MSSC 6040 - Take-home Final Exam, Fall 2022

Henri Medeiros Dos Reis

Due: Thurs, Dec 15, 11:59 pm by upload to D2L dropbox.

Instructions:

1. **Choose 4 of the 6 problems to do**; each problem you do is worth 25 pts, and your total score will be out of 100 pts.
2. Submit your completed exam to the D2L dropbox by the specified deadline. No late work will be accepted.
3. To speed up grading, **submit your exam as one ".pdf" file**; please, no MS Word ".docx" files.
4. All code and plots need to be included in the single pdf. Do not submit any MATLAB *.m* files.
5. **RULES FOR OUTSIDE SOURCES**
   It is permissible to consult:
   (a) The textbook.
   (b) *Your own* course notes.
   (c) Any resources on D2L, including lecture notes, HW solution guides, and MATLAB scripts.
   (d) The internet for base-level questions: MATLAB syntax, basic definitions of terms, etc.
   (e) Me, via email, for clarification on the problem statement, but I will not give out hints for any of the problems.

   It is NOT permissible to consult:
   (a) Any classmates, other students, other professors, etc.
   (b) Anybody else's course notes (it is okay if you had previously borrowed someone else's notes because you missed a day, etc).
   (c) The internet for anything remotely approaching the actual question asked.
   - For example, it is okay to search the internet for "How to compute eigenvalues in MATLAB", but it is not okay to search the internet for the actual question.
   - If you are in doubt about what is okay, email me!

Any instances of plagiarism, working with classmates, or other cheating, will result in a score of 0 for the entire final exam.

Problem 1. The least squares problem $\min_x \|Ax-b\|_2$ can be solved via conjugate gradients (CG) applied to the normal equations $A^*Ax = A^*b$. However, the naive application of CG to the normal equations leads to a potentially unstable algorithm, especially when A has a large condition number. A more stable re-ordering of the CG updates, known as conjugate gradient least squares (CGLS), is described below:

Algorithm: Conjugate Gradients Least Squares (CGLS)
Initialize x_0, and set $r_0 = b - Ax_0$, $s_0 = A^*r_0$, $p_0 = s_0$, $\delta_0 = s_0^*s_0$.
For $k = 0, 1, 2, ...$ until $\delta_k < \varepsilon_{tol}^2\|A^*b\|_2^2$ or a maximum number of iterations is reached, do
$q_k = Ap_k$
$\alpha_k = \delta_k/(q_k^*q_k)$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k q_k$
$s_{k+1} = A^*r_{k+1}$
$\delta_{k+1} = s_{k+1}^*s_{k+1}$
$p_{k+1} = s_{k+1} + (\delta_{k+1}/\delta_k)p_k$
end

(a) Define a MATLAB function `[x,flag,relres,iter] = cgls(x0,b,A,At,tol,maxit)` that implements the CGLS algorithm above, where `x0` is a vector representing the initialization x_0, `b` is the b vector in the least squares problem, `A` is a function handle such that `A(x)` computes Ax, `At` is a function handle such that `At(y)` computes A^*y, `tol` is the tolerance parameter ε_{tol} appearing in the exit condition, and `maxit` is the maximum number of iterations to run. The output `x` is the final x-iterate returned by CGLS, `flag` is equal to `0` if CGLS reached the exit condition $\delta_k < \varepsilon_{tol}^2\|A^*b\|_2^2$ within `maxit` iterations and is equal to `1` otherwise, `relres` is the value of the relative residual $\|Ax-b\|_2/\|b\|_2$ for the returned x, and `iter` is the total number of CGLS iterations that were run.

Include a printout/screenshot of the `cgls` function you defined in your writeup.

(b) Test out your `cgls` function on the small-scale least squares problem where:

$$A = \begin{bmatrix} 5 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 5 \\ 1 & 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Call your `cgls` function using the following code:

```
Amat = [5 1 1; 1 5 1; 1 1 5; 1 1 1]; b = [1; 2; 3; 4];
A = @(x) Amat*x;
At = @(x) Amat'*x;
x0 = [0; 0; 0];
[x,flag,relres,iter] = cgls(x0,b,A,At,1e-10,10);
```

Report the returned values of `flag`, `relres`, and `iter`. Compare the solution `x` you obtained via CGLS with the solution obtained via backslash: `Amat\b`. These should be identical.

(c) Apply your `cgls` function to solve the deblurring problem setup for you in the provided script `problem1.m`. Here, function handles for the blurring operator A and its adjoint A^* are defined for you in `A` and `At`, and the b vector (the vectorized blurry image) is loaded as `b`. The image to deblur is shown below:

Set the initialization to be a vector of all zeros `x0 = zeros(prod(dim),1)`, and set `tol = 1e-5`, and `maxit = 200`. Report the returned values of `flag`, `relres`, and `iter`. Include a visualization of the deblurred image `x` with the commands:

```
imagesc(reshape(x,dim),[0,1]);
axis image; axis off; colormap gray
```

Solution 1. a-) CGLS algorithm in MATLAB

```
%% Conjugate Gradients Least Squares
function [x,flag,relres,iter] = cgls(x0,b,A,At,tol,maxit)
    x = x0;
    r = b-A(x);
    s = At(r);
    p = s;
    delta = s'*s;
    for k=1:maxit
        q = A(p);
        alpha = delta/(q'*q);
        x = x + alpha*p;
        r = r - alpha*q;
        s = At(r);
        delta_prev = delta;
        delta = s'*s;
```

```
        p = s+(delta/delta_prev)*p;
        iter = k;

        if delta < tol*norm(At(b),2)
            flag = 0;
            relres = norm(A(x)-b,2)/norm(b,2);
            break;
        end

        if k == maxit
            flag = 1;
        end
    end
end
```

b-) The results of applying my function are:
flag = 0
relres = 0.5570
iter = 2

```
disp(x)
disp(Amat\b)
%-----------OUTPUT----------------
%    0.0962
%    0.3462
%    0.5962

%    0.0962
%    0.3462
%    0.5962
```

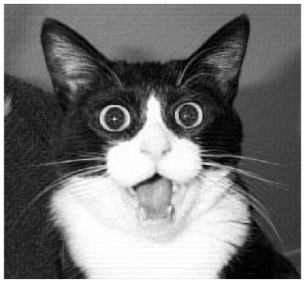
c-) Code and results of using my function to deblur the image:

```
%% load blurry image
load blurryimg %loads vectorized blurry image, b
dim = [472, 510]; %image dimensions

figure; % show blurry image
imagesc(reshape(b,dim),[0,1]); axis image; axis off;
   colormap gray
title('blurry image, b')

%% define A and At as function handles
```

```
vec = @(x) x(:); %helper functions for reshaping
unvec = @(x) reshape(x,dim);
h = ones(9,9)/81; %9x9 uniform blur kernel
A = @(x) vec(conv2(unvec(x),h,'same'));
At = A; %A is self-adjoint

%% Todo: run CGLS algorithm to deblur the image
% visualize the output x with the command below
x0 = zeros(prod(dim),1);
tol = 1e-5;
maxit = 200;
[x , flag , relres , iter ] = cgls (x0 ,b ,A , At ,tol ,
   maxit) ;
figure(2)
imagesc(reshape(x,dim),[0,1]); axis image; axis off;
   colormap gray
```

Results:
flag $= 0$
relres $= 0.0013$
iter $= 25$

Deblured image



Problem 2. The condition number of a matrix A is the ratio of largest to smallest singular values. In particular, if $A \in \mathbb{C}^{m\times m}$ is Hermitian, $\kappa(A) = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}$, where $\lambda_{\max}$ is the largest magnitude eigenvalue of A, and $\lambda_{\min}$ is the smallest magnitude eigenvalue of A.

(a) Devise and implement an algorithm in MATLAB that computes an approximation of the condition number $\kappa(A)$ of a Hermitian matrix $A \in \mathbb{C}^{m\times m}$ *assuming you only know how to compute matrix-vector products with* A. In other words, assume you are given a function `A()` in MATLAB such that `A(x)` gives the matrix-vector product Ax for any vector x.

Do not search in the literature or on the internet for such an algorithm; you are to come up with the idea on your own. You *may not* use the commands `condest`, `svds` or `eigs`. You *may* use the command `pcg` if you wish.

(b) Test your algorithm on the "matrix" $A \in \mathbb{R}^{10\times 10}$ defined by the function:

```
A = @(x) (1:10)'.*x;
```

Note that $\kappa(A) = 10$. Ensure that your algorithm returns a value within $[9.9, 10.1]$

(c) Use your algorithm to estimate the condition number of the discrete Laplacian "matrix" $A \in \mathbb{R}^{40{,}000\times 40{,}000}$ defined in MATLAB by the following code-snippet:

```
% define A as a function handle
dim = [200,200];
vec = @(x) x(:); %helper functions for reshaping
unvec = @(x) reshape(x,dim);
h = [0 -1 0; -1 4 -1; 0 -1 0]; %discrete Laplacian
A = @(x) vec(conv2(unvec(x),h,'same'));

% test out A function
x = randn(40000,1);
y = A(x); %equivalent to A*x
```

Solution 2. a-) Function that implements my algorithm to compute the condition number of a matrix:

```
function cond_num = condition(size,A)
    % find largest eigen value
    v = randn(size,1); %random initialization
    v = v/norm(v);
    for k = 1:1000
        u = A(v);
        v = u/norm(u);
        lam_larg = v'*(A(v));
        relres = norm(A(v)-lam_larg*v)/norm(v); %should be
           close to zero;
        if(relres < 1e-10)
            break
        end
    end
    % find smallest eigen value
    v = randn(size,1); %random initialization
    v = v/norm(v);
    for k = 1:1000
```

```
        % need to solve Au = v
        u = v;
        r = v-A(u);
        p = r;
        delta = r'*r;
        for k=1:5000
            S = A(p);
            alpha = delta/(p'*S);
            u = u + alpha*p;
            r = r - alpha*S;
            prev_delt = delta;
            delta = r'*r;
            p = r+delta/prev_delt*p;

            if delta < 0.01
                %fprintf('reached exit tol at iter %d\n',k);
                break;
            end
        end
        v = u/norm(u);
        lam_small = v'*(A(v));
        % verify we found an eigenvalue:
        relres = norm(A(v)-lam_small*v)/norm(v); %should be
           close to zero
        if(relres < 1e-10)
            break
        end
    end
    cond_num = lam_larg/lam_small;
    disp(lam_small)
    disp(lam_larg)
end
```

b-) Code and results for testing my algorithm:

```
size = 10;
A = @( x ) (1:10)'.* x ;
my_con = condition(size, A);
disp(my_con)
%------Output----------------
% 10
```

c-) Code and results for testing my algorithm:

```
% define A as a function handle
dim = [200 ,200];
vec = @( x ) x (:) ; % helper functions for reshaping
unvec = @( x ) reshape (x , dim ) ;
h = [0 -1 0; -1 4 -1; 0 -1 0]; % discrete Laplacian
A = @( x ) vec ( conv2 ( unvec ( x ) ,h , 'same') ) ;
% test out A function
x = randn (40000 ,1) ;
y = A ( x ) ; % equivalent to A*x

my_con = condition(40000, A);
disp(my_con)
%------Output----------------
% 1.6360e+04
```

Problem 3. Suppose we observe an incomplete subset of the entries of a matrix X, and we wish to somehow recover the missing entries. For example, consider the following 4×4 matrix where we only observe 9 of the 16 entries:

$$X = \begin{bmatrix} 1 & & 3 & \\ & -2 & & -4 \\ & 4 & 6 & \\ -2 & & -6 & -8 \end{bmatrix}.$$

Low-rank matrix completion is a strategy used to fill in the missing entries under the assumption that the completed matrix is low-rank. Your task for this problem is to implement a simple algorithm for low-rank matrix completion called "iterative singular value hard thresholding" (ISVHT). The ISVHT algorithm proceeds as follows: First, to initialize, we replace all the missing entries with zeros. Then we alternate between two steps: (1) low-rank approximation of the matrix by a rank-r truncated SVD, and (2) put back in the observed entries. These two steps are repeated until convergence. Pseudocode for ISVHT is below:

Algorithm: Iterative Singular Value Hard Thresholding (ISVHT)
Choose a target rank r. Initialize X by putting in zeros for missing entries.
For $k = 0, 1, 2, \ldots$ until converged, do
  $[U, \Sigma, V] = \text{svd}(X)$
  $X \leftarrow U_r \Sigma_r V_r^*$ `%step 1: replace X with its rank-r truncated SVD`
  $x_{i,j} \leftarrow \tilde{x}_{i,j}$ for all $(i, j) \in \Omega$ `%step 2: put back in the observed entries`

Here $A \leftarrow B$ means we overwrite variable A with variable B; $U_r \in \mathbb{R}^{m \times r}$ and $V_r \in \mathbb{R}^{n \times r}$ are the restrictions of U and V to their first r columns, and $\Sigma_r \in \mathbb{R}^{r \times r}$ is the restriction of Σ to its first r rows and first r columns. Also, Ω denotes the set of indices of observed entries, and $\tilde{x}_{i,j}$ denotes the observed (i, j)-entry of X.

(a) Perform matrix completion on the 4×4 matrix X given above by running 1000 iterations of the ISVHT algorithm with rank $r = 1$. Verify that ISVHT recovers a matrix that is approximately rank one and agrees with the observed entries. *Note: in MATLAB it is convenient to use a logical mask to put back in the observed entries. For example, the following code snippet can be used to initialize X for the ISVHT algorithm.*

```
Omega = logical([1 0 1 0; 0 1 0 1; 0 1 1 0; 1 0 1 1]);
obs_val = [1; -2; -2; 4; 3; 6; -6; -4; -8];
X = zeros(4,4); % initialize X with all zeros
X(Omega) = obs_val; %put back in observed entries
```

(b) Load the variables stored in the provided file `matcomplete.mat` using the command `load matcomplete`. This loads a logical mask `Omega` and a vector of observed values `obs_val` from a 411×601 matrix. The observed values are a random subset of half the pixels from an image of a famous artwork. A zero-filled version of the incomplete matrix is shown below:

Your goal "restore" this artwork, i.e., fill in the missing pixels, using ISVHT. To do so, run 100 iterations of ISVHT for a few different integer values of the rank cutoff r in the range $[5, 50]$ (try at least three). Visualize the completed matrix with the command:

```
imagesc(X,[0,1]); axis image; axis off; colormap gray
```

Evaluate the visual quality of the reconstructions for the different values of the rank cutoff r, and select what you think is the "best" one.

(c) In one paragraph, briefly describe a possible strategy for automatically selecting the "best" rank cutoff parameter r (i.e., without a human in the loop) in the ISVHT algorithm applied to a general low-rank matrix completion problem.

Solution 3. a-) The matrix generated by the algorithm is

$$\begin{bmatrix} 1.0000 & 2.0000 & 3.0000 & 4.0000 \\ -1.0000 & -2.0000 & -3.0000 & -4.0000 \\ 2.0000 & 4.0000 & 6.0000 & 8.0000 \\ -2.0000 & -4.0000 & -6.0000 & -8.0000 \end{bmatrix}$$

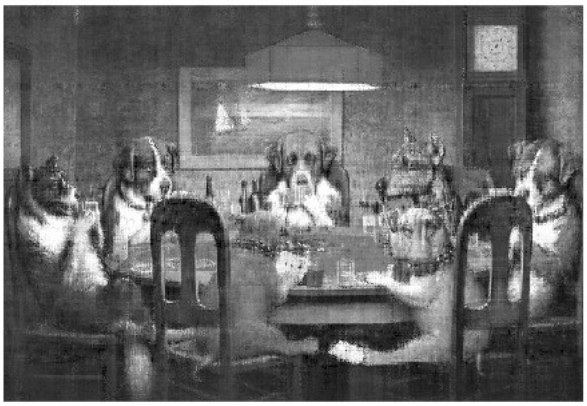
And the code used:

```
r = 1;  r
for i = 1:1000
    [U,S,V] = svd(X);
    X = U(:,1:r)*S(1:r,1:r)*V(:,1:r)';
    X(Omega) = obs_val;
end
```

b-) Code and results:

```
load matcomplete
X = zeros(411,601);
X ( Omega ) = obs_val ;

r = 14; % chose a target rank between [5,50]
% initialize X, defined above?

for i = 1:100 %change no num interations
    [U,S,V] = svd(X);
    X = U(:,1:r)*S(1:r,1:r)*V(:,1:r)';
    X(Omega) = obs_val;
end
% visualize
imagesc(X,[0,1]);axis image;axis off; colormap gray
```

Reconstructed image



c-) In order to select the "best" rank cutoff for a matrix we need to pick the smallest number possible that would still give out good results. But the problem is that the definition of good depends on the application and also on the size of the matrix. A good approach that we could use is to look at the largest singular values and the number above a certain number of singular values would define the rank. Unfortunately, even though this seems like a nice way to solve this problem, a human would still has to set the number that would define what a large singular value is. Then, the only other solution that I could think is to base the rank on the size of the matrix and take care of some special cases.

```
if(min(size(MyMatrix)) < 20  % check if the matrix is too
   small
    r = 1                    % if so, rank can probably be 1
else
    r = fix(min(size(MyMatrix)*0.05)
    % if not, use 5% of the smallest side of the matrix
end
```

Problem 4. A Hermitian matrix A is positive definite if and only if all *pivot entries*[1] encountered while performing Gaussian elimination (without pivoting) on A are positive.

(a) Define a general function `is_posdef(A)` in MATLAB that uses this property to determine whether a Hermitian matrix `A` is positive definite. Your function should return `1`

[1]Gaussian elimination reduces a matrix A to an upper triangular matrix U; the "pivot entries" are the entries appearing along the main diagonal of U.

if `A` is positive definite and `0` if `A` is not positive definite. Your function should avoid performing any unnecessary calculations, and exit as soon as a non-positive pivot is encountered (the command `break` may be useful for this). You *may not* use any built-in matrix factorization routines in MATLAB such as `lu`, `chol`, `ldl`, `qr`, `eig`, `svd`, etc. Your function does not need to check whether A is square or Hermitian.

(b) Test your function on the following matrices:

```
A = 101*eye(100)-ones(100,100);
B = gallery('fiedler',1:10);
C = invhilb(5);
D = diag(1.5*ones(1,10)) - diag(ones(1,9),1) ...
    - diag(ones(1,9),-1);
```

Solution 4. a-) The MATLAB function to check if it is positive definite:

```
function posdef = is_posdef(A)
    s = size(A,1);
    flag = 0;
    for k = 1:s-1 %cols
        if(A(k,k)<=0)
            posdef = 0;
            flag = 1;
            break
        end

        if(k == s-1)
            if(A(k+1,k+1)<0)
                flag = 1;
                posdef = 0;
                break
            end
        end
        if(flag == 1)
            break;
        end
        for i = k+1:s %rows
            l = A(i,k)/A(k,k);
            for j = k+1:s
               A(i,j) = A(i,j)-l*A(k,j);
            end
        end
    end
    if(flag ~= 1)
```

```
        posdef=1;
    end
end
```

b-) Code to test and output:

```
is_posdef(A) % 1
is_posdef(B) % 0
is_posdef(C) % 1
is_posdef(D) % 0
```

A is positive definite.
B is not positive definite.
C is positive definite
D is not positive definite

Problem 5.

(a) Devise and implement an algorithm to numerically approximate the induced p-norm of a matrix for any $p \in [1, \infty)$. *Do not search in the literature or on the internet for such an algorithm; you are to come up with the idea on your own.* Explain how your algorithm works, including any pictures/plots that you think are helpful.

(b) Use your algorithm to approximate the p-norm of each of the matrices below for the values $p = 1, 2, 4, 10, 100$.

```
A = [5 1 -5; -4 3 -8; 1 -1 5];
B = [5 8 8 -3; 9 -9 7 6; -7 5 -3 -8; -3 -9 -5 2];
C = [-2 -5 4 -6 -5; -6 7 2 -2 -5;...
     -7 -9 -6 3 9; -5 -1 -8 -9 5; -3 5 9 7 -7];
```

(c) MATLAB only lets you compute matrix p-norms for $p = 1, 2, \infty$, so compare your answers to the 1- and 2-norm computed by MATLAB.

Problem 6. (Trefethen-Bau, Exercise 12.3) The goal of this problem is to explore some properties of random matrices. Your job is to be a laboratory scientist, performing experiments that lead to conjectures and more refined experiments. Do not try to prove anything. Do produce well-designed plots, which are worth a thousand numbers.

Define a random matrix to be an $m \times m$ matrix whose entries are independent samples from the real normal distribution with mean zero and standard deviation $m^{-1/2}$. (In MATLAB: `A = randn(m,m)/sqrt(m)`.) The factor $\sqrt{m}$ is introduced to make the limiting behavior clean as $m \to \infty$.

(a) What do the eigenvalues of a random matrix look like? What happens, say, if you take 100 random matrices and superimpose all of their eigenvalues in a single plot? If you do this for $m = 8, 16, 32, 64, \ldots$, what pattern is suggested? How does the spectral radius $\rho(A)$ (Exercise 3.2) behave as $m \to \infty$?

(b) What about norms? How does the 2-norm of a random matrix behave as $m \to \infty$? Of course, we must have $\rho(A) \leq \|A\|_2$ (Exercise 3.2). Does this inequality appear to approach as equality as $m \to \infty$?

(c) What about condition numbers (see the section for the definition of the condition number of a matrix)—or more simply, the smallest singular value $\sigma_{\min}$? Even for fixed m this question is interesting. What proportions of random matrices in $\mathbb{R}^{m \times m}$ seem to have $\sigma_{\min} \leq 2^{-1}, 4^{-1}, 8^{-1}, \ldots$? In other words, what does the tail of the probability distribution of smallest singular values look like? How does the scale of all this change with m?

(d) How do the answers to (a)–(c) change if we consider random triangular instead of full matrices, i.e., upper-triangular matrices whose entries are samples from the same distribution as above?