Nathan Jo, Edward Holappa, Griffin Weinhold

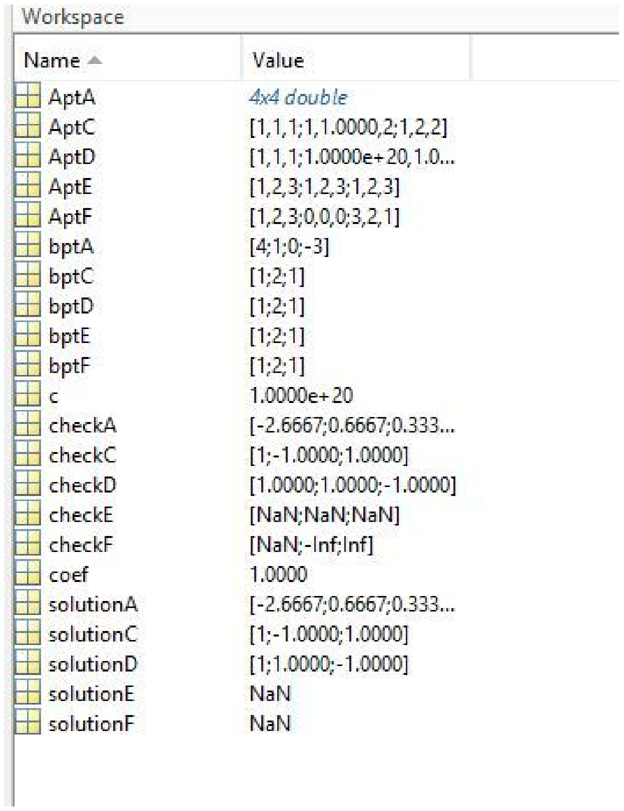
MATH 458 Homework 5

Problem 1)

a) A forward or backward solve costs n^2 flops.
b) Decomposing a matrix using LU costs (⅔)n^3 flops.
c) Not necessarily. The 2x2 matrix A with row vectors (0,1) and (1,0) is nonsingular, but a(1,1) = 0.
d) Computing the inverse of a matrix to solve a linear system can be wasteful in storage, is more computationally expensive, and will give rise to more pronounced roundoff errors.
e) Expressions for error terms aren't possible for us to compute, because we must know the approximate solution as well as the exact solution. If we knew the exact solution- we know that the error would be 0. Residuals act as upper bounds for errors and they are made from numbers we can directly calculate.
f) No. For example, let matrix A have the row vectors (1.2969, .8648) and (.2161, .1441) and let **b** be the column vector (.8642,.1440). Using some algorithm, we get an approximate solution **x** which is the column vector (.9911,-.4870). The residual has an infinity norm of 10^-8. But, since the exact solution is the column vector (2,-2), the infinity norm of the error is 1.513. Approximately 10^8 times bigger than the residual.
g) Finding the exact condition number is costly, because we must calculate the inverse of a matrix. We just want to know the ballpark of the condition number, so we can calculate a cheap estimate of it.

Problem 2 & 3)

```
%Problem 2
%The answer to problem 2 is embedded in ainvb and plu
%Problem 3

%Part A

AptA = [1 1 0 1; 2 1 -1 1; 4 -1 -2 2; 3 -1 -1 1];
bptA = [4;1;0;-3];
solutionA = ainvb(AptA, bptA);
checkA = AptA \ bptA;
%The two solutions are equal

%Part B
%AptB = [1 1 0 1; 2 1 -1 1; 4 -1 -2 2; 3 -1 -1 2];
%bptB = [4;1;0;-3];
%solutionB = ainvb(AptB, bptB);
%checkB = AptB \ bptB;

%System crashes. The matrix is singular.

%Part C

AptC = [1 1 1; 1 (1 + 10^-15) 2; 1 2 2];
bptC = [1;2;1];
solutionC = ainvb(AptC, bptC);
checkC = AptC \ bptC;
%The two solutions are equal

%Part D
c = 10^20;
coef = 1 + (10^-15);
AptD = [1 1 1; c (c*coef) (2*c); 1 2 2];
bptD = [1;2;1];
solutionD = ainvb(AptD, bptD);
checkD = AptD \ bptD;
%The two solutions are equal

%Part E
AptE = [1 2 3; 1 2 3; 1 2 3;];
bptE = [1;2;1];
solutionE = ainvb(AptE, bptE);
checkE = AptE \ bptE;
%Matrix is singular- both ainvb and matlab discovered this
%The two solutions are equal

%Part F
AptF = [1 2 3; 0 0 0; 3 2 1];
bptF = [1;2;1];
solutionF = ainvb(AptF, bptF);
checkF = AptF \ bptF;
%The matrix is singular
%The two solutions are equal

function x = ainvb(A,b)
%
% function x = ainvb(A,b)
%
% solve Ax = b
if (det(A) == 0)
    msg = "The Matrix A Is Singular. x is now NaN.";
    x = nan;
    display(msg)
else
    [p,LU] = plu (A);
    y = forsub (LU,b,p);
    x = backsub (LU,y);
end
end

function [p,A] = plu (A)
%
% function [p,A] = plu (A)
%
% Perform LU decomposition with partial pivoting.
% Upon return the coefficients of L and U replace those
% of the input n-by-n nonsingular matrix A. The row interchanges
% performed are recorded in the 1D array p.

n = size(A,1);

% initialize permutation vector p
p = 1:n;

% LU decomposition with partial pivoting
for k = 1:n-1

  % find row index of relative maximum in column k
  [val,q] = max(abs(A(k:n,k)));
  q = q + k-1;

  % interchange rows k and q and record this in p
  A([k,q],:)=A([q,k],:);
  p([k,q])=p([q,k]);

  % compute the corresponding column of L
  J=k+1:n;
  if (A(k,k) >= (2.22*10^-16))
    A(J,k) = A(J,k) / A(k,k);
    % update submatrix by outer product
    A(J,J) =  A(J,J) - A(J,k) * A(k,J);
  else
      A = 0;
      return;
  end



end
end

function x = backsub (A,b)...

function y = forsub (A,b,p)...
```



Workspace

Name	Value
AptA	4x4 double
AptC	[1,1,1;1,1.0000,2;1,2,2]
AptD	[1,1,1;1.0000e+20,1.0...
AptE	[1,2,3;1,2,3;1,2,3]
AptF	[1,2,3;0,0,0;3,2,1]
bptA	[4;1;0;-3]
bptC	[1;2;1]
bptD	[1;2;1]
bptE	[1;2;1]
bptF	[1;2;1]
c	1.0000e+20
checkA	[-2.6667;0.6667;0.333...
checkC	[1;-1.0000;1.0000]
checkD	[1.0000;1.0000;-1.0000]
checkE	[NaN;NaN;NaN]
checkF	[NaN;-Inf;Inf]
coef	1.0000
solutionA	[-2.6667;0.6667;0.333...
solutionC	[1;-1.0000;1.0000]
solutionD	[1;1.0000;-1.0000]
solutionE	NaN
solutionF	NaN

4.

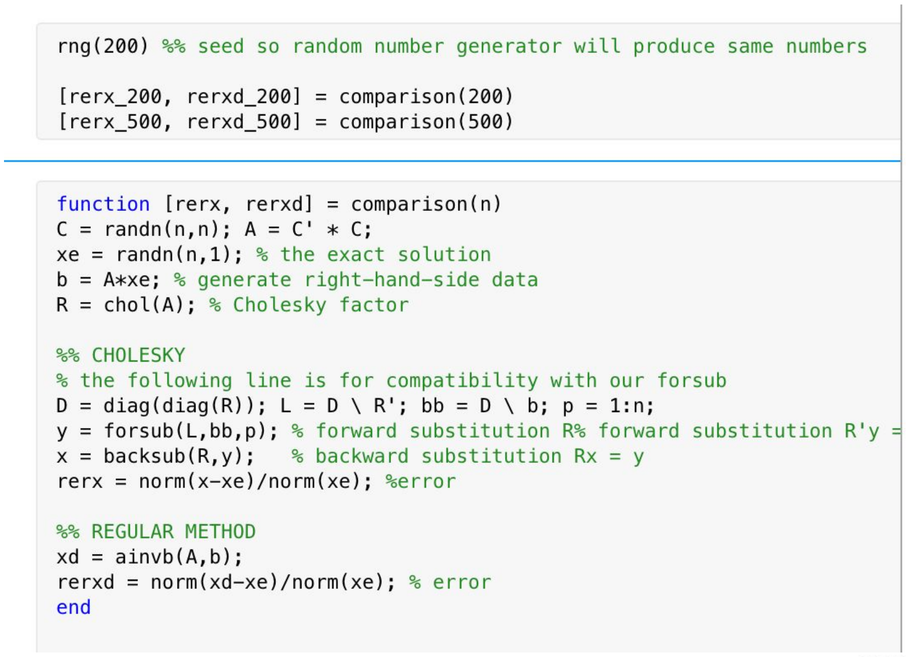
```
function A=Cholesky(A)

[n n_unused]=size(A); % n_unused is just a placeholder so n doesn't become a vector
for k = 1:n-1
    A(k, k) = sqrt(A(k, k));
    for i = k+1:n
        A(i,k) = A(i,k)/A(k,k);
    end
    for j = k+1:n
        for i = j:n
            A(i,j) = A(i,j)-A(j,k) * A(i,k);

        end
    end
end
A(n,n) = sqrt(A(n,n));
end
```

%PART B

```
function A=Cholesky_vectorized(A)

[n n_unused]=size(A); % n_unused is just a placeholder so n doesn't become a vector
for k = 1:n-1
    A(k, k) = sqrt(A(k, k));
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    for j = k+1:n
        A(j:n,j) = A(j:n,j)-A(j,k) * A(j:n,k);
    end
end
A(n,n) = sqrt(A(n,n));
end
```

```
function A = Cholesky_oneforloop(A)
    n = length(A);
    for i = 1:n
        A(i:n, i) = A(i:n, i) - A(i:n, 1:i-1) * A(i, 1:i-1)';
        A(i, i) = sqrt(A(i, i));
        A(i+1:n, i) = A(i+1:n, i) / A(i, i);
    end
end
```

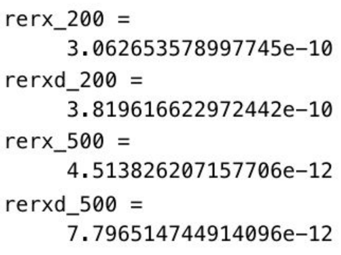
5.



```
rng(200) %% seed so random number generator will produce same numbers

[rerx_200, rerxd_200] = comparison(200)
[rerx_500, rerxd_500] = comparison(500)
```

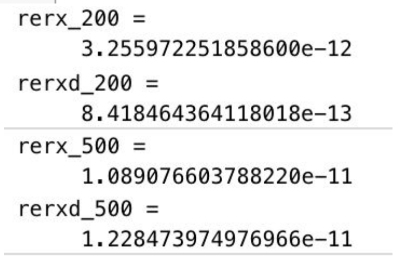
```
function [rerx, rerxd] = comparison(n)
C = randn(n,n); A = C' * C;
xe = randn(n,1); % the exact solution
b = A*xe; % generate right-hand-side data
R = chol(A); % Cholesky factor

%% CHOLESKY
% the following line is for compatibility with our forsub
D = diag(diag(R)); L = D \ R'; bb = D \ b; p = 1:n;
y = forsub(L,bb,p); % forward substitution R% forward substitution R'y =
x = backsub(R,y);   % backward substitution Rx = y
rerx = norm(x-xe)/norm(xe); %error

%% REGULAR METHOD
xd = ainvb(A,b);
rerxd = norm(xd-xe)/norm(xe); % error
end
```

Seed of 200



```
rerx_200 =
     3.062653578997745e-10
rerxd_200 =
     3.819616622972442e-10
rerx_500 =
     4.513826207157706e-12
rerxd_500 =
     7.796514744914096e-12
```

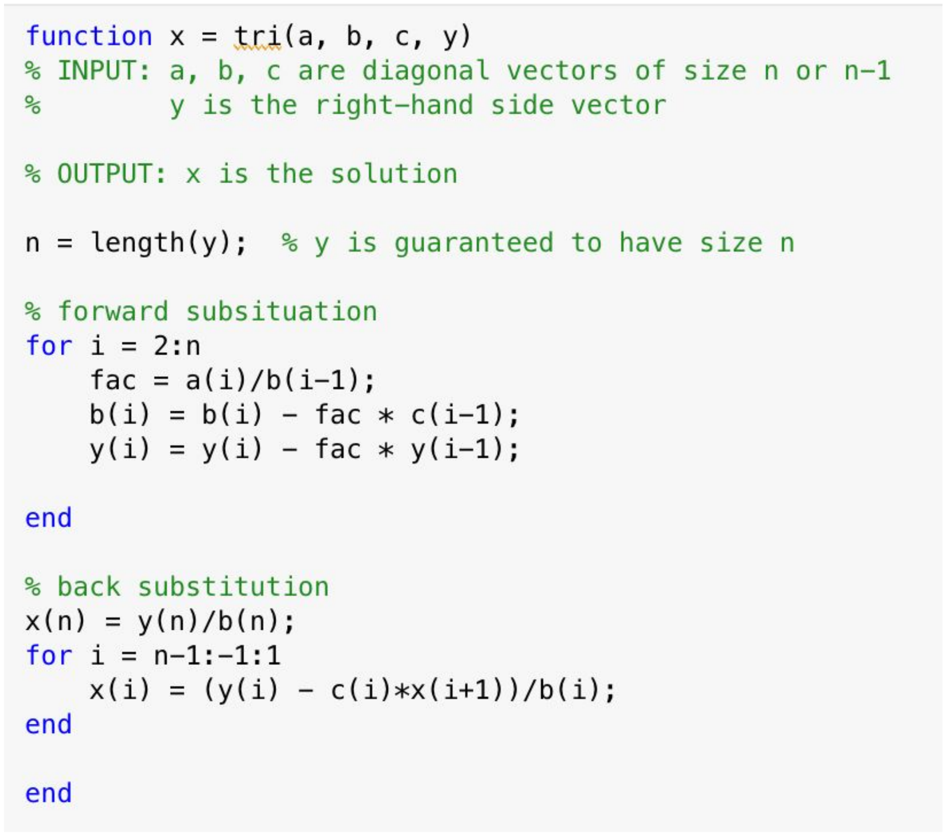
Seed of 0



```
rerx_200 =
     3.255972251858600e-12
rerxd_200 =
     8.418464364118018e-13
rerx_500 =
     1.089076603788220e-11
rerxd_500 =
     1.228473974976966e-11
```

The relative error using the Cholesky decomposition algorithm is consistently smaller than the regular method involving partial pivoting. This makes sense it turns out ainvb uses unnecessary row permutations. Also, there is no partial pivoting used in the Cholesky decomposision. Since the seed guarantees that our random numbers are consistent in multiple runs, these errors are reliable indicators of how each process performs.

6.



```
function x = tri(a, b, c, y)
% INPUT: a, b, c are diagonal vectors of size n or n-1
%        y is the right-hand side vector

% OUTPUT: x is the solution

n = length(y);  % y is guaranteed to have size n

% forward subsituation
for i = 2:n
    fac = a(i)/b(i-1);
    b(i) = b(i) - fac * c(i-1);
    y(i) = y(i) - fac * y(i-1);

end

% back substitution
x(n) = y(n)/b(n);
for i = n-1:-1:1
    x(i) = (y(i) - c(i)*x(i+1))/b(i);
end

end
```

7.

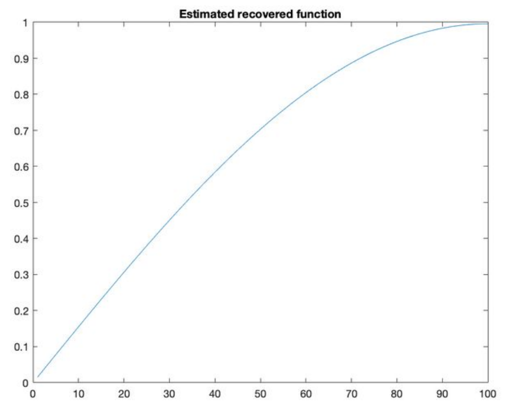
```
% QUESTION 7: Applying the tri function
N = 100;
h = 1/N;

t = linspace(0, 1, N); %% creating the 100 evenly spaced points from 0 to 1

% subdiagonal consisting of -1/(h^2). Its nth entry is -2/(h^2)
% for both subdiagonal and superdiagonal, size is N but a(1) and c(N) are not used
a = (-1/(h*h)) * ones(N, 1);
a(N) = a(N)*2;

% main diagonal consisting of 2/(h^2)
b = (2/(h*h)) .* ones(N, 1);

% superdiagonal consisting of -1/(h^2)
c = (-1/(h*h)) .* ones(N, 1);

% calculate g given the 100 points
g = fun(t);

% estimate the recovered function's values using the tridiagonal algorithm
x = tri(a, b, c, g);
plot(x)
title('Estimated recovered function')

% actual function's values
i = linspace(1, 100, 100);
u = real(i, h);
plot(u)
title('Actual function')

% compare the estimated and actual values, calculating the infinity norm
abs_err = abs(x-u)
inf_norm = norm(abs_err, inf)
```

Supplementary functions

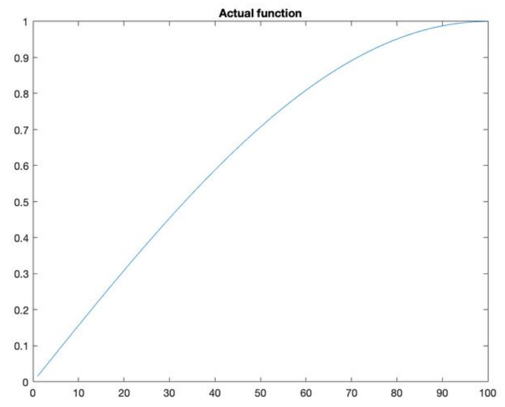
```
function u = real(t, h)
%% This is the actual function that corresponds to integrating g twice.
u = sin((pi/2)*t*h);
end
```

```
function g = fun(t)
%% Our given g function
g = (pi/2)^2 * sin(pi/2 * t);
end
```

Results:

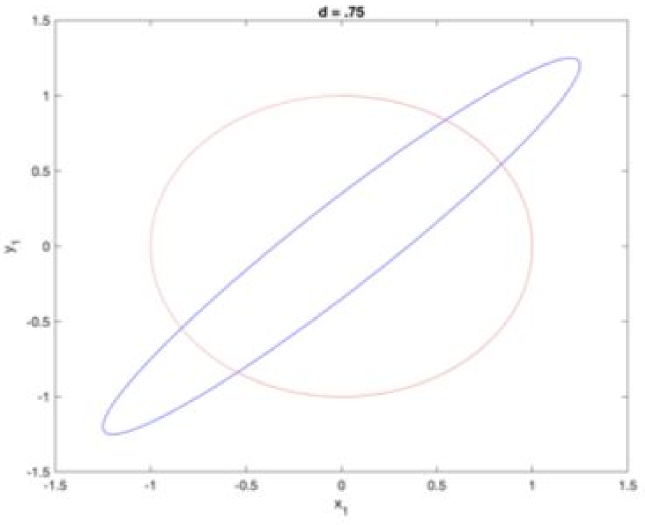


```
abs_err = 1×100
     0.000156759922414   0.000309644299314 ...

inf_norm =
   0.004328913170439
```

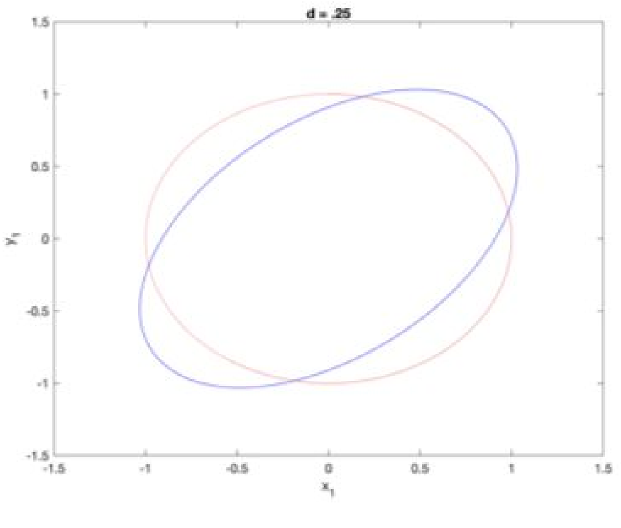


Problem 9)

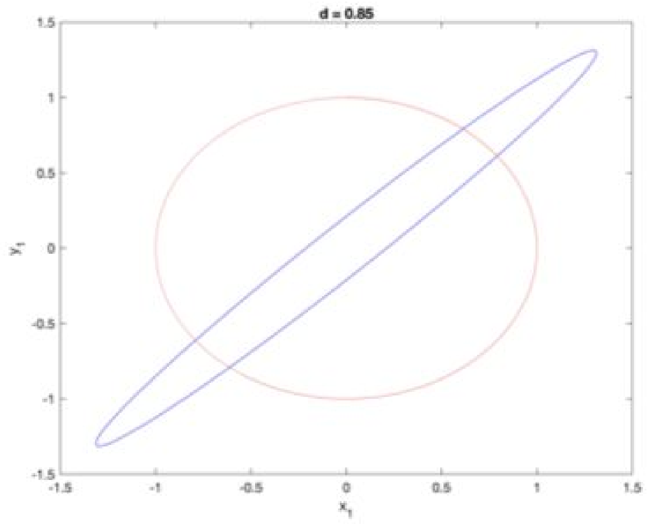
```
%Problem 9
%This problem is having us stretch a unit circle by a symmetric positive
%definite transformation
%As d gets approaches -1, the circle gets more an more stretched,
%approaching the form of y = -x
d = .75;
A = [1,d;d,1];
t = 0:.01:10; m = length(t);
x(1,1:m) = sin(t);
x(2,1:m) = cos(t);
y = A*x;
plot(y(1,:),y(2,:), 'b', x(1,:), x(2,:), 'r:')
xlabel('x_1')
ylabel('y_1')
title("d = .75")
```



```
d = .25;
A = [1,d;d,1];
t = 0:.01:10; m = length(t);
x(1,1:m) = sin(t);
x(2,1:m) = cos(t);
y = A*x;
plot(y(1,:),y(2,:), 'b', x(1,:), x(2,:), 'r:')
xlabel('x_1')
ylabel('y_1')
title("d = .25")
```



```
d = .85;
A = [1,d;d,1];
t = 0:.01:10; m = length(t);
x(1,1:m) = sin(t);
x(2,1:m) = cos(t);
y = A*x;
plot(y(1,:),y(2,:), 'b', x(1,:), x(2,:), 'r:')
xlabel('x_1')
ylabel('y_1')
title("d = 0.85")
```



```
d = -0.75;
A = [1,d;d,1];
t = 0:.01:10; m = length(t);
x(1,1:m) = sin(t);
x(2,1:m) = cos(t);
y = A*x;
plot(y(1,:),y(2,:), 'b', x(1,:), x(2,:), 'r:')
xlabel('x_1')
ylabel('y_1')
title("d = -0.75")
```