Name:______________________________ **Assigned: 2016.03.03**

Math 433/833 **Assignment # 5** **Due: 2016.03.10**

Thanks to Gary Wright for writing up some of these solutions.

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 Descent, 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.

   Use the steepest-descent method to solve: *minimize* $f(x_1, x_2) = 4x_1^2 + 2x_2^2 + 4x_1x_2 - 3x_1$, starting from the point $(2, 2)^T$. Perform three iterations.

   Solution. First we'll need to find what A and $\vec{b}$ are. We can find A by finding the Hessian of f and we can find $\vec{b}$, by finding the gradient and considering the values which contain no variables. Doing this we get:

$$A = \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \text{ and } \vec{b} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$$

   Next, we'll perform the three iterations of steepest-descent.

first iteration:

$$\vec{r_0} = \vec{b} - A\vec{x_0} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} - \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} -21 \\ -16 \end{pmatrix}$$

$$\alpha_0 = \frac{\vec{r_0}^T \vec{r_0}}{\vec{r_0}^T A \vec{r_0}} = \frac{\begin{pmatrix} -21 \\ -16 \end{pmatrix}^T \begin{pmatrix} -21 \\ -16 \end{pmatrix}}{\begin{pmatrix} -21 \\ -16 \end{pmatrix}^T \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} -21 \\ -16 \end{pmatrix}} \approx 0.09627$$

$$\vec{x_1} = \vec{x_0} + \alpha_0 \vec{r_0} = \begin{pmatrix} 2 \\ 2 \end{pmatrix} + 0.09627 \begin{pmatrix} -21 \\ -16 \end{pmatrix} \approx \begin{pmatrix} -0.02169 \\ 0.45967 \end{pmatrix}$$

second iteration:

$$\vec{r_1} = \vec{b} - A\vec{x_1} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} - \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} -0.02169 \\ 0.45967 \end{pmatrix} \approx \begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix}$$

$$\alpha_1 = \frac{\vec{r_1}^T \vec{r_1}}{\vec{r_1}^T A \vec{r_1}} = \frac{\begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix}^T \begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix}}{\begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix}^T \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix}} \approx 0.62011$$

$$\vec{x_2} = \vec{x_1} + \alpha_1 \vec{r_1} = \begin{pmatrix} -0.02169 \\ 0.45967 \end{pmatrix} + 0.62011 \begin{pmatrix} 1.33481 \\ -1.75193 \end{pmatrix} \approx \begin{pmatrix} 0.80604 \\ -0.62672 \end{pmatrix}$$

third iteration:

$$\vec{r_2} = \vec{b} - A\vec{x_2} = \begin{pmatrix} 3 \\ 0 \end{pmatrix} - \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} 0.80604 \\ -0.62672 \end{pmatrix} \approx \begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix}$$

$$\alpha_2 = \frac{\vec{r_2}^T \vec{r_2}}{\vec{r_2}^T A \vec{r_2}} = \frac{\begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix}^T \begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix}}{\begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix}^T \begin{pmatrix} 8 & 4 \\ 4 & 4 \end{pmatrix} \begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix}} \approx 0.09627$$

$$\vec{x_3} = \vec{x_2} + \alpha_2 \vec{r_2} = \begin{pmatrix} 0.80604 \\ -0.62672 \end{pmatrix} + 0.09627 \begin{pmatrix} -0.94144 \\ -0.71728 \end{pmatrix} \approx \begin{pmatrix} 0.71541 \\ -0.69577 \end{pmatrix}$$

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:

```
function [x,iter] = steepestDescent(A,b,x0,maxIter,tol)
% 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).

Solution. Answers may vary considerably. Below is one possible solution.

```
function [x,iter] = steepestDescent(A,b,x0,maxIter,tol)
% Use Steepest Descent algorithm to solve Ax = b.
% Example usage:
%   n = 100; A = rand(n)-0.5; A = A'+A+n*eye(n);
%   x = rand(n,1); b = A*x;
%   [x_sd,iter] = steepestDescent(A,b,b,100,1e-6);
%   display([norm(x-x_sd)/norm(x),iter]);

x = x0;
r = b-A*x;
for iter = 1:maxIter
    r_sq = r'*r;
    if (r_sq < tol)
        break
    end
    Ar = A*r;
    alpha = r_sq/(r'*Ar);
    x = x + alpha*r;
    r = r - alpha*Ar;
end
```

Naïve version of the algorithm:

```
function [x,iter] = steepestDescentNaive(A,b,x0,maxIter,tol)
% Use Steepest Descent algorithm to solve Ax = b.
% Example usage:
%   n = 100; A = rand(n)-0.5; A = A'+A+n*eye(n);
%   x = rand(n,1); b = A*x;
%   [x_sd,iter] = steepestDescentNaive(A,b,b,100,1e-6);
%   display([norm(x-x_sd)/norm(x),iter]);

x = x0;
for iter = 1:maxIter
    r = b-A*x;
    r_sq = r'*r;
    if (r_sq < tol)
        break
    end
    alpha = r'*r/(r'*(A*r));
    x = x + alpha*r;
end
```

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

```
% A program to test linear solvers.
n = 100;
L = tril(rand(n,n)); % Random lower-triangular matrix
A = L*L'; % Random SPD matrix;
x_exact = rand(n,1); % Random vector (the exact solution)
b = A*x_exact; % Now x_exact is the exact solution of Ax = b.
tic; % Mark the starting time
%
% Here, call your steepestDescent program to solve Ax = b
% WITHOUT using x_exact.  Find x and the number of iterations.
%
time = toc;
error = norm(x_exact - x)/norm(x);
output_string = 'Error = %g, time = %g, iterations = %d';
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-3` 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?

Solution. Answers may vary greatly here. Roughly speaking, the number of iterations should increase as the matrix size increases. However, `maxIter` can easily be reached quickly.

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 you can, 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.

Name:______________________________ **Assigned: 2016.03.03**

Math 433/833 **Assignment # 5** **Due: 2016.03.10**

Solution. Answers may vary considerably. Below is one possible implementation of the conjugate gradient method. It is not optimal, but hopefully it is somewhat easier to read that an optimized code.

```
function [x,iter] = conjugateGradient(A,b,x0,maxIter,tol)
% Use Conjugate Gradient algorithm to solve Ax = b.
% Example usage:
%     n = 1000; A = tril(rand(n)); A = A*A';
%     x = rand(n,1); b = A*x;
%     [x_sd,iter] = CG(A,b,b,100,1e-6);
%     display([norm(x-x_sd)/norm(x),iter]);

x = x0;
r = b - A*x; % residual
p = r;       % search direction
for iter = 1:maxIter
    r_sq = r'*r; % Precompute for speed.
    if (sqrt(r_sq) < tol)
        % If residual is small enough, stop.
        break
    end
    Ap = A*p; % Precompute for speed.
    alpha = r_sq/(p'*Ap);
    x = x + alpha*p;
    r = r - alpha*Ap;
    beta = r'*r/r_sq; % r is now the new r, r_sq uses old r
    p = r + beta*p;
end
```