# 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

This project was code 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 MATLAB 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.

# 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

A number and a line

Description automatically generated with medium confidence

A diagram of a curve

Description automatically generated

1. Find the appropriate initial angle θ0, 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 𝜀𝑠 = 0.01%

Given:

v0 = 30/ms

y0 = 1.8m

y = 1m

g = 9.81 m/s2

We can use the following code to find our first initial angle for θ0:

A computer screen shot of a code

Description automatically generated

A computer code with black text

Description automatically generated

Using these as our θ0:

[xL, xu] = [0.661967, 0.663540]

For the Bisection method, we get the following:

A screenshot of a computer

Description automatically generated

We can se here that both the root and the error are able to converge.

For the False Position method, we get the following: A screenshot of a computer

Description automatically generated

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:

A screenshot of a computer

Description automatically generated

This equation will then be used for the fixed-point iteration method, along side the same initial value for x0:

A screenshot of a computer

Description automatically generated

Moving on, for the newton method, we first need to find the derivative of the equation. After that, we can plug it into code like so:

A white background with black text

Description automatically generated

Then, we just pass the same initial guess as always:

A screenshot of a computer

Description automatically generated

Using the secant method:

A screenshot of a computer

Description automatically generated

Using Modified Secant:

A screenshot of a computer

Description automatically generated

Using `fzero` from Octave:

A screenshot of a computer

Description automatically generated

Lastly, the Muller method:

A screenshot of a computer

Description automatically generated

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

Using a different initial guess:

A screenshot of a computer

Description automatically generated

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

A screenshot of a computer

Description automatically generated

1. 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.

1. Which algorithm is the fastest?

The fastest algorithm is the Newton algorithm as it is the fastest to reduce the error to approximate a value, and the one that takes the least amount of iterations.

# 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:

Where . Find the distance where the force is 1.25 N if and are C for a ring with a radius of 0.85 m.

You need to run all codes when possible.

A circle with a line and arrows

Description automatically generated with medium confidence

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

A white background with colorful text

Description automatically generated

Which yields the following possible roots:

A white background with black text

Description automatically generated

[x1, x2] = [0.240240, 0.242242]

[x1, x2] = [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~:

A screenshot of a computer

Description automatically generated

# Problem 4

For fluid flow in pipes, friction is described by a dimensionless number, the Fanning friction factor . 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 . A formula that predicts given is the von Karman equation:

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 given a user-supplied value of 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:

A screenshot of a computer code

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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:

A screenshot of a computer

Description automatically generated