Kyle Salitrik
E SC 407H
10/17/13
HW #3

OUTPUT

Problem 1:

A =

1	1	1	1	1	1
1	2	4	8	16	32
1	3	9	27	81	243
1	4	16	64	256	1024
1	5	25	125	625	3125
1	6	36	216	1296	7776

L =

1	0	0	0	0	0
1	1	0	0	0	0
1	2	1	0	0	0
1	3	3	1	0	0
1	4	6	4	1	0
1	5	10	10	5	1

U =

1	1	1	1	1	1
0	1	3	7	15	31
0	0	2	12	50	180
0	0	0	6	60	390
0	0	0	0	24	360
0	0	0	0	0	120

X =

201.2600
-128.8210
40.6742
-7.4229
0.7408
-0.0311

Y =

106.4000
-48.6100
23.7200
-12.2100
6.5900
-3.7300

LSolved =

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

USolved =

1	1	1	1	1	1
0	1	3	7	15	31
0	0	2	12	50	180
0	0	0	6	60	390
0	0	0	0	24	360
0	0	0	0	0	120

Problem 2:

NRsol =

1/3
2/3
2/3

NRiterations =

1

JCsol =

0.3333
0.6667
0.6667

JCiterations =

28

SORsol =

0.3333
0.6667
0.6667

SORiterations =

39

DRIVER.m

```
fprintf('Problem 1:\n\n\n')
A = MatrixGenerator(6);
B = [106.4; 57.79; 32.9; 19.52; 12.03; 7.67];
[A, L, U] = LUD(A)
[X, Y, LSolved, USolved] = LU_Solver(L, U, B)


fprintf('\n\n\nProblem 2:\n\n\n')
syms x y z;
NR = [ 3*x + 4*y - z - 3; x - 4*y + 2*z + 1; -2*x - y + 5*z - 2];
JC = [ 3*x, 4*y, -z; x, -4*y, 2*z; -2*x, -y, 5*z];
b = [ 3; -1; 2 ];
x0 = [0;0;0];
relaxation = .5;

[NRsol, NRiterations] = Newton_Raphson_SOE(NR, x0, 20, 10e-8)
[JCsol, JCiterations] = Jacobi_Method(JC, b, x0, 100, 10e-8)
[SORsol, SORiterations] = SOR(JC, b, x0, relaxation, 100, 10e-8)
```

Matrix_Generator.m

```
function [ A ] = MatrixGenerator( n )
%MATRIXGENERATOR Summary of this function goes here
%   Detailed explanation goes here

A = zeros(n:n);

for i = 1:n
    for j = 1:n
        A(i,j) = i^(j-1);
    end
end

end
```

GAUSSIAN ELIMINATION CODE

Gaussian Elimination.m

```matlab
function [ B, x ] = Gaussian_Elimination( A, b )
%GAUSSIAN_ELIMINATION Summary of this function goes here
%   Detailed explanation goes here

%Forward Elimination
[B, c] = FWDElim(A,b);

%Backward Substitution
x = BWDSub(B, c);
end
```

FWDElim.m

```matlab
function [ B, c ] = FWDElim( A, b )
%FWDELIM Performs naive Gaussian forward elimination given a matrix and
%its solution vector.

[ m, n ] = size(A);

B = A; %preserve original matrix and return augmented matrix;
c = b; %preserve original solution vector and return augmented solutions;

for i = 1:m
    for j = i+1:n
        divi = B(j,i)/B(i,i);
        B(j,i) = B(j,i) - B(i,i)*divi;
        for(k=i+1:n)
            B(j,k) = B(j,k) - divi*B(i,k);
        end
        c(j) = c(j) - divi*c(i);
    end
end
```

BWDSub.m

```matlab
function [ x ] = BWDSub( B, b)
%BWDSUB performs backward substitution on a matrix processed via forward
%Gaussian elimination
[m, n] = size(B);

C = B;
x = zeros(n,1);

x(n)=b(n)/B(n,n);
for i=n-1:-1:1
    sum = 0;
    for j=i+1:n
        sum = sum + B(i,j)*x(j);
    end
    x(i)=(b(i)-sum)/B(i,i);
end
end
```

LU DECOMPOSITION CODE

LUD.m

```
function [ A, L, U ] = LUD( A )
%LUD Summary of this function goes here
%   Detailed explanation goes here

% Check to see if matrix is square
[ i, j ] = size(A);
if (i ~= j)
    fprintf('The matrix is not square!\n');
    error('non-square matrix')
end

% Check to see if pivoting is necessary
for(x = 1:i)
    if A(x,x) == 0;
        error('pivoting neecesary');
    end
end

% Initialize L & U
L = eye(i);
U = A;   % Setting U = A allows us to perform the decomposition and keep
         % the original matrix untouched.

for(x = 1:i)
    for(y = (x+1):i)
        L(y,x) = U(y,x)/U(x,x);
        for(z = 1:i)
            U(y,z) = U(y,z) - L(y,x)*U(x,z);
        end
    end
end

%Check to see if operation succeeded.
if (L*U ~= A)
    fprintf('LU Decomposition failed')
    error('LU Decomp. failure')
end

end
```

LU_Solver.m

```
function [ X, Y, LSolved, USolved ] = LU_Solver( L, U, B )
%LU_SOLVER Summary of this function goes here
%   Detailed explanation goes here
[ m, n ] = size(L);

X = zeros(m,1);
Y = zeros(m,1);

[LSolved, Y] = Gaussian_Elimination(L,B);
[USolved, X] = Gaussian_Elimination(U,Y);
```

Iterative Solution Methods

Newton_Raphson_SOE.m

```
function [ x, iterations ] = Newton_Raphson_SOE( A, x0, max_iter,tol, vars)
%NEWTON_RAPHSON_SOE  - Uses the Newton-Raphson method to determine
% solution to a system of equations.
%   A - input matrix
%   b - right side vector
%   x0 - initial guess
%   max_iter - maximum number of iterations before quitting
%   tolerance - error tolerance
%   vars - variables appearing in system of equations

if(nargin < 5)
    syms x y z
    vars = [ x; y; z ];

x_curr = x0;
iterations = 0;
Jac = jacobian(A);
Jac_inverse = inv(Jac);
err = 1;

while(iterations < max_iter && err > tol)
    A_eval = double(subs(A, vars, x_curr));
    J_eval = double(subs(Jac_inverse, vars, x_curr));
    Subtract = J_eval*A_eval;
    x_curr = x_curr - Subtract;
    A_eval = double(subs(A, vars, x_curr));
    err = max(abs(A_eval));
    iterations = iterations+1;
end

x = sym(x_curr);
end
```

Jacobi Method.m

```
function [ X, iterations ] = SOR( A, b, x0, relax, max_iter, tol, vars )
%SOR - Uses successive overrelaxation method to determine solution to
% a system of equations.
%   A - input matrix
%   b - right side vector
%   x0 - initial guess
%   relax - relaxation coefficient
%   max_iter - maximum number of iterations before quitting
%   tolerance - error tolerance
%   vars - variables appearing in system of equations

%Check if symbolic vector needs created
if(nargin < 7)
    syms x y z
    vars = [x;y;z];
end

%Create vector of ones to obtain coefficient matrix from variable
% functions.
number_of_ones = size(vars);
one = ones(number_of_ones(1), 1);

%Determine number of iterations
[ m, n ] = size(A);

%%Begin SOR Method
A_plug = subs(A, vars, one);
iterations = 1;
x_curr = x0;
b;
err = 100;
while(iterations < max_iter &&  err > tol)
    x_old = x_curr;
    for i = 1:n
        x_sum = 0;
        for j = 1:n
            if j ~=i
                x_sum = x_sum + A_plug(i,j)*x_curr(j);
            end
        end

        x_curr(i) = (1-relax)*x_curr(i) + relax*(b(i) - x_sum)/A_plug(i,i);
    end
    err = norm(max(abs((x_curr - x_old))));
    iterations = iterations + 1;
end

X = x_curr;
```

SOR.m

```
function [ X, iterations ] = Jacobi_Method( A, b, x0, max_iter, tol, vars )
%JACOBI_METHOD  - Uses the Jacobi method to determine solution to
% a system of equations.
%   A - input matrix
%   b - right side vector
%   x0 - initial guess
%   max_iter - maximum number of iterations before quitting
%   tolerance - error tolerance
%   vars - variables appearing in system of equations


%Check if symbolic vector needs created
if(nargin < 6)
    syms x y z
    vars = [x;y;z];
end

%Create vector of ones to obtain coefficient matrix from variable
% functions.
number_of_ones = size(vars);
one = ones(number_of_ones(1), 1);

%Determine number of iterations
[ m, n ] = size(A);

%%Begin Jacobi Method
A_plug = subs(A, vars, one);
iterations = 1;
x_curr = x0;
err = 100;
while(iterations < max_iter &&  err > tol)
    x_old = x_curr;
    for i = 1:n
        x_sum = 0;
        for j = 1:n
            if j ~=i
                x_sum = x_sum + A_plug(i,j)*x_curr(j);
            end
        end

        x_curr(i) = (b(i) - x_sum)/A_plug(i,i);
    end
    err = norm(max(abs((x_curr - x_old))));
    iterations = iterations + 1;
end

X = x_curr;
```