Note 1: This assignment involves coding. It is OK to talk with other people, but please write your own code. It is very obvious when one person just copies another persons code, or reproduces something they found on the internet without understanding it. (It is especially obvious when you ask them to explain their code!) So, try to code things by yourself, talking to other people or checking other resources only if you get _really_ stuck. This will make you stronger! Also, staring at code for a few minutes and being confused does not count as getting stuck. This is part of coding! Progress is usually not continuous. Programming is a puzzle that you solve, not a bucket that you fill.

Note 2: Next week on 2016.03.10 (i.e., Thursday March 10th), we will learn about another tool to make your codes even faster for certain types of matrices. We will have a short bonus assignment over that weekend which is a contest to see who can code the best and fastest solver! The details of this contest will be released on 2016.03.10. In the meantime, try to make your codes as good as possible to be ready for the contest!

-1. Review the Gram-Schmidt method from Linear Algebra.

0. Read sections 12.1, 12.2, 13.1, 13.2 in the book. You might also want to read my notes on Steepest Descen,t posted on the webpage at:
   www.math.unl.edu/~alarios2/courses/2016_spring_M433/content.shtml

1. Page 408, #2.1. Do the calculations by hand (show your work). This will help you understand what is going on, and give you a feeling for how fast the computations are. Think about what the matrix A and the vector $\mathbf{b}$ are in this problem.

2. Code the Steepest Descent algorithm. First try the "Naïve form" we saw in class, then the "improved form". Is one actually faster than the other? Does the speed depend on anything, such as the matrix size, etc.? If so, when at what size can you see a difference? To keep things uniform, please start you code like this:

```
1  function [x,iter] = steepestDescent(A,b,x0,maxIter,tol)
2  % Use Steepest Descent algorithm to solve Ax = b.
```

where `x0` is the initial guess. `maxIter` is the maximum number of iterations, and `tol` is the tolerance. The code should stop once `maxIter` iterations have occured, or the desired level of accuracy is reached, determined by the tolerance `tol`. It is up to you what to base the tolerance on though (make it reasonable though).

Next, test you code, maybe try something like this:

```
1  % A program to test linear solvers.
2  n = 100;
3  L = tril(rand(n,n)); % Random lower-triangular matrix
4  A = L*L'; % Random SPD matrix;
5  x_exact = rand(n,1); % Random vector (the exact solution)
6  b = A*x_exact; % Now x_exact is the exact solution of Ax = b.
7  tic; % Mark the starting time
8  %
9  % Here, call your steepestDescent program to solve Ax = b
10 % WITHOUT using x_exact.  Find x and the number of iterations.
11 %
12 time = toc;
13 error = norm(x_exact - x)/norm(x);
14 output_string = 'Error = %g, time = %g, iterations = %d';
15 display(sprintf(output_string,error,time,iter));
```

3. Plot a graph of the matrix size n vs. the number of iterations for tolerance
   tol = 1e-6 for n = 100, 110, 120,…,1000. You may want to review the
   Matlab intro on plotting, located at:

www.math.unl.edu/~alarios2/courses/2016_spring_M433/documents/matlabIntroLarios.pdf

   It may help to turn the above test code into a function, or wrap it in a loop.
   Does the plot show what you expect it to show?

4. Repeat problems 2 and 3 for the Conjugate Gradient algorithm (see page 454
   in your book for the algorithm). Try to make your code as fast and efficient as
   possible, while still being readable!

> Print out all your codes and the two graphs (plot titles and labels for the x
> and y axes are required!), and turn them in, stapled to your homework.