Table of Contents

Table of Figures

Table of Tables

Please give bonus marks to save my plummeting GPA. Thanks.

Question 1: Root Approximation

Question1: Implement the Bisection, Newton-Raphson, Secant, and False-Position methods in a MATLAB program.

Use the non-linear function $e^x - x^3 + 10$ as a test problem and select suitable starting values for each method mentioned above and compute the root. Also, prepare a table of iterations for different stopping criteria and comment on the performance of each method.

The function $f(x) = e^x - x^3 + 10$ has two roots: 3.57647 and 3.87591 (correct to 5 decimal places. These roots were derived using Wolfram Alpha and the MATLAB data cursor.

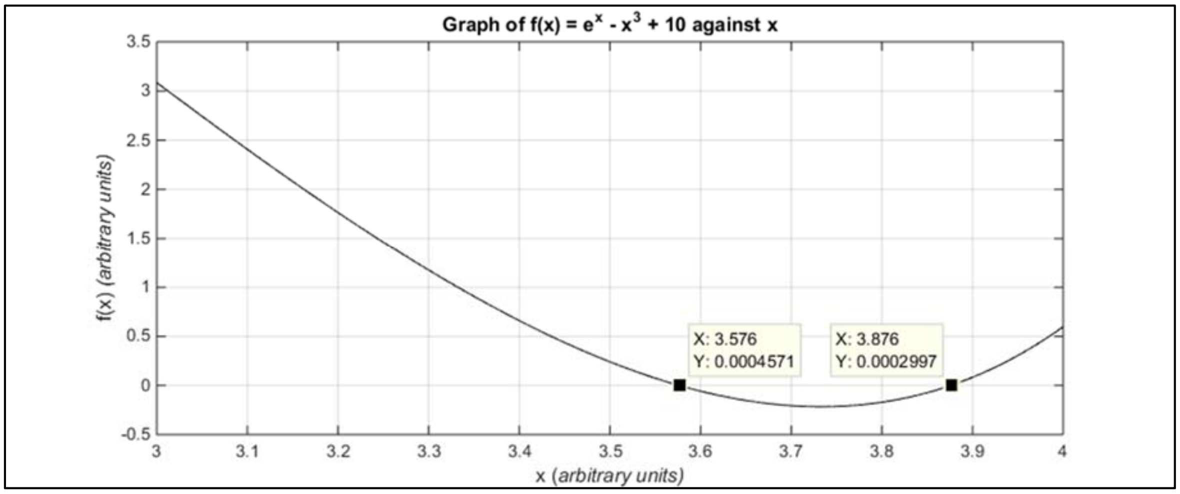

Figure 1: Graph of $f(x)$ against x, showing two roots

Code 01 (Appendix 1) was used to investigate the relative speed of convergence of each method. For testing each of the four aforementioned root finding methods, the smaller root i.e. $x_0 \approx 3.57647$ was chosen. The interval chosen for the bisection, secant, and false position methods was $[3.5, 3.6]$, with the initial approximation for the Newton-Raphson method being 3.5. The number of iterations taken for each method to converge to the given root for different tolerance values is summarized in the table below.

Method	$\lvert f(x)\rvert < 10^{-5}$	$\lvert f(x)\rvert < 10^{-10}$	$\lvert f(x)\rvert < 10^{-15}$
Bisection	13	30	44
Newton-Raphson	3	4	5
Secant	3	5	6
False Position	5	12	17

Table 1: Iterations until Convergence for Root Finding Methods

For a better comparison of the data, the number of iterations taken by each method until convergence for each tolerance level was plotted on grouped bar chart (Figure 2).

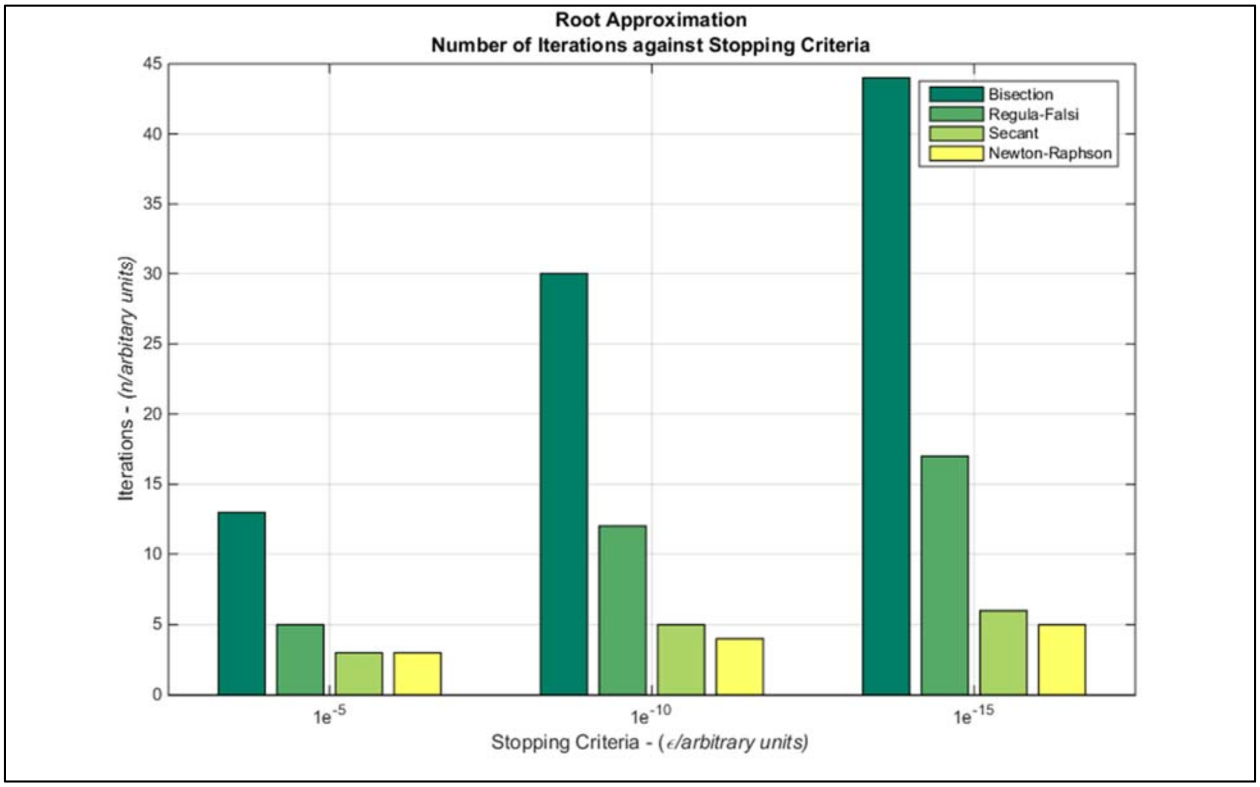

Figure 2: Comparing Convergence of Root Approximation Methods

<u>General Comments on Results</u>

- Decreasing value of stopping criteria corresponds to increasing accuracy, which is why for all root approximation methods tested, the number of iterations until convergence increased with decreasing value of stopping criteria.
- For all error/tolerance levels, the Newton-Raphson method took the least number of iterations to converge, followed by the Secant method and the Regula-Falsi (false position) method.
- The bisection method took the most iterations to converge to a root for all tolerance levels. It is clearly the slowest of all root finding methods tested.
- For the highest tested tolerance level (10^{-5}), the Newton-Raphson and Secant methods took the same number of iterations to converge. For the lowest tolerance level (10^{-15}), the Newton-Raphson method is only slightly faster than the secant method.
- The more complex the implementation of a root approximation method, the faster the method is to converge (case in point: Newton-Raphson).

Comments on Individual Methods

Method	Comments
Newton-Raphson	The fastest method for approximating the root of the polynomial, since it takes the least number of iterations to converge. However, it is not guaranteed to converge. It is also susceptible to oscillating between two root approximation that are symmetric about the actual root, as was experienced for tolerance level $1e^{-15}$ (iterations were derived by stepping through the method with MATLAB debugger and examining the workspace).
Secant	Almost as fast as the Newton-Raphson method in terms of iterations taken until convergence, and is not susceptible to the oscillation caused by symmetric roots. However, this method is sensitive to the initial upper and lower bounds provided as root approximations. If these values are not close enough to the root, the method can fail to converge, or even diverge.
Regula-Falsi	Faster than the bisection method but slower than the closely related Secant method, the Regula-Falsi method has a linear rate of convergence. While the Newton-Raphson and Secant methods were observed to fail to converge (or even diverge) under certain conditions for both roots, the Regula-Falsi method always converged. This is a good method to use for problems in which the speed of convergence is not necessarily a priority as long as convergence does occur.
Bisection	The slowest of all root approximation methods to converge. Took the largest number of iterations until convergence across all tested tolerance levels. However, since it is a bracketing method, it is guaranteed to converge to the root given enough computation time. Compared to the regula-falsi method, which also guarantees convergence, the bisection method is very slow. Hence, there is very little reason to use this method unless simplicity of implementation is a priority.

Table 2: Comments on Root Finding Methods

Conclusion: Newton-Raphson method is the best choice for this root finding problem.

Question 2: Regression Models

Question 2: Generate 100 data for the function $y(t) = t^2 - 2t + e^t$, where t ranges from 0 to 1. Fit a linear model, exponential model, and power model on the data set. Give numerical backed comment, which method describes the generated data best.

Using Code 2 (Appendix 2), the function $y(t) = t^2 - 2t + e^t$ was plotted on a figure. The data for the function was then used to generate three different regression models to fit the data: linear, exponential, and power.

$$y_{linear} = ax + b$$

$$y_{exponential} = me^{nx}$$

$$y_{power} = mc^{x}$$

The program was then used to plot all three regression models on the same figure as $y(t)$ (Figure 3) for a quick, visual inspection of which model fit the data best.

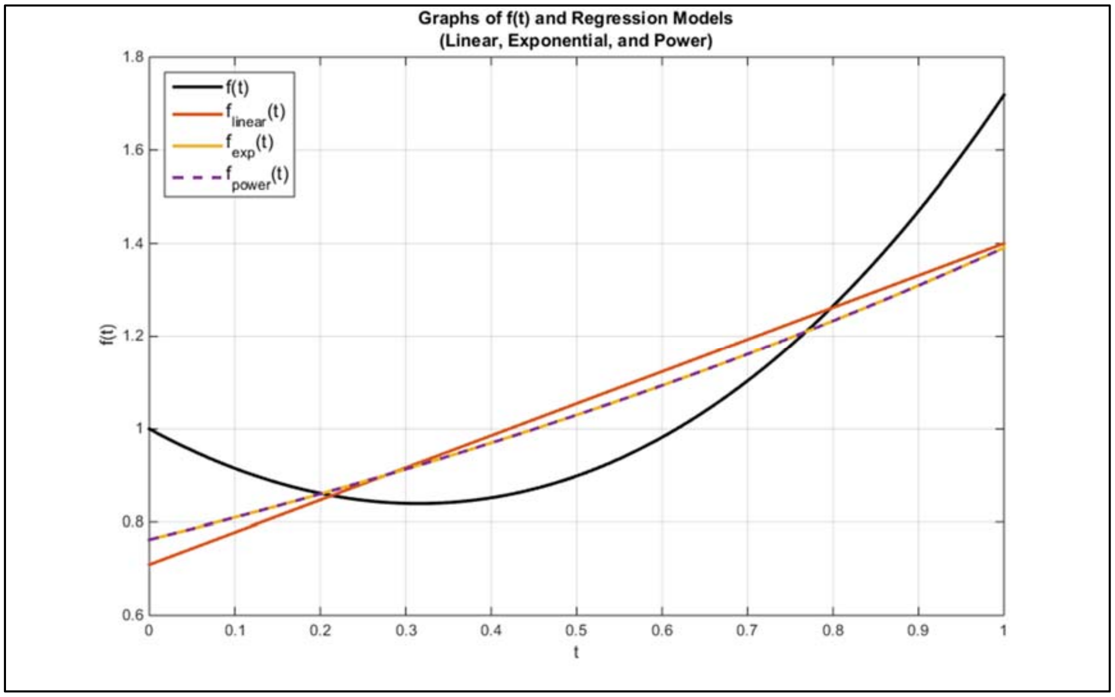

Figure 3: Regression Models for $y(t)$

The code also calculated and displayed the root-mean squared (RMS) errors between the values of each regression model and the function $y(t)$ for each value of the independent variable t. Figure 4 shows the RMS errors between each model and the actual function.

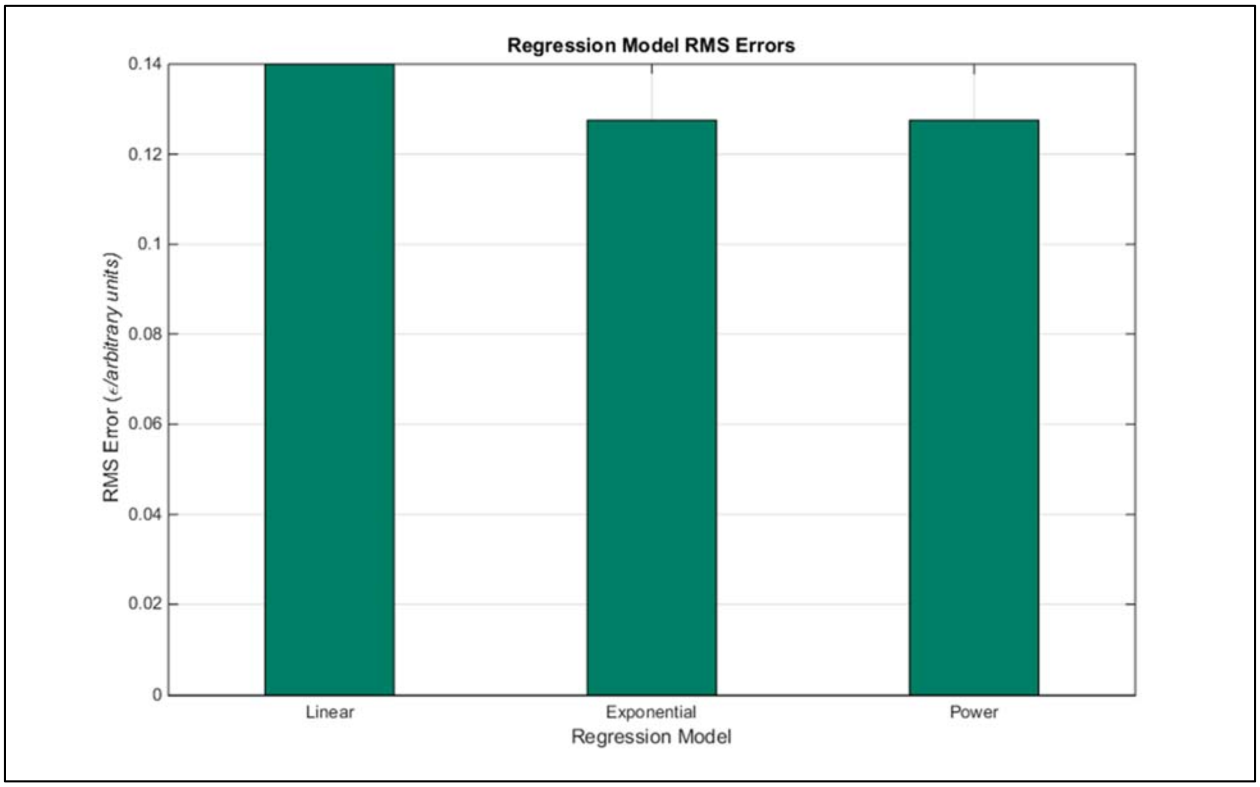

Figure 4: RMS Errors for Each Regression Model

The three models derived to fit this data are as follows

$$y_{linear} = 0.6912 + 0.7091x$$
$$y_{exponential} = 0.7627e^{0.6007x}$$
$$y_{power} = 0.7627e^{1.8324}$$

The root-mean squared errors for each regression model as computed by Code 02 are

Model	RMS Error	Best Fit
Linear	0.139961205576074	No
Exponential	0.127525790778834	Yes
Power	0.127525790778834	Yes

Table 3: RMS Error between Regression Models and $y(t)$

<u>Comments on Regression Results</u>

- The RMS error compared to the actual values of $y(t)$ is considerably large for all three models, which suggests that a different kind of regression model (for instance, a polynomial model) is more suitable for this data set.
- The exponential and power models have the smallest RMS errors, which means they are the best choices amongst all three models tested for this data.
- The linear model RMS error is only 9.75% higher than that of exponential and power models.
- The RMS errors of exponential and power models are identical, which makes sense since the exponential model is a special case of the power model.
- **Best Model to Describe Given Data: Exponential or Power Model**

Question 3: Systems of Linear Equations

Question 3: Implement the Gauss-Seidel iterative method for an $(n \times n)$ order of linear system of equations in MATLAB. Assume a system of equation of your choice for verification purpose. However, code should be written in general for any order of system of linear equations. Comment on the number of iterations.

Code 03 (Appendix 3) uses the Gauss-Seidel iterative method to find the solutions to an $(n \times n)$ system of linear equations.

$$\underline{A}\,\underline{X} = \underline{B}$$

The code first checks that a solution to the system of equations exists, and that the number of equations and unknowns is equal. In order for the Gauss-Seidel method to converge, the coefficient matrix $\underline{A}$ must be diagonally dominant.

The code was tested using the following system of equations, but works for any $(n \times n)$ system.

$$\begin{bmatrix} 6 & -2 & 1 & 1 \\ 1 & -7 & 2 & 2 \\ -1 & 2 & 8 & 4 \\ 2 & 1 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \end{bmatrix}$$

The Gauss-Seidel method successfully calculated approximate solutions for the each of the unknowns in the system of equations for various tolerance levels. The approximate roots of the equation for a tolerance of $1e^{-10}$ are

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \approx \begin{bmatrix} 1.038993710695537 \\ -0.809643605865251 \\ 1.517819706508471 \\ 0.628930817604477 \end{bmatrix}$$

<u>Comments on Number of Iterations</u>

To test the relationship between the tolerance level and the number of iterations taken by the Gauss-Seidel method to converge to an approximate solution, Code 03 was run with tolerances of $10^{0}, 10^{-5}, 10^{-10}$ and 10^{-15}. Additionally, the RMS error between the actual roots computed by MATLAB and those calculated by the Gauss-Seidel method was also calculated as a way of investigating how the accuracy of solutions improved with decreasing tolerance levels.

Figure 4 shows number of iterations required by the Gauss-Seidel method to converge for a given tolerance level.

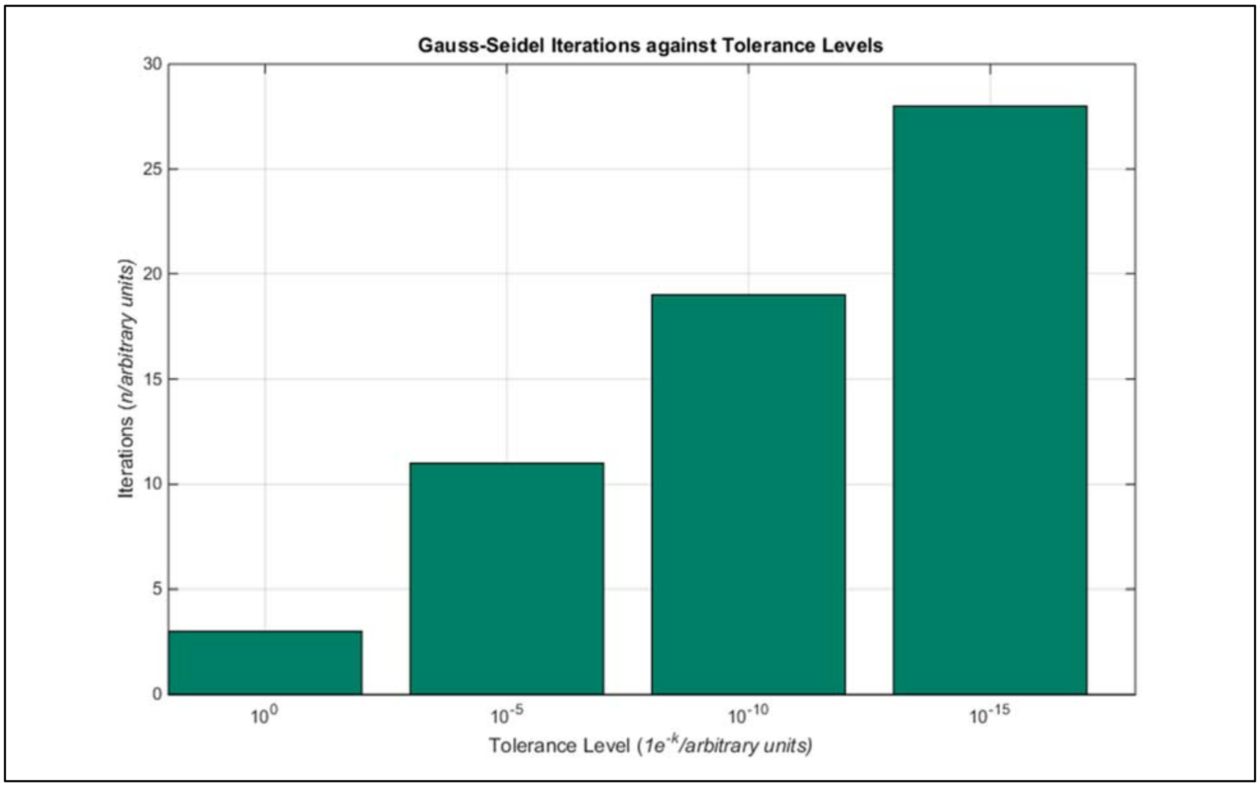

Figure 5: Number of Gauss-Seidel Iterations Until Convergence against Tolerance

Figure 5 clearly shows that the number of iterations taken by the Gauss-Seidel method to converge increases with decreasing tolerance level. Another way of interpreting this is to consider decreasing tolerance levels as increasing accuracy. This makes sense because the more accurate the desired the solution, the more times the Gauss-Seidel method has to compute values for the unknowns until the norm of the unknown vector is small enough to satisfy $\underline{A}\,\underline{X} - \underline{B} \approx 0$.

Furthermore, the number of computations involved in the

Comments on Accuracy

To assess whether decreasing tolerance has a significant effect on solution accuracy, the RMS error between the solution computed by MATLAB and that calculated by the Gauss-Seidel method was evaluated for different tolerance levels. The results are shown in Figure 6.

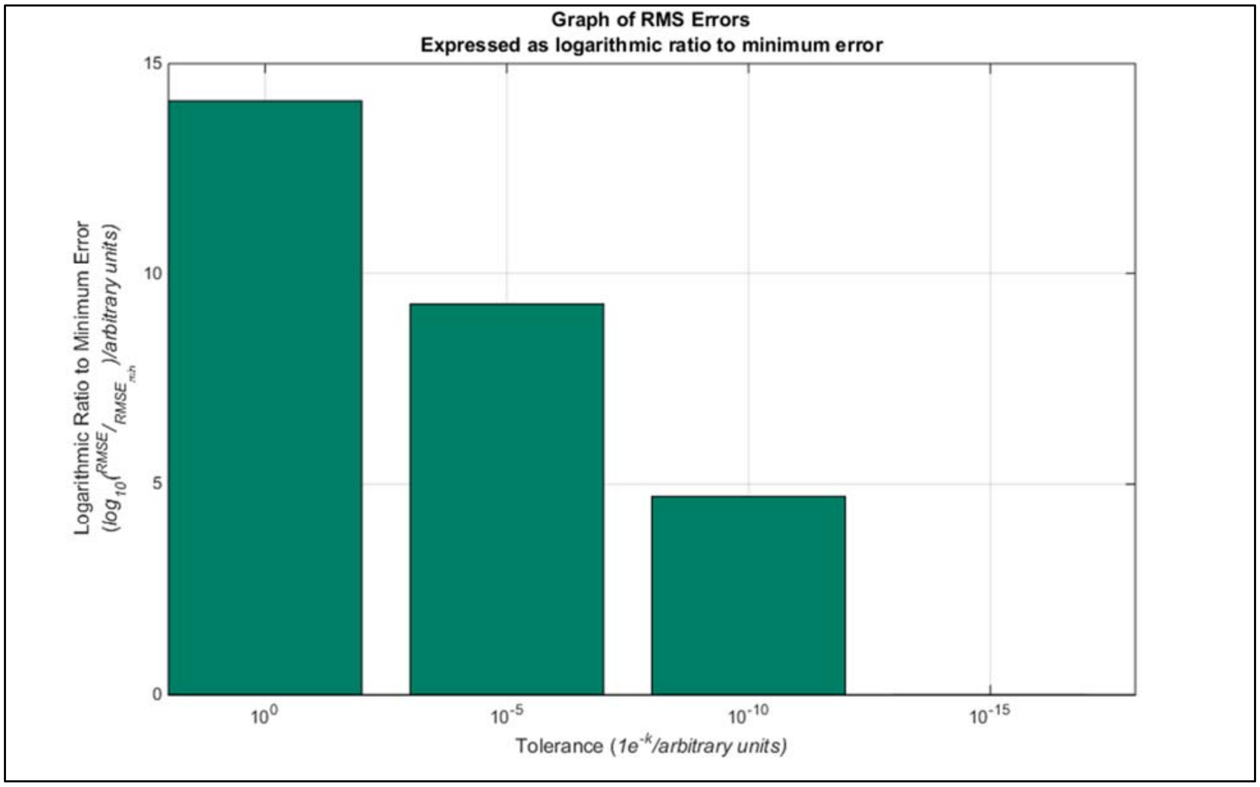

Figure 6: Comparing Relative Magnitudes of RMS Errors

The RMS errors for different tolerance levels differed by several orders of magnitude, with the error being 0.01604 for a tolerance of 10^0 to 1.24×10^{-16} for a tolerance of 10^{-15}. To better compare the relative differences in their orders of magnitude, the RMS errors were expressed the logarithm of the ratio to the minimum RMS error for the given test suite.

$$Relative\ RMS\ Error = \log_{10}\left(\frac{Raw\ RMS\ Error}{Minimum\ RMS\ Error}\right)$$

Figure 6 shows there is indeed benefit to gained from lower tolerance levels, as the RMS error with a tolerance of 10^0 is 14 orders of magnitude higher than that at e^{-15}. Since minimum RMS error and RMS error at tolerance of 10^{-15} had a ratio of 1, the relative error was 0 $\because \log_{10}\left(\frac{1.26\times10^{-16}}{1.26\times10^{-16}}\right) = \log_{10}(1) = 0.$

<u>**Final Comments**</u>

- **Decreasing tolerance level increased the number of iterations required for the Gauss-Seidel method to converge to a solution.**
- **Increasing the order of the system of linear equations increased the number of computations required for convergence, and hence increased program running time.**

Appendix A Root Approximation Code

Program Organization: Question 1

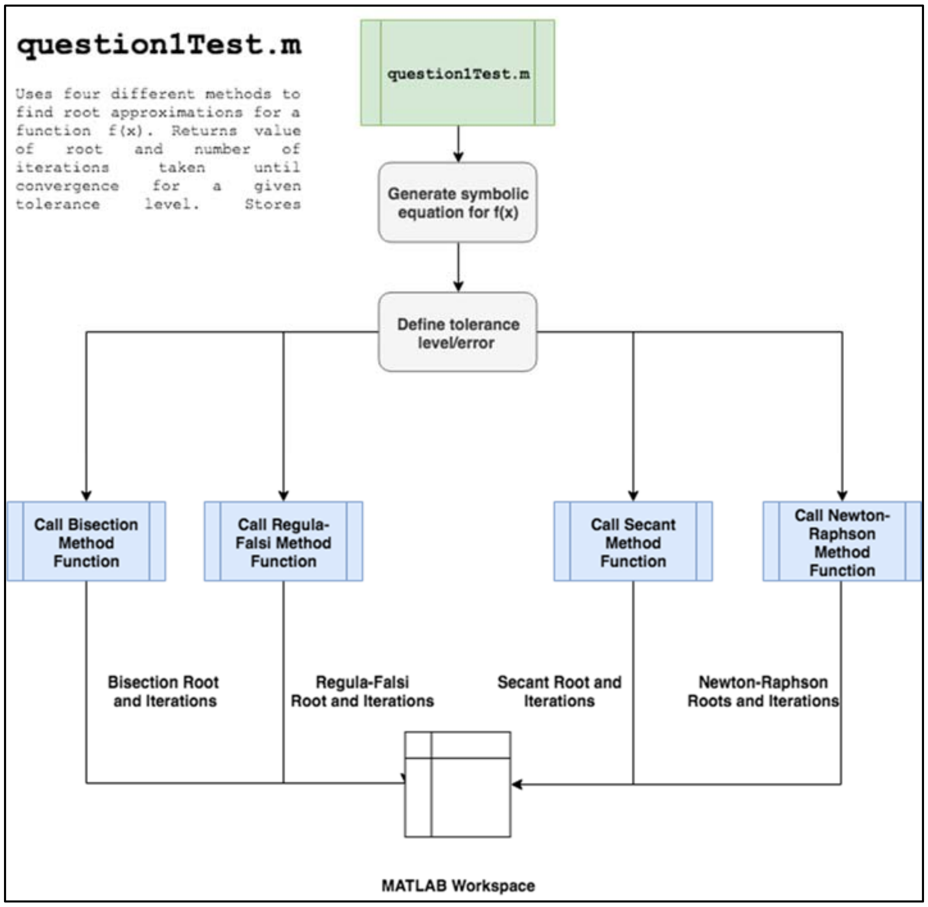
question1Test.m
Uses four different methods to find root approximations for a function f(x). Returns value of root and number of iterations taken until convergence for a given tolerance level. Stores
question1Test.m
Generate symbolic equation for f(x)
Define tolerance level/error
Call Bisection Method Function
Call Regula-Falsi Method Function
Call Secant Method Function
Call Newton-Raphson Method Function
Bisection Root and Iterations
Regula-Falsi Root and Iterations
Secant Root and Iterations
Newton-Raphson Roots and Iterations
MATLAB Workspace

Figure 7: Program Organization: Question 1

Pseudocode A1: `question1Test.m`

```
1. Initialise symbolic object for function whose root is to be found $f(x)$
2. Initialise tolerance level/error $\epsilon$
3. Call the bisectionSearch function with the following arguments
    a. equation whose root is to be found $f(x)$
    b. lower bound for interval containing root $lower$
    c. upper bound for interval containing root $upper$
    d. error $\epsilon$
4. Call the newtonRaphson function with the following arguments
    a. equation whose root is to be found $f(x)$
    b. error $\epsilon$
```

   c. initial root approximation x_0
5. Call the regulaFalsi function with the following arguments
   a. equation whose root is to be found $f(x)$
   b. lower bound for interval containing root $lower$
   c. upper bound for interval containing root $upper$
   d. error ϵ
6. Call the secantMethod function with the following arguments
   a. equation whose root is to be found $f(x)$
   b. lower bound for interval containing root $lower$
   c. upper bound for interval containing root $upper$
   d. maximum expected iterations n_{max}
   e. error ϵ
7. Echo the root approximations and number of iterations returned by each function

Code A1: `question1Test.m`

```
%%  question1Test.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   Finds roots of an equation using different numerical methods and
%   for levels of accuracy, and compares errors for each case.

%% Defining Equation to be used in Question 1
f_x = @(x) exp( x )- x^3 + 10;
error = 1e-15;

%% Testing Bisection Method
[x_approx_bisection, iters_bisection ] = bisectionSearch( f_x, 3.8, 3.9,
error );

%% Testing Newton-Raphson Method
[x_approx_nr, iters_nr ] = newtonRaphson( f_x, error, 3.8 );

%% Testing Regula-Falsi Method
[x_approx_rf, iters_rf ] = regulaFalsi( f_x, 3.8, 3.9, error );

%% Testing Secant Method
[x_approx_sec, iters_sec ] = secantMethod( f_x, 3.8, 3.9, 1000, error );
```
Code A1: question1Test.m

Pseudocode A2: `isRootInInterval.m`

1. Evaluate function at lower bound $f(lower)$
2. Evaluate function at upper bound $f(upper)$
3. Evaluate product of values from steps 1 and 2

```
        a. If the product $f(upper) \times f(lower)$ is less than 0, the root lies in the interval.
           Set the appropriate flag variable to true.
        b. If the product is greater than 0, the root does not lie in the interval. Set the flag
           variable to false.
    4. Return the flag variable.
```

Code A2: **isRootInInterval.m**

```
%%  isRootInInterval.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall
'18
%   a function that determines whether the root of an equation lies in
the
%   interval spanned by [lower, upper] using intermediate value theorem
%   corollary called Bonzano's theorem - f(lower) * f(upper) < 0

function rootFlag = isRootInInterval( lower, upper, equation )
if ( equation( lower ) * equation( upper ) < 0 )
    rootFlag = true;
else
    rootFlag = false;
end
```

Code A2:isRootInInterval.m

Pseudocode A3: **bisectionSearch.m**

```
1. Initialise all variables
    a. Set lower bound to value passed as argument.
    b. Set upper bound to value passed as argument.
    c. Set number of iterations to zero.
    d. Set loop continuation condition to true.
2. Use isRootInInterval function to determine if root does exist in [lower, upper].
3. If the result of isRootInInterval
    a. TRUE
        i. Until convergence occurs
            1. Increment iterations
            2. Set the root approximation to the midpoint of the current interval
               i.e. $x_{approx} = \frac{lower + upper}{2}$
            3. Evaluate the value of the function at the approximation
               $f(x_{approx})$
            4. Is the absolute value of the function at the current approximation
               'close enough to zero i.e. $|f(x_{approx})| < \epsilon$?
                a. YES
                    i. Set loop continuation condition to false.
                b. NO
            i. Does the root lie in [$lower, x_0$]?
                1. Set the upper bound to current approximation $x_0$.
            ii. Does the root lie in [$x_0, upper$]?
                1. Set the lower bound to current approximation $x_0$.
    b. FALSE
        i. Tell the user the root does not lie in the given interval.
        ii. Prompt user to repeat with a valid interval.
        iii. Set the root approximation result and total iterations to the
             predetermined invalid value to indicated method unsuccessful.
4. Return the root approximation and number of iterations.
```

Code A3: `bisectionSearch.m`

```matlab
%%  bisectionSearch.m -% Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   iteratively finds root of a given equation for specified stopping
%   criteria and interval bounds. Returns approximate root and num of
%   iterations taken as result. If root not in interval, returns [NaN, NaN]

function [x_approx, iters ] = bisectionSearch( equation, lower_bound,...
    upper_bound, stop_criteria )
% initialising bounds to prevent arguments changing
lower = lower_bound;
upper = upper_bound;
iters = 0;                 % number of iterations until root found
continueIters = true;      % loop continuation condition

% first, check if root lies in interval
validInterval = isRootInInterval( lower_bound, upper_bound, equation );

if ( validInterval )
    % while continues as long as the flag not changed by the loop
    while ( continueIters == true )
        x_approx = ( lower + upper ) / 2;     % new root = interval mid
        f_approx = equation( x_approx );      % value at new root
        f_lower = equation( lower );          % value at lower bound
        iters = iters + 1;                    % increment iterations

        % if value of the function at root is close enough to zero
        if ( abs( f_approx ) < stop_criteria )
            continueIters = false;            % stop iterating
        end     % end else block to handle loop continuation

        % adjusting bounds for next iteration
        if ( f_approx * f_lower < 0 )    % root in [lower, x_approx]
            upper = x_approx;
        else
            lower = x_approx;            % root in [x_approx, upper]
        end     % end if-else block for adjusting lower and upper bounds
    end     % end while loop
else
    % tell user they need to enter a different interval
    fprintf( '%s\n%s\n%s\n', 'Bisection search failed', 'Root does not lie in this interval.', ...
        'Please try again' );
    % init results to NaN to mark program unsuccessful
    x_approx = NaN; iters = NaN;
    return;
end % end else if-else block to check for valid interval
```

Code A3: bisectionMethod.m

Pseudocode A4: `regulaFalsi.m`

```
1. Initialise all variables
    a. Set lower bound to value passed as argument.
    b. Set upper bound to value passed as argument.
    c. Set number of iterations to zero.
    d. Set loop continuation condition to true.
2. Does the root lie in the given interval?
    a. YES
        i. Repeat Until Convergence
            1. Increment iterations.
            2. Find new approximation $x_0 = \frac{lower \times f(upper) - upper \times f(lower)}{f(upper) - f(lower)}$
            3. Is the current approximation close enough to the actual root i.e. is $|f(x_0)| < \epsilon$?
                a. YES: convergence has occurred.
                b. NO: Adjust interval bounds.
                    i. If root in $[lower, x_0]$, set upper to $x_0$.
                    ii. If root in $[x_0, upper]$, set lower to $x_0$.
    b. NO
        i. Tell user that a root does not exist in the current interval.
        ii. Ask user to repeat with a valid interval.
        iii. Set both $x_0$ and iterations to invalid values to indicate unsuccessful convergence.
3. Return the root approximation $x_0$ and the number of iterations.
```

Code A4: **regulaFalsi.m**

```
%%  regulaFalsi.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   uses Regula-Falsi method to approximate root of a given equation for
%   a given interval and stopping criteria.

function [x_approx, iters] = regulaFalsi( equation, lower_bound,
upper_bound,...
    stop_criteria )
% storing bounds in new variables to prevent argument mutation
lower = lower_bound;
upper = upper_bound;

% initialising iterations to 0
iters = 0;

% check that provided interval contains root
validInterval = isRootInInterval( lower, upper, equation );
continueIters = true;                   % loop continuation condition

% if the user has provided a valid interval, start iterating
if ( validInterval )
    while ( continueIters )
        iters = iters + 1;              % incrementing iterations
        f_lower = equation( lower );    % value at lower bound
        f_upper = equation( upper );    % value at upper bound

        % new root = [ a * f(b) - b * f(a)] / [ f(b) - f(a) ]
        x_approx = ( lower * f_upper - upper * f_lower ) / ...
            ( f_upper - f_lower );
        f_approx = equation( x_approx );        % value at approximation

        % if the stopping criteria has been met
        if ( abs( f_approx ) < stop_criteria )
            continueIters = false;

        % stopping criteria not met? Adjust interval bounds
        else
            if ( f_approx * f_upper > 0 )   % root in [lower, x_approx]
                upper = x_approx;
            else
                lower = x_approx;           % root in [x_approx, upper]
            end     % end if-else block to handle bound adjustment
        end     % end if block to check for convergence
    end     % end while loop to find approximate root

```

```
% if invalid interval, return NaN, NaN and prompt user to repeat
else
    fprintf( '%s\n%s\n', 'Regula-Falsi method failed. Root not in
interval',...
        'Please repeat with a different interval.' );
    x_approx = NaN; iters = NaN;
end     % end if block to check for valid user-specified interval
```

Code A4: regulaFalsi.m

Pseudocode A5: `newtonRaphson.m`

```
1. Initialize all variables
    a. Set number of iterations to 0.
    b. Set initial root approximation $x_0$
    c. Set expression for function derivative $f'(x)$
    d. Set loop continuation condition to true.
2. Repeat until Convergence
    a. Increment iterations.
    b. Evaluate value of function at current root approximation $f(x_0)$.
    c. Evaluate value of derivative at current root approximation $f'(x_0)$.
    d. Calculate new root approximation $x_{new} = x_0 - \frac{f(x_0)}{f\prime(x_0)}$
    e. Is the method diverging i.e. is $|f(x_{new})| > |f(x_0)|$?
        i. YES: Method is diverging. Set loop continuation condition to false.
        ii. Tell user to try again with a different initial point.
        iii. Set root approximation and iterations to invalid values to signal
             unsuccessful convergence.
    f. Is the new approximation is close enough to the actual root i.e. $|f(x_{new})| < \epsilon$
        i. YES: Convergence achieved. Set loop continuation condition to false.
        ii. NO:
            1. Have the maximum number of iterations $n_{max}$ been exceeded?
                a. YES: Could not converge. Set loop continuation condition
                   to false. Set root approximation and iterations to invalid
                   values to signal unsuccessful convergence.
3. Return the root approximation and number of iterations.
```

Code A5: `newtonRaphson.m`

```
%%  newtonRaphson.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   uses Newton-Raphson iterative method to find root of an equation
%   given equation's derivative, a stopping criteria, and initial root
%   returns approximation and number of iterations if root found, else
%   returns [NaN, NaN]

```

```
function [x_approx, iters] = newtonRaphson( equation, stop_criteria, x_0 )
% defining the derivative of the equation
df_x = matlabFunction(diff( sym( equation )));

% Define parameters - initial guess, value of f, value of derivative, iters
iters = 0; x_current = x_0;
continueIters = true;

% continueIters becomes false if stopping criteria met, or divergence
while ( continueIters )
    fx_current = equation( x_current );     % f(x_0)
    df_x_current = df_x( x_current );       % f'(x_0)

    x_approx = x_current - fx_current / df_x_current;

    fx_approx = equation( x_approx );       % value at current approx
    iters = iters + 1;                      % increment iterations

    % if the function is beginning to diverge, stop iterations
    if ( abs( fx_approx ) > abs( fx_current ) )
        fprintf( 'NR Method diverging. Stopping iterations. Try new initial point.\n' );
        x_approx = NaN; iters = NaN;
        return;

    % if not diverging, then could have converged i.e. met criteria
    else
        % if the stopping criteria has been met, stop iterating
        if ( abs( fx_approx ) < stop_criteria )
            continueIters = false;
        else
            % otherwise update x_current for next iteration
            x_current = x_approx;
        end     % end if-else to handle possible convergence
    end     % end if-else to handle to handle divergence-convergence
end     % end while loop
```

Code A5: newtonRaphson.m

Pseudocode A6: `secantMethod.m`

```
1. Initialise all variables
    a. Set lower root approximation to $x_0$
    b. Set upper root approximation to $x_1$
    c. Set number of iterations to 0.
```

```
        d. Set loop continuation condition to true.
2. Repeat until convergence
        a. Increment iterations
        b. Calculate the values of the function at the lower and upper root approximations
           i.e. $f(x_0)$ and $f(x_1)$.
        c. Calculate next root approximation $x_{new} = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$
        d. If the maximum number of iterations has been exceeded
                i. Set loop continuation condition to false.
                ii. Tell user that the method could not converge to a root for the given
                    number of iterations, and that they must repeat with higher iterations or
                    different interval.
                iii. Set the root approximation and number of iterations to invalid values.
        e. If the maximum number of iterations has not been exceeded and the root
           approximation is close enough to the actual root $|f(x_{new})| < \epsilon$
                i. Set loop continuation condition to false.
        f. If the maximum number of iterations has not been exceeded and the root has yet
           to converge
                i. Set first initial root value to second initial root $x_0 = x_1$.
                ii. Set second initial root to current root approximation $x_1 = x_0$.
3. Return the root approximation and the number of iterations.
```

Code A6: `secantMethod.m`

```matlab
%%  secantMethod.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   approximates the root of a given equation to an accuracy defined by
%   the stopping criteria. Assumes user will provide two approximations
%   for the root which need not form an interval containing the root.
%   if root not found until max_iters, returns [NaN, NaN]

function [x_approx, iters] = secantMethod( equation, root_1, root_2,...
    max_iters, stop_criteria )
% creating new variables to store root arguments to prevent mutation
x0 = root_1;
x1 = root_2;
iters = 0;                          % initialising iterations to 0
continueIters = true;               % loop continuation condition

while( continueIters )
    f_x0 = equation( x0 );          % value at previous approximation
    f_x1 = equation( x1 );          % value at current approximation

    % next root approximation
    x_approx = x1 - f_x1 * ( x1 - x0 ) / ( f_x1 - f_x0 );
    f_x_approx = equation( x_approx );
    iters = iters + 1;              % incrementing iterations

    % if function has not converged after maximum iterations
    if ( iters > max_iters )
        continueIters = false;      % change loop continuation condition
        fprintf( '%s\n%s\n%s\n', 'Secant method exceeded max iterations.', ...
            'Please try again with different initial approximations',...
            'or increase max iterations.' );
        x_approx = NaN; iters = NaN;

    % if the function has converged i.e. has met stopping criteria
    elseif ( abs( f_x_approx ) < stop_criteria )
        continueIters = false;

    % if function yet to exceed max_iters and hasn't converged
    else
        % adjusting root values for next iteration
        x0 = x1;
        x1 = x_approx;
    end     % end if-elseif-else block for post-approximation step
end     % end while loop to find root approximation
```

Code A6: secantMethod.m

Appendix B
Regression Models Code

Program Organization: Question 2

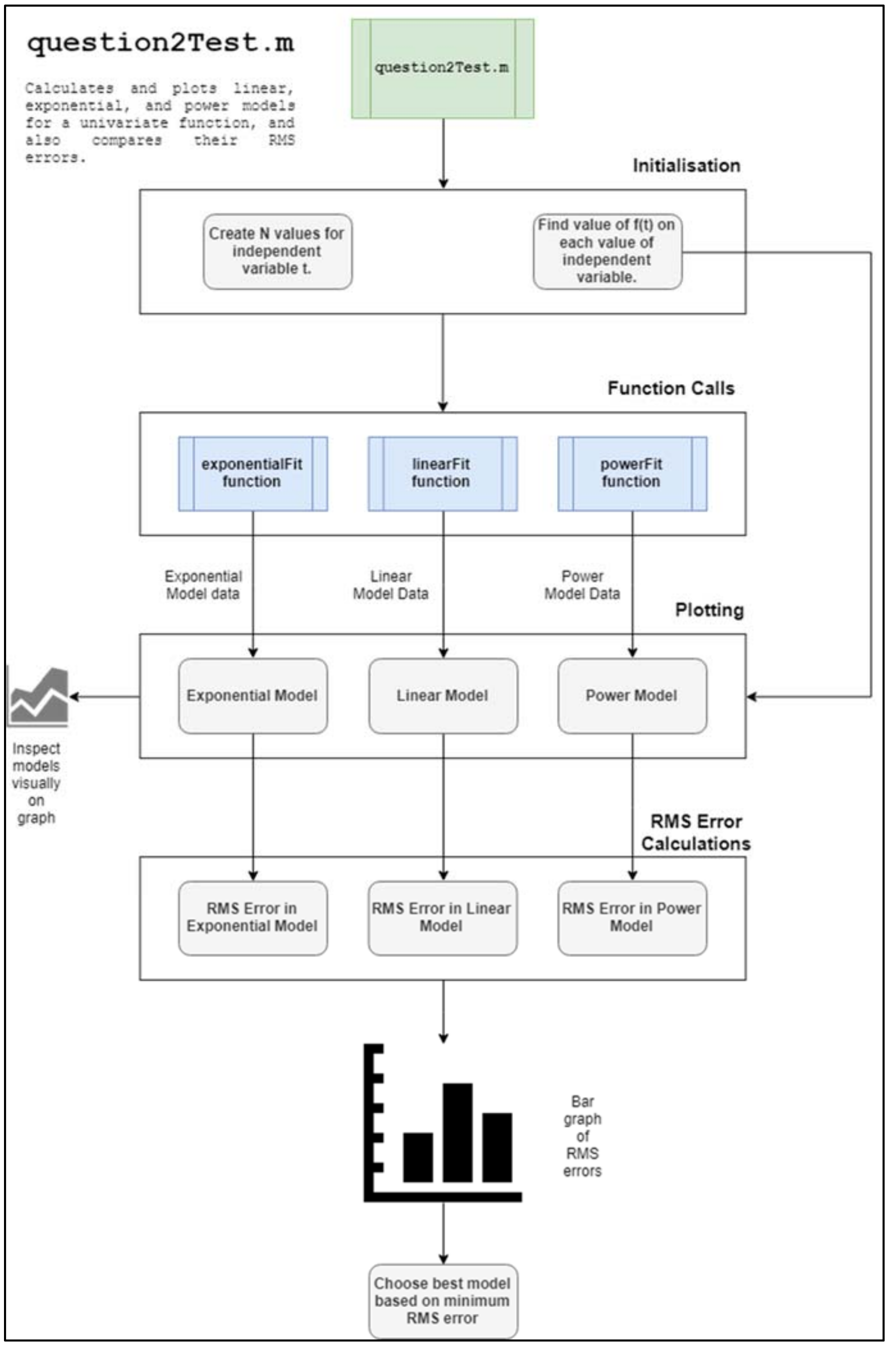
question2Test.m
Calculates and plots linear, exponential, and power models for a univariate function, and also compares their RMS errors.
question2Test.m
Initialisation
Create N values for independent variable t.
Find value of f(t) on each value of independent variable.
Function Calls
exponentialFit function
linearFit function
powerFit function
Exponential Model data
Linear Model Data
Power Model Data
Plotting
Exponential Model
Linear Model
Power Model
Inspect models visually on graph
RMS Error Calculations
RMS Error in Exponential Model
RMS Error in Linear Model
RMS Error in Power Model
Bar graph of RMS errors
Choose best model based on minimum RMS error

Figure 8: Program Organization: Question 2

Pseudocode B1: `question2Test.m`

For tractability, the indexed sum $\sum_{i=0}^{n} p_i$ will be represented by $\sum p$ for any variable p.

```
1. Create domain of $N$ values to represent the independent variable $t$.
2. Evaluate the function at the all values of $t$ to calculate dependent variable values $f(t)$.
3. Call the linear regression method to get the values of constants $a$, $b$ and $f(t)$ in $y_{linear} = ax + b$.
4. Call the exponential regression method to get the values of constants $m, n$, and $f(t)$ in $y_{exponential} = me^{nx}$.
5. Call the power regression method to get the values of constants $m, c$, and $f(t)$ in $y_{power} = mc^{x}$.
6. Plot the data for $f(t)$, $y_{linear}$, $y_{exponential}$, and $y_{power}$ on the same graph to compare regression models visually.
7. Calculate and store root-mean squared errors between $f(t)$ and each of its regression models i.e.
    a. $RMS_{linear} = \sqrt{\frac{\left(f(t) - y_{linear}(t)\right)^2}{N}}$
    b. $RMS_{exponential} = \sqrt{\frac{\left(f(t) - y_{exponential}(t)\right)^2}{N}}$
    c. $RMS_{power} = \sqrt{\frac{\left(f(t) - y_{power}(t)\right)^2}{N}}$
8. Plot a bar graph comparing the RMS errors of all three regression models.
```

Code B1: `question2Test.m`

```matlab
%%  question2Test.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   Implements linear, exponential, and power regression models for a
%   linearly spaced data, plots them, calculates RMS error
%   to find best fit for the data.

%% creating domain of evenly spaced values between 1 and 100
t = linspace( 0, 1, 100 );

%% function to be modelled
f_t = t.^2 - 2 * t + exp( t );

%% storing coefficients and model data for linear, exp, and power fits
[lin_a, lin_b, f_t_linear] = linearFit( t, f_t );
[exp_m, exp_n, f_t_exp] = expFit( t, f_t );
[pwr_m, pwr_c, f_t_pwr] = powerFit( t, f_t );

%% plotting all models on a graph
plot( t, f_t, t, f_t_linear, t, f_t_exp, t, f_t_pwr );
xlabel( 't' ); ylabel( 'f(t)' ); grid;
title( { 'Graphs of f(t) and Regression Models', ...
    '(Linear, Exponential, and Power)'} );

%% calculating RMS errors for each model
rmse_linear = rms( abs ( f_t - f_t_linear ) );
rmse_exp = rms( abs ( f_t - f_t_exp ) );
rmse_pwr = rms( abs( f_t - f_t_pwr ) );

%% plotting RMS error bar graph to determine min error and best model
figure; bar( [ rmse_linear, rmse_exp, rmse_pwr ] );
xlabel( 'Regression Model' ); ylabel( 'RMS Error (\it{\epsilon/arbitrary
units})');
title( 'Regression Models RMS Errors' ); grid;
```

Code B1: question2Test.m

Pseudocode B2: `linearFit.m`

```
1. Initialise the following data
    a. Number of terms $N$
    b. Sum of all values of independent variable $\sum x$
    c. Sum of all values of dependent variable $\sum y$
    d. Sum of squares of all values of independent variable $\sum x^2$
    e. Sum of products of corresponding independent and dependent variables $\sum xy$
2. Calculate constants $a$, $b$ in the linear model $y_{linear} = ax + b$ as follows
    a. Slope $a = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2}$
    b. Intercept $b = \frac{\sum y - a\sum x}{n}$
3. Calculate regression model values by substituting $a, b$ into $y_{linear} = ax + b$.
4. Return constants $a, b$ and $y_{linear}$.
```

Code B2: `linearFit.m`

```
%%  linearFit.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   uses linear regression to derive a linear model for a given data set
%   x is the array of independent variable, y is array of corresponding
%   function values. Returns coefficients a and b in y = ax + b
%   along with the values of the model at each independent value

function [a, b, linearModel]= linearFit( x, y )
n = length ( x );               % number of data points
sum_x = sum( x );               % sum of all values in array x
sum_y = sum( y );               % sum of all values in array y
sum_x2 = sum( x.^2 );           % sum of squares of values in x
sum_xy = sum( x .* y );         % sum of element-wise products of x,y

% slope of linear model
a = ( n * sum_xy - sum_x * sum_y ) / ( n * sum_x2 - sum_x ^ 2 );
```

```

% y-intercept of linear mode
b = ( sum_y - a * sum_x ) / n;

% linear model implemented as y = ax + b
linearModel = a * x + b;
```

Code B2: linearFit.m

Pseudocode B3: expFit.m

```
1. Calculate the natural logarithm of all values of the dependent variable $\ln(y) = Y$.
2. Initialise the following data
    a. Number of terms $N$
    b. Sum of all values of independent variable $\sum x$
    c. Sum of all values of transformed dependent variable $\sum Y$
    d. Sum of squares of all values of independent variable $\sum x^2$
    e. Sum of products of corresponding independent and transformed dependent variables $\sum xY$
3. Calculate the constants $A, B$ in the transformed exponential model $Y = Ax + B$
    a. Slope $A = \frac{N\sum xY - \sum x \sum Y}{N\sum x^2 - (\sum x)^2}$
    b. Intercept $B = \frac{\sum Y - A\sum x}{N}$
4. Use constants $A, B$ to find the equivalent constants in $y_{exponential} = me^{nx}$
    a. $n = A$
    b. $m = e^B$
5. Calculate exponential model values using $y_{exponential} = me^{nx}$.
6. Return $m, n$ and $y_{exponential}$.
```

Code B3: expFit.m

```
%%  expFit.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   implements univariate exponential-based regression model for a
dataset [x,y]
%   x - independent variable values, y - dependent variable
%   m, c, and powerModel are values of coefficients and model evaluated
%   at each instance of independent variable. Model is y = (m)e^(nx)

function [m, n, expModel ] = expFit( x, y )
N = length( x );                        % number of data points

% Using linear law to find linear regression equivalent of exp model
% y = me^(nx) => ln(y) = ln(me^(nx)) => ln(y) = ln(e^(nx)) + ln(m)
% y = nx + ln(m) <=> Y = Ax + B
Y = log( y );                           % linearized dependent variable
values
```

```
sum_x = sum( x );                  % sum of independent values - sigma x
sum_x2 = sum( x.^2 );              % sum of squares of ind values - sigma (x2)
sum_Y = sum( Y );                  % sum of linearized dependent vals - sigma Y
sum_xY = sum( x .* Y );            % sum of products of x and Y

% Finding slope in linearized model - no reverse transform needed
n = ( N * sum_xY - sum_x * sum_Y ) / ( N * sum_x2 - sum_x^2 );

% Finding Y intercept in linearized model - ln(m) = B => m = exp(B)
B = ( sum_Y - n * sum_x ) / N;
m = exp( B );

% Initializing exponential model m.e^(nx)
expModel = m * exp( n * x );
```

Code B3: expFit.m

Pseudocode B4: **powerFit.m**

```
1. Calculate the natural logarithm of all values of the dependent variable $\ln(y) = Y$.
2. Initialise the following data
    a. Number of terms $N$
    b. Sum of all values of independent variable $\sum x$
    c. Sum of all values of transformed dependent variable $\sum Y$
    d. Sum of squares of all values of independent variable $\sum x^2$
    e. Sum of products of corresponding independent and transformed dependent variables $\sum xY$
3. Calculate the constants $A, B$ in the transformed exponential model $Y = Ax + B$
    a. Slope $A = \frac{N\sum xY - \sum x \sum Y}{N\sum x^2 - (\sum x)^2}$
    b. Intercept $B = \frac{\sum Y - A\sum x}{N}$
4. Use constants $A, B$ to find the equivalent constants in $y_{exponential} = mc^x$
    a. $c = e^A$
    b. $m = e^B$
5. Calculate exponential model values using $y_{power} = mc^x$.
6. Return $m, n$ and $y_{power}$.
```

Code B4: **powerFit.m**

```
%%  powerFit.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   implements univariate exponential-based regression model for a dataset [x,y]
%   x - independent variable values, y - dependent variable
%   m, c are constants in power model y = m.c^(x) and powerModel is the
%   value of the regression model evaluated at each indpendent variable
```

```matlab

function [ m, c, powerModel ]= powerFit( x, y )
N = length( x );                % number of data points

% Using linear law to find linear regression equivalent of exp model
% y = m.c^(x) => ln(y) = ln(m.c^(x)) => ln(y) = ln(c^(x)) + ln(m)
% => ln(y) = ln(c)(x) + ln(m) <=> Y = Ax + B
Y = log( y );                   % linearized dependent variable values
sum_x = sum( x );               % sum of independent values - sigma x
sum_Y = sum( Y );               % sum of linearized dependent vals - sigma Y
sum_x2 = sum( x.^2 );           % sum of squares of ind values - sigma (x2)
sum_xY = sum( x .* Y );         % sum of products of x and Y

% Finding slope in linearized model - since A = ln(c), c = exp( A );
A = ( N * sum_xY - sum_x * sum_Y ) / ( N * sum_x2 - sum_x ^ 2 );
c = exp( A );                   % n =

% Finding y interept in linearized model - since B = ln(m), m = exp( B );
B = ( sum_Y - A * sum_x ) / N;
m = exp( B );

% Power model is implemented as
powerModel = m * c .^ x;
```

Code B4: powerFit.m

Appendix C
Gauss-Seidel Method Code

Program Organization: Question 3

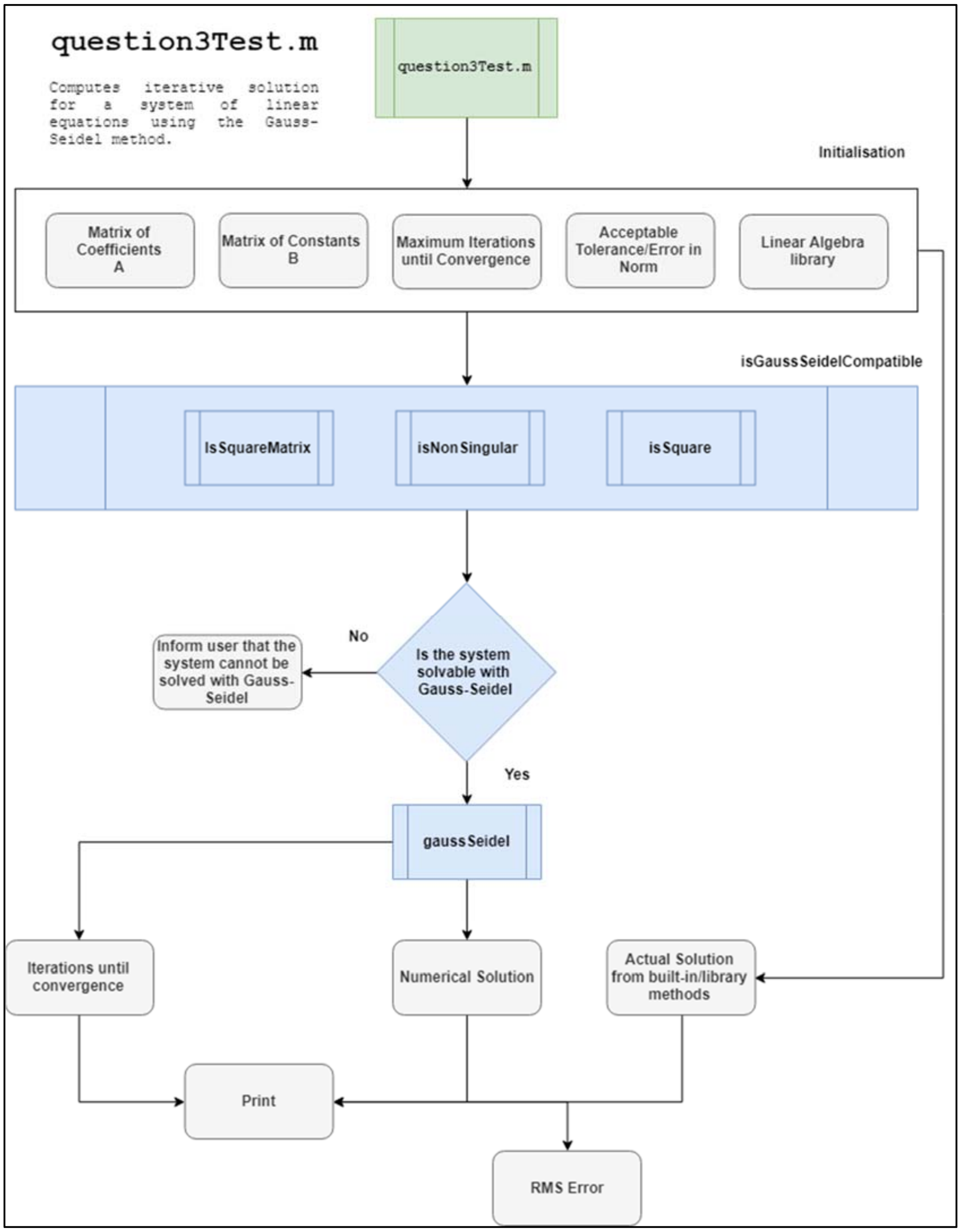
question3Test.m
Computes iterative solution for a system of linear equations using the Gauss-Seidel method.
question3Test.m
Initialisation
Matrix of Coefficients A
Matrix of Constants B
Maximum Iterations until Convergence
Acceptable Tolerance/Error in Norm
Linear Algebra library
isGaussSeidelCompatible
IsSquareMatrix
isNonSingular
isSquare
Is the system solvable with Gauss-Seidel
No
Inform user that the system cannot be solved with Gauss-Seidel
Yes
gaussSeidel
Iterations until convergence
Numerical Solution
Actual Solution from built-in/library methods
Print
RMS Error

Figure 9: Program Organization: Question 3

Pseudocode C1: `question3Test.m`

```
1. Initialise coefficient matrix for system of equations $A$.
2. Initialise constants matrix for system of equations B.
3. Define the tolerance/error/maximum value $\epsilon$ of norm for $AX - B$.
4. Define maximum number of iterations for iterative solution $n_{max}$.
5. Compute the actual solution $X_{actual}$ of the system of equations using built-in tools.
6. Can the system of equations be solved using the Gauss-Seidel method?
    a. YES
        i. Inform the user that the program will attempt to use the Gauss-Seidel method to solve the system iteratively.
        ii. Call GaussSeidel method to return the solutions matrix $X_{Gauss-Seidel}$ and the number of iterations $n_{Gauss-Seidel}$ taken until convergence.
        iii. Echo $X_{actual}$.
        iv. Echo $X_{Gauss-Seidel}$ to visually inspect similarity (or lack thereof) to $X_{actual}$.
        v. Echo the number of iterations taken until convergence $n_{Gauss-Seidel}$.
        vi. Echo the RMS error between the $X_{actual}$ and $X_{Gauss-Seidel}$.
    b. NO
        i. Inform user that the given system cannot be solved using the Gauss-Seidel method.
```

Code C1: `question3Test.m`

```matlab
%%  question3Test.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   finds solution of a n x n system of equations using
%   Gauss-Seidel iterative method. Tested with a 4 x 4 matrix

%% Defining parameters for test system of equations - Ax = B
A = [ 6, -2, 1, 1;          % coefficient matrix
      1, -7, 2, 2;
     -1,  2, 8, 4;
      2,  1, 4, 9 ];
B = [ 10; 11; 12; 13 ];     % constants matrix
err = 1e-10;                % stopping criteria
max_iters = 100;            % in case it doesn't converge
x_actual = A \ B;           % MATLAB-derived solution for comparison

%% Only proceed if the coefficient matrix meets all criteria
if ( isGaussSeidelCompatible( A ) )
    % inform user that system can be solved with Gauss-Seidel method
    disp( 'System is Gauss-Seidel compatible.' );

    % Returns the matrix of solutions and iterations taken
    [ x_gauss_seidel, iterations ] = gaussSeidel( A, B, err );

    % echo MATLAB-derived solution
    fprintf( 'The actual solution is \n' );
    disp( x_actual );

    % echo iterative solution for verification
    fprintf( '\nThe Gauss-Seidel iterative solution is\n' );
    disp( x_gauss_seidel );

    % echo iterations taken and current error/tolerance level
    fprintf( '\nFor error level %s, this solution took %d\n
iterations.\n',...
        num2str(err), iterations );

    % echo RMS error between MATLAB and Gauss-Seidel solutions
    fprintf( 'RMS Error between the two is %s\n', ...
        num2str(rms(abs(x_gauss_seidel - x_actual))));
        %num2str( rms( ( x_gauss_seidel - x_actual ) ) ) );
else
    % tell user this system can't be solved with GS method
    fprintf( 'System cannot be solved with Gauss-Seidel Method.\n');
end
```

Code C1: question3Test.m

Pseudocode C2: `isGaussSeidelCompatible.m`

```
1. If the coefficient matrix $A$ of the system of equations is square, non-singular, and diagonally dominant?
    a. YES: return true.
    b. NO: return false.
```

Code C2: `isGaussSeidelCompatible.m`

```
%%  isGaussSeidelCompatible.m - Saad Siddiqui, EE-16163, Section D, TE-
EE Fall '18
%   determines if argument matrix fulfills all three conditions for
GaussSeidel compatibility

function gs_compatible = isGaussSeidelCompatible( coeff_mat )
% Is the matrix square? If not square, can't be solved using GS
is_square = isSquareMatrix( coeff_mat );

% Is the matrix singular? If singular, A^-1 doesn't exist. No solution
sol_exists = isCoeffNonSingular( coeff_mat );

% Is the matrix diagonally dominant?
is_diag_dom = isDiagDominant( coeff_mat );
```

```matlab

% If all three conditions fulfilled, can solve with Gauss-Seidel
if ( is_square && sol_exists && is_diag_dom )
    gs_compatible = true;
else
    gs_compatible = false;
end
```

Code C2: isGaussSeidelCompatible.m

Pseudocode C3: `isCoeffSingular.m`

```
1. Is the determinant of the matrix 0?
    a. YES: return true
    b. NO: return false
```

Code C3: `isCoeffSingular.m`

```matlab
%%  isCoeffSingular.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   determines whether a given matrix is singular

function coeffNonSingular = isCoeffNonSingular( coeff_matrix )
if ( det( coeff_matrix ) == 0 )
    coeffNonSingular = false;
    fprintf( 'Matrix is singular\n' );
else
    coeffNonSingular = true;
end
```

Code C3: isCoeffSingular.m

Pseudocode C4: isDiagDominant.m

```
1. Initialise diagonally dominant flag variable to true (assuming is diagonally dominant).
2. For all rows in the coefficient matrix
    a. Find the absolute value of the diagonal element $a_{ii}$.
    b. Find the sum of absolute values of all elements in each row of $a_{11} + a_{12} + a_{13} + \cdots + a_{1n}$.
    c. Do the sums for this row satisfy $2|a_{ii}| > |a_{11} + a_{12} + a_{13} + \cdots + a_{1n}|$ (another way of expressing condition for diagonal dominance).
        i. NO: set diagonally dominant flag to false.
3. Return diagonally dominant flag.
```

Code C4: `isDiagDominant.m`

```matlab
%%  isDiagDominant.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   determines whether a given matrix is diagonally dominant

```

```
function diagDominant = isDiagDominant( coeff_mat )
if ( all((2*abs(diag(coeff_mat))) >= sum(abs(coeff_mat),2)) )
    diagDominant = true;
else
    diagDominant = false;
    fprint( 'Matrix is not diagonally dominant.\n' );
end
```

Code C4: isDiagDominant.m

Pseudocode C5: `isSquareMatrix.m`

```
1. Get the number of rows of the coefficient matrix $r$.
2. Get the number of columns of the coefficient matrix $c$.
3. Are neither $r$ nor $c$ are equal to 0 and are both equal to each other?
    a. YES: return true.
    b. NO: return false.
```

Code C5: `isSquareMatrix.m`

```
%%  isSquareMatrix.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall
'18
%   determines whether a given matrix is square matrix

function squareFlag = isSquareMatrix( coeff_mat )
[ n_row, n_col ] = size( coeff_mat );

% if both rows and cols equal and neither is 0 i.e. not null matrix
if ( n_row == n_col && n_row ~= 0 )
    squareFlag = true;
else
    squareFlag = false;
    fprintf( 'Coefficient Matrix is not square.\n' );
end
```

Code C5: isSquareMatrix.m

Pseudocode C6: `gaussSeidel.m`

```
1. Initialise the following variables
    a. number of unknowns in the system of equations $n$.
    b. an initial approximation $X_0$ of n zeroes.
    c. a loop continuation condition set to true.
    d. number of iterations $k$
2. Repeat until convergence
    a. Increment iterations
    b. For every row in the coefficient matrix
        i. Initialise the row $sum$ to 0.
```

```
            ii. For every column in the current row of the coefficient matrix
                1. Compute the sum $\sum_{j=0, j\neq i}^{n} a_{ij}.x_j$ i.e. the sum of all coefficients and
                   corresponding values of unknowns except for the diagonal element
                   for the current row.
            iii. Set the current unknown $x_i$ to $\frac{b_j - \sum_{j=0, j\neq i}^{n} a_{ij}.x_j}{a_{ii}}$
        c. Calculate the norm of the matrix $AX_0 - B$.
        d. Is the norm of the matrix close enough to 0 i.e. is $|AX_0 - B| < \epsilon$?
            i. YES: Convergence has occurred. Set loop continuation condition to false.
3. Return the approximated solution set $X_0$ and the number of iterations.
```

Code C6: `gaussSeidel.m`

```matlab
%%  gaussSeidel.m - Saad Siddiqui, EE-16163, Section D, TE-EE Fall '18
%   uses Gauss-Seidel method to find solution of (n x n) linear system
%   of equations AX = B, given coefficient matrix coeff_mat,
%   constants matrix const_mat, and a stopping criteria stop_criteria.
%   Assumes coeff_mat is non-singular, diagonally dominant
%   and solution to AX = B exists.

function [ X, iters ] = gaussSeidel( coeff_mat, const_mat, stop_criteria )
% A (n x n) system will have n x 1 solution matrix
% Since nonsingular, trivial solution must exist so init initial guess
% to a matrix of zeros and same dimensions as coeff_mat
size_coeff = size( coeff_mat );          % [ n, n ]
n = size_coeff( 1, 1 );                  % will determine num of iters
X = zeros( n, 1 );                       % [ n, 1 ]
continue_iters = true;                   % loop continuation condition
iters = 0;                               % keeping track of iterations

% until iterative solution not close enough to satisfying Ax = B
while ( continue_iters )
    iters = iters + 1;    % record additional iteration
    for i = 1 : n
        sum = 0;          % initialise to zero at beginning of each iter

        % summing all terms before the diagonal element
        for j = 1 : i - 1
            sum = sum + coeff_mat( i, j ) * X( j );
        end     % end inner for loop 1 (j)

        % summing all terms after the diagonal element
        for j = i + 1 : n
            sum = sum + coeff_mat( i, j ) * X( j );
        end     % end inner for loop 2 (j)

        % the ith root for the current iteration
        X( i ) = ( const_mat( i ) - sum ) / coeff_mat( i, i ) ;
    end     % end outer for loop (i)

    % checking if current solution is close enough to actual solution
    % Stop criteria expressed as norm of vector Ax - B because if
    % x satisfies Ax = B, it must also satisfy Ax - B ~ 0
    % implies the norm of Ax - B should be ~ 0.
    if ( norm( coeff_mat * X - const_mat ) < stop_criteria )
        continue_iters = false;
    end     % end if block to handle loop continuation s
end     % end while loop
```

Code C6: gaussSeidel.m