Universidad Ana G. Mendez Recinto de Gurabo

School of Engineering

Gurabo, Puerto Rico

# Numerical Methods – Project 1

Carlos A. Roque Fontanez

S01299757

COMP-411

Dr. Yahya Masalmah

November 6th, 2023

## Table of Contents

## Introduction

In the quest to mimic the precision of an experienced right fielder's throw or to optimize fluid flow within pipes, we as emerging engineers are tasked with numerical challenges that not only demand accuracy but also an understanding of practical applications in our field. This project takes us on a journey through the application of computational methods to analyze trajectories and forces in a real-world context. First, we calculate the ideal initial angle for a baseball throw, given a set of initial conditions, ensuring the ball's path from the fielder's hand to the catcher is both precise and practical. Next, we delve into the electrostatic force exerted by a charged ring on a point charge, a problem that tests our understanding of electric field distributions. Finally, we tackle the complexities of fluid dynamics by computing the Fanning friction factor for different Reynolds numbers, using the von Karman equation. Through the use of various numerical methods—ranging from Bisection to Muller's method—we not only derive solutions but also evaluate the methods themselves for their efficiency and accuracy, ensuring our calculations are within a tight error threshold of 0.01%. This report details our approach, findings, and insights gained from employing these methods in solving each problem, highlighting the synergy between theoretical knowledge and its practical application in engineering.

## Project Structure

### Main Functions for Numerical Solutions of Equations

This project was coded in Octave with the following functionality:

`menu_options` - The script functions as an interactive tool that repeatedly presents a menu to the user, allowing them to choose from various numerical methods for root-finding. Upon selection, the user is prompted to enter necessary parameters and the chosen algorithm proceeds to calculate an approximation of the root of a mathematical function. The core of the script is designed to facilitate the application of different algorithms like Bisection and Newton's method, each utilizing its own iterative procedure and convergence criteria. The process continues in a loop, providing results for each method until the user opts to exit, ensuring a user-driven experience without restarting the script.

` bisection_method_project1` - This script implements the bisection method, a numerical technique for finding roots of a continuous function. The method requires an initial bracket, given by xl and xu, within which the root is believed to lie, and iteratively narrows down this bracket to pinpoint the root's location. The user supplies the function for which the root is sought, an acceptable error tolerance es, and a maximum number of iterations `maxit`.

At the start, the function checks whether the function values at the initial bracket boundaries have opposite signs—a necessary condition for the presence of a root. If this condition isn't met, it halts with an error.

The iterative process begins, calculating the midpoint of the current bracket and evaluating the function at that point. If the function value is zero or the error tolerance is met (the relative difference between the current and previous midpoints is less than es), the function concludes the root has been found. Otherwise, it decides which half of the bracket contains the root by checking the sign of the function at the midpoint, then repeats the process with the narrowed bracket.

The algorithm continues until the error tolerance is satisfied or the maximum number of iterations is reached. It records the midpoint estimates and the approximate relative errors at each step, returning these lists as outputs—`xr` for the root estimates and `ea` for the errors. If the maximum number of iterations is reached without meeting the error tolerance, it notifies the user that the root was not found to the desired precision.

` false_position_method_project1` - This scripted function applies the false position (or "regula falsi") method to find a root of a given function. This method is an improvement over the bisection method in that it potentially converges faster. It uses a linear interpolation to approximate the root, taking into account the function values at the bracketing interval.

The user provides an initial guess for the lower (xl) and upper (xu) bounds of the interval, the function (`func`) for which the root is sought, an allowable tolerance (es), and a maximum number of iterations (`maxit`).

The function starts by evaluating the function at the initial guesses to ensure that they bracket a root (i.e., the function values have different signs). If not, an error is displayed and execution stops.

For each iteration up to `maxit`, the method calculates an approximation of the root (xi) by interpolating the line between the points (xl, f(xl)) and (xu, f(xu)) and finding where it crosses the x-axis. After each new approximation, it updates the error estimate (`ea`) and checks if the function value at the new approximation is close enough to zero or if the approximation is within the desired tolerance. If so, it stops and declares convergence.

If the function value at the new approximation has a different sign from the value at the lower bound, it replaces the upper bound with the approximation (and vice versa). This maintains the bracket around the root.

When the loop ends, either from finding a satisfactory root or reaching the maximum number of iterations, it returns the last approximation of the root (`xr`) and the corresponding error estimate (`ea`). If the loop finishes because the maximum number of iterations was reached without satisfying the error tolerance, it indicates that the root has not been located within the desired accuracy.

`fixed_point_iteration_project1` - The scripted function implements the Fixed-Point Iteration method, used to approximate solutions of the equation f(x) = 0. This method assumes that the equation can be rewritten in the form x = g(x), so the root finding problem becomes an iteration process starting from an initial guess x_0.

The user supplies the initial guess x_0, the function f (passed as a string f_func_str) which needs to be solved, a corresponding function g (passed as a string g_func_str) used for iterations, a tolerance level e_a, and a maximum number of iterations i.

The function initializes the iteration counter and an empty list to keep track of all approximate root values (x_vals) and the corresponding error estimates (ea_vals). For each iteration, it applies the iteration function g to the current approximation, checks the value of the function f at this new point to see if it is sufficiently close to zero (within the tolerance), and records the error. If the stopping criteria are met—either the function value is within the error tolerance or the maximum number of iterations is reached—the process halts.

At each step, it calculates the relative error based on the current and previous approximations and stores it. This process repeats until either the function value is less than the error tolerance (indicating that the root is found) or the maximum number of iterations is reached without achieving the required tolerance. The function outputs the final approximation and its corresponding error, along with the number of iterations taken to reach the result.

`newton_method_project1` - This function encapsulates Newton's method, a classic root-finding algorithm leveraging calculus to iteratively converge to a solution of f(x) = 0. It requires an initial approximation `x_0`, and the function `func_handle` alongside its derivative `func_prime_handle`.

At each iteration, the method assesses the current guess by applying the Newton formula: a new guess `x` is computed as the previous `x` minus the quotient of the function value at `x` and the derivative at `x`.

This process is repeated until the change in successive estimates is less than the specified tolerance `e_a`, or until `i` iterations are completed. The vectors `x_vals` and `ea_vals` capture the succession of approximations and their corresponding errors, providing insight into the convergence behavior of the method.

`secant_method_project1` - This function implements the secant method, an iterative root-finding technique that approximates solutions to the equation f(x) = 0. Unlike Newton's method, the secant method does not require the calculation of derivatives. It starts with two initial approximations, `x0` and `x1`, and iteratively produces new estimates using the values of the function `func_str` at these points.

In each iteration, the method uses the previous two estimates to compute a new approximation of the root. This is achieved by calculating the secant line through the points (x0, f(x0)) and (x1, f(x1)) and then finding the x-intercept of this line.

The process continues until the absolute value of the function at the latest estimate is less than the specified tolerance `eps`, or until a maximum number of iterations `max_iter` has been reached. The vectors `x_vals` and `ea_vals` are updated at each iteration with the latest root estimate and its corresponding relative error, respectively.

If the solution is found within the specified tolerance before the maximum number of iterations, the function prints the solution and the number of function calls made. If the process exceeds `max_iter` without finding an adequate solution, it aborts execution and indicates that a solution was not found.

`modified_secant_method_project1` - This function is a scripted implementation of the Modified Secant method, a numerical technique used for finding roots of a function f(x) = 0. The method is a variation of the Secant method, introducing a small perturbation `delta` to the current estimate `x0` to avoid the need for two initial approximations.

The primary procedure `modified_secant` takes a function handle `func_handle`, an initial approximation `x0`, a perturbation factor `delta`, a tolerance `eps`, and a maximum number of iterations `max_iter`. It starts the root-finding process, iterating to refine the estimate of the root until the absolute value of the function at the current estimate is less than the tolerance, or the maximum number of iterations is reached.

The iterative process consists of generating a perturbed point `x1` from the current estimate `x0`, evaluating the function at both `x0` and `x1`, and then using these values to calculate a new estimate. This new estimate is obtained by taking the current estimate `x0` and adjusting it by a factor related to the function values and the perturbation.

Throughout the iterations, the function maintains arrays `x_vals` and `ea_vals` to record the sequence of estimates and their corresponding relative errors, respectively. After each iteration, it updates these arrays with the new estimate and its relative error.

Upon completion of the iteration process, if a solution is found within the specified tolerance, the function prints the number of function calls and the found solution. If the function fails to find a solution within the tolerance after the maximum number of iterations, it prints a message indicating that execution is aborted.

If the function terminates due to an error (such as a zero denominator), it immediately stops execution and prints an error message. If the final function value is above the specified tolerance, the `no_iterations` is set to -1 to indicate that the desired accuracy was not achieved.

`fzero_method_project1` - This function provides a MATLAB interface to the built-in `fzero` function, which is designed to find the roots of a nonlinear equation defined by `f_func`. The user supplies an initial guess `x0`, a tolerance `tol`, and a maximum number of iterations `max_iter`.

The function sets up an options structure `options` that specifies the tolerance for the root and the maximum number of iterations the `fzero` function will execute. An output function `@outfun` is also defined and passed to `fzero` via the options structure. This output function is called at each iteration of the root-finding process.

Inside the `outfun`, the relative approximate error `ea` is computed, provided the current estimate `x` is not zero, to avoid division by zero. The value of `ea` is then stored in the array `ea_vals`, and the current estimate `x` is stored in `x_vals` for output. The `x_old` variable is updated with the current estimate `x` for use in the next iteration's error calculation.

After running the `fzero` function with the provided options, the root is stored in the variable `root`, and the total number of iterations taken to find the root is recorded in `iter`.

The function concludes by printing out the found root and the number of iterations required to reach it to the MATLAB Command Window. These print statements are currently uncommented and will execute when the function is called.

By capturing the sequence of estimates and their corresponding errors in `x_vals` and `ea_vals`, this function not only finds the root but also provides insight into the convergence behavior of the `fzero` method.

`muller_project1` - This function implements the Muller method for finding roots of a given function. It is a root-finding algorithm that uses quadratic interpolation to approximate the roots of a real-valued function. The function accepts an initial set of points `p0`, `p1`, `p2`, the function for which the root is sought `func`, a tolerance `TOL`, and a maximum number of iterations `N0`.

The method begins by calculating the differences `h1` and `h2` between consecutive initial points, as well as the divided differences `DELTA1` and `DELTA2`. These are used to compute the coefficient `d` of the quadratic interpolating polynomial.

The algorithm proceeds iteratively, updating the value of `p` (the current estimate of the root) using Muller's method, which involves solving a quadratic equation derived from the interpolating polynomial. The discriminant `D` is calculated, ensuring that the correct sign is chosen to avoid cancellation errors when computing the new estimate `h`.

The estimated root `p` and the absolute error `abs(h)` are then stored in the arrays `estimated_roots` and `approx_errors` respectively. If the absolute error is less than the specified tolerance `TOL`, the function displays the estimated root and terminates.

If the tolerance condition is not met, the method updates the points `p0`, `p1`, and `p2` for the next iteration and recalculates the necessary differences and divided differences. This process repeats until the tolerance is met or the maximum number of iterations `N0` is reached.

Should the method exceed `N0` iterations without finding a root within the desired tolerance, it prints a message indicating failure to converge within the specified number of iterations.

The function uses the `format long;` statement at the beginning to ensure that results are displayed with maximum precision, which is particularly useful for observing the behavior of iterative root-finding methods.

## Other Helper Functions

These functions were created for the purpose of aiding the user inserting necessary values for each method in the menu without having complications:

`g_of_problem1_equation` - This function calculates and returns an angle `theta_0` such that a projectile launched with that angle from an initial height `y_0` with an initial velocity `v_0` will reach a target at a horizontal distance `x` and a final height `y`. The calculation is based on the projectile motion equations under uniform gravity `g`:

- It defines the gravitational acceleration `g`, the initial launch velocity `v_0`, the horizontal distance to the target `x`, the initial height `y_0`, and the target height `y`.
- Using these parameters, the function calculates the angle `theta_0` using the inverse tangent function `atan`. This is derived from the kinematic equations of projectile motion, taking into account the initial and final vertical positions of the projectile, the distance traveled, the effect of gravity, and the initial velocity.
- The term `(g * x^2) / (2 * v_0^2 * (cos(theta_0))^2)` represents the vertical displacement due to gravity when the projectile has traveled a horizontal distance `x`.
- The angle `theta_0` is recalculated by rearranging the projectile motion equation to solve for `theta_0`.

`g_of_problem2_equation` - This function is used to find a new value `x_new` which is used in a context of fixed-point iteration for an electrical force problem. The function uses constants such as the permittivity of free space `e_0`, the charges `q` and `Q`, the radius of a ring `a`, and a target force `F_target`:

- The permittivity of free space `e_0` is a fundamental physical constant used to calculate electric forces in vacuum.
- The charges `q` and `Q` are the magnitudes of two point charges or distributed charges involved in the problem.
- The constant `a` represents the radius of a ring, suggesting a charge distribution along a circular path.
- `F_target` is the target electric force that we are trying to achieve.

The equation for `x_new` is derived from Coulomb's law, adjusted for the geometry of the problem which seems to involve a point charge `q` at a distance `x` from the center of a charged ring with a total charge `Q` and radius `a`. The force `F_target` is set to be the desired force at this point.

The expression `(x^2 + a^2)^(3/2)` is part of the formula that calculates the electric field due to a ring of charge at a point along the axis passing through the center of the ring, and `x_new` is calculated such that the electric force `F_target` would be produced at that point.

In the context of fixed-point iteration, this function is likely called repeatedly with an initial guess for `x`, and the output `x_new` is then used as the new input for the next iteration, until the value converges to a stable point where `x` does not change significantly between iterations.

`g_of_problem3_equation` - This function generates an anonymous function `fhandle` which calculates the Colebrook equation for fluid flow in a pipe. This equation is commonly used to estimate the friction factor `f` for turbulent flow in a circular pipe based on the Reynolds number `Re`:

- It takes the Reynolds number `Re` as an input, which is a dimensionless quantity in fluid mechanics used to predict flow patterns in different fluid flow situations.
- The function then returns an anonymous function `fhandle`, which calculates the inverse of the square of the friction factor `f`. This is based on an implicit equation from the Colebrook-White formula, which is typically solved using numerical methods such as the Newton-Raphson method or Fixed-Point Iteration.

`problem1_d_equation` - This function calculates the derivative of the projectile's motion equation with respect to the launch angle `theta_0`. It is used to find the angle that would make a projectile hit a certain target point, given initial conditions:

- It defines `g` as the gravitational acceleration, `v_0` as the initial velocity of the projectile, and `x` as the horizontal distance to the target.
- The function `dy` computes the derivative of the projectile's trajectory equation with respect to `theta_0`, which is essential for methods like Newton-Raphson that need the derivative to find the root of an equation.

`problem1_equation` - This MATLAB function defines the equation of the trajectory of a projectile based on the angle of launch (`theta_0`). It determines the vertical position (`y`) of a projectile at a certain horizontal distance (`x`), considering initial velocity (`v_0`), gravitational acceleration (`g`), and initial height (`y_0`):

1. The term tan(theta_0) * x calculates the linear increase in height based on the angle theta_0.
2. The term -(g * x^2) / (2 * v_0^2 * (cos(theta_0))^2) represents the parabolic arc due to gravity's influence on the projectile.
3. y_0 - 1 adjusts the equation based on the initial height from which the projectile is launched, minus 1 meter from the target height to account for some specific condition or requirement in the problem.

`problem2_d_equation` - Derivative of the electrostatic force between a point charge and a charged ring.

4. Constants `e_0`, `q`, `Q`, and `a` are defined for the calculation.
5. `numerator` computes the difference in forces adjusted for distance `x`.
6. `denominator` normalizes the force over the cube of the distance plus the square of the ring radius.
7. `F_diff_prime` gives the rate of change of force with respect to `x`.

`problem2_equation` - This function calculates the difference between the electrostatic force at a point `x` from the center of a charged ring and a target force value.

8. Utilizes the constant for vacuum permittivity `8.9e-12` $C^2/(Nm^2)$.
9. Charge values `2e-5` C for both the point charge and the ring are used.
10. The radius `a` of the ring is 0.85 meters.
11. `calc` computes the electrostatic force at distance `x` from the ring's center and subtracts the target force of `1.25` Newtons.

`problem2_possible_root ` - This MATLAB script is designed to identify intervals where the root of the `problem2_equation` may exist by checking for sign changes in the function values over a specified range of `x`.

12. `problem2_equation` is an anonymous function that calculates the difference between electrostatic force at a point and a target force value.
13. `x_vals` is an array of 1000 linearly spaced points between 0 and 2.
14. `F_diff_vals` stores the computed values of `problem2_equation` for each `x` in `x_vals`.
15. The script iterates over `F_diff_vals` to check for sign changes, indicating potential brackets around roots.
16. When a sign change is detected, it reports the interval as a possible bracket where a root can be found.

The `problem3_d_equation` function takes in the Reynolds number (`Re`) and returns an anonymous function `dfhandle`, which represents the derivative of the von Karman equation with respect to the friction factor `f`:

17. The `derivative_of_problem3_equation` is a nested function where the actual derivative calculation is supposed to be implemented based on the von Karman equation for fluid flow in pipes.
18. The placeholder derivative shown in the function `derivative_of_problem3_equation` is just an illustrative example and is not the correct derivative of the von Karman equation.
19. Users must replace the placeholder derivative with the correct derivative expression derived from the von Karman equation.

`problem3_equation` function is designed to return an anonymous function `fhandle` that captures the Reynolds number `Re` for subsequent evaluations:

20. The inner function `problem3_equation_internal` calculates the von Karman equation for pipe flow, which is used to find the friction factor `f` given a Reynolds number `Re`.
21. The left-hand side (`lhs`) of the von Karman equation is `1 / sqrt(f)`, and the right-hand side (`rhs`) is `4 * log10(Re * sqrt(f)) - 0.4`.
22. The difference between the left-hand side and right-hand side (`residual`) represents the discrepancy from the root, which the numerical methods aim to minimize or eliminate to find an accurate value of `f`.

`problem3_possible_root` - The script visualizes the von Karman equation for a range of Reynolds numbers (`Re_values`) by plotting the equation values against the friction factor `f`:

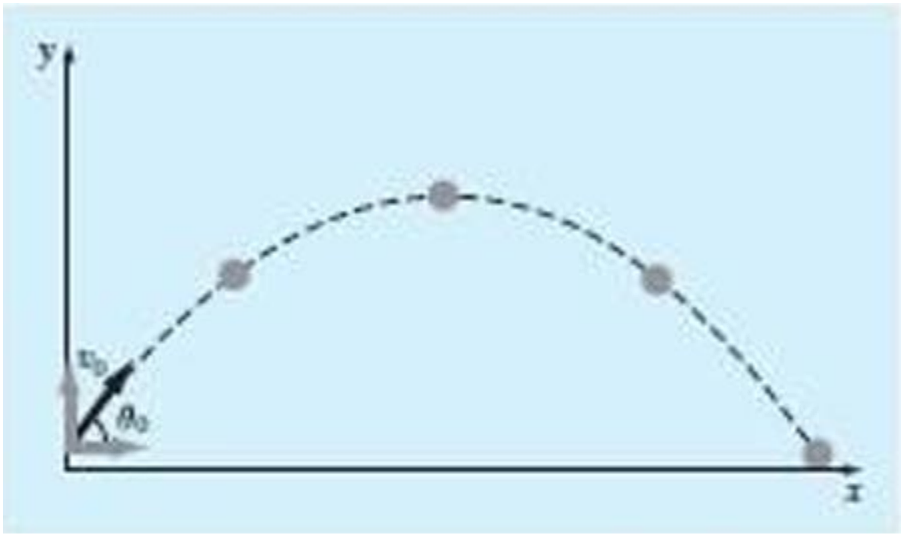
23. `Re_values` are created with a `linspace` function, generating 10 values evenly spaced between 10,000 and 500,000.
24. The plot is initiated with `hold on` to allow multiple plots on the same figure.
25. A for-loop iterates over the `Re_values`, creating an anonymous function `colebrook_eq` for each `Re`, which represents the von Karman equation.
26. `f_vals` is a linear space of 1000 points between 0.001 and 0.01, representing possible friction factor values.
27. `arrayfun` is used to evaluate `colebrook_eq` over the range of `f_vals`, and the results are plotted with a label for each `Re`.
28. Inside the loop, another loop checks for sign changes in consecutive `colebrook_vals`, which indicate potential brackets where the root of the equation might be found, and prints the bracket range.
29. The plot is finalized with a grid, labeled axes, a title, and a legend showing different `Re` values.
30. `hold off` is used to finalize the plot, indicating that no more elements will be added.

## Methodology

### Problem 2

Aerospace engineers sometimes compute the trajectories of projectiles such as rockets. A related problem deals with the trajectory of a thrown ball. The trajectory of a ball thrown by a right fielder is defined by the (x, y) coordinates as displayed in Fig. 1. The trajectory can be model

$$y = \tan(\theta_0) \times x - \frac{g \times x^2}{2 \times v_0^2 \times \cos^2(\theta_0)} + y_0$$



a) Find the appropriate initial angle $\theta_0$, if v0 = 30 m/s, and the distance to the catcher is 90 m. Note that the throw leaves the right fielder's hand at an elevation of 1.8 m and the catcher receives it at 1 m. Use $\varepsilon s$ = 0.01%

$$y = \tan\theta_0 x - \frac{g}{2v^2 cos^2(\theta_0)} x^2 + y_0$$

Given:

$v_0$ = 30/ms

$y_0$ = 1.8m

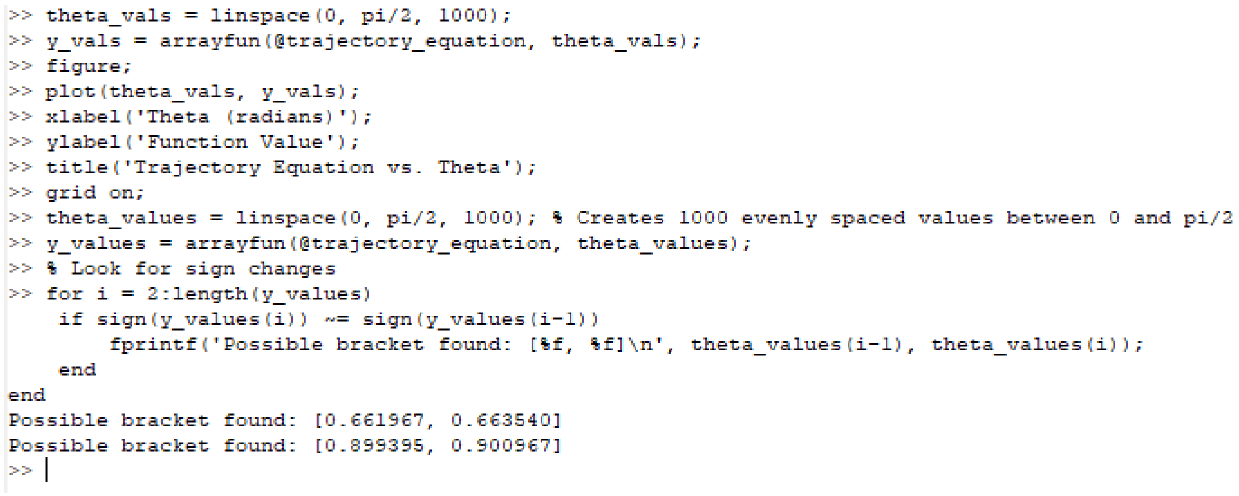
y = 1m

g = 9.81 m/$s^2$

$x = 90$

We can use the following code to find our first initial angle for $\theta_0$:

```
function y = trajectory_equation(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance
    y_0 = 1.8; % initial height

    y = tan(theta_0) * x - (g * x^2) / (2 * v_0^2 * (cos(theta_0))^2) + y_0 - 1;
endfunction
```

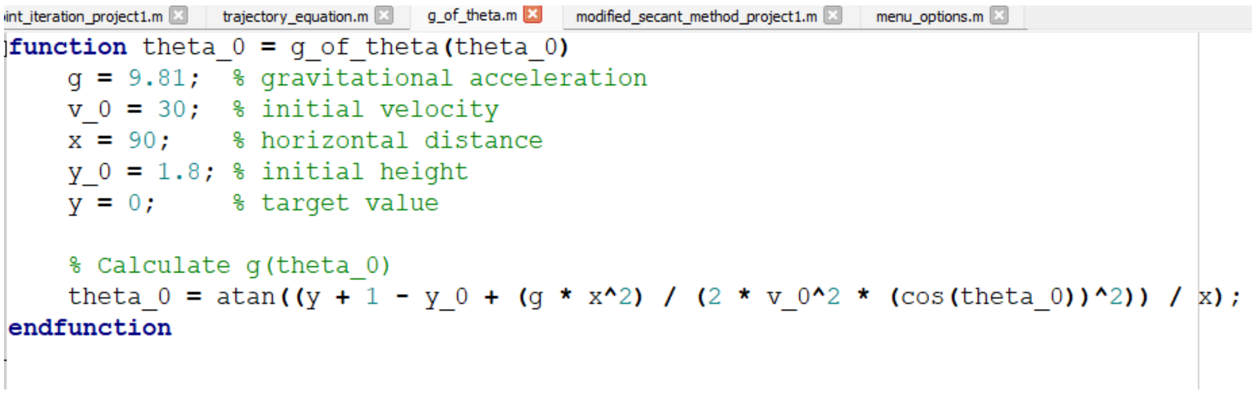


```
>> theta_vals = linspace(0, pi/2, 1000);
>> y_vals = arrayfun(@trajectory_equation, theta_vals);
>> figure;
>> plot(theta_vals, y_vals);
>> xlabel('Theta (radians)');
>> ylabel('Function Value');
>> title('Trajectory Equation vs. Theta');
>> grid on;
>> theta_values = linspace(0, pi/2, 1000); % Creates 1000 evenly spaced values between 0 and pi/2
>> y_values = arrayfun(@trajectory_equation, theta_values);
>> % Look for sign changes
>> for i = 2:length(y_values)
    if sign(y_values(i)) ~= sign(y_values(i-1))
        fprintf('Possible bracket found: [%f, %f]\n', theta_values(i-1), theta_values(i));
    end
end
Possible bracket found: [0.661967, 0.663540]
Possible bracket found: [0.899395, 0.900967]
>> |
```

Using these as our $\theta_0$:

$[x_L, x_u]$ = [0.661967, 0.663540]

For the fixed-point iteration method, we have to derive our g(x) first before using the method. The following code is used for the g(x) of the equation:



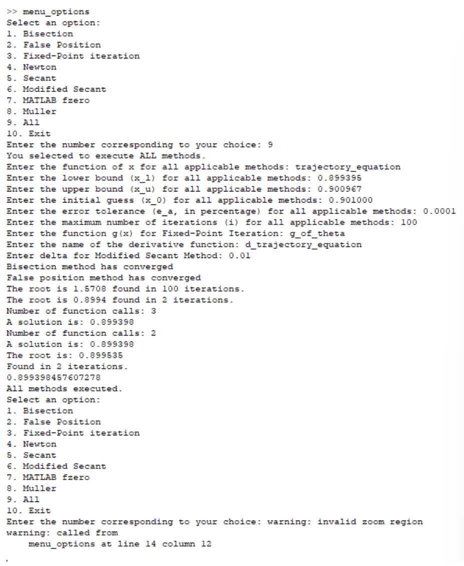
```
function theta_0 = g_of_theta(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance
    y_0 = 1.8; % initial height
    y = 0;     % target value

    % Calculate g(theta_0)
    theta_0 = atan((y + 1 - y_0 + (g * x^2) / (2 * v_0^2 * (cos(theta_0))^2)) / x);
endfunction
```

For the Newton method, we first need to find the derivative of the equation. After that, we can plug it into code like so:

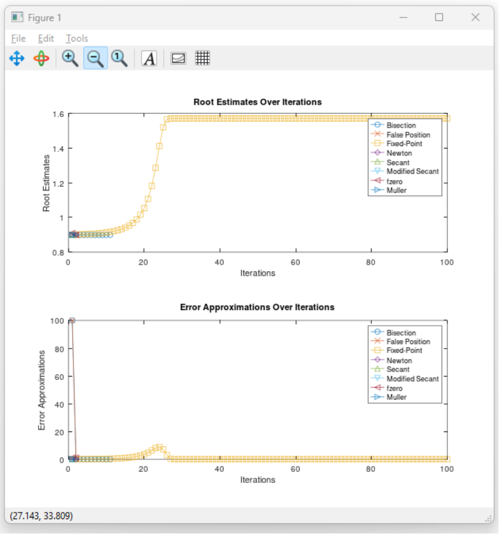
```matlab
function dy = d_trajectory_equation(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance

    dy = x * (sec(theta_0))^2 - (g * x^2 * sin(theta_0)) / (v_0^2 * (cos(theta_0))^3);
end
```

b) Repeat part a using different initial guesses (3 different values where applicable).

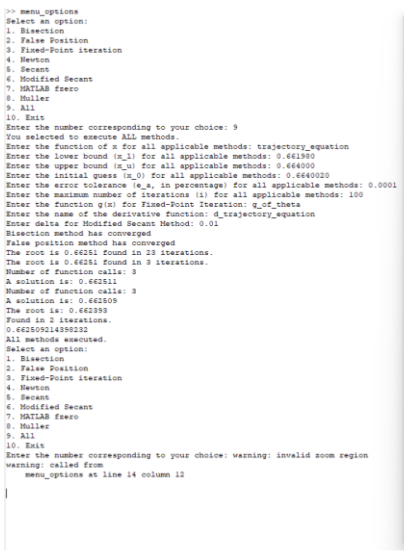
Using a different initial guess:



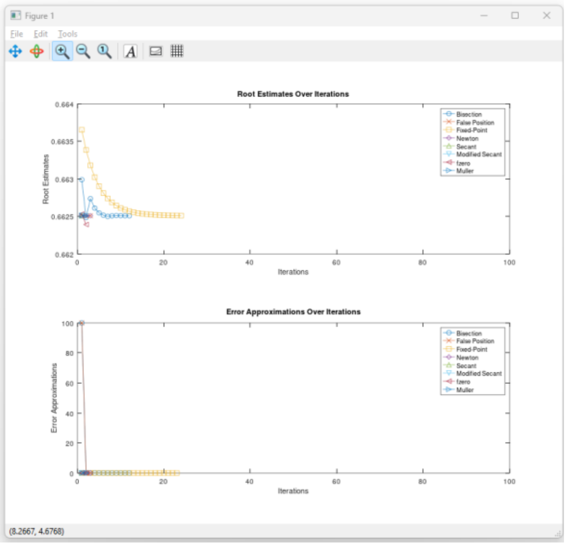
```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: trajectory_equation
Enter the lower bound (x_l) for all applicable methods: 0.899395
Enter the upper bound (x_u) for all applicable methods: 0.900967
Enter the initial guess (x_0) for all applicable methods: 0.901000
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_theta
Enter the name of the derivative function: d_trajectory_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 1.5708 found in 100 iterations.
The root is 0.8994 found in 2 iterations.
Number of function calls: 3
A solution is: 0.899398
Number of function calls: 2
A solution is: 0.899398
The root is: 0.899535
Found in 2 iterations.
0.899398457607278
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: warning: invalid zoom region
warning: called from
    menu_options at line 14 column 12
```



Using a second value closer to the first initial guess using in (a):



```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: trajectory_equation
Enter the lower bound (x_l) for all applicable methods: 0.661980
Enter the upper bound (x_u) for all applicable methods: 0.664000
Enter the initial guess (x_0) for all applicable methods: 0.6640020
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_theta
Enter the name of the derivative function: d_trajectory_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 0.66251 found in 23 iterations.
The root is 0.66251 found in 3 iterations.
Number of function calls: 3
A solution is: 0.662511
Number of function calls: 3
A solution is: 0.662509
The root is: 0.662393
Found in 2 iterations.
0.662509214398232
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: warning: invalid zoom region
warning: called from
    menu_options at line 14 column 12
```



c) Plot a graph of the approximation percentage error for all the used algorithms in part a.

All plots in (a) and (b) include Root Estimation plot and Error Approximation plot.

d) Which algorithm is the fastest?

| Name | Initial Guesses | Tolerance | Iterations | Estimated Root | Iterations Taken |
|---|---|---|---|---|---|
| Bisection | xl = 0.661967, xu = 0.663540 | 0.0001 | 100 | 0.66251 | 12 |
| False-Position | xl = 0.661967, xu = 0.663540 | 0.0001 | 100 | 0.66251 | 3 |
| Fixed-Point Iteration | x0 = 0.661967 | 0.0001 | 100 | 0.66251 | 22 |
| Newton-Raphson | x0 = 0.661967 | 0.0001 | 100 | 0.66251 | 2 |
| Secant | xl = 0.661967, xu = 0.663540 | 0.0001 | 100 | 0.66251 | 3 |
| Modified Secant | x0 = 0.661967 | 0.0001 | 100 | 0.66251 | 3 |
| Fzero | x0 = 0.661967 | 0.0001 | 100 | 0.66251 | 4 |
| Muller | xl = 0.661967, xu = 0.663540, x0 = 0.663580 | 0.0001 | 100 | 0.66251 | 4 |

The Newton-Raphson method, generally considered fast for finding roots due to its quadratic convergence, where the number of accurate digits doubles with each iteration. This method uses the function's value and its derivative, providing an efficient prediction of the root, which often results in achieving the desired accuracy with fewer iterations compared to other methods like bisection or secant. The adaptive step size, which adjusts based on the slope of the function, further optimizes the convergence process. However, the method's performance is contingent upon a good initial guess and a well-behaved function; otherwise, it may not converge or could be outperformed by other algorithms.

## Problem 3

A total charge $Q$ is uniformly distributed around a ring-shaped conductor with radius $a$. A charge $q$ is located at a distance $x$ from the center of the ring (Fig. 2). The force exerted on the charge by the ring is given by:

$$F = \frac{1}{4\pi e_0} \frac{qQx}{(x^2 + a^2)^{\frac{3}{2}}}$$

Where $e_0 = 8.9 * 10^{-12} \frac{C^2}{Nm^2}$ . Find the distance $x$ where the force is 1.25 N if $q$ and $Q$ are $2 * 10^{-5}$C for a ring with a radius of 0.85 m.

You need to run all codes when possible.

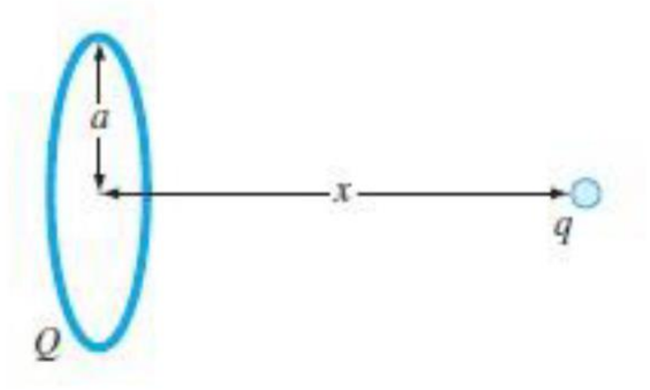
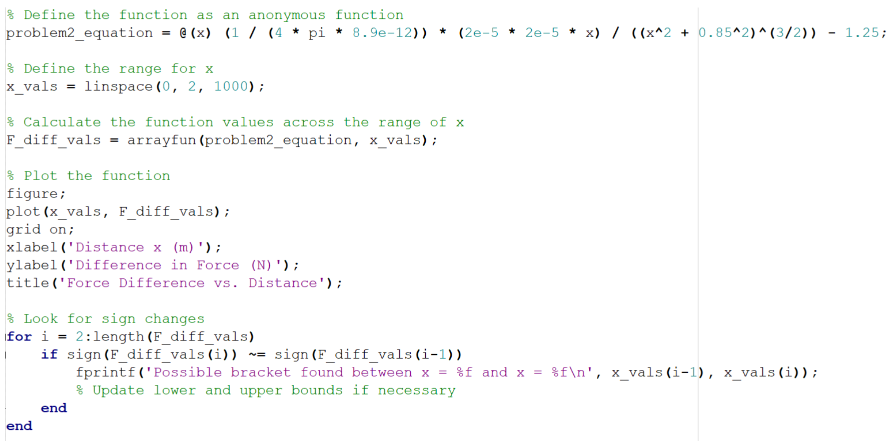


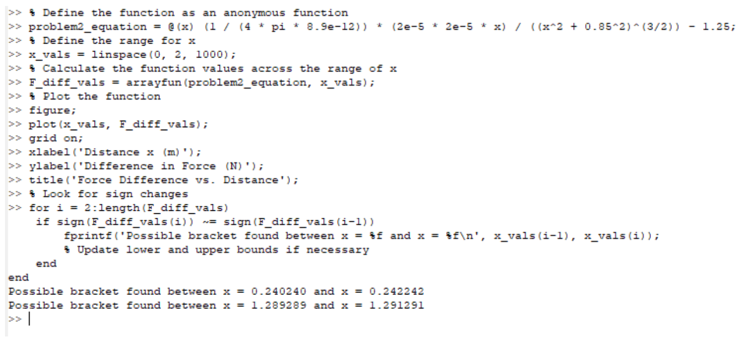
Fig. 2

We begin similarly with Problem 2 by finding possible roots with the following code:



```
% Define the function as an anonymous function
problem2_equation = @(x) (1 / (4 * pi * 8.9e-12)) * (2e-5 * 2e-5 * x) / ((x^2 + 0.85^2)^(3/2)) - 1.25;

% Define the range for x
x_vals = linspace(0, 2, 1000);

% Calculate the function values across the range of x
F_diff_vals = arrayfun(problem2_equation, x_vals);

% Plot the function
figure;
plot(x_vals, F_diff_vals);
grid on;
xlabel('Distance x (m)');
ylabel('Difference in Force (N)');
title('Force Difference vs. Distance');

% Look for sign changes
for i = 2:length(F_diff_vals)
    if sign(F_diff_vals(i)) ~= sign(F_diff_vals(i-1))
        fprintf('Possible bracket found between x = %f and x = %f\n', x_vals(i-1), x_vals(i));
        % Update lower and upper bounds if necessary
    end
end
```

Which yields the following possible roots:

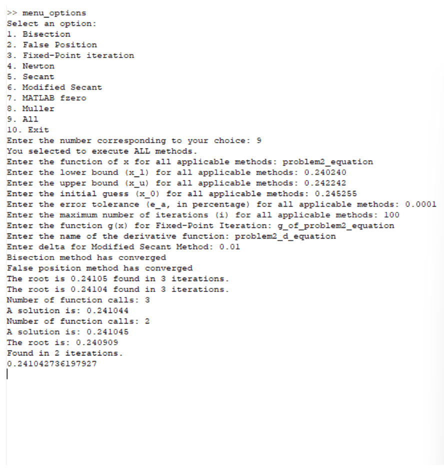


```
>> % Define the function as an anonymous function
>> problem2_equation = @(x) (1 / (4 * pi * 8.9e-12)) * (2e-5 * 2e-5 * x) / ((x^2 + 0.85^2)^(3/2)) - 1.25;
>> % Define the range for x
>> x_vals = linspace(0, 2, 1000);
>> % Calculate the function values across the range of x
>> F_diff_vals = arrayfun(problem2_equation, x_vals);
>> % Plot the function
>> figure;
>> plot(x_vals, F_diff_vals);
>> grid on;
>> xlabel('Distance x (m)');
>> ylabel('Difference in Force (N)');
>> title('Force Difference vs. Distance');
>> % Look for sign changes
>> for i = 2:length(F_diff_vals)
    if sign(F_diff_vals(i)) ~= sign(F_diff_vals(i-1))
        fprintf('Possible bracket found between x = %f and x = %f\n', x_vals(i-1), x_vals(i));
        % Update lower and upper bounds if necessary
    end
end
Possible bracket found between x = 0.240240 and x = 0.242242
Possible bracket found between x = 1.289289 and x = 1.291291
>> |
```

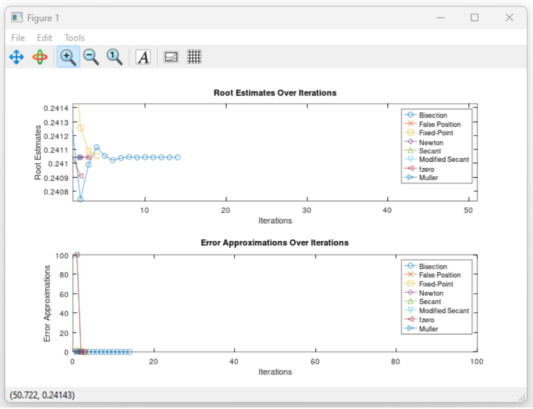
$[x_1, x_2] = [0.240240, 0.242242]$

$[x_1, x_2] = [1.289289, 1.291921]$

Using all numerical methods at once, passing various inputs needed for each piece of code, but specific initial guesses as the ones found, we can see that all solutions converge onto 0.2410~:



```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: problem2_equation
Enter the lower bound (x_l) for all applicable methods: 0.240240
Enter the upper bound (x_u) for all applicable methods: 0.242242
Enter the initial guess (x_0) for all applicable methods: 0.245255
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_problem2_equation
Enter the name of the derivative function: problem2_d_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 0.24105 found in 3 iterations.
The root is 0.24104 found in 3 iterations.
Number of function calls: 3
A solution is: 0.241044
Number of function calls: 2
A solution is: 0.241045
The root is: 0.240909
Found in 2 iterations.
0.241042736197927
|
```



| Name | Initial Guesses | Tolerance | Iterations | Estimated Root | Iterations Taken |
|---|---|---|---|---|---|
| Bisection | xl = 0.240240, xu = 0.242242 | 0.0001 | 100 | 0.241 | 14 |
| False-Position | xl = 0.240240, xu = 0.242242 | 0.0001 | 100 | 0.241 | 3 |
| Fixed-Point Iteration | x0 = 0.242240 | 0.0001 | 100 | 0.241 | 4 |
| Newton-Raphson | x0 = 0.242240 | 0.0001 | 100 | 0.241 | 2 |
| Secant | xl = 0.240240, xu = 0.242242 | 0.0001 | 100 | 0.241 | 4 |
| Modified Secant | x0 = 0.242240 | 0.0001 | 100 | 0.241 | 4 |
| Fzero | x0 = 0.242240 | 0.0001 | 100 | 0.241 | 5 |
| Muller | xl = 0.240240, xu = 0.242242, x0 = 0.242240 | 0.0001 | 100 | 0.241 | 6 |

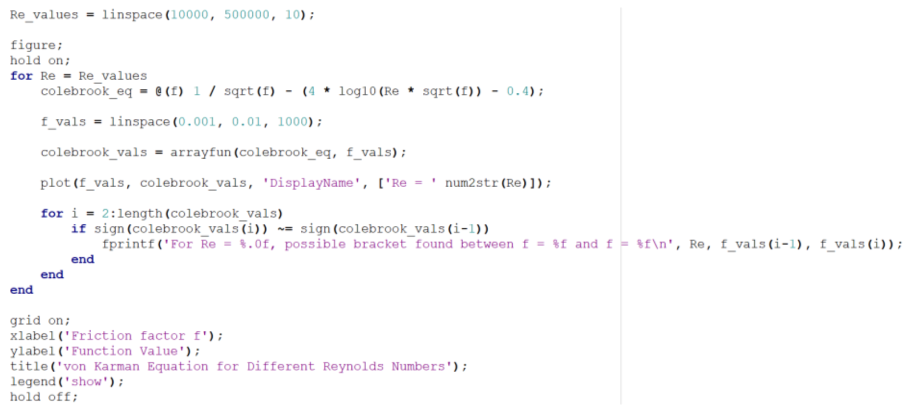
## Problem 4

For fluid flow in pipes, friction is described by a dimensionless number, the Fanning friction factor $f$. The Fanning friction factor is dependent on a number of parameters related to the size of the pipe and the fluid, which can all be represented by another dimensionless quantity, the Reynolds number $Re$. A formula that predicts $f$ given $Re$ is the von Karman equation:

$$\frac{1}{\sqrt{f}} = 4\log_{10}\left(Re\sqrt{f}\right) - 0.4$$

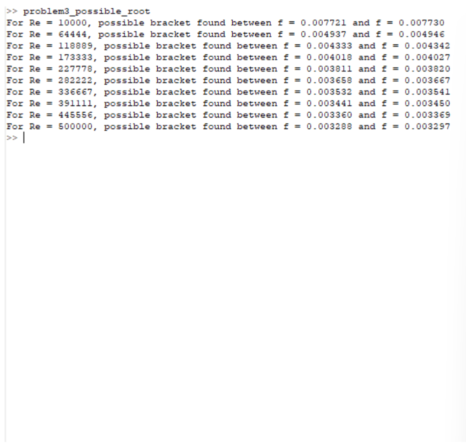
Typical values for the Reynolds number for turbulent flow are 10,000 to 500,000 and for the Fanning friction factor are 0.001 to 0.01. Develop a function to solve for $f$ given a user-supplied value of $Re$ between 2,500 and 1,000,000. Design the function so that it ensures that the absolute error in the result is E < 0.000005.

For this problem, we had to modify our usual steps to find possible brackets:

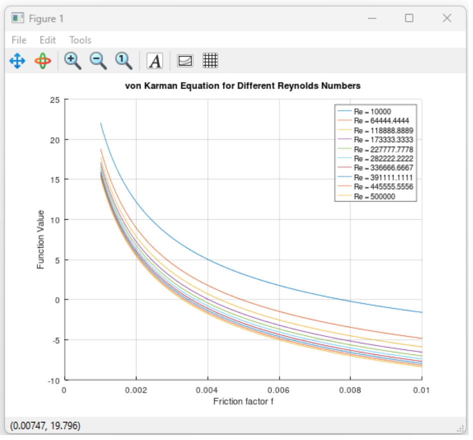


```
Re_values = linspace(10000, 500000, 10);

figure;
hold on;
for Re = Re_values
    colebrook_eq = @(f) 1 / sqrt(f) - (4 * log10(Re * sqrt(f)) - 0.4);

    f_vals = linspace(0.001, 0.01, 1000);

    colebrook_vals = arrayfun(colebrook_eq, f_vals);

    plot(f_vals, colebrook_vals, 'DisplayName', ['Re = ' num2str(Re)]);

    for i = 2:length(colebrook_vals)
        if sign(colebrook_vals(i)) ~= sign(colebrook_vals(i-1))
            fprintf('For Re = %.0f, possible bracket found between f = %f and f = %f\n', Re, f_vals(i-1), f_vals(i));
        end
    end
end

grid on;
xlabel('Friction factor f');
ylabel('Function Value');
title('von Karman Equation for Different Reynolds Numbers');
legend('show');
hold off;
```

This code, would yield several possible brackets perfect for our methods:



```
>> problem3_possible_root
For Re = 10000, possible bracket found between f = 0.007721 and f = 0.007730
For Re = 64444, possible bracket found between f = 0.004937 and f = 0.004946
For Re = 118889, possible bracket found between f = 0.004333 and f = 0.004342
For Re = 173333, possible bracket found between f = 0.004018 and f = 0.004027
For Re = 227778, possible bracket found between f = 0.003811 and f = 0.003820
For Re = 282222, possible bracket found between f = 0.003658 and f = 0.003667
For Re = 336667, possible bracket found between f = 0.003532 and f = 0.003541
For Re = 391111, possible bracket found between f = 0.003441 and f = 0.003450
For Re = 445556, possible bracket found between f = 0.003360 and f = 0.003369
For Re = 500000, possible bracket found between f = 0.003288 and f = 0.003297
>> |
```

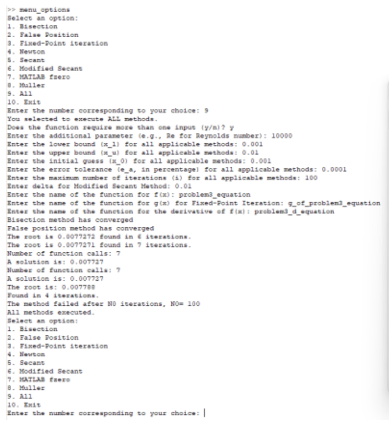


We can use the first bracket and the values given inside the problem description. Namely:

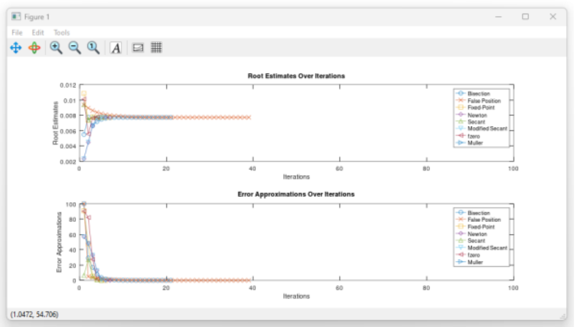
Re = 2,500 through 1,000,000

[x_1, x_2] = [0.001, 0.01]

With these values, we can go over to plug them into our numerical methods functions:



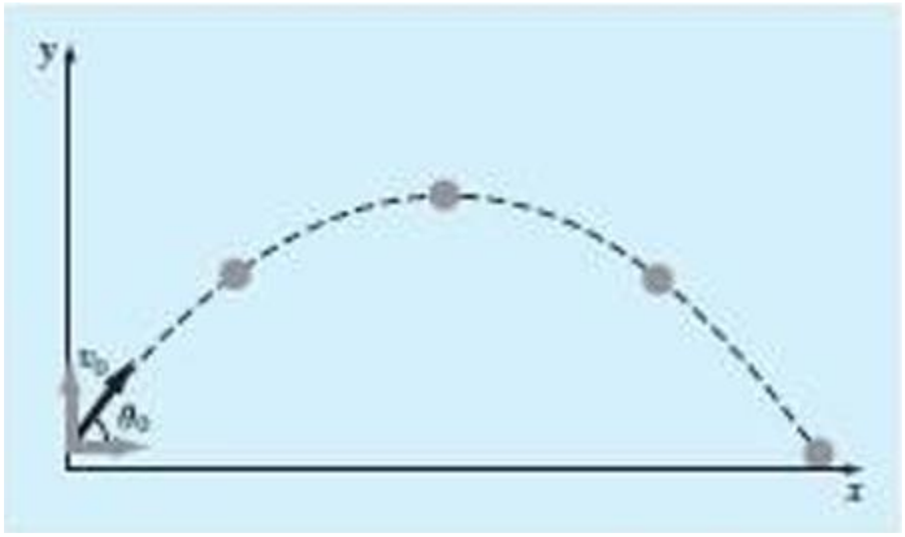
```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Does the function require more than one input (y/n)? y
Enter the additional parameter (e.g., Re for Reynolds number): 10000
Enter the lower bound (x_l) for all applicable methods: 0.001
Enter the upper bound (x_u) for all applicable methods: 0.01
Enter the initial guess (x_0) for all applicable methods: 0.001
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter delta for Modified Secant Method: 0.01
Enter the name of the function for f(x): problem3_equation
Enter the name of the function for g(x) for Fixed-Point Iteration: g_of_problem3_equation
Enter the name of the function for the derivative of f(x): problem3_d_equation
Bisection method has converged
False position method has converged
The root is 0.0077272 found in 6 iterations.
The root is 0.0077271 found in 7 iterations.
Number of function calls: 7
A solution is: 0.007727
Number of function calls: 7
A solution is: 0.007727
The root is: 0.007788
Found in 4 iterations.
The method failed after N0 iterations, N0= 100
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: |
```



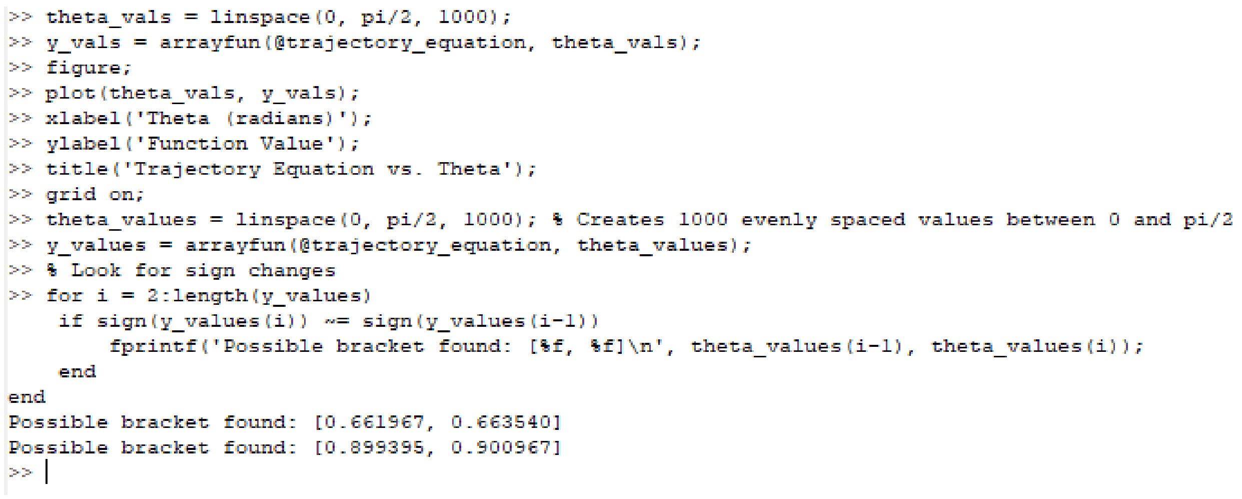
| Name | Initial Guesses | Tolerance | Iterations | Estimated Root | Iterations Taken |
|---|---|---|---|---|---|
| Bisection | Re = 10000, xl = 0.001, xu = 0.01 | 0.0001 | 100 | 0.77272 | 7 |
| False-Position | Re = 10000, xl = 0.001, xu = 0.01 | 0.0001 | 100 | 0.77272 | 8 |
| Fixed-Point Iteration | Re = 10000 | 0.0001 | 100 | 0.77272 | 7 |
| Newton-Raphson | Re = 10000 | 0.0001 | 100 | 0.77272 | 4 |
| Secant | Re = 10000 | 0.0001 | 100 | 0.77272 | 6 |
| Modified Secant | Re = 10000 | 0.0001 | 100 | 0.77272 | 7 |
| Fzero | Re = 10000 | 0.0001 | 100 | 0.77272 | 6 |
| Muller | Re = 10000 | 0.0001 | 100 | 0.77272 | 9 |

## Figures

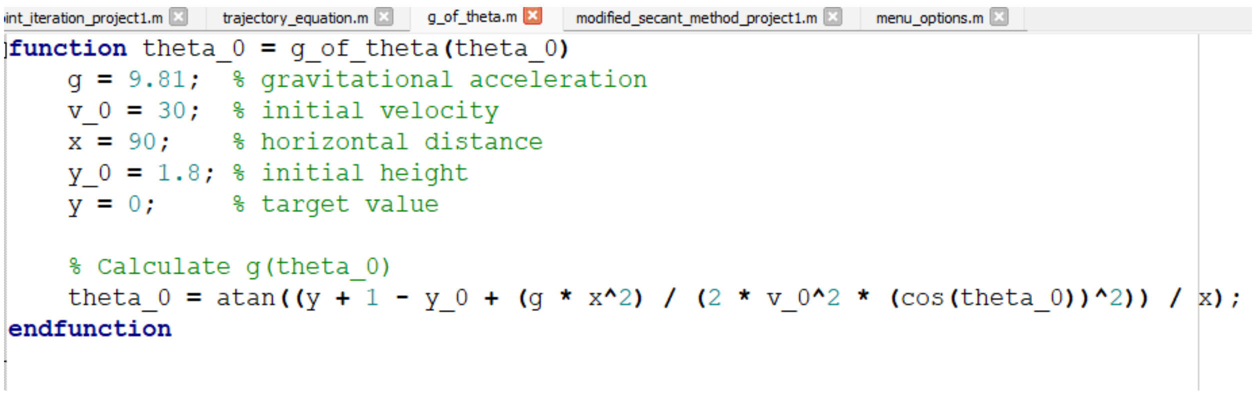
$$y = \tan(\theta_0) \times x - \frac{g \times x^2}{2 \times v_0^2 \times \cos^2(\theta_0)} + y_0$$



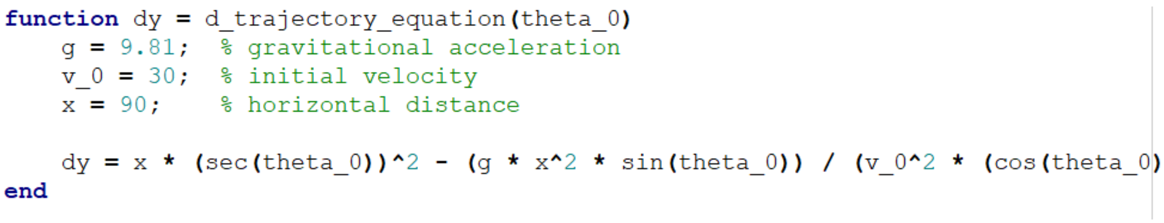
```
function y = trajectory_equation(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance
    y_0 = 1.8; % initial height

    y = tan(theta_0) * x - (g * x^2) / (2 * v_0^2 * (cos(theta_0))^2) + y_0 - 1;
endfunction
```



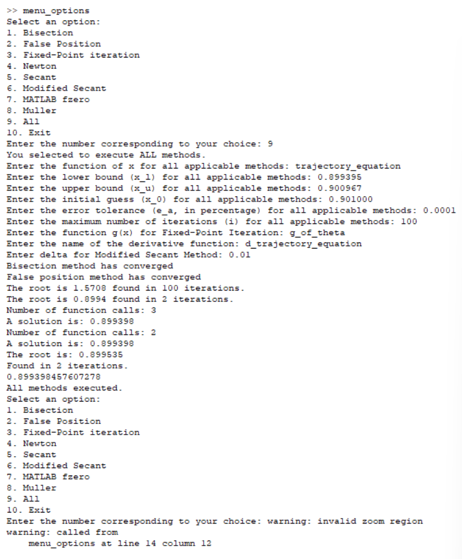
```
>> theta_vals = linspace(0, pi/2, 1000);
>> y_vals = arrayfun(@trajectory_equation, theta_vals);
>> figure;
>> plot(theta_vals, y_vals);
>> xlabel('Theta (radians)');
>> ylabel('Function Value');
>> title('Trajectory Equation vs. Theta');
>> grid on;
>> theta_values = linspace(0, pi/2, 1000); % Creates 1000 evenly spaced values between 0 and pi/2
>> y_values = arrayfun(@trajectory_equation, theta_values);
>> % Look for sign changes
>> for i = 2:length(y_values)
    if sign(y_values(i)) ~= sign(y_values(i-1))
        fprintf('Possible bracket found: [%f, %f]\n', theta_values(i-1), theta_values(i));
    end
end
Possible bracket found: [0.661967, 0.663540]
Possible bracket found: [0.899395, 0.900967]
>> |
```



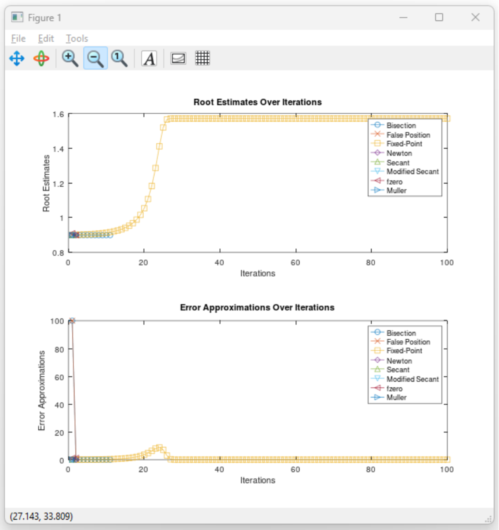
```
function theta_0 = g_of_theta(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance
    y_0 = 1.8; % initial height
    y = 0;     % target value

    % Calculate g(theta_0)
    theta_0 = atan((y + 1 - y_0 + (g * x^2) / (2 * v_0^2 * (cos(theta_0))^2)) / x);
endfunction
```

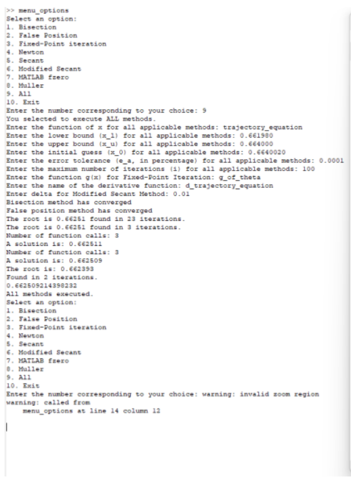


```
function dy = d_trajectory_equation(theta_0)
    g = 9.81;  % gravitational acceleration
    v_0 = 30;  % initial velocity
    x = 90;    % horizontal distance

    dy = x * (sec(theta_0))^2 - (g * x^2 * sin(theta_0)) / (v_0^2 * (cos(theta_0))^3);
end
```

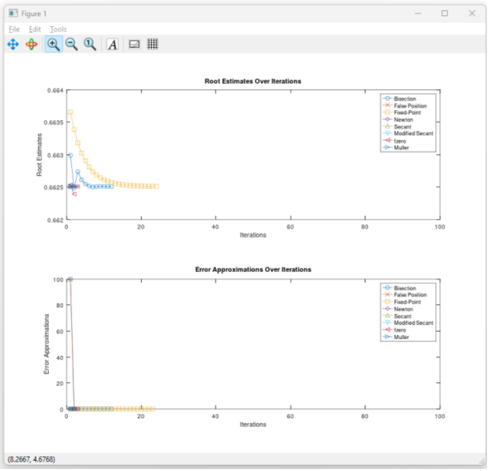


```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: trajectory_equation
Enter the lower bound (x_l) for all applicable methods: 0.899395
Enter the upper bound (x_u) for all applicable methods: 0.900967
Enter the initial guess (x_0) for all applicable methods: 0.901000
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_theta
Enter the name of the derivative function: d_trajectory_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 1.5708 found in 100 iterations.
The root is 0.8994 found in 2 iterations.
Number of function calls: 3
A solution is: 0.899398
Number of function calls: 2
A solution is: 0.899398
The root is: 0.899535
Found in 2 iterations.
0.899398457607278
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: warning: invalid zoom region
warning: called from
    menu_options at line 14 column 12
```





```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: trajectory_equation
Enter the lower bound (x_l) for all applicable methods: 0.661980
Enter the upper bound (x_u) for all applicable methods: 0.664000
Enter the initial guess (x_0) for all applicable methods: 0.6640020
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_theta
Enter the name of the derivative function: d_trajectory_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 0.66251 found in 23 iterations.
The root is 0.66251 found in 3 iterations.
Number of function calls: 3
A solution is: 0.662511
Number of function calls: 3
A solution is: 0.662509
The root is: 0.662393
Found in 2 iterations.
0.662509214398232
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: warning: invalid zoom region
warning: called from
    menu_options at line 14 column 12
```



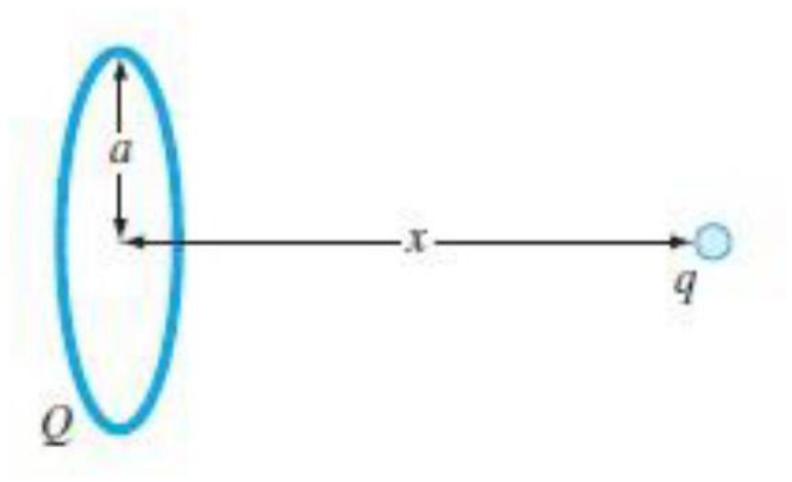
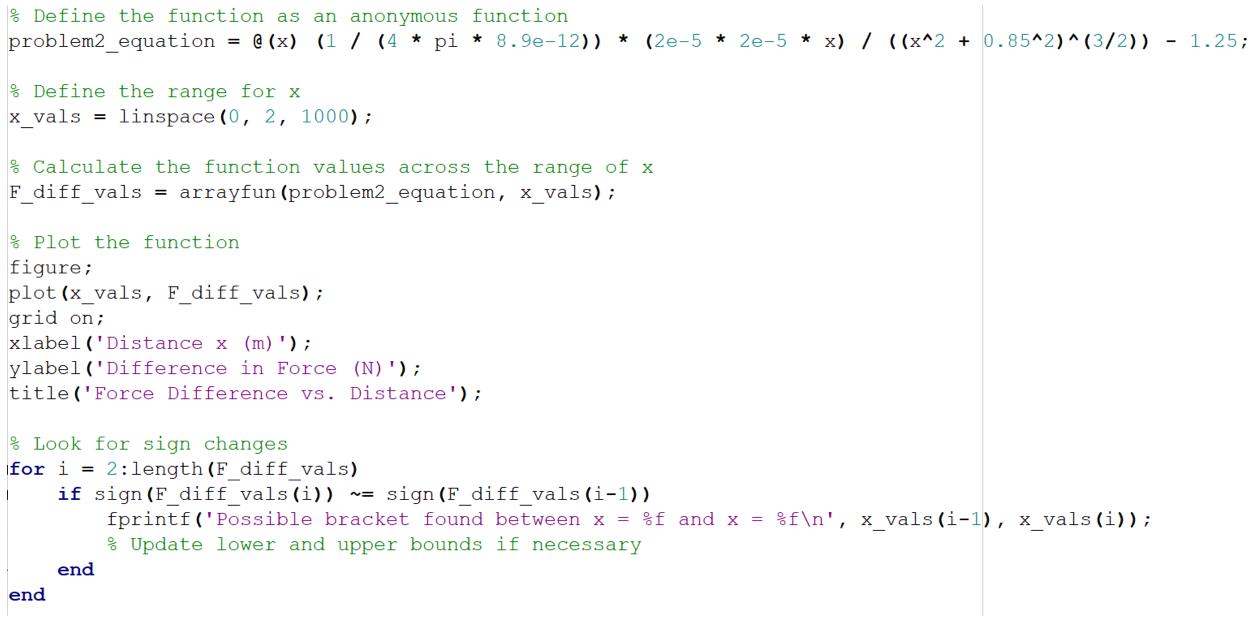
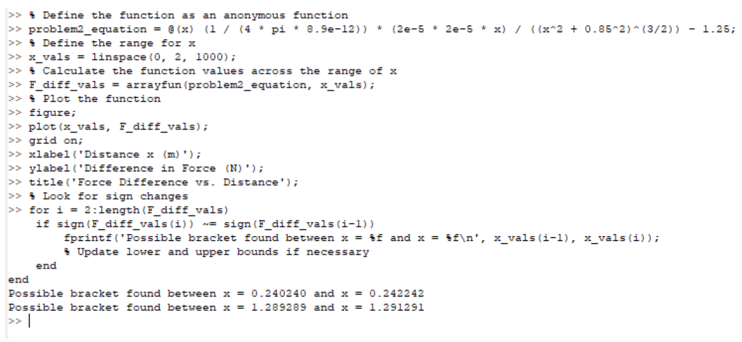


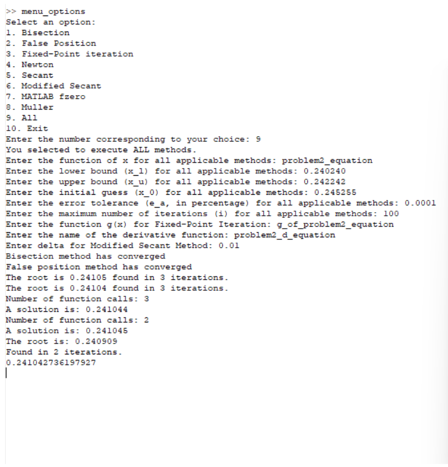
Fig. 2



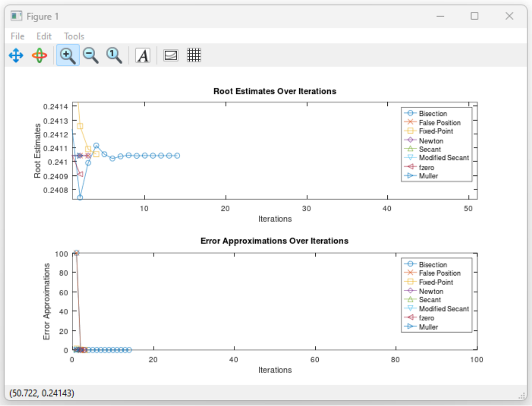
```
% Define the function as an anonymous function
problem2_equation = @(x) (1 / (4 * pi * 8.9e-12)) * (2e-5 * 2e-5 * x) / ((x^2 + 0.85^2)^(3/2)) - 1.25;

% Define the range for x
x_vals = linspace(0, 2, 1000);

% Calculate the function values across the range of x
F_diff_vals = arrayfun(problem2_equation, x_vals);

% Plot the function
figure;
plot(x_vals, F_diff_vals);
grid on;
xlabel('Distance x (m)');
ylabel('Difference in Force (N)');
title('Force Difference vs. Distance');

% Look for sign changes
for i = 2:length(F_diff_vals)
    if sign(F_diff_vals(i)) ~= sign(F_diff_vals(i-1))
        fprintf('Possible bracket found between x = %f and x = %f\n', x_vals(i-1), x_vals(i));
        % Update lower and upper bounds if necessary
    end
end
```



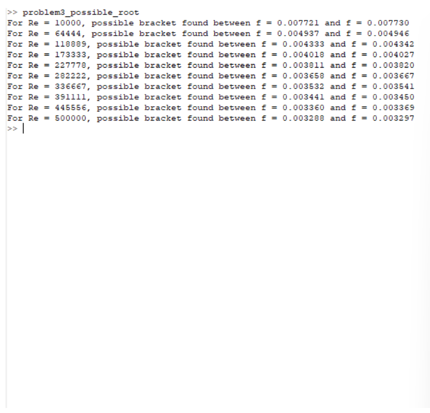
```
>> % Define the function as an anonymous function
>> problem2_equation = @(x) (1 / (4 * pi * 8.9e-12)) * (2e-5 * 2e-5 * x) / ((x^2 + 0.85^2)^(3/2)) - 1.25;
>> % Define the range for x
>> x_vals = linspace(0, 2, 1000);
>> % Calculate the function values across the range of x
>> F_diff_vals = arrayfun(problem2_equation, x_vals);
>> % Plot the function
>> figure;
>> plot(x_vals, F_diff_vals);
>> grid on;
>> xlabel('Distance x (m)');
>> ylabel('Difference in Force (N)');
>> title('Force Difference vs. Distance');
>> % Look for sign changes
>> for i = 2:length(F_diff_vals)
    if sign(F_diff_vals(i)) ~= sign(F_diff_vals(i-1))
        fprintf('Possible bracket found between x = %f and x = %f\n', x_vals(i-1), x_vals(i));
        % Update lower and upper bounds if necessary
    end
end
Possible bracket found between x = 0.240240 and x = 0.242242
Possible bracket found between x = 1.289289 and x = 1.291291
>> |
```



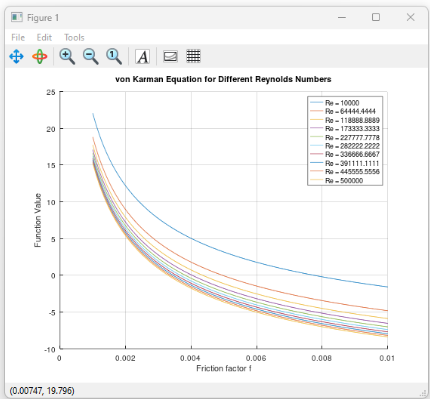
```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Enter the function of x for all applicable methods: problem2_equation
Enter the lower bound (x_l) for all applicable methods: 0.240240
Enter the upper bound (x_u) for all applicable methods: 0.242242
Enter the initial guess (x_0) for all applicable methods: 0.245255
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter the function g(x) for Fixed-Point Iteration: g_of_problem2_equation
Enter the name of the derivative function: problem2_d_equation
Enter delta for Modified Secant Method: 0.01
Bisection method has converged
False position method has converged
The root is 0.24105 found in 3 iterations.
The root is 0.24104 found in 3 iterations.
Number of function calls: 3
A solution is: 0.241044
Number of function calls: 2
A solution is: 0.241045
The root is: 0.240909
Found in 2 iterations.
0.241042736197927
```

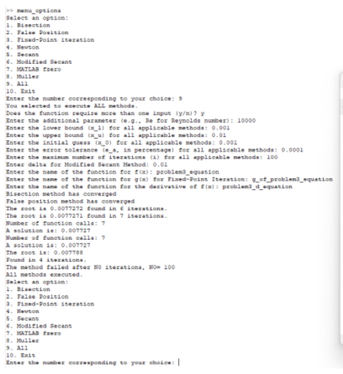


```
Re_values = linspace(10000, 500000, 10);

figure;
hold on;
for Re = Re_values
    colebrook_eq = @(f) 1 / sqrt(f) - (4 * log10(Re * sqrt(f)) - 0.4);

    f_vals = linspace(0.001, 0.01, 1000);

    colebrook_vals = arrayfun(colebrook_eq, f_vals);

    plot(f_vals, colebrook_vals, 'DisplayName', ['Re = ' num2str(Re)]);

    for i = 2:length(colebrook_vals)
        if sign(colebrook_vals(i)) ~= sign(colebrook_vals(i-1))
            fprintf('For Re = %.0f, possible bracket found between f = %f and f = %f\n', Re, f_vals(i-1), f_vals(i));
        end
    end
end

grid on;
xlabel('Friction factor f');
ylabel('Function Value');
title('von Karman Equation for Different Reynolds Numbers');
legend('show');
hold off;
```

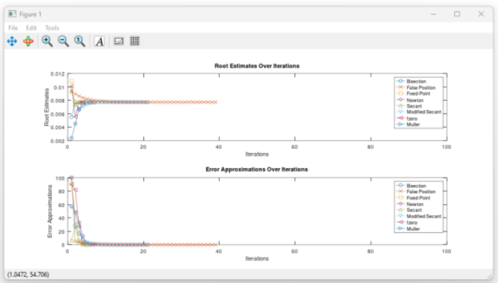


```
>> problem3_possible_root
For Re = 10000, possible bracket found between f = 0.007721 and f = 0.007730
For Re = 64444, possible bracket found between f = 0.004937 and f = 0.004946
For Re = 118889, possible bracket found between f = 0.004333 and f = 0.004342
For Re = 173333, possible bracket found between f = 0.004018 and f = 0.004027
For Re = 227778, possible bracket found between f = 0.003811 and f = 0.003820
For Re = 282222, possible bracket found between f = 0.003658 and f = 0.003667
For Re = 336667, possible bracket found between f = 0.003532 and f = 0.003541
For Re = 391111, possible bracket found between f = 0.003441 and f = 0.003450
For Re = 445556, possible bracket found between f = 0.003360 and f = 0.003369
For Re = 500000, possible bracket found between f = 0.003288 and f = 0.003297
>> 
```





```
>> menu_options
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 9
You selected to execute ALL methods.
Does the function require more than one input (y/n)? y
Enter the additional parameter (e.g., Re for Reynolds number): 10000
Enter the lower bound (x_l) for all applicable methods: 0.001
Enter the upper bound (x_u) for all applicable methods: 0.01
Enter the initial guess (x_0) for all applicable methods: 0.001
Enter the error tolerance (e_a, in percentage) for all applicable methods: 0.0001
Enter the maximum number of iterations (i) for all applicable methods: 100
Enter delta for Modified Secant Method: 0.01
Enter the name of the function for f(x): problem3_equation
Enter the name of the function for g(x) for Fixed-Point Iteration: g_of_problem3_equation
Enter the name of the function for the derivative of f(x): problem3_d_equation
Bisection method has converged
False position method has converged
The root is 0.0077272 found in 6 iterations.
The root is 0.0077271 found in 7 iterations.
Number of function calls: 7
A solution is: 0.007727
Number of function calls: 7
A solution is: 0.007727
The root is: 0.007788
Found in 4 iterations.
The method failed after N0 iterations, N0= 100
All methods executed.
Select an option:
1. Bisection
2. False Position
3. Fixed-Point iteration
4. Newton
5. Secant
6. Modified Secant
7. MATLAB fzero
8. Muller
9. All
10. Exit
Enter the number corresponding to your choice: 
```



## Appendix

### Codes

#### Bisection

```
function [xr, i, ea] = bisection_method_project1(xl, xu, func, es, maxit)
  % Find root using the bisection method.
  % Input: xl      lower bound
  %        xu      upper bound
  %        func    function to find the root for
  %        es      error tolerance
  %        maxit   maximum number of iterations
  % Output: xr     estimated root
  %         ea     approximate error

  a(1) = xl; b(1) = xu;
  ya(1) = feval(func, a(1)); yb(1) = feval(func, b(1));
  if ya(1) * yb(1) > 0.0
      error('Function has same sign at end points');
  end

  ea = []; % To store approximate relative error
  xi = []; % To store estimated roots

  for i = 1:maxit
      xi(i) = (a(i) + b(i)) / 2;
      y(i) = feval(func, xi(i));

      if i > 1
          ea(i) = abs((xi(i) - xi(i-1)) / xi(i)) * 100;
```

```
        else
            ea(i) = 100;
        end

        if y(i) == 0.0
            disp('Exact zero found'); break;
        elseif y(i) * ya(i) < 0
            a(i+1) = a(i); ya(i+1) = ya(i);
            b(i+1) = xi(i); yb(i+1) = y(i);
        else
            a(i+1) = xi(i); ya(i+1) = y(i);
            b(i+1) = b(i); yb(i+1) = yb(i);
        end

        if (i > 1) && (ea(i) < es)
            disp('Bisection method has converged'); break;
        end
    end

    if i >= maxit
        disp('Zero not found to desired tolerance');
    end

    xr = xi;
    ea = ea;
end
```

### False Position

```
function [xr, i, ea] = false_position_method_project1(xl, xu, func, es,
maxit)
```

```
% Find root near x1 using the false position method.
% Input: func     string containing name of function
%       xl, xu    initial guesses
%       es        allowable tolerance in computed root
%       maxit     maximum number of iterations
% Output: xr      row vector of approximations to root
%         ea      row vector of approximate relative error

% Initialize
a(1) = xl;
b(1) = xu;
ya(1) = feval(func, a(1));
yb(1) = feval(func, b(1));

if ya(1) * yb(1) > 0.0
  error('Function has same sign at end points');
end

ea = []; % To store approximate relative error

for i = 1:maxit
  % Compute the root estimate
  xi(i) = b(i) - yb(i)*(b(i)-a(i))/(yb(i)-ya(i));
  y(i) = feval(func, xi(i));

  % Compute error after the first iteration
  if i > 1
    ea(i) = abs((xi(i) - xi(i-1)) / xi(i)) * 100;
  else
    ea(i) = 100; % For the first iteration, set error to 100%
    end

    % Check for exact zero
    if y(i) == 0.0
      disp('exact zero found');
      break;
    elseif y(i) * ya(i) < 0
      a(i+1) = a(i);
      ya(i+1) = ya(i);
      b(i+1) = xi(i);
      yb(i+1) = y(i);
    else
      a(i+1) = xi(i);
      ya(i+1) = y(i);
      b(i+1) = b(i);
      yb(i+1) = yb(i);
    end

    % Check for convergence
    if (i > 1) && (ea(i) < es)
      disp('False position method has converged');
      break;
    end
  end

  % Return the root and error vectors
  xr = xi;
  ea = ea;
end
```

## Fixed-Point Iteration

```
function [x_vals, iter, ea_vals] = fixed_point_iteration_project1(x_0,
f_func_str, g_func_str, e_a, i)

    iter = 0;

    x = x_0;


    x_vals = []; % To store root approximations

    ea_vals = []; % To store error approximations


    while iter < i

        x_old = x; % Store old value for error calculation


        % Step 1: Calculate y = g(x_0)

        y = feval(g_func_str, x);


        % Step 2: Let x1 = g(x_0)

        x1 = y;


        % Step 3: Examine if x1 is the solution of f(x) = 0

        f_x1 = feval(f_func_str, x1);


        x_vals = [x_vals x1]; % Store the new x value for plotting


        if abs(f_x1) < e_a

            break;

        end


        % Step 4: If not, repeat the iteration

        x = x1;
        ea = abs((x1 - x_old) / x1) * 100;

        ea_vals = [ea_vals ea]; % Store the new error value for plotting


        iter = iter + 1;

    end


    % Return the root and error vectors

    disp(['The root is ', num2str(x1), ' found in ', num2str(iter), '
iterations.']);

end
```

## Newton Method

```
function [x_vals, iter, ea_vals] = newton_method_project1(x_0, func_handle,
func_prime_handle, e_a, i)

    iter = 0;

    x = x_0;


    % Initialize vectors for storing x-values and errors for output

    x_vals = [];

    ea_vals = [];


    while iter < i

        x_old = x;


        % Compute the new estimate using the Newton formula

        x = x_old - feval(func_handle, x_old) / feval(func_prime_handle,
x_old);


        % Store the new x value for output

        x_vals = [x_vals x];
        iter = iter + 1;


        if x ~= 0

            ea = abs((x - x_old) / x) * 100;


            % Store the new error value for output

            ea_vals = [ea_vals ea];


            if ea < e_a

                break;

            end

        end

    end


    % Return the root and error vectors

    disp(['The root is ', num2str(x), ' found in ', num2str(iter), '
iterations.']);

end
```

## Secant Method

```
function [x_vals, no_iterations, ea_vals] = secant_method_project1(x0, x1,
func_str, eps, max_iter)

    [solution, no_iterations, x_vals, ea_vals] = secant(func_str, x0, x1,
eps, max_iter);


    if no_iterations > 0  % Solution found

        fprintf('Number of function calls: %d\n', 2 + no_iterations);

        fprintf('A solution is: %f\n', solution);

    else

        fprintf('Abort execution. Solution not found within max_iter.\n');

    end
```

```
end


function [solution, no_iterations, x_vals, ea_vals] = secant(func_str, x0,
x1, eps, max_iter)
    f_x0 = feval(func_str, x0);
    f_x1 = feval(func_str, x1);
    iteration_counter = 0;

    x_vals = NaN(1, max_iter);  % Initialize with NaN values
    ea_vals = NaN(1, max_iter); % Initialize with NaN values

    while iteration_counter < max_iter
        denominator = f_x1 - f_x0;
        if denominator == 0
            fprintf('Error! - denominator zero for x = %f\n', x1);
            solution = NaN;
            return;
        end

        x = x1 - (f_x1 * (x1 - x0) / denominator);
        ea = abs((x - x1) / x) * 100;

        if ea <= eps  % Convergence criterion
            break;
        end

        x0 = x1;
        x1 = x;
        f_x0 = f_x1;
        f_x1 = feval(func_str, x1);
        % Store the new x value and error value
        x_vals(iteration_counter + 1) = x;
        ea_vals(iteration_counter + 1) = ea;

        iteration_counter = iteration_counter + 1;
    end

    solution = x1;
    no_iterations = iteration_counter;
end
```

## Modified Secant Method

```
function [x_vals, no_iterations, ea_vals] =
modified_secant_method_project1(x0, delta, func_handle, eps, max_iter)
    [solution, no_iterations, x_vals, ea_vals] = modified_secant(func_handle,
x0, delta, eps, max_iter);

    if no_iterations > 0  % Solution found
        fprintf('Number of function calls: %d\n', 1 + no_iterations);
        fprintf('A solution is: %f\n', solution);
    else
        fprintf('Abort execution.\n');
    end
end

function [solution, no_iterations, x_vals, ea_vals] =
modified_secant(func_handle, x0, delta, eps, max_iter)
    f_x0 = feval(func_handle, x0);
    iteration_counter = 0;
```

```
x_vals = [];
ea_vals = [];

while abs(f_x0) > eps && iteration_counter < max_iter
    try
        x1 = x0 + delta * x0; % Perturbed point
        f_x1 = feval(func_handle, x1);
        denominator = (f_x1 - f_x0);
        x = x0 - (f_x0 * (delta * x0)) / denominator;
    catch
        fprintf('Error! - denominator zero for x = %f\n', x0);
        break
    end

    ea = abs((x - x0) / x) * 100;

    x_vals = [x_vals x];
    ea_vals = [ea_vals ea];

    x0 = x;
    f_x0 = feval(func_handle, x0);
    iteration_counter = iteration_counter + 1;
end

if abs(f_x0) > eps
    iteration_counter = -1;
end

solution = x0;
no_iterations = iteration_counter;
```

```
end
```

## MATLAB fzero

```
function [x_vals, iter, ea_vals] = fzero_method_project1(f_func, x0, tol,
max_iter)

    x_vals = [];
    ea_vals = [];
    x_old = x0;
    iter = 0;

    options = optimset('TolX', tol, 'MaxIter', max_iter, 'OutputFcn',
@outfun);

    root = fzero(f_func, x0, options);

    function stop = outfun(x, optimValues, state)
        stop = false;
        iter = iter + 1;

        % Compute the approximate error if x is not zero
        if x ~= 0
            ea = abs((x - x_old) / x) * 100;
            ea_vals = [ea_vals, ea];
        end

        x_vals = [x_vals, x];
        x_old = x;
    end
    % Uncomment the following lines if you want to display the results in the
Command Window

    fprintf('The root is: %f\n', root);

    fprintf('Found in %d iterations.\n', iter);

end
```

## Muller Method

```
function [estimated_roots, i, approx_errors] = muller_project1(p0, p1, p2,
func, TOL, N0)

    format long;


    h1 = p1 - p0;

    h2 = p2 - p1;

    DELTA1 = (feval(func, p1) - feval(func, p0)) / h1;

    DELTA2 = (feval(func, p2) - feval(func, p1)) / h2;

    d = (DELTA2 - DELTA1) / (h2 + h1);

    i = 3;


    % Initializing arrays to store estimated roots and approximations

    estimated_roots = [];

    approx_errors = [];


    while i <= N0

        b = DELTA2 + h2 * d;

        D = sqrt(b^2 - 4 * feval(func, p2) * d);


        if abs(b - D) < abs(b + D)

            E = b + D;

        else

            E = b - D;

        end
        h = -2 * feval(func, p2) / E;
        p = p2 + h;

        % Storing the estimated root and approximation error
        estimated_roots = [estimated_roots, p];
        approx_errors = [approx_errors, abs(h)];

        if abs(h) < TOL
            disp(p);
            break;
        end

        p0 = p1;
        p1 = p2;
        p2 = p;
        h1 = p1 - p0;
        h2 = p2 - p1;
        DELTA1 = (feval(func, p1) - feval(func, p0)) / h1;
        DELTA2 = (feval(func, p2) - feval(func, p1)) / h2;
        d = (DELTA2 - DELTA1) / (h2 + h1);
        i = i + 1;
    end

    if i > N0
        formatSpec = 'The method failed after N0 iterations, N0= %d \n';
        fprintf(formatSpec, N0);
    end
end
```