1) is true, denote $\mathbf{d} = (d_1, d_2, \dots)^T$ and do the following calculation.

$$\begin{aligned} H\mathbf{d} &= \mathbf{d} - 2\frac{(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)^T}{(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)^T(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)}\mathbf{d} \\ &= \mathbf{d} - \frac{2}{\|\mathbf{d}\|^2 - 2d_k\|\mathbf{d}\|) + \|\mathbf{d}\|^2}(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k)(\|\mathbf{d}\|^2 - \|\mathbf{d}\|d_k) \\ &= \mathbf{d} - \frac{2(\|\mathbf{d}\|^2 - \|\mathbf{d}\|d_k)}{(2\|\mathbf{d}\|^2 - 2d_k\|\mathbf{d}\|)}(\mathbf{d} - \|\mathbf{d}\|\mathbf{e}_k) \\ &= \|\mathbf{d}\|\mathbf{e}_k \end{aligned}$$

You can find further discussion in Quarteroni, Sacco & Saleri (Sec. 5.6.1), in a Planet Math encyclopedia article `http://planetmath.org/householdertransformtion`, or many other references.

There is a way to use this idea to take any column of a matrix and make those entries below the diagonal entry be zero, as is done in the k^{th} step of Gaußian factorization. This will lead to another matrix factorization.

Consider a column vector of length n split into two

$$\mathbf{b} = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ \hline b_k \\ b_{k+1} \\ \vdots \\ b_n \end{array}\right],$$

and denote part of $\mathbf{b}$ below the line as $\mathbf{d}$, a column vector of length $n-k+1$.

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-k+1} \end{bmatrix} = \begin{bmatrix} b_k \\ b_{k+1} \\ \vdots \\ b_n \end{bmatrix}.$$

Construct an $(n-k+1)\times(n-k+1)$ matrix H_d that changes $\mathbf{d}$ so it has a nonzero first component and zeros below it. Then

$$\left[\begin{array}{c|c} I & 0 \\ \hline 0 & H_d \end{array}\right] \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ \hline b_k \\ b_{k+1} \\ \vdots \\ b_n \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ \hline \|\mathbf{d}\| \\ 0 \\ \vdots \\ 0 \end{array}\right].$$

Setting

$$H = \begin{bmatrix} I & 0 \\ 0 & H_d \end{bmatrix}$$

allows us to write

$$H\mathbf{b} = H \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ \hline b_k \\ b_{k+1} \\ \vdots \\ b_n \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ \hline \|\mathbf{d}\| \\ 0 \\ \vdots \\ 0 \end{array}\right]. \tag{2}$$

The algorithm for computing H_d involves constructing a $(n-k+1)$-vector $\mathbf{v}$, the same size as $\mathbf{d}$. Once $\mathbf{v}$ has been constructed, it will be expanded to a n-vector $\mathbf{w}$ by adding $k-1$ leading zeros.

$$\mathbf{w} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-k+1} \end{bmatrix}$$

It should be clear that $H = I - 2\mathbf{w}\mathbf{w}^T$ satisfies

$$H = \begin{bmatrix} I & 0 \\ 0 & H_d \end{bmatrix}.$$

The following algorithm for constructing $\mathbf{v}$ and $\mathbf{w}$ includes a choice of sign that minimizes roundoff errors. This choice results in the sign of $(H\mathbf{b})_k$ in (2) being either positive or negative.

Constructing a Householder matrix

1. Set $\alpha = \pm\|\mathbf{d}\|$ with signum$(\alpha) = -$signum(d_1), and $\alpha > 0$ if $d_1 = 0$. (The sign is chosen to minimize roundoff errors.)
2. Set $v_1 = \sqrt{\frac{1}{2}\left[1 - \frac{d_1}{\alpha}\right]}$
3. Set $p = -\alpha v_1$
4. Set $v_j = \frac{d_j}{2p}$ for $j = 2, 3, \ldots, (n-k+1)$
5. Set $\mathbf{w} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \mathbf{v} \end{bmatrix}$.
6. Set $H = I - 2\mathbf{w}\mathbf{w}^T$.

The vector $\mathbf{v}$ is specified above in (1) and clearly is a unit vector. It is hard to see that the vector $\mathbf{v}$ as defined in the algorithm above is a unit vector satisfying (1). The calculation confirming that $\mathbf{v}$ is a unit vector is presented here. You should look carefully at the algorithm to convince yourself that $\mathbf{v}$ also satsifies (1).

Suppose you are given a vector $\mathbf{d}$. Define

$$\begin{aligned} \alpha &= \|\mathbf{d}\| \\ v_1 &= \sqrt{\frac{1}{2}\left(1 - \frac{d_1}{\alpha}\right)} \\ p &= -\alpha v_1 \\ v_j &= \frac{d_j}{2p} \text{ for } j \geq 2 \end{aligned}$$

Then the following calculation shows that **v** is a unit vector.

$$\begin{aligned}
\|\mathbf{v}\|^2 &= \frac{1}{2}\left(1-\frac{d_1}{\alpha}\right)+\sum\left(\frac{d_j^2}{4\alpha^2\left(\frac{1}{2}\left(1-\frac{d_1}{\alpha}\right)\right)}\right)\\
&= \frac{\alpha^2\left(1-\frac{d_1}{\alpha}\right)^2+\sum d_j^2}{2\alpha^2\left(1-\frac{d_1}{\alpha}\right)}\\
&= \frac{(\alpha-d_1)^2+\alpha^2-d_1^2}{2\alpha(\alpha-d_1)}\\
&= \frac{(\alpha-d_1)+(\alpha+d_1)}{2\alpha}=1
\end{aligned}$$

In the following exercise you will write a Matlab function to implement this algorithm.

Exercise 4:

(a) Start a function m-file named `householder.m` with signature and beginning lines

```
function H = householder(b, k)
% H = householder(b, k)
% more comments

% your name and the date

n = size(b,1);
if size(b,2) ~= 1
  error('householder: b must be a column vector');
end

d(:,1) = b(k:n);
if d(1)>=0
  alpha = -norm(d);
else
  alpha =  norm(d);
end
```

(b) Add comments at the beginning, being sure to explain the use of the variable `k`.

(c) In the case that `alpha` is exactly zero, the algorithm will fail because of a division by zero. For the case that `alpha` is zero, return

```
H = eye(n);
```

otherwise, complete the above algorithm.

(d) Test your function on the vector `b=[10;9;8;7;6;5;4;3;2;1];` with `k=1` and again with `k=4`. Check that `H` is orthogonal and `H*b` has zeros in positions `k+1` and below.
Debugging hints:

- **w** is an $n \times 1$ (column) unit vector.
- Orthogonality of `H` is equivalent to the vector **w** being a unit vector. If `H` is not orthogonal, check **w**.
- **v** is a $(n-k+1)\times 1$ (column) unit vector.
- No individual component of a unit vector can ever be larger than 1.

This `householder` function can be used for the QR factorization of a matrix by proceeding through a series of partial factorizations $A = Q_k R_k$, where Q_0 is the identity matrix, and R_0 is the matrix A. When we begin the k^{th} step of factorization, our factor R_{k-1} is only upper triangular in columns 1 to $k-1$. Our goal on the k-th step is to find a better factor R_k that is upper triangular through column k. If we can do this process $n-1$ times, we're done. Suppose, then, that we've partially factored the matrix A, up to column $k-1$. In other words, we have a factorization

$$A = Q_{k-1} R_{k-1}$$

for which the matrix Q_{k-1} is orthogonal, but for which the matrix R_{k-1} is only upper triangular for the first $k-1$ columns. To proceed from our partial factorization, we're going to consider taking a Householder matrix H, and "inserting" it into the factorization as follows:

$$\begin{aligned} A &= Q_{k-1} R_{k-1} \\ &= Q_{k-1} H^T H R_{k-1} \end{aligned}$$

Then, if we define

$$\begin{aligned} Q_k &= Q_{k-1} H^T \\ R_k &= H R_{k-1} \end{aligned}$$

it will again be the case that:

$$A = Q_k R_k$$

and we're guaranteed that Q_k is still orthogonal. Our construction of H guarantees that R_k is actually upper triangular all the way through column k.

Exercise 5: In this exercise you will see how the Householder matrix can be used in the steps of an algorithm that passes through a matrix, one column at a time, and turns the bottom of each column into zeros.

(a) Define the matrix `A` to be the magic square of order 5, `A=magic(5)`.

(b) Compute `H1`, the Householder matrix that knocks out the subdiagonal entries in column 1 of `A`, and then compute `A1=H1*A`. Are there any non-zero values below the diagonal in the first column?

(c) Now let's compute `H2`, the Householder matrix that knocks out the subdiagonal entries in column 2 of `A1` (not `A`), and compute `A2=H2*A1`. This matrix should have subdiagonal zeros in column 2 as well as in column 1. You should be convinced that you can zero out any subdiagonal column you want. Since zeroing out one subdiagonal column doesn't affect the previous columns, you can proceed sequentially to zero out *all* subdiagonal columns.

6 Householder Factorization

For a rectangular M by N matrix A, the Householder QR factorization has the form

$$A = QR$$

where the matrix Q is $M \times M$ (hence square and truly orthogonal) and the matrix R is $M \times N$, and upper triangular (or upper trapezoidal if you want to be more accurate.)

If the matrix A is not square, then this definition is different from the Gram-Schmidt factorization we discussed before. The obvious difference is the *shape* of the factors. Here, it's the Q matrix that is square. The other difference, which you'll have to take on faith, is that the Householder factorization is generally more accurate (smaller arithmetic errors), and easier to define compactly.

Householder QR Factorization Algorithm:

```
Q = I;
R = A;
for k = 1:min(m,n)
  Construct the Householder matrix H for column k of the matrix R;
  Q = Q * H';
  R = H * R;
end
```

Exercise 6:

(a) Write an m-file `h_factor.m`. It should have the form

```
function [ Q, R ] = h_factor ( A )
% [ Q, R ] = h_factor ( A )
% more comments

% your name and the date
```

Use the routine `householder.m` in order to compute the `H` matrix that you need at each step.

(b) A simple test is the matrix

```
A = [ 0 1 1
      1 1 1
      0 0 1 ]
```

You should see that simply interchanging the first and second rows of `A` turns it into an upper triangular matrix. The matrix `Q` that you get is equivalent to a permutation matrix, execpt possibly with (-1) in some positions. You can guess what the solution is. Be sure that `Q*R` is `A`.

(c) Test your code by computing the QR factorization of the matrix `A=magic(5)`. Visually compare your results with those from your `gs_factor`. **Warning:** `Q` remains an orthogonal matrix if one of its columns is multiplied by (-1). This is equivalent to multiplying on the right by a matrix that is like the identity matrix except with one (-1) on the diagonal instead of all (+1). Call this matrix `J`. The product `Q*J` is clearly orthogonal and can be incorporated into the QR factorization, but R is changed, too, by multiplying on the left by `J`. This changes the sign of the corresponding row of `R`. Aside from these signs, the two methods should yield the same results.

(d) There is a Matlab function for computing the QR factorization called `qr`. Visually compare your results from `h_factor` applied to the matrix `A=magic(5)` with those from `qr`.

(e) Householder factorization is *much* less sensitive to roundoff errors. Test `h_factor` on the same Hilbert matrix, `A=hilb(10)` that you used before in Exercise 2. Call the resulting orthogonal matrix `Q3`. Show that `B3=Q3'*Q3` is much closer to the identity matrix than either `B2` or `B1` (from Exercise 2) are.

7 The QR Method for Linear Systems

If we have computed the Householder 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 $Q^TQ = I$:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ QR\mathbf{x} &= \mathbf{b} \\ Q^TQR\mathbf{x} &= Q^T\mathbf{b} \\ R\mathbf{x} &= Q^T\mathbf{b} \qquad (3) \end{aligned}$$

so all we have to do is form the new right hand side $Q^T\mathbf{b}$ and then solve the upper triangular system. And that's easy because it's the same as the u_solve step of our old PLU solution algorithm from the previous lab.

Exercise 7: Make a copy of your file u_solve.m from the previous lab (upper-triangular solver), or get a copy of my version. Write a file called h_solve.m. It should have the signature

```
function x = h_solve ( Q, R, b )
% x = h_solve ( Q, R, b )
% more comments

% your name and the date
```

The matrix R is upper triangular, so you can use u_solve to solve (3) above. Assume that the QR factors come from the h_factor routine. Set up your code to compute $Q^T\mathbf{b}$ and then solve the upper triangular system $Rx = Q^T\mathbf{b}$ using u_solve.

When you think your solver is working, test it out on a system as follows:

```
    n = 5;
    A = magic ( n );
    x = [ 1 : n ]';
    b = A * x;
    [ Q, R ] = h_factor ( A );
    x2 = h_solve ( Q, R, b );
    norm(x - x2)/norm(x) % 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 in a later lab.

8 Extra Credit: Cholesky factorization (8 points)

In the case that the matrix A is symmetric positive definite, then A can be factored uniquely as LL^T, where L is a lower-triangular matrix with positive values on the diagonal. This factorization is called the "Cholesky factorization" and is both highly efficient (in terms of both storage and time) and also not very sensitive to roundoff errors.

You can find more information about Cholesky factorization in your text and also in a Wikipedia article http://en.wikipedia.org/wiki/Cholesky_decomposition.

Exercise 8: The Cholesky factorization can be described by the following equations.

$$\begin{aligned} L_{jj} &= \sqrt{A_{jj} - \sum_{m=1}^{j-1} L_{jm}^2} \\ L_{kj} &= \frac{1}{L_{jj}} \left(A_{kj} - \sum_{m=1}^{j-1} L_{km} L_{jm} \right) \qquad \text{for } j < k \le n \end{aligned}$$

(a) As you recall, the Matlab command A=pascal(6) generates a symmetric positive definite matrix based on Pascal's triangle, and the command L1=pascal(6,1) generates a lower triangular matrix based on Pascal's triangle. Further, you know that A=L1*L1'. Why is L1 *not* the Cholesky factor of A?

(b) Write a function m-file with signature and comments

```
function L=cholesky(A)
% L=cholesky(A)
% more comments

% your name and the date
```

(c) Test your function on the matrix `A=(eye(6)+pascal(6))` by computing `L=cholesky(A)`. Check that `L*L'=A`.

(d) Matlab has a built-in version of Cholesky factorization named `chol`. By default this function returns an *upper* triangular matrix, but `chol(A,'lower')` returns a lower triangular matrix. Compare the results from `chol` with those from your `cholesky` function on a matrix of your choice. Be sure to describe your test fully in your summary.

(e) Download my copy of `l_solve.m`. This function is similar to the one you wrote in the previous lab, but it is modified to avoid the assumption that the lower triangular matrix has ones on the diagonal.

(f) Use the following code to generate a 50×50 positive definite symmetric matrix (using Gershgorin's theorem) along with solution and right side vectors.

```
N = 50;
A = rand(N,N);         % random numbers between 0 and 1
A = .5 * (A + A');     % force A to be symmetric
A = A + diag(sum(A)); % make A positive definite by Gershgorin
x = rand(N,1);         % solution
b = A*x;               % right side
```

(g) Use your `cholesky` function along with `l_solve` and `u_solve` to solve the system $Ax_0 = b$ and show that $\|x_0 - x\|/\|x\|$ is small. It turns out that solving by Cholesky factorization is generally the most efficient way to solve a positive definite symmetric matrix system.

Last change `$Date: 2017/01/31 16:24:40 $`