Lab 1 & 2

2202545E
School of Physics & Astronomy, University of Glasgow
27^{th} of November 2017

1 Lab 1

Gaussian Elimination

Gaussian elimination reduces a square matrix to be an upper diagonal matrix consisting of only zero entries below the diagonal. This is done by iterating over the rows in the matrix eliminating initial entries on each row by subtracting (or adding) the ratio between the first entry in the row and the first row in the iteration (further description of the computational method can be found in appendix 1.) In order to show the intermediate results five lines of code were added printing a message describing the iteration number followed by the corresponding matrices. The code is tested using the four simultaneous equations found in system of equations (1) where the answer also can be seen.

$$\begin{aligned} x_1 + 5x_2 + 2x_3 + 3x_4 &= 0 \\ 2x_1 + 3x_2 + x_3 + 4x_4 &= 8 \\ 3x_1 + 5x_2 + 3x_3 + 2x_4 &= 1 \\ 4x_1 + 5x_2 + 2x_3 + 2x_4 &= 1 \end{aligned} \quad \leftrightarrow \quad \begin{aligned} x_1 &= 17/13 \\ x_2 &= -33/13 \\ x_3 &= 17/13 \\ x_4 &= 38/17 \end{aligned} \tag{1}$$

Gaussian elimination and backwards substitution was used in order to solve the system of equations that can be seen in table 1 on the next page. For a) the result is $[3, 2, 1]$ whilst b) cannot be solved using this method since we use the first entry on the first row for division and thereby divide by 0. This is a general disadvantage of the Gaussian elimination method which occurs whenever a row contains zero-entries from the beginning. Other issues with using Gaussian elimination include rounding off errors which can occur when coefficients are much smaller than 1 or when the equations are given with a specific number of decimals. In order to solve this problem scaled partial pivoting is introduced.

a)	**b)**	**c)**	**d)**
$-x_1 + 2x_2 + 5x_3 = 6$	$-x_2 + 2x_3 = 1$	$6x_1 + 2x_2 + 2x_3 = -2$	$6x_1 + 2x_2 + 2x_3 = -2$
$4x_1 - 6x_2 - x_3 = -1$	$-x_1 + 4x_2 - 6x_3 = -1$	$2x_1 + \frac{2}{3}x_2 + \frac{1}{3}x_3 = 1$	$2x_1 + 0.6667x_2 + 0.3333x_3 = 1$
$2x_1 - x_2 + 3x_3 = 7$	$3x_1 + 2x_2 - x_3 = 7$	$x_1 + 2x_2 - x_3 = 0$	$x_1 + 2x_2 - x_3 = 0$

Table 1: Table containing systems of equations.

Gaussian Elimination with Scaled Partial Pivoting

Scaled Partial Pivoting consist of exchanging the row-order of a matrix whilst doing Gaussian Elimination in order to avoid division by zero and reducing round-off error. (The code can be found in appendix 2.) Using this method on b) from exercise 2, we now get the solution $[1, 3, 2]$. The difference is that the code recognizes that the first coefficient in the first row is zero and hence swaps row 1 and 3 from where the Gaussian Elimination can be completed without further row-swapping.
The last two systems of equations found in table 1 are solved using Gaussian elimination with

scaled partial pivoting. They look the same but one is defined using fractions whilst the other uses numbers with 4 decimals. For the third system of equations c), we get the answer $[2.6, -3.8, -5.0]$ whilst using the same code on d) we get $[2.5999, -3.7999, -4.9999]$. This is due to round-off errors that are caused by the number of decimals. Looking at the intermediate steps (outputs for both tasks can be found in appendix 2.) we see that after the first elimination that the (2,2) entry is 0.0000 instead of 0. By using the GEshow, we see that the entry is 0.000033 (as seen in appendix 3) which leads to the minor error in the result.

Round-off Errors

As we saw, scaled partial pivoting can solve some issues with Guassian elimination though it cannot assure an absolutely precise answer due to round off errors. The code for Gaussian elimination with scaled partial pivoting was modified to take another input variable r_{dec} which determines the number of decimals of the result and therefore can be used to explore the number of decimals influence on the result. r_{dec} was implemented into the code when eliminating row entries during the Gaussian Elimination which can be seen in the code in appendix 4.

$$\begin{pmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.217 \\ 0.254 \end{pmatrix} \tag{2}$$

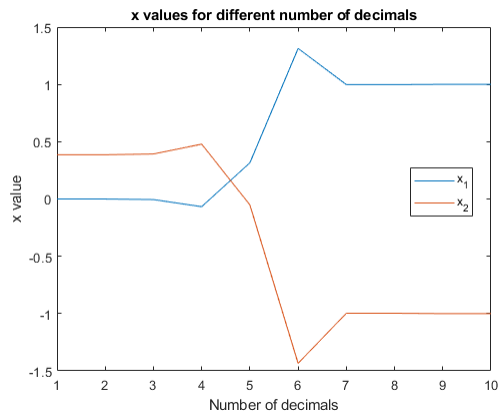


Figure 1: x values corresponding to varying number of decimals used in calculation

A separate function, which can be found in appendix 4, was created in order to run the previous function several times with an increasing number of decimals. System of equations (2) was be used and the number of decimals were in the interval [1,10]. The varying values of x_1 and x_2 as solutions for the system of equations were plotted against the number of decimals which can be seen in figure 1. We can on the graph see that the x-values vary until we consider 7 decimals. Round off errors are difficult to resolve unless you include many decimals in all intermediate steps and results.

Condition Number

One method to determine a systems sensitivity to small errors is finding the condition number for the system's corresponding matrix **A**. A simple program, that outputs the matrix that was inputted into the program together with it's condition number was written and can be found in appendix 5. As we can see in table 2, the condition number for the matrices corresponding to the system of equations used previously are fairly similar and low meaning that their corresponding solutions are stable and do not change much depending on number of decimals. But for system (2), we can see that it has a very high condition number which matches the varying x-values.

System of equation	Condition No
System in (1)	19.23042
System a) in table 1	11.76601
System b) in table 1	22.50920
System c) in table 1	31.50212
System d) in table 1	31.50152
System in (2)	2398.352

Table 2: Table of condition numbers for the matrices used in previously.

Interpolation

Interpolation is the estimation of a function based on a set of x values and corresponding y values. An approximate polynomial of order $N-1$ will be created for a set of N coordinates using equation (3) where it's coefficients a_0 to a_{N-1} are given by equation (4). In order to explore interpolation, a program was written (found in appendix 6.) It creates a matrix containing each $(x_k)^j$ entry then using Gaussian elimination with scaled partial pivoting the specific a_j that satisfies equation (4) are found. These are then used to define the approximate polynomial for the data-set using (3).

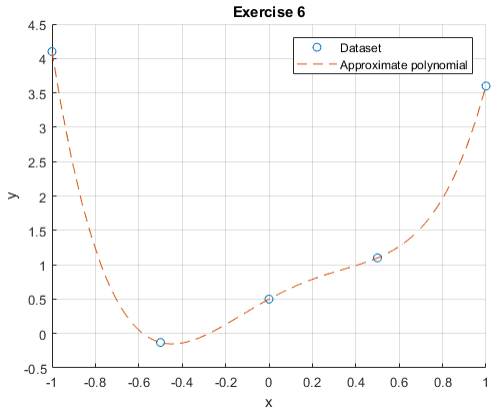


Figure 2: Interpolation function for data-set found in table 3.

$$F(x) = \sum_{j=0}^{N-1} a_j x^j \tag{3}$$

$$\sum_{j=0}^{N-1} a_j x_k^j = y_k \tag{4}$$

The estimate function for the coordinates in table 3 can be seen in 2 where it is clear that it fits quite well.

x_k	-1.0	-0.5	0.0	0.5	1.0
y_k	+4.1	-1.3	0.5	1.1	3.6

Table 3: Coordinates used for interpolation

Interpolations using functions

We can interpolate data sets in terms of specified functions as seen in equations (5) and (6).

$$F(x) = \sum_{j=0}^{N-1} a_j g_j(x^j) \tag{5}$$

$$\sum_{j=0}^{N-1} a_j g_j(x_k) = y_k \tag{6}$$

This method is mostly the same as in the previously though each matrix entry is individually defined by a specific function. An example can be seen in equation (7) which is used to interpolate the coordinates in table 4. The approximate polynomial can be seen in figure 3 and it was found using the code found in appendix 7.

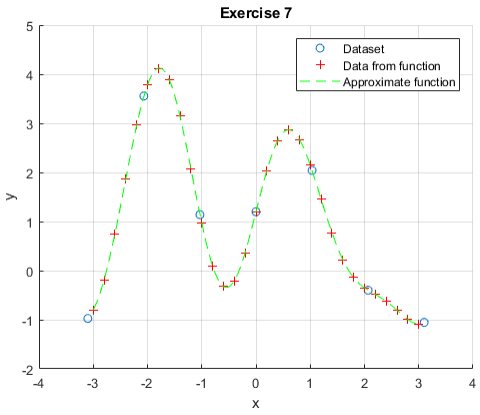


Figure 3: Interpolation function for data-set found in table 4.

$$a_0 + a_1 sin(x) + a_2 cos(x) + a_3 sin(2x) + a_4 cos(2x) + a_5 sin(3x) + a_6 cos(3x) \tag{7}$$

x_k	-3.0999	-2.0666	-1.0333	0.0000	1.0333	2.0666	3.0999
y_k	-0.97309	3.5557	1.1402	1.2000	2.0412	-0.39794	-1.0535

Table 4: Coordinates used for interpolation

2 Lab 2

Bisection Method

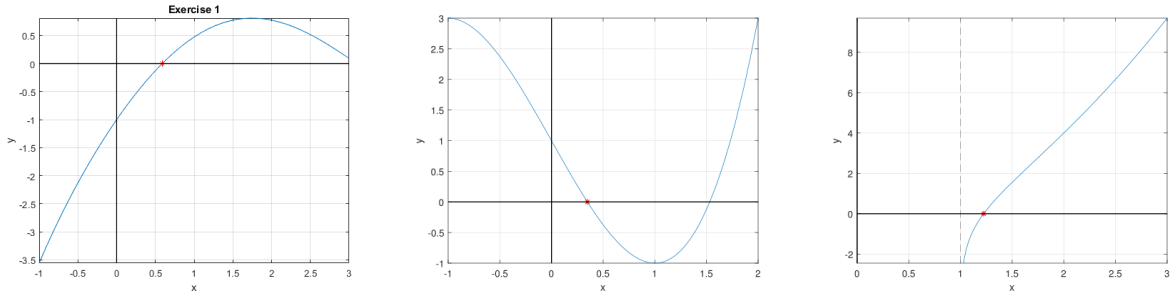
The Bisection Method finds the midpoint of an interval, checks whether it is the root and changes the interval if not. This is iterated until the root is found (within a given error-band) or until the max number of iterations is reached. The given code was modified by changing the switch cases in the function f to be the functions (8), (9) and (10) as seen underneath. For a visual representation, a plot of the wanted function together with the root found is created by the program. (Code can be found in appendix 1.)Plotting the functions we see that the first and third function have their root in the interval [0,2] though the third function will be executed on the interval [1,2] in order to avoid complex numbers arriving from ln(0). The second function will be evaluated on [0,1] to get the first and smallest positive root.

$$\mathrm{e}^{-\mathrm{x}} = \sin(\mathrm{x}) \qquad (8) \qquad x^3 - 3x + 1 = 0 \qquad (9) \qquad \ln(\mathrm{x} - 1) = -\mathrm{x}^2 \qquad (10)$$

Setting the error band to be 0.0001 and the maximum number of iterations 100 we get the results found in table 5. Figure 4 contains the visual representation of the three functions together with the found root.

	Root	**Error Bound**	**Iteration Count**
$e^{-x} = \sin(x)$	0.5886	$6.1035 * 10^{-5}$	14
$x^3 - 3x + 1 = 0$	0.3474	$6.1035 * 10^{-5}$	13
$\ln(x - 1) = -x^2$	1.2236	$6.1029 * 10^{-5}$	13

Table 5: Table of results for using the Bisection Method



(a) $e^{-x} = \sin(x)$ (b) $x^3 - 3x + 1 = 0$ (c) $\ln(x - 1) = -x^2$

Figure 4: Plots of each function with its root

Secant Method

The secant method creates a straight line between two points a and b then using the intersection of this line with the x-axis estimates a root. The intersection, p, is found using the equation (11) and if $f(p)$ is outside the given error bound, $(p, f(p))$ replaces $(a, f(a))$ which replaces $(b, f(b))$. This is iterated until the root is found (within an error margin) or the maximum number of iterations is met. (Code can be found in appendix 2.)

$$p = a - f(a)\frac{b - a}{f(b) - f(a)} \qquad (11)$$

The secant method used on the equations (8), (9) and (10) gives the results found in table 6.

	Root	Error Bound	Iteration Count
$e^{-x} = \sin(x)$	0.5885	$-9.7592 * 10^{-6}$	8
$x^3 - 3x + 1 = 0$	0.3473	$1.5707 * 10^{-6}$	6
$\ln(x-1) = -x^2$	1.2237	$1.5141 * 10^{-5}$	6

Table 6: Table of results using the Secant method

Newton-Raphson Method

The Newton-Raphson method uses, like the secant method, linear approximation to estimate roots by finding the derivative of points as seen in equation (12) and (13). Otherwise the method is exactly the same as the secant method.

$$m = \left(\frac{df}{dx}\right)_{x=a} \quad (12)$$

$$p = a - \frac{f(a)}{m} \quad (13)$$

The code seen in appendix 3 is used on functions (1), (2) and (3) in order to find their roots and the results can be found in table 7.

	Root	Error Bound	Iteration Count
$e^{-x} = \sin(x)$	0.5885	$3.3314 * 10^{-6}$	4
$x^3 - 3x + 1 = 0$	0.3473	$7.4131 * 10^{-5}$	3
$\ln(x-1) = -x^2$	1.2237	$2.9187 * 10^{-6}$	8

Table 7: Table of results using the Newton-Raphson method

Comparison of Methods

We can in table 8 see that the Newton-Raphson method converges much faster than the other two which is also clear from the convergence of the error of the three methods that can be seen on the next page ((14) for bisection method, (15) for secant and (16) for Newton-Raphson.) Both the bisection and secant method need to have two initial function values in order to work whilst the Newton-Raphson method only needs one though for this method it is required that the function is continuous and differentiable on the interval it's used on (ie. where the root lies) and as seen later in this report it can fail under certain circumstances. The bisection method never fails but will take longer in order to find the root which is more requires more computational power. For the secant method, it fails if $f(x_n) = f(x_{n-1})$ since this causes division by 0 and overall if $f(x_n) - f(x_{n-1}) \leq \epsilon$ then it can cause error when obtaining x_{n+1}. The Newton-Raphson method does not converge if x_0 is too far away from the root, $f'(x_0) = 0$ or $x_2 = x_0$ for which the iteration will be cyclic (an example of this can be seen in the next section.) Hence the faster the method converges the higher the chance is of it not converging at all and one has to be careful when choosing a method.

$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \quad (14)$$
for m = 1

$$\epsilon_{n-1} = \text{constant} \times (\epsilon_n)^m \quad (15)$$
for $m = \frac{1+\sqrt{5}}{3} \approx 1.62$

$$\epsilon_{n-1} = \text{constant} \times (\epsilon_n)^m \quad (16)$$
for m = 2

As one can see in table 8, the computer run time for the different methods vary quite a lot. The bisection method is generally slower than the other two though the Newton-Raphson method is dependent on the function and hence the run time differes drastically. All run times are under a second since the equations are fairly simple but with more complex equations one can expext the difference in run time to be much larger which makes choosing the right method more important for the effectiveness of a program.

Number of Iterations	Bisection Method	Secant Method	Newton-Raphson Method
$e^{-x} = \sin(x)$	14	8	4
$x^3 - 3x + 1 = 0$	13	6	3
$\ln(x-1) = -x^2$	14	6	8
Computer Time	**Bisection Method**	**Secant Method**	**Newton-Raphson Method**
$e^{-x} = \sin(x)$	0.264598s	0.078196s	0.067924s
$x^3 - 3x + 1 = 0$	0.338931s	0.074012s	0.117424s
$\ln(x-1) = -x^2$	0.105590s	0.067284s	0.106145s

Table 8: Table of comparison of number of iterations and computer run time for the three different methods

Newton-Raphson Method for Functions with more than one root

For functions with more than one root, the starting x value determines which root the Newton-Raphson Method will converge to. Table 9 shows the roots for three different x_0 values found for the function $x^3 + 2x^2 - x + 5$. The code for this can be found in appendix 4 where the code found in appendix 3 is used three times in a loop where each iteration uses a different x_0.

	Root	Error Bound	Iteration Count
$x_0 = 1 + i$	$0.4629 + i1.2225$	$-3.1332 * 10^{-8} + i2.1971 * 10^{-9}$	5
$x_0 = 1 - i$	$0.4629 - i1.2225$	$-3.1332 * 10^{-8} - i2.1971 * 10^{-9}$	5
$x_0 = -4$	-2.9259	$6.0097 * 10^{-7}$	5

Table 9: Table of number of iterations of the three different methods

Failures of the Newton-Raphson method

As seen on figure 5, the root for equation (17) is at $x = 2$ and that the graph is symmetric about this point. This means that the Newton-Raphson method will go back and forth between two values and continue forever until it reaches the maximum number of iteration. The root can therefore not be calculated.

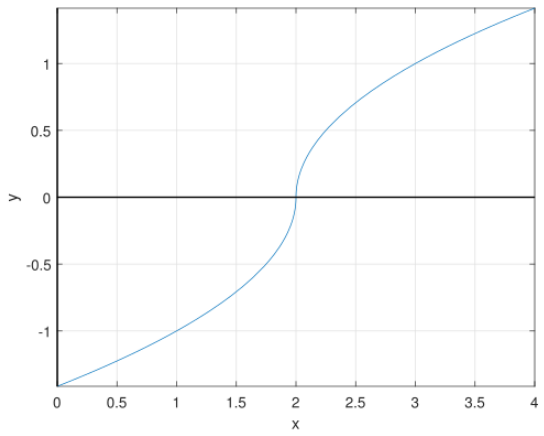


Figure 5: $f(x) = sign(x-2)\sqrt{|\ x-2\ |}$

$$f(x) = \text{sign(x} - 2)\sqrt{|\text{x} - 2|} \tag{17}$$

$$f'(x) = \frac{f(x)}{2(x-2)} \tag{18}$$

Both the secant and bisection methods finds the root of this function since they do not use the derivative. The secant method finds the root $x = 2$, and hence the error is 0, using only 2 iterations. The bisection methods finds $x = 2.0001$ as a root after 15 iterations with an error of $6.1035 * 10^{-5}$. We therefore see an example of why you should be careful when choosing which method to use. In this case, the secant method was the most precise and the most effective. The outputs using the three different methods can be found in appendix 5.

Appendices

Apendices Lab 1

Appendix 1

Gaussian Elimination

Pseudo Code

Forward elimination

real array $(a_{ij})_{N\times N}, (b_i)_N$
integer i,j,N
for j=2 **to** N **do**
    for i=j **to** N **do**
        $m = \frac{a_{(i,j-1)}}{a_{(j-1,j-1)}}$
        $a_{i,:} = a_{i,:} - m \times a_{(j-1,:)}$
        $b_i = b_i - m \times b_{(j-1)}$
    end for
end for

Back substitution

real array $(a_{ij})_{N\times N}, (x_i)_N$
integer i,j,N
real sum
$x_N \leftarrow \frac{b_N}{a_{N,N}}$
for i = N-1 **to** 1 **step** -1 **do**
    $sum \leftarrow b_i$
    for j=i+1 **to** N **do**
        $sum \leftarrow sum - a_{ij}x_{ij}$
    end for
    $x_i \leftarrow \frac{sum}{a_{ij}}$
end for

MatLab Code

```
function [x] = gaussElim2(A,b)
% File gaussElim.m
%   This pogram will perform Gaussian elmination
%   on the input matrix.
%   i.e., given A and b it can be used to find x,
%         Ax = b
%
%   To run this file you will need to specify several
%   things:
%   A - matrix for the left hand side.
%   b - vector for the right hand side
%
%   The routine will return the vector x.
%   ex: [x] = gaussElim(A,b)
```

```
%      this will perform Gaussian elminiation to find x.
tic;
%The dimension of matrix A used for itteration
 N = max(size(A));

% Perform Gaussian Elimination

%Starting
fprintf('You have entered matrices; \n');
      fprintf('A= \n');
      disp(A);
      fprintf('b= \n');
      disp(b);

 for j=2:N
      for i=j:N
         m = A(i,j-1)/A(j-1,j-1); %multiple of first equation to eliminate first en
         A(i,:) = A(i,:) - A(j-1,:)*m; %substracting multiples of first equation
         b(i) = b(i) - m*b(j-1);
      end
      % Printing matrices after each step
      fprintf("The matrices after %d step(s) are; \n", (j-1));
      fprintf('A= \n');
      disp(A);
      fprintf('b= \n');
      disp(b);
 end

%Perform back substitution
 x = zeros(N,1);
 x(N) = b(N)/A(N,N);

 for j=N-1:-1:1
   x(j) = (b(j)-A(j,j+1:N)*x(j+1:N))/A(j,j);
 end
 toc
end
```

Appendix 2

Gaussian Elimination with Scaled Partial Pivoting

Output from functions

```
Begin forward elimination with Augmented system:
    6.0000    2.0000    2.0000   -2.0000
    2.0000    0.6667    0.3333    1.0000
    1.0000    2.0000   -1.0000         0

After elimination in column 1 with pivot = 6.000000
    6.0000    2.0000    2.0000   -2.0000
         0         0   -0.3333    1.6667
         0    1.6667   -1.3333    0.3333

Swap rows 2 and 3;  new pivot = 1.66667

After elimination in column 2 with pivot = 1.666667
    6.0000    2.0000    2.0000   -2.0000
         0    1.6667   -1.3333    0.3333
         0         0   -0.3333    1.6667

ans =

    2.6000
   -3.8000
   -5.0000
```

(a) Results for Exercise 3 a) using GEPivShow

```
Begin forward elimination with Augmented system:
    6.0000    2.0000    2.0000   -2.0000
    2.0000    0.6667    0.3333    1.0000
    1.0000    2.0000   -1.0000         0

After elimination in column 1 with pivot = 6.000000
    6.0000    2.0000    2.0000   -2.0000
         0    0.0000   -0.3334    1.6667
         0    1.6667   -1.3333    0.3333

Swap rows 2 and 3;  new pivot = 1.66667

After elimination in column 2 with pivot = 1.666667
    6.0000    2.0000    2.0000   -2.0000
         0    1.6667   -1.3333    0.3333
         0         0   -0.3333    1.6667

ans =

    2.5999
   -3.7999
   -4.9999
```

(b) Results for Exercise 3 b) using GEPivShow

Figure 6: Results for Exercise 3

Pseudo Code
real array $(a_{ij})_{n\times n}, (b_i)_n, (x_i)_n$
integer i,j,k,l,n, p, ip
real pivot
for i=1 **to** n **do**
    $pivot = a_{i,1}$
    p = i
    for j=1 **to** n **do**
        if $a_{i,j} > pivot$ **then**
            $pivot = a_{i,j}$
            ip = i
        end if
        if $ip \neq p$ **then**
            $a_{i,:} = a_{ip,:}$ & & $a_{ip,:} = a_{i,:}$
            $b_i = b_{ip}$ & & $b_{ip} = b_i$
        end if
        $pivot = a_{i,i}$
        Gaussian elimination and back substitution is used on the matrix
        The pseudo code can be found in appendix 1
    end for
end for

MatLab Code

```
function x = GEPivShow(A,b,ptol)
% GEPivShow  Show steps in Gauss elimination with partial pivoting and
%            back substitution
%
% Synopsis:  x = GEPivShow(A,b)
%            x = GEPivShow(A,b,ptol)
%
% Input:     A,b  = coefficient matrix and right hand side vector
%            ptol = (optional) tolerance for detection of zero pivot
%                   Default:  ptol = 50*eps
%
% Output:    x = solution vector, if solution exists

if nargin<3, ptol = 50*eps;  end
[m,n] = size(A);
if m~=n,  error('A matrix needs to be square');  end
nb = n+1;   Ab = [A b];     % Augmented system
fprintf('\nBegin forward elimination with Augmented system:\n');  disp(Ab);

% --- Elimination
for i = 1:n-1                              % loop over pivot row
  [pivot,p] = max(abs(Ab(i:n,i)));         % value and index of largest available pivot
  ip = p + i - 1;                          % p is index in subvector i:n
  if ip~=i                                 % ip is true row index of desired pivot
    fprintf('\nSwap rows %d and %d;  new pivot = %g\n',i,ip,Ab(ip,i));
    Ab([i ip],:) = Ab([ip i],:);           % perform the swap
  end
  pivot = Ab(i,i);
  if abs(pivot)<ptol, error('zero pivot encountered after row exchange');
end
  for k = i+1:n                  % k = index of next row to be eliminated
     Ab(k,i:nb) = Ab(k,i:nb) - (Ab(k,i)/pivot)*Ab(i,i:nb);
  end
  fprintf('\nAfter elimination in column %d with pivot = %f\n',i,pivot);
  disp(Ab);
end

% --- Back substitution
x = zeros(n,1);              % preallocate memory for and initialize x
x(n) = Ab(n,nb)/Ab(n,n);
for i=n-1:-1:1
  x(i) = (Ab(i,nb) - Ab(i,i+1:n)*x(i+1:n))/Ab(i,i);
end
```

Appendix 3

Gaussian Elimination with shown intermediate steps

Output for system of equations d)

```
Begin forward elmination with Augmented system:
    6.0000    2.0000    2.0000   -2.0000
    2.0000    0.6667    0.3333    1.0000
    1.0000    2.0000   -1.0000         0


After elimination in column 1 with pivot = 6.000000
    6.0000    2.0000    2.0000   -2.0000
         0    0.0000   -0.3334    1.6667
         0    1.6667   -1.3333    0.3333


After elimination in column 2 with pivot = 0.000033
   1.0e+04 *

    0.0006    0.0002    0.0002   -0.0002
         0    0.0000   -0.0000    0.0002
         0         0    1.6667   -8.3333


ans =

    2.5999
   -3.7999
   -4.9999
```

Figure 7: Results for Exercise 3 b) using GEshow

Pseudo Code

This code is the same as seen in Appendix 2 except that the intermediate steps are printed after every iteration.

MatLab Code

```
function x = GEshow(A,b,ptol)
% GEshow  Show steps in Gauss elimination and back substitution
%         No pivoting is used.
%
% Synopsis:  x = GEshow(A,b)
%            x = GEshow(A,b,ptol)
%
% Input:     A,b  = coefficient matrix and right hand side vector
%            ptol = (optional) tolerance for detection of zero pivot
%                   Default:  ptol = 50*eps
%
% Output:    x = solution vector, if solution exists
tic;
if nargin<3, ptol = 50*eps;  end
[m,n] = size(A);
if m~=n,  error('A matrix needs to be square');  end
nb = n+1;   Ab = [A b];     % Augmented system
fprintf('\nBegin forward elmination with Augmented system:\n');  disp(Ab);

% --- Elimination
for i = 1:n-1                  % loop over pivot rows
  pivot = Ab(i,i);
  if abs(pivot)<ptol, error('zero pivot encountered');  end
  for k = i+1:n                % k = index of next row to be eliminated
     Ab(k,i:nb) = Ab(k,i:nb) - (Ab(k,i)/pivot)*Ab(i,i:nb);
  end
  fprintf('\nAfter elimination in column %d with pivot = %f\n',i,pivot);
  disp(Ab);
end

% --- Back substitution
x = zeros(n,1);              % preallocate memory for and initialize x
x(n) = Ab(n,nb)/Ab(n,n);
for i=n-1:-1:1
  x(i) = (Ab(i,nb) - Ab(i,i+1:n)*x(i+1:n))/Ab(i,i);
end
toc;
end
```

Appendix 4

Gaussian elimination with scaled partial pivoting and specific number of decimals

Pseudo Code

The pseudo code is exactly the same as seen in appendix 2 though the only difference is during the Gaussian elimination when subtracting multiples of the first equation from the successive equations. Here the "gradient" m is rounded to having r_dec number of decimals.
ie. the change in pseudo code is the following;

integer r_{dec}
$m = round\left(\frac{a_{(i,j-1)}}{a_{(j-1,j-1)}}, r_dec\right)$

MatLab Code

```
function x = GEPivShow2(A,b,r_dec,ptol)
% GEPivShow  Show steps in Gauss elimination with partial pivoting and
%            back substitution
%
% Synopsis:  x = GEPivShow2(A,b)
%            x = GEPivShow2(A,b,ptol)
%
% Input:     A,b  = coefficient matrix and right hand side vector
%            r_dec = number of decimals used in calculations
%            ptol = (optional) tolerance for detection of zero pivot
%                   Default:  ptol = 50*eps
%
% Output:    x = solution vector, if solution exists

fprintf('\nDecimal place rounding is %d', r_dec);

if nargin<4, ptol = 50*eps;  end
[m,n] = size(A);
if m~=n,  error('A matrix needs to be square');  end
nb = n+1;   Ab = [A b];      % Augmented system
%fprintf('\nBegin forward elimination with Augmented system:\n');  disp(Ab);

% --- Elimination
for i = 1:n-1                              % loop over pivot row
  [pivot,p] = max(abs(Ab(i:n,i)));         % value and index of largest available pivot
  ip = p + i - 1;                          % p is index in subvector i:n
  if ip~=i                                 % ip is true row index of desired pivot
    %fprintf('\nSwap rows %d and %d;  new pivot = %g\n',i,ip,Ab(ip,i));
    Ab([i ip],:) = Ab([ip i],:);           % perform the swap
  end
  pivot = Ab(i,i);
  if abs(pivot)<ptol, error('zero pivot encountered after row exchange');
end
```

```
  for k = i+1:n                 %  k = index of next row to be eliminated
     Ab(k,i:nb) =Ab(k,i:nb) - round((Ab(k,i)/pivot),r_dec)*Ab(i,i:nb);
     %Here round is used in order to only use r_dec number of decimals in
     %the calculation.
  end
  %fprintf('\nAfter elimination in column %d with pivot = %f\n',i,pivot);
  %disp(Ab);
end

% --- Back substitution
x = zeros(n,1);               %  preallocate memory for and initialize x
x(n) = Ab(n,nb)/Ab(n,n);
for i=n-1:-1:1
  x(i) = (Ab(i,nb) - Ab(i,i+1:n)*x(i+1:n))/Ab(i,i);
end
fprintf("The results is:\n");
disp(x);

end
```

Calculations of x-values using different number of decimals

Pseudo Code
real array $(a_{ij})_{n\times n}, (b_i)_n, (\text{values\_x1}_i)_n, (\text{values\_x2}_i)_n, (x_i)_n$
integer n,k
for k=1 **to** 10 **do**
Gaussian Elimination using Scaled Partial Pivoting and number of decimals is used here with k as the number of decimals. The pseudo code can be found in the previous section here in appendix 3.
$\text{values\_x1}_{1,k} = x_1$(from output of GEPivShow2)
$\text{values\_x2}_{1,k} = x_2$(from output of GEPivShow2)
$x_{1,k} = k$
end for
Graph of both x_1 and x_2 values are plotted on the y-axis against the list of k-values as the x-axis.

MatLab Code

```
function x = decimal_calculations(A,b)
% This program will calculate solutions for a system of equations
% It is done using a varying number of decimals (from 1 to 10)
% The function GEPivShow2 is used in order to perform Gaussian Elimination
% with Scaled Partial Pivoting on the system of equations with the specific
% number of decimals.
% The output from this function is an array [ x_1 x_2 ].
%
% input = A - matrix of coefficients for the system of equations
%         b - coloumn vector with the RHS of the system of equations
```

```
%
% output = values_x1 - a vector containing the different x_1 values
%          values_x2 - a vector containing the different x_ values
%          a graph depicting the two different x-values against the number
%          of decimals.

% Defining arrays for later use
values_x1 = zeros(1,size(A,1)); % will contain list of x_1 values
values_x2 = zeros(1,size(A,1)); % will contain list of x_2 values
x_axis = zeros(1,size(A,1)); % for plotting graph

for k=1:10 % number of decimals
    res = GEPivShow2(A,b,k); % Gaussian Elimination using scaled partial pivoting
    values_x1(1,k) = res(1); % the kth entry is equal to the specific x_1
    values_x2(1,k) = res(2); % the kth entry is equal to the specific x_2
    x_axis(1,k) = k; % Adding the number of decimals to the x-axis for graphing
end

%Printing the lists of x_1 and x_2 values
fprintf('x1 values; \n');
disp(values_x1);
fprintf('x2 values; \n');
disp(values_x2);

%Graph
figure
plot(x_axis,values_x1,x_axis,values_x2);
title('x values for different number of decimals');
xlabel('Number of decimals');
ylabel('x value');
legend('x_1','x_2');
end
```

Appendix 5

Calculation of condition number

Pseudo Code
real array $(a_{ij})_{n\times n}$
integer n
The MatLab function cond() is used in order to calculate the condition number for the input matrix A. This result is printed to the command window.

MatLab Code

```
function x = cond_calc(A)
% This program will calculate the condition number for a matrix
% Input = A - matrix
% Output = cond(A) - the condition number for matrix A.
%

% Display matrix entered.
fprintf('The matrix entered is; \n');
disp(A);

% The value is just printed but one could have put it into a variable
% This wouldn't be particularly usefull so I didn't in order to save
% memory.
fprintf('The condition number for the matrix is %d. \n', cond(A));

end
```

Appendix 6

Interpolation

Pseudo Code
real array $((x)_i)_n, ((y)_i)_n, (matrix_{ij})_{n\times n}, (a_i)_n$
integer n,k,j
for k=1 **to** n **do**
    for j=0 **to** n-1 **do**
        $matrix_{(k,j+1)} = (x_k)^j$
    end for end for

Gaussian elimination using scaled partial pivoting is used in order to find solutions for augmented matrix (creatively called matrix) this list of solutions for the system of equations is put into the a matrix. The pseudo code for this can be found in appendix 2.

$f = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + a_4x^4$

This is the approximate function which is afterwards plotted in a graph together with the initial data points.

MatLab Code

```
%This program will interpolate a given dataset.
% This will be done using the following equations
% The approximate polynomial is of the order N-1, where N is the number of
% datapoints, and will be given by;
%   F(x) = sum from j=0 to N-1 of a_j * x^j
% where
%   sum from j=0 to N-1 of a_j * (x_k)^j = y_k

x_k = [ -1.0 -0.5 0.0 0.5 1.0 ];
y_k = [ 4.1 -.13 0.5 1.1 3.6 ];

% We find N to use for later calculations.
N = length(x_k);

 matrix = zeros(N,N); % augmented matrix

 % We now calcualte each augmented matrix entry
for k=1:N
    for j=0:(N-1)
        matrix(k,(j+1)) = (x_k(k)).^j; % calculate the coefficients for the simulta
    end
end
fprintf('This is the matrix of coefficients;');
disp(matrix);

a = GEPivShow(matrix,y_k') %We now have the a-values by Gaussian Elimination
```

```
% We can now find the approximate polynomial
f = @(x) a(1)*x^0 + a(2)*x^1 + a(3)*x^2 + a(4)*x^3 + a(5)*x^4

% Graph
scatter(x_k,y_k) %Data points
grid on
hold on
fp = fplot(f,[-1,1]) %Function
fp.LineStyle = '- -';
title('Exercise 6')
legend('Dataset','Approximate polynomial')
xlabel('x')
ylabel('y')
hold off
```

Appendix 7

Interpolation using function

Pseudo Code
real array $((x)_i)_n, ((y)_i)_n, (matrix_{ij})_{n\times n}, (a_i)_n$
integer n,k,j
for k=1 **to** n **do**
  for j=0 **to** n-1 **do**
    if j = 0 **then**
      $matrix_{(k,j+1)} = 1$
    else if j = 1 **then**
      $matrix_{(k,j+1)} = sin(x_k)$
    else if j = 2 **then**
      $matrix_{(k,j+1)} = cos(x_k)$
    else if j = 3 **then**
      $matrix_{(k,j+1)} = sin(2x_k)$
    else if j = 4 **then**
      $matrix_{(k,j+1)} = cos(2x_k)$
    else if j = 5 **then**
      $matrix_{(k,j+1)} = sin(3x_k)$
    else if j = 6 **then**
      $matrix_{(k,j+1)} = cos(3x_k)$
    end if
  end for
end for

Gaussian elimination using scaled partial pivoting is used in order to find solutions for augmented matrix (creatively called matrix) this list of solutions for the system of equations is put into the a matrix. The pseudo code for this can be found in appendix 2.

$f = a_0 + a_1 sin(x) + a_2 cos(x) + a_3 sin(2x) + a_4 cos(2x) + a_5 sin(3x) + a_6 cos(3x)$

This is the approximate function which is afterwards plotted in a graph together with the initial data points.

MatLab code

```
%Interpolation of set of data using an equation of the form
% a_0 + a_1*sin(x) + a_2*cos(x) + a_3*sin(2x) + a_4*cos(2x) + a_5*sin(3x) + a_6*cos

x_k = [ -3.0999 -2.0666 -1.0333 0.0000 1.0333 2.0666 3.0999 ];
y_k = [ -0.97309 3.5557 1.1402 1.2000 2.0412 -0.39794 -1.0535 ];

% We find N to use for later calculations.
N = length(x_k);

 matrix = zeros(N,N); % augmented matrix
```

```
% The coefficients in the matrix now depends on which a-value.
% Hence for a_0 we have the entry equals 1, for a_1 it is sin(x), a_2
% cos(x) and so on.

for k=1:N
    for j=0:(N-1)
        %Depending on the value of j, the euqation coefficent is differnet.
        % hence all the if-elseif statements.
        if j == 0
            matrix(k,(j+1)) = 1;
        elseif j == 1
            matrix(k,(j+1)) = sin(x_k(k));
        elseif j == 2
            matrix(k,(j+1)) = cos(x_k(k));
        elseif j == 3
            matrix(k,(j+1)) = sin(2* (x_k(k)));
        elseif j == 4
            matrix(k,(j+1)) = cos(2* (x_k(k)));
        elseif j == 5
            matrix(k,(j+1)) = sin(3* (x_k(k)));
        elseif j == 6
            matrix(k,(j+1)) = cos(3* (x_k(k)));
        end
    end
end
fprintf("This is the matrix of coefficients;");
disp(matrix);

a = GEPivShow(matrix,y_k') %We now have the a-values byt Gaussian Elimination

% We can now find the approximate function using the first equation stated
f = @(x) a(1) + a(2)*sin(x) + a(3)*cos(x) + a(4)*sin(2*x) + a(5)*cos(2*x) + a(6)*si

%Graph
scatter(x_k,y_k) % Data points
grid on
hold on
fp = fplot(f,[-3,3]) %Function
fp.LineStyle = '- -';
title('Exercise 7')
legend('Dataset','Approximate function')
xlabel('x')
ylabel('y')
hold off

x_values = zeros(N,1);
y_values = zeros(N,1);
```

```
i=0;
while i <= (6/0.2)
    x_values((i+1),1) = -3.0000 + i*0.2;
    y_values((i+1),1) = f(x_values((i+1),1));

    i=i+1;
end

x_values
y_values

scatter(x_k,y_k)
hold on
scatter(x_values,y_values,'+','r')
grid on
hold on
fp = fplot(f,[-3,3],'g')
fp.LineStyle = '- -';
title('Exercise 7')
legend('Dataset','Data from function','Approximate function')
xlabel('x')
ylabel('y')
hold off
```

Appendices Lab 2

Appendix 1

Bisection Method

Pseudo Code
integer it_count, max_iterate
real a, b, c, a0, b0, e, fa, fb, fc
a = a0, b = b0, fa = f(a), fb=f(b), it_count = 0
while $b - c > ep$ **and** $it\_count < max\_iterate$ **do**
    it_count += 1
    fc = f(c)
    if $sign(fb) * sign(fc) \leq 0$ **then**
        a=c, fa=fc
    else
        b=c, fb=fc
    end if
c = $\frac{a+b}{2}$
end while

The resultant root c is plotted together with the corresponding function which is chosen by a an input to the overall program in a switch statement in a function defined in the program and then called.

MatLab Code

```
function root=bisect2(a0,b0,ep,max_iterate,index_f)
%
% function bisect(a0,b0,ep,max_iterate,index_f)
%
% This is the bisection method for solving an equation f(x)=0.
%
% The function f is defined below by the user. The function f is
% to be continuous on the interval [a0,b0], and it is to be of
% opposite signs at a0 and b0. The quantity ep is the error
% tolerance. The routine guarantees this as an error bound
% provided: (1) the restrictions on the initial interval are
% correct, and (2) ep is not too small when the machine epsilon
% is taken into account. Most of these conditions are not
% checked in the program! The parameter max_iterate is an upper
% limit on the number of iterates to be computed.
%
% For the given function f(x), an example of a calling sequence
% might be the following:
%    root = bisect(1,1.5,1.0E-6,10,1)
% The parameter index_f specifies the function to be used.
%
% The following will print out for each iteration the values of
```

```
%       count, a, b, c, f(c), (b-a)/2
% with c the current iterate and (b-a)/2 the error bound for c.
% The variable count is the index of the current interate.  Tap
% the carriage return to continue with the iteration.
tic;
if a0 >= b0
    disp('a0 < b0 is not true.  Stop!')
    return
end

format short e
a = a0; b = b0;
fa = f(a,index_f); fb = f(b,index_f);

if sign(fa)*sign(fb) > 0
    disp('f(a0) and f(b0) are of the same sign.  Stop!')
    return
end

c = (a+b)/2;
it_count = 0;
while b-c > ep & it_count < max_iterate
    it_count = it_count + 1;
    fc = f(c,index_f);
%   Internal print of bisection method. Tap the carriage
%   return key to continue the computation.
    iteration = [it_count a b c fc b-c];
    if sign(fb)*sign(fc) <= 0
        a = c;
        fa = fc;
    else
        b = c;
        fb = fc;
    end
    c = (a+b)/2;
end

format long
root = c
format short e
error_bound = b-c
format short
it_count


% Plot specifications
switch index_f
case 1
```

```
    fplot(@(z) sin(z)-exp(-z),[a0-1,b0+1]); %function
    hold on
    plot(c,sin(c)-exp(-c),'r*'); %root

case 2
    fplot(@(z) z.^3 - 3*z + 1,[a0-1,b0+1]);
    hold on
    plot(c,c.^3 - 3*c + 1,'r*');

 case 3
    fplot(@(z)log(z-1) + z.^2,[a0-1,b0+1]);
    hold on
    plot(c,log(c-1) + c.^2,'r*');

end
    grid on
    xL = xlim; yL = ylim; line(xL,[0,0],'color','k','linewidth',1); line([0,0],yL,
    xlabel('x'); ylabel('y'); %axis labels
    hold off

toc;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = f(x,index)

% function to define equation for rootfinding problem.

switch index
case 1
    value = sin(x) - exp(-x);
case 2
    value = x.^3 -3*x +1;
case 3
    value = log(x-1) + x.^2;
case 4
    value = sign(x-2)*sqrt(abs(x-2));
end
end
```

Appendix 2

Secant Method

Pseudo Code
integer max_iterate,it_count,
real e, x0, x1, x2, error_bd, fx0, fx1, error
error = 1, fx0 = f(x0), it_count = 0
while $|$ *error* $|>$ *error_bd* **and** *it_count* $\leq$ *max_iterate* **do**
    it_count += 1, fx1 = f(x1)
    if $fx1 - fx0 = 0$ **then**
        break out of function
    end if
$x2 = x1 - fx1 * \frac{x1-x0}{fx1-fx0}$, error = x2 - x1,
x0 = x1, x1 = x2, fx0 = fx1
end while

The resultant root c is plotted together with the corresponding function which is chosen by a an input to the overall program in a switch statement in a function defined in the program and then called.

MatLab Code

```
function root = secant2(x0,x1,error_bd,max_iterate,index_f)
%
% function secant(x0,x1,error_bd,max_iterate,index_f)
%
% This implements the secant method for solving an
% equation f(x) = 0. The function f(x) is given below.
%
% The parameter error_bd is used in the error test for the
% accuracy of each iterate.  The parameter max_iterate is an
% upper limit on the number of iterates to be computed. Two
% initial guesses, x0 and x1, must also be given.
%
% For the given function f(x), an example of a calling sequence
% might be the following:
%    root = secant(x0,x1,1.0E-12,10,1)
% The parameter index_f specifies the function to be used.
%
% The program prints the iteration values
%      iterate_number, x, f(x), error
% The value of x is the most current initial guess, called
% previous_iterate here, and it is updated with each iteration.
% The value of error is
%    error = newly_computed_iterate - previous_iterate
% and it is an estimated error for previous_iterate.
% Tap the carriage return to continue with the iteration.
```

```
tic;
% For plotting later
a0 = x0;
b0 = x1;

format short e
error = 1;
fx0 = f(x0,index_f);
it_count = 0;
iteration = [it_count x0 fx0];
while abs(error) > error_bd & it_count <= max_iterate
    it_count = it_count + 1;
    fx1 = f(x1,index_f);
    if fx1 - fx0 == 0
        disp('f(x1) = f(x0); Division by zero;  Stop')
        return
    end
    x2 = x1 - fx1*(x1-x0)/(fx1-fx0);
    error = x2 - x1;
%   Internal print of secant method. Tap the carriage
%   return key to continue the computation.
    iteration = [it_count x1 fx1 error];
    x0 = x1;
    x1 = x2;
    fx0 = fx1;
end

if it_count > max_iterate
    disp('The number of iterates calculated exceeded')
    disp('max_iterate.  An accurate root was not')
    disp('calculated.')
else
    format long
    root = x2
    format short e
    error
    format short
    it_count
end

% Plot specifications
switch index_f
case 1
    fplot(@(z) sin(z)-exp(-z),[a0-1,b0+1]); %function
    hold on
    plot(root,sin(root)-exp(-root),'r*'); %root

case 2
```

```
    fplot(@(z) z.^3 - 3*z + 1,[a0-1,b0+1]);
    hold on
    plot(root,root.^3 - 3*root + 1,'r*');
 case 3
    fplot(@(z) log(z-1) + z.^2,[a0-1,b0+1]);
    hold on
    plot(root,log(root-1) + root.^2,'r*');
end
    grid on
    xL = xlim; yL = ylim; line(xL,[0,0],'color','k','linewidth',1); line([0,0],yL,
    xlabel('x'); ylabel('y'); %axis labels
    title('Exercise 2') % plot title
    hold off
toc;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = f(x,index)

% function to define equation for rootfinding problem.

switch index
case 1
    value = sin(x) - exp(-x);
case 2
    value = x.^3 -3*x +1;
 case 3
    value = log(x-1) + x.^2;
case 4
    value = sign(x-2)*sqrt(abs(x-2));
end
end
```

Appendix 3

Newton-Raphson Method

Pseudo Code

integer max_iterate, it_count
real x0, x1, error_bd, error, fx, dfx
error = 1, it_count = 0
while | *error* |> *error_bd* **and** *it_count* $\leq$ *max_iterate* **do**
    fx = f(x0), dfx = f'(x)
    if dfx = 0 **then**
        break out of function
    end if $x1 = x0 - \frac{fx}{dfx}$, error = x1 - x0
    x0 = x1, it_count += 1
end while

The resultant root c is plotted together with the corresponding function which is chosen by a an input to the overall program in a switch statement in a function defined in the program and then called.

MatLab Code

```
function root = newton2(x0,error_bd,max_iterate,index_f)
%
% function newton(x0,error_bd,max_iterate,index_f)
%
% This is Newton's method for solving an equation f(x) = 0.
%
% The functions f(x) and deriv_f(x) are given below.
% The parameter error_bd is used in the error test for the
% accuracy of each iterate.  The parameter max_iterate
% is an upper limit on the number of iterates to be
% computed.  An initial guess x0 must also be given.
%
% For the given function f(x), an example of a calling sequence
% might be the following:
%    root = newton(1,1.0E-12,10,1)
% The parameter index_f specifies the function to be used.
%
% The program prints the iteration values
%      iterate_number, x, f(x), deriv_f(x), error
% The value of x is the most current initial guess, called
% previous_iterate here, and it is updated with each iteration.
% The value of error is
%    error = newly_computed_iterate - previous_iterate
% and it is an estimated error for previous_iterate.
% Tap the carriage return to continue with the iteration.
tic;
% For plotting later
```

```
a0 = x0;

format short e
error = 1;
it_count = 0;
while abs(error) > error_bd & it_count <= max_iterate
    fx = f(x0,index_f);
    dfx = deriv_f(x0,index_f);
    if dfx == 0
        disp('The derivative is zero.  Stop')
        return
    end
    x1 = x0 - fx/dfx;
    error = x1 - x0;
%   Internal print of newton method. Tap the carriage
%   return key to continue the computation.
    iteration = [it_count x0 fx dfx error];
    x0 = x1;
    it_count = it_count + 1;
end

if it_count > max_iterate
    disp('The number of iterates calculated exceeded')
    disp('max_iterate.  An accurate root was not')
    disp('calculated.')
else
    format long
    root = x1
    format short e
    error
    format short
    it_count

end
    % Plot specifications
switch index_f
case 1
    fplot(@(z) sin(z)-exp(-z),[a0-1,3]); %function
    hold on
    plot(root,sin(root)-exp(-root),'r*'); %root

case 2
    fplot(@(z) z.^3 - 3*z + 1,[a0-1,3]);
    hold on
    plot(root,root.^3 - 3*root + 1,'r*');
 case 3
    fplot(@(z)log(z-1) + z.^2,[a0-1,3]);
    hold on
```

```
    plot(root,log(root-1) + root.^2,'r*');

    case 4
     fplot( @(z) sign(z-2)*sqrt(abs(z-2)),[a0,4]);
     hold on
end
    grid on
    xL = xlim; yL = ylim; line(xL,[0,0],'color','k','linewidth',1); line([0,0],yL,
    xlabel('x'); ylabel('y'); %axis labels
    %title('Exercise 3') % plot title
    hold off
toc;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = f(x,index)

% function to define equation for rootfinding problem.

switch index
case 1
    value = sin(x) - exp(-x);
case 2
    value = x.^3 -3*x +1;
 case 3
    value = log(x-1) + x.^2;
case 4
    value = sign(x-2)*sqrt(abs(x-2));
end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = deriv_f(x,index)

% Derivative of function defining equation for rootfinding
% problem.

switch index
case 1
    value = cos(x) + exp(-x);
case 2
    value = 3*x.^2 - 3;
 case 3
    value = (1)/(x-1) + 2*x;
case 4
    value = (sign(x-2)*sqrt(abs(x-2))) / (2*(x-2));
end
```

end

Appendix 4

Newton-Raphson Method for more than one root

Pseudo Code
real array $(x_j)_3$
integer max_iterate, it_count, j
real error_bd, error, fx, fdx, x1
x = [1+i 1-i -4]
for j=0 **to** 2 **do**
  $error = 1$, $it\_count = 0$
  while $|\ error\ | > error\_bd$ **and** $it\_count \leq max\_iterate$ **do**
    $fx = f(x_j)$, $dfx = f'(x)$
    if $dfx = 0$ **then**
      break out of function
    end if $x1 = x_j - \frac{fx}{dfx}$, $error = x1 - x_j$
    $x_j = x1$, $it\_count+ = 1$
end while

MatLab Code

```
function root = newton3(error_bd,max_iterate)
%
% function newton(x0,error_bd,max_iterate,index_f)
%
% This is Newton's method for solving an equation f(x) = 0.
%
% The functions f(x) and deriv_f(x) are given below.
% The parameter error_bd is used in the error test for the
% accuracy of each iterate.  The parameter max_iterate
% is an upper limit on the number of iterates to be
% computed.  An initial guess x0 must also be given.
%
% For the given function f(x), an example of a calling sequence
% might be the following:
%    root = newton(1,1.0E-12,10,1)
% The parameter index_f specifies the function to be used.
%
% The program prints the iteration values
%      iterate_number, x, f(x), deriv_f(x), error
% The value of x is the most current initial guess, called
% previous_iterate here, and it is updated with each iteration.
% The value of error is
%    error = newly_computed_iterate - previous_iterate
% and it is an estimated error for previous_iterate.
% Tap the carriage return to continue with the iteration.

tic;
for x0 = [ 1+i 1-i -4 ] % iterating over the three initial x-values
```

```
    fprintf( "x0 = %1.1f + %1.1fi \n",x0, x0/1i); % priting the initial x-value
    format short e
    error = 1;
    it_count = 0;
    while abs(error) > error_bd & it_count <= max_iterate
        fx = f(x0);
        dfx = deriv_f(x0);
        if dfx == 0
            disp('The derivative is zero.  Stop')
            return
        end
        x1 = x0 - fx/dfx;
        error = x1 - x0;
    %   Internal print of newton method. Tap the carriage
    %   return key to continue the computation.
        iteration = [it_count x0 fx dfx error];
        x0 = x1;
        it_count = it_count + 1;
    end

    if it_count > max_iterate % if the max number of iterations is met.
        disp('The number of iterates calculated exceeded')
        disp('max_iterate.  An accurate root was not')
        disp('calculated.')
    else

        format long
        root = x1;
        fprintf( "root = %.4f + %.4fi \n",root, root/1i);
        format short e
        error
        format short
        it_count

    end
end
toc;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = f(x)

% function to define equation for rootfinding problem.

value = x.^3 +2*x.^2 - x + 5;

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function value = deriv_f(x)

% Derivative of function defining equation for rootfinding
% problem.

value = 3*x.^2 + 4*x -1;

end
```

Appendix 5

Failures of the Newton-Raphson Method

Output from the different methods

```
>> newton2(1,0.0001,100,4)
The number of iterates calculated exceeded
max_iterate.  An accurate root was not
calculated.
```

Figure 8: Newton-Raphson Method

```
>> bisect2(1,3,0.0001,100,4)

root =

   2.000061035156250


error_bound =

   6.1035e-05


it_count =

    14


ans =

    2.0001
```

(a) Bisection Method

```
>> secant2(1,3,0.0001,100,4)

root =

     2


error =

     0


it_count =

     2


ans =

     2
```

(b) Secant Method

Figure 9: Outputs using MatLab Code