APPM3021 Numerical Methods • Lab 1
Tyson Cross 1239448
--

Exercises

exercise1.m :

```
% APPM3021 Lab 1, Exercise 1

clc
clear all

A = [ 1 1 1 2;...
      1 2 4 1;...
     -1 0 3 1;...
      2 0 2 4]

rows = length(A);
b = randi(10,rows,1)

solution = gaussElimination(A,b)                    % Here is the function

% Output and check
check = A\b;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

exercise2.m :

```
% APPM3021 Lab 1, Exercise 2

clc
clear all

rows = randi(8)+1;
A = magic(rows)
rows = length(A);
B = randi(10,rows,rows)

solution = gaussMultipleSystems(A,B)                % Here is the function

% Output and check
check = A\B;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

exercise3.m :

```
% APPM3021 Lab 1, Exercise 3

clc
clear all

rows = randi(8)+1;
A = magic(rows)
rows = length(A);
b = randi(10,rows,1)

[L, U] = LUFactorization(A)                         % Here is the function

% Check the function works
lu_check = L*U;
if ~isequal(A,lu_check)
    warning(['Function is inaccurate, by a max difference of ',...
        num2str(max(max(abs(A - lu_check))))])
    disp(' ')
```

```
end

% Solve the matrix using LU decomposition
% Ax=b , A=LU, so Ax=LUx=b
% Ux=y <--- Ly=b

Y = gaussElimination(L,b);
solution = gaussElimination(U,Y)

% Output and check
check = A\b;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

Questions

Question 1a)

```
% APPM3021 Lab 1, Question 1a

clc
clear all

A = [ 2,  1, -1,  2;...
      4,  5, -3,  6;...
     -2,  5, -2,  6;...
      4, 11, -4,  8]

b = [5; 9; 4; 2]

% Gauss Elimination w/o partial pivoting
% Forward elimination and back substitution
solution = gaussElimination(A,b)

% Output and check
check = A\b;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

Question 1b)

```
% APPM3021 Lab 1, Question 1b

clc
clear all

A = [ 3,  1, -1;...
      1, -4,  2;...
     -2, -1,  5]

b = [3; -1; 2]

% Gauss Elimination w/o partial pivoting
% Forward elimination and back substitution
solution = gaussEliminationAltered(A,b)

% Output and check
check = A\b;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

Question 1c)

```
% APPM3021 Lab 1, Question 1c
```

```
clc
clear all

A = [ 1, -1,  2, -1;...
      2, -2,  3, -3;...
      1,  1,  1,  0;...
      1, -1,  4,  3]

B = [ -8, -10, -100;...
     -20, -20, -250;...
      -2,  -2,  -25;...
       4,   8,   80]

solution = gaussMultipleSystems(A,B)                % Here is the function

% Output and check
check = A\B
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

Question 1d)

```
% APPM3021 Lab 1, Question 1d

clc
clear all

A = [ 1, -1,  2, -1;...
      2, -2,  3, -3;...
      1,  1,  1,  0;...
      1, -1,  4,  3]

B = [ -8, -10, -100;...
     -20, -20, -250;...
      -2,  -2,  -25;...
       4,   8,   80]

[L, U] = LUFactorization(A)                         % Here is the function

% Check the function works
lu_check = L*U;
if A ~= lu_check
    warning(['Function is inaccurate, by a max difference of ',...
        num2str(max(max(abs(A - lu_check))))])
    disp(' ')
end

% Solve the matrix using LU decomposition
% Ax=b , A=LU, so AX=LUX=B
% UX=Y <--- LY=b

Y = gaussMultipleSystems(L,B);
solution = gaussMultipleSystems(U,Y);

% Output and check
check = A\B;
if ~isequal(solution,check)
    warning(['Solution is inaccurate, by a max difference of ',...
        num2str(max(max(abs(solution-check))))])
end
```

Functions and Code

isSolvable.m :

```
function x = isSolvable( A )
% Checks if input matrix is square and non-singular

x = true;
n = size(A);
if n(1) ~= n(2)
    disp('Matrix is not square')
    x = false;
    return
end

if det(A)==0
    disp('Matrix is singular')
    x = false;
    return
end

end
```

swapRow.m :

```
function X = swapRow(A,row_1,row_2)
%   Swaps row_1 with row_2 in matrix A

temp_row = A(row_1,:);                                  % store temp_row
A(row_1,:) = A(row_2,:);                                % assign new row_1
A(row_2,:)= temp_row;                                   % assign new row_2
X = A;                                                  % return matrix

return
```

backSubstitution.m :

```
function x = backSubstitution(A,b)
%   Solves for variables and substitutes them (upwards from the bottom)
%   in a upper triangular matrix (forward eliminated system of equations)


if ~isSolvable(A)                                       % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

n = length(b);
x = zeros(n,1);                                         % initialise solution vector
if A(n,n) ~= 0
    x(n) = b(n) / A(n,n);                               % solution to variable in bottom row
end

for i = (n-1):-1:1                                      % work ascending from the last row up
    value = b(i);                                       % find the factor
    for j = (i+1):n                                     % finish rest of entries in row
        value = value -(A(i,j) .* x(j));
    end
    if A(i,i) ~= 0
        x(i) = value / A(i,i);                          % solve for variable
    end
end
```

forwardSubstitution.m :

```
function x = forwardSubstitution(A,b)
%   Solves for variables and substitutes them (downwards from the top)
%   in a upper triangular matrix (forward eliminated system of equations)
```

```
if ~isSolvable(A)                                    % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

n = length(b);
x = zeros(n,1);                                      % initialise solution vector
x(1) = b(1) / A(1,1);                                % solution to variable in top row
for i = 2:n                                          % work ascending from the second row down
    value = b(i);                                    % find the factor
    for j = (i+1):n                                  % finish rest of entries in row
        value = value - (A(i,j) .* x(j));
    end
    x(i) = value / A(i,i);                           % solve for variable
end
```

forwardElimination.m :

```
function [X,y] = forwardElimination(A,b)
%   Forward elimination method, takes a Matrix and vector
%   Puts Matrix A in upper triangular form

if ~isSolvable(A)                                    % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

n = length(b);
for row = 1:(n-1)                                    % for each row
    for i = (row+1):n                                % for each pivot along the main diagonal
        if A(row,row) ~= 0
            m = A(i,row) / A(row/row);               % find the factor
            else m = 1;
        end
        for j = row:n                                % finish rest of entries in row
            A(i,j)=A(i,j)-(m*A(row,j));              % set entry in A
        end
        b(i) = b(i) - ( m .* b(row));                % set entry in b

    end
X = A;                                               % Output assignments
y = b;
end
```

backElimination.m :

```
function [X,y] = backElimination(A,b)
%   "Back" elimination method, takes a Matrix and vector
%   Puts Matrix A in upper, reverse triangular form

if ~isSolvable(A)                                    % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

n = length(b);
for row = n:-1:2                                    % for each row
    for i = n:-1:(row+1)                             % for each pivot along the main diagonal
        if A(row,row) ~= 0
            m = A(i,row) / A(row/row);               % find the factor
        end
        for j = n:-1:row                             % finish rest of entries in row
            A(i,j)=A(i,j)-(m*A(row,j));              % set entry in A
        end
        b(i) = b(i) - ( m .* b(row));                % set entry in b

    end
X = A;                                               % Output assignments
y = b;
end
```

forwardEliminationWithPivoting.m :

```
function [X,y] = forwardEliminationWithPivoting(A,b)
%   Forward elimination method, takes a Matrix and vector
%   Uses partial pivoting
```

```
%   Puts Matrix A in upper triangular form

if ~isSolvable(A)                                   % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

n = length(b);
for row = 1:(n-1)                                   % for each row
    for pivot = (row+1):n                           % check for need to do partial-pivoting
        if (A(row,row)<A(pivot,row))                % pivot is smaller than current entry
            A = swapRow(A,row,pivot);               % swap rows in A
            b = swapRow(b,row,pivot);               % swap row in b
        end
    end
    for i = (row+1):n                               % for each pivot along the main diagonal
        if A(row,row) ~= 0
            m = A(i,row) / A(row/row);              % find the factor
        else m = 1;
        end
        for j = row:n                               % finish rest of entries in row
            A(i,j)=A(i,j)-(m*A(row,j));             % set entry in A
        end
        b(i) = b(i) - ( m .* b(row));               % set entry in b
    end
end
X = A;                                              % Output assignments
y = b;
end
```

gaussElimination.m :

```
function x = gaussElimination(A,b)
%   Gaussian elimination method for solving a single system of equations i.e. Ax = b
%   Using forward elimination and back substitution, without partial pivoting

if ~isSolvable(A)                                   % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

[M,y] = forwardElimination(A,b);
x = backSubstitution(M, y);

end
```

gaussEliminationAltered.m

```
function x = gaussElimination(A,b)
%   Gaussian elimination method for solving a single system of equations i.e. Ax = b
%   Using back elimination and forward substitution, without partial pivoting
%   This function returns the front, top triangular values as 0

if ~isSolvable(A)                                   % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

[M,y] = backElimination(A,b)
x = forwardSubstitution(M, y)

end
```

gaussMultipleSystems.m :

```
function X = gaussMultipleSystems( A,B )
%   Gaussian elimination method for solving multiple system of equations i.e. AX = B
%   Using forward elimination and back substitution, with partial pivoting

if ~isSolvable(A)                                   % check is matrix is square and non-singular
    error(strcat('Matrix is not solvable'))
end

[n,m] = size(B);
Y = zeros(n,m);
M = zeros(size(A));
```

```
for i = 1:m
    [M,Y(:,i)] = forwardEliminationWithPivoting(A,B(:,i));
end

X = zeros(n,m);
for i = 1:m
    X(:,i) = backSubstitution(M, Y(:,i));
end


end
```

gaussEliminationLUFactorization

```
function [L,U] = LUFactorization(A)
%   Permuted LU-factorization splits a matrix into Upper and Lower matrices
%   using Doolitle Decomposition

if ~isSolvable(A)                                       % check is matrix is square and non-
singular
    error(strcat('Matrix is not solvable'))
end

n = length(A);

L = eye(n);                                             % L diagonal entries will be 1
U = zeros(n);                                           % pre-allocate upper matrix U

for i = 1:n                                             % Loop through from second row
        % Lower matrix calculation
        for j = 1:(i-1)                                 % Loop through columns
            L(i,j) = A(i,j);                            % Use current A(i,j) value
            for k = 1:(j-1)                             % for each entry (lower ?)
                L(i,j)=L(i,j)-double(L(i,k)*U(k,j));    % double maintains precision of inner
 product
            end
            if U(j,j) ~= 0
                L(i,j)= L(i,j)/U(j,j);                  % completes the entry's calculation
 equation
            end
        end

        % Upper matrix calculation
        for j = i:n                                     % Loop through columns
            U(i,j) = A(i,j);                            % Use current A(i,j) value
            for k = 1:(i-1)                             % for each entry in the row & column
                U(i,j)=U(i,j)-(double(L(i,k)*U(k,j)));  % assign entry (upper ?)
            end
        end
end
end
```