UNIVERSITÀ DI TORINO

Fisica Torino

NUMERICAL ALGORITHMS FOR PHYSICS

FRANCESCO MARCHISOTTI

# REALISTIC PROJECTILE MOTION

FINAL PROJECT

SUPERVISOR:
ANDREA MIGNONE

ACADEMIC YEAR 2023/2024

## Abstract

The problem of the projectile motion is one that every physicist encounters during their first year. It's quite a simple problem, but it always gets solved neglecting air friction. This is because, when taking air friction into account, the problem gets much more complex and does not have an analytical solution.

However, such a solution can be found numerically, and the question of "how far will the projectile travel given these initial conditions" is quite an easy one to answer. The problem gets much more complicated when the variable to solve for is the initial launch angle given a target distance, and the solution presents a few unique challenges that can enrich a physicist's numerical toolbox.

# Contents



# List of Figures

# 1 Problem statement

Given a distance $L$ from the target and a target elevation $h$ (not necessarily positive), find the angle needed for a projectile launched with speed $v_0 = |\vec{v}(t=0)|$ to hit the target.

# 2 Equations of motion

The derivation of the equations of motion for a projectile with air friction is relatively straightforward. We start by considering the total sum of the forces on the projectile:

$$\vec{F}_{net} = \vec{F}_g + \vec{F}_{drag} = -mg\hat{y} - B|\vec{v}|\vec{v} \tag{1}$$

where $g = 9.81\,\mathrm{m/s^2}$.
Since $\vec{v} = (\dot{x}\hat{x} + \dot{y}\hat{y})$, and, by Newton's second law, $\vec{F}_{net} = m\vec{a}$:

$$m\vec{a} = -mg\hat{y} - B|\vec{v}|\left(\dot{x}\hat{x} + \dot{y}\hat{y}\right)$$

or, in components:

$$\begin{cases} m\ddot{x} = -B|\vec{v}|\dot{x} \\ m\ddot{y} = -mg - B|\vec{v}|\dot{y} \end{cases} \tag{2}$$

with $|\vec{v}| = \sqrt{\dot{x}^2 + \dot{y}^2}$.

## 2.1 Nondimensionalisation

In order to solve the equations numerically, we shall now define new nondimensional variables. These variables measure their respective quantities not in SI units, but relative to a characteristic value for that quantity (denoted by a Greek letter). When producing plots and reading results, the values obtained from the algorithm are then multiplied by the appropriate dimensional factors.

$$\begin{cases} \vec{x} = \chi\vec{x}' \\ t = \tau t' \\ m = \mu m' \\ B = \frac{\mu}{\chi}B' \\ g = \frac{\chi}{\tau^2}g' \end{cases}$$

which give us the dimensionless equations:

$$\begin{cases} m'\ddot{x}' = -B'|\vec{v}'|\dot{x}' \\ m'\ddot{y}' = -m'g' - B'|\vec{v}'|\dot{y}' \end{cases}$$

where a dot above a variable now represents the derivative with respect to $t'$ of that variable.

We can choose the values of the dimensional constants such that many coefficients become 1, or we set them to a natural characteristic dimension of the system:

$$\begin{cases} \chi = L \\ \mu \text{ s.t. } m' = 1 \Rightarrow \mu = m \\ \tau \text{ s.t. } g' = 1 \Rightarrow \tau = \sqrt{L/g} \end{cases}$$

and thus we obtain the final form of our equations (where the $'$ have been dropped since they are no longer needed):

$$\begin{cases} \ddot{x} = -B|\vec{v}|\dot{x} \\ \ddot{y} = -1 - B|\vec{v}|\dot{y} \end{cases} \tag{3}$$

The system of two second order ODEs can now be rewritten as a system of four first order ODEs:

$$\begin{cases} \dot{x} = u \\ \dot{y} = v \\ \dot{u} = -Bu\sqrt{u^2+v^2} \\ \dot{v} = -1 - Bv\sqrt{u^2+v^2} \end{cases} \tag{4}$$

## 2.2 Exact solution

These equations do not have an analytical solution, but such a solution exists when air friction is neglected (i.e. when $B = 0$). The full Cauchy problem is defined by giving the boundary conditions:

$$\begin{cases} x(t=0) = 0 \\ y(t=0) = 0 \\ y(x=1) = 0 \\ \vec{v}(t=0) = u_0\hat{x} + v_0\hat{y} = |\vec{v}_0| \left(\cos\theta\hat{x} + \sin\theta\hat{y}\right) \end{cases}$$

and the solution is

$$\begin{cases} x(t) = u_0 t \\ y(t) = -\frac{1}{2}t^2 + v_0 t \end{cases}$$

The equation for the trajectory can be obtained by inverting the first equation, thus getting $t(x)$, and substituting it in the second one

$$y(x) = -\frac{1}{2}\left(\frac{x}{u_0}\right)^2 + \frac{v_0}{u_0}x \tag{5}$$

Imposing the second boundary condition, we should get the initial launch angle needed to hit the target

$$\begin{aligned} y(x=1) &= \frac{2u_0v_0 - 1}{2u_0^2} = 0 \\ &\to u_0v_0 = \frac{1}{2} \\ &\to |\vec{v}_0|^2 \cos\theta \sin\theta = \frac{|\vec{v}_0|^2}{2} \sin 2\theta = \frac{1}{2} \\ &\to \theta_1 = \frac{1}{2} \arcsin \frac{1}{|\vec{v}_0|^2}, \quad \theta_2 = \frac{\pi}{2} - \frac{1}{2} \arcsin \frac{1}{|\vec{v}_0|^2} \end{aligned} \tag{6}$$

This result provides a benchmark against which to test the numerical solution.

# 3 Numerical algorithm

This section aims to explain the numerical algorithm used to find a solution to the problem. The basic code structure involves finding the roots of the residual function using a root solver (§ 3.4).

## 3.1 Residual function

The residual function takes in a value of $\theta$ and performs the following operations:

1. integrates the equations using a $4^{\text{th}}$-order Runge-Kutta method (§ 3.2), stopping the integration when the x-coordinate of the projectile exceeds the x-coordinate of the target
2. stores the x- and y-coordinates of the last projectile position, along with $n$ previous positions, where $n$ is the order of the polynomial interpolation
3. interpolates with a polynomial over the last $n+1$ positions of the projectile
4. returns the difference between the value of the polynomial interpolation at $x = L$ and $h$

## 3.2 Runge-Kutta

Given the IVP

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t), \quad y(t_0) = y_0$$

and a step size $\Delta t$, the Runge-Kutta approximation of $y(t_{n+1})$ is defined as

$$\begin{aligned} y_{n+1} &= y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ t_{n+1} &= t_n + \Delta t \end{aligned}$$

$k_i$ is the slope of $y(t)$ estimated at four different points:

- $k_1 = f(t_n, y_n)$ is the slope at the beginning of the interval, using $y$
- $k_2 = f(t_n + \Delta t/2, y_n + hk_1/2)$ is the slope at the midpoint of the interval, using $y$ and $k_1$
- $k_3 = f(t_n + \Delta t/2, y_n + hk_2/2)$ is the slope at the midpoint of the interval, using $y$ and $k_2$
- $k_4 = f(t_n + \Delta t, y_n + hk_3)$ is the slope at the end of the interval, using $y$ and $k_3$

When the four slopes are averaged in calculating $y_{n+1}$, greater weight is given to the slopes at the midpoint. If $f(y,t) = f(t)$, so that the equation is equivalent to a simple integral, then RK4 is Simpson's integration rule.

## 3.3 Polynomial interpolation

The polynomial interpolation function solves a linear system of $n+1$ equations (using Gauss Elimination to reduce the coefficient matrix to upper triangular form) in order to find the coefficients of the polynomial of order $n$ passing through all the points. It then computes and returns the value of the polynomial at the specified value of $x$.

## 3.4 Root finder

The root finder algorithm first brackets the roots, then uses the secant method to find the roots in the intervals identified by the bracketing function.

**Bracketing function** The bracketing function loops through a number of subintervals of the specified range and returns the boundary of all the intervals in which the function changes sign an even number of times.

**Secant method** The secant method is a root-finding algorithm that approximates the function in a specified range with the segment passing through the value assumed by the function at the range boundary. The algorithm then splits the initial range at the point where the segment crosses the x-axis and checks in which of the two new sub-ranges the function changes sign an even number of times and it repeats using this sub-range as the initial range until the range width is smaller than the required tolerance.

# 4 Numerical solution

**Shooting method** Numerically, a boundary value problem is quite challenging in and of its own: a typical method for solving such a problem is the shooting method, that reduces the boundary value problem to an initial value problem. This method involves finding the solution to the IVP for different initial conditions ("shooting"), until one is found that also matches the boundary condition of the BVP.

In mathematical terms, given the BPV

$$\begin{cases} y''(t) = f(t, y(t), y'(t)) \\ y(t_0) = y_0 \\ y(t_1) = y_1 \end{cases}$$

let $y(t; a)$ be a solution of the IVP

$$\begin{cases} y''(t) = f(t, y(t), y'(t)) \\ y(t_0) = y_0 \\ y'(t_0) = a \end{cases}$$

if $y(t_1; a) = y_1$, then $y(t; a)$ is also a solution of the BVP.
This is equivalent to finding the roots of the residual function

$$F(a) = y(t_1; a) - y_1$$

or, in terms of the projectile motion notation

$$F(\theta) = y(x = 1; \theta) - h$$

**Polynomial interpolation** Finding the launch angle to shoot a projectile at in order to hit a target has an added difficulty: the boundary condition is given with respect to a dependent variable (the x-position) rather than the independent one (time). This means that, since numerical integration solves the ODEs at discrete time steps, it's not possible to know the value of $y(x = 1; \theta)$ for every possible value of $\theta$.

A possible solution to this problem that does not excessively increase the computational cost is to replace $y(x = 1; \theta)$ in the residual function with the straight line passing through the point just before the x-coordinate of the target and the point just after, evaluated at the x-coordinate of the target. However, especially for larger time steps, this is cause of a non-negligible error, as shown in Figure 5, generated using a time step size of $1.0 \times 10^{-3}$, resulting in an error of about $10^{-7}$. Figure 5 is obtained with the method explained in section 4.1.

Since the analytical solution of the frictionless problem shown in equation (5) is a parabola, using a $2^{\text{nd}}$ order polynomial interpolation would solve exactly for the trajectory, giving an error in the machine precision order. In light of this result, a second order polynomial is used in this project to interpolate the last three points of the trajectory.

## 4.1 Comparison with analytical solution

In order to test the accuracy of the algorithm, it is useful to produce a plot showcasing the absolute difference between the analytical solution and the frictionless numerical solution with the following boundary conditions:

$$\begin{cases} x(t=0) = 0.0\,\mathrm{m} \\ y(t=0) = 0.0\,\mathrm{m} \\ |\vec{v}(t=0)| = 10.0\,\mathrm{m/s} \\ y(x=10.0\,\mathrm{m}) = 0.0\,\mathrm{m} \end{cases}$$

The appropriate range of angles to search the roots of the residual function in can be estimated by creating the shooting plot, in which several trajectories (each corresponding to a different launch angle) are computed and plotted.

This range can be refined further by looking at the residual plot, where the zeros of the residual function are clearly shown.

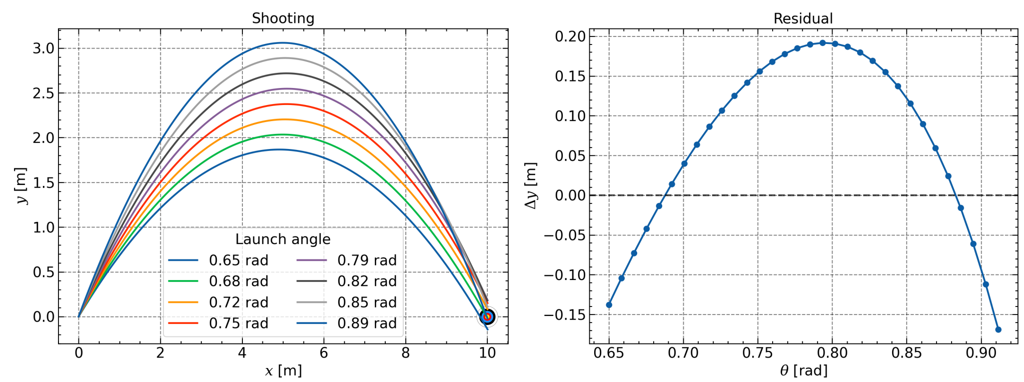


Figure 1: Shooting and residual plots without friction.

As seen in Figure 2, the difference over the course of the whole trajectory is at most of the order of $10^{12}$, i.e. the machine precision. This is because, since the analytical solution is only a function of the second order, the RK4 method (combined with a $2^{\text{nd}}$ order polynomial interpolation) solves exactly the problem.

Table 1 shows the obtained results.

| | $\theta_1$ | $\theta_2$ |
|---|---|---|
| analytical | 0.68777523221 | 0.88302109458 |
| numerical | 0.68777523222 | 0.88302109459 |

Table 1: Comparison between analytical and numerical launch angles.

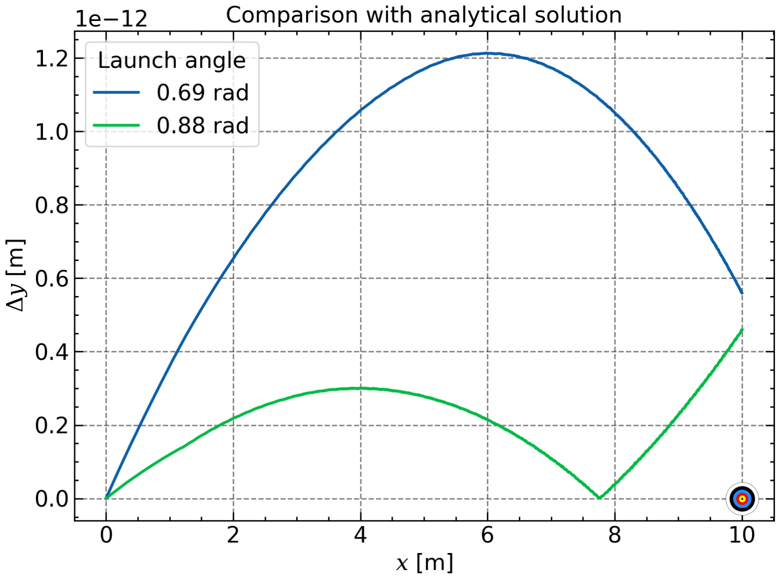


Figure 2: Absolute difference between numerical and analytical solutions.

## 4.2 Time step size

In order to further improve the results of the algorithm, it is possible to find the time step size that gives the most accurate result: by looking at Figure 3, in which the normalised $\theta$ is plotted against the time step size, it appears clear that using a time step smaller than about $2 \times 10^{-5}$ does not yield a better result, and only serves to increase the computational load. Therefore, a time step of $1.0 \times 10^{-5}$ is used throughout this project (except where otherwise stated). If, instead of accuracy, a faster execution is favoured, then a time step in the range $2 \times 10^{-3} - 2 \times 10^{-4}$ would be a better choice.

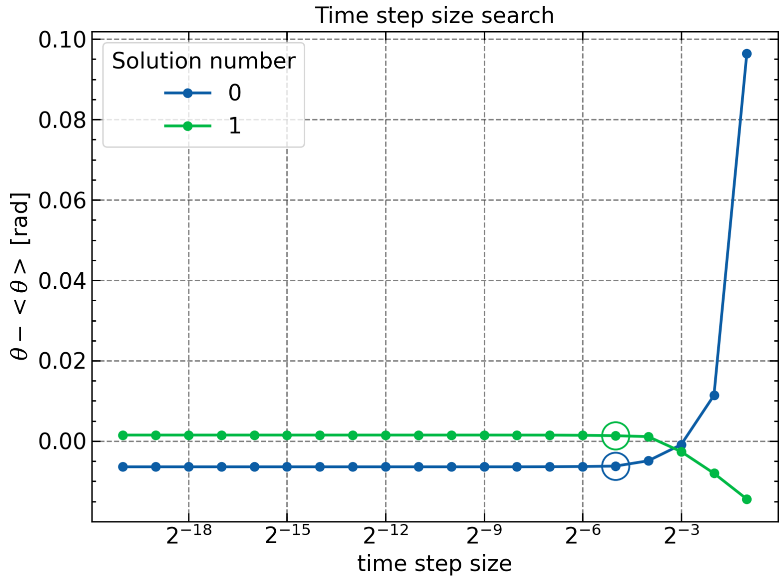


Figure 3: Optimal time step size search.

### 4.3 Solutions with friction

Now that the algorithm has been tested against the analytical benchmark, it's safe to assume that the numerical solution is precise enough and provides a meaningful answer. For instance, given a drag coefficient of $B = 4.0 \times 10^{-5}\,\text{kg/m}$, unit mass, and boundary conditions

$$\begin{cases} x(t=0) = 0.0\,\text{m} \\ y(t=0) = 0.0\,\text{m} \\ |\vec{v}(t=0)| = 9.9\,\text{m/s} \\ y(x=10.0\,\text{m}) = -0.2\,\text{m} \end{cases} \tag{7}$$

The roots of the residual function found by the algorithm are

| Sol. number | $\theta$ [rad] |
|---|---|
| 1 | 0.67833021369 |
| 2 | 0.87259533450 |

Table 2: Launch angles with friction.

and the trajectories obtained with those angles are shown in Figure 4.

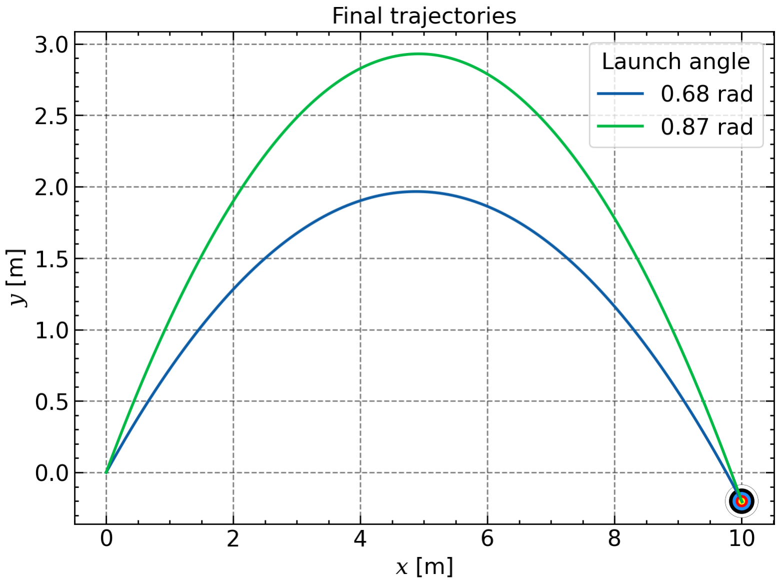


Figure 4: Trajectories with friction.

# A Extra plots

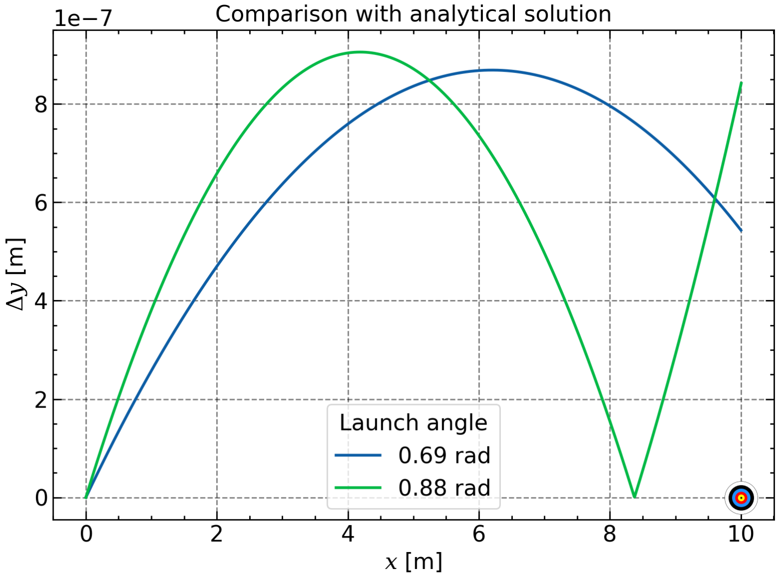


Figure 5: Comparison with analytical solution (linear interpolation).

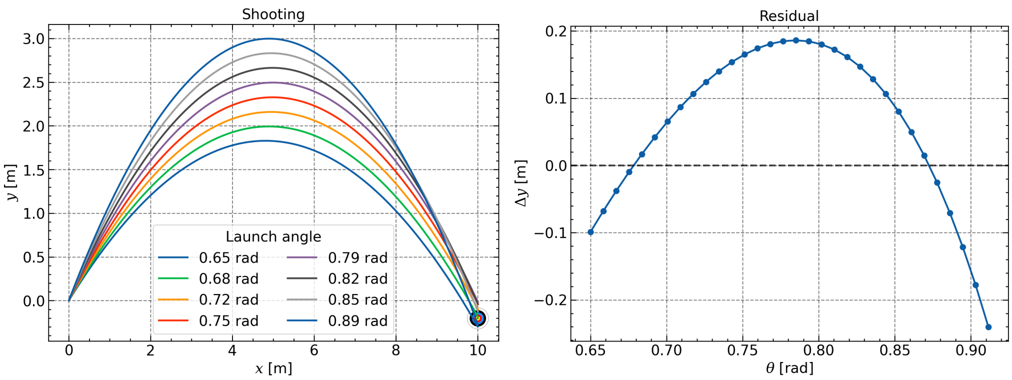


Figure 6: Shooting and residual plots with drag.

# B Code listing

## B.1 ProjectileMotion/src/main.cpp

```
/**
 * @file main.cpp
 *
 * @author Francesco Marchisotti
 *
 * @brief Main file for the course's final project.
 *
 * @date 2024-05-15
 */

#include "../../Libs/include/exception.hpp"
#include "../../Libs/include/lin_alg.hpp"
#include "../../Libs/include/ode_solver.hpp"
#include "../../Libs/include/root_finder.hpp"

#include <cmath>
#include <fstream>
#include <iostream>

using std::cerr;
using std::cin;
using std::cout;
using std::endl;

#define FRICTION 1
const static int gOrder = 2;  //<! Selects order of polynomial interpolation

int numIntegrations = 0;  //!< Number of integrations of the ODEs performed

double g_dt = 1.0e-5;

// Problem data
#if FRICTION
const static double B     = 4.0e-5;  //!< Drag coefficient [kg/m]
const static double V0    = 9.90;    //!< Initial velocity [m/s]
const static double L     = 10.0;    //!< Target distance  [m]
const static double YTarg = -0.2;    //!< Target height    [m]
double gTheta;                       //!< Initial launch angle
#else
const static double B     = 0.0;   //!< Drag coefficient [kg/m]
const static double V0    = 10.0;  //!< Initial velocity [m/s]
const static double L     = 10.0;  //!< Target distance  [m]
const static double YTarg = 0.0;   //!< Target height    [m]
double gTheta;                     //!< Initial launch angle
```

```
#endif

// Dimensional factors
const static double chi = L;              //!< Space dimensional factor [m]
const static double mu  = 1.0;            //!< Mass dimensional factor [kg]
const static double g   = 9.81;           //!< Gravity [m/s^2]
const static double tau = sqrt(chi / g);  //!< Time dimensional factor [s]

const static double b     = B * chi / mu;    //!< Adimensional friction
const static double v0    = V0 * tau / chi;  //!< Adimensional speed
const static double xTarg = 1.0;             //!< Adimensional target distance
const static double yTarg = YTarg / L;       //!< Adimensional target height

/**
 * @brief Prints problem data and dimensional constants to file.
 */
void printConstants();

/**
 * @brief     This function returns the linear interpolation between two points
 *            evaluated at a certain x.
 *
 * @param[in] x   The point at which to evaluate the interpolation.
 * @param[in] x1  The x-coordinate of the first point.
 * @param[in] y1  The y-coordinate of the first point.
 * @param[in] x2  The x-coordinate of the second point.
 * @param[in] y2  The y-coordinate of the second point.
 *
 * @return    The interpolated line evaluated at x.
 */
double linearInterp(const double &x, const double &x1, const double &y1,
                    const double &x2, const double &y2);

/**
 * @brief               This function returns the polinomial interpolation
 *                                               between a sufficient number of points evaluated at a
 *                                               certain x.
 *
 * @param[in] x         The point at which to evaluate the interpolation.
 * @param[in] xLast     The x-coordinates of the first ``order`` points.
 * @param[in] yLast     The y-coordinates of the first ``order`` points.
 * @param[in] xCurrent  The x-coordinate of the last point.
 * @param[in] yCurrent  The y-coordinate of the last point.
 * @param[in] order     The order or the polynomial.
 *
 * @return    The interpolated line evaluated at x.
 */
double polInterp(const double &x, double xLast[], double yLast[],
```

```
                 const double &xCurrent, const double &yCurrent,
                 const int &order);

/**
 * @brief      Right Hand Side of the system of ODEs.
 *
 * @param[in]  Y  The input values of the variables.
 * @param[out] R  The output values of the variables.
 */
void RHS(const double &t, double Y[], double R[]);

/**
 * @brief     Computes the exact trajectory.
 *
 * @param[in] x          The point at which to evaluate the trajectory.
 * @param[in] solNumber  The number of the solution.
 *                       - 0: the solution with theta < M_PI / 4
 *                       - 1: the solution with theta > M_PI / 4
 *
 * @return    double
 */
double exact(const double &x, const int &solNumber);

/**
 * @brief     Produces the convergence plot.
 *
 * @param[in] dt_0     The first (and largest) value of dt.
 * @param[in] nPoints  The number of dts to explore.
 * @param[in] factor   The factor that scales dt.
 */
void convergence(const double &dt_0, const int &nPoints,
                 const double factor = 0.5);

/**
 * @brief     Generates data for shooting plot.
 *
 * @param[in] thetaMin  Lower bound for launch angle.
 * @param[in] thetaMax  Upper bound for launch angle.
 * @param[in] nTheta    Number of launch angles explored.
 */
void shootingPlot(const double &thetaMin, const double &thetaMax,
                  const double &nTheta);

/**
 * @brief          Integration interface.
 *
 * @param[in,out]  y        Array with the variables.
 * @param[in]      theta    Launch angle.
 * @param[out]     xLast    Array with x-position at previous time step.
```

```
 * @param[out]     yLast    Array with y-position at previous time step.
 * @param[in]      order    Numer of previous times to save.
 * @param[in]      outFile  Output file.
 */
void integrate(double y[], const double &theta, double xLast[], double yLast[],
               const int &order, std::ofstream &outFile);

/**
 * @overload
 *
 * @brief          Integration interface (without file output).
 *
 * @param[in,out]  y      Array with the variables.
 * @param[in]      theta  Launch angle.
 * @param[out]     xLast  Array with x-position at previous time step.
 * @param[out]     yLast  Array with y-position at previous time step.
 * @param[in]      order  Numer of previous times to save.
 */
void integrate(double y[], const double &theta, double &xLast, double &yLast);

/**
 * @overload
 *
 * @brief          Integration interface (without past values).
 *
 * @param[in,out]  y        Array with the variables.
 * @param[in]      theta    Launch angle.
 * @param[in]      outFile  Output file.
 */
void integrate(double y[], const double &theta, std::ofstream &outFile);

/**
 * @overload
 *
 * @brief          Integration interface (without past values and file output).
 *
 * @param[in,out]  y      Array with the variables.
 * @param[in]      theta  Launch angle.
 */
void integrate(double y[], const double &theta);

/**
 * @brief          Function that performs the integration and prints to file.
 *
 * @param[in]      RHSFunc     Right Hand Side of the system of ODEs.
 * @param[in, out] y           Array with the variables. Should be already
 *                             initialised.
 * @param[in]      nEq         Number of equations in the system.
 * @param[in]      t           Starting time of the integration.
```

```
 * @param[in]      dt          Time step size.
 * @param[in]      theta       Launch angle.
 * @param[out]     xLast       Array with x-position at previous time step.
 * @param[out]     yLast       Array with y-position at previous time step.
 * @param[in]      order       Numer of previous times to save.
 * @param[in]      maxStep     Maximum number of integration steps.
 * @param[in]      outFile  Output file.
 */
void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
                 double y[], const int &nEq, double t, const double &dt,
                 const double &theta, double xLast[], double yLast[],
                 const int &order, const int &maxStep, std::ofstream &outFile);

/**
 * @overload
 *
 * @brief          Function that performs the integration and prints to file
 *                 with exact difference.
 *
 * @param[in]      RHSFunc     Right Hand Side of the system of ODEs.
 * @param[in]      exactFunc   Exact trajectory.
 * @param[in, out] y           Array with the variables. Should be already
 *                             initialised.
 * @param[in]      sol_number  Solution number.
 * @param[in]      theta       Launch angle.
 * @param[in]      outFile  Output file.
 */
void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
                 double (*exactFunc)(const double &x, const int &sol_number),
                 double y[], const int &sol_number, const double &theta,
                 std::ofstream &outFile);

/**
 * @brief     Residual function for the BVP.
 *
 * @param[in] theta  Launch angle.
 *
 * @return    y(x = 1) - yTarg
 */
double Residual(const double &theta);

int main() {
  convergence(0.5, 20);

#if FRICTION
  cout << "===== FRICTION =====" << endl << endl;
#else
  cout << "===== NO FRICTION =====" << endl << endl;
  for (int i = 0; i < 2; i++) {
```

```
    double theta =
      0.5 * (i * M_PI + (i == 0 ? 1.0 : -1.0) * asin(1 / (v0 * v0)));
    cout.precision(16);
    cout << "Analytical theta" << i << " = " << theta << endl;
  }
  cout << endl;
#endif

  printConstants();

  const double thetaMin = 0.65;    // Minimum launch angle
  const double thetaMax = 0.92;    // Maximum launch angle
  const int nTheta      = 32;      // Number of launch angles explored
  const double thetaTol = 1.0e-7;  // Tolerance for root searching

  shootingPlot(thetaMin, thetaMax, nTheta);

  /* +------------------+
   * | Problem solution |
   * +------------------+ */
  numIntegrations = 0;
  double roots[4];
  int nRoots = -1;
  try {
    findRoots(Residual, thetaMin, thetaMax, thetaTol, roots, nRoots, 4,
              "secant");

    cout << "Integrations performed: " << numIntegrations << endl;

    cout.precision(16);
    cout << "Optimal thetas [rad] (+/- " << thetaTol << "): ";
    printVector(roots, nRoots);
  } catch (std::exception &err) {
    cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
  } catch (...) {
    cerr << "Sorry, could not recognise the error." << endl;
  }

  std::ofstream finTraj;
#if FRICTION
  finTraj.open("data/finTraj.csv");
  finTraj << "t,x,y,u,v,theta" << endl;

  for (int i = 0; i < nRoots; i++) {
    double y[4];
    integrate(y, roots[i], finTraj);
  }
#else
  finTraj.open("data/noFriction.csv");
```

```
  finTraj << "t,x,delta_y,u,v,theta" << endl;
  for (int i = 0; i < nRoots; i++) {
    double y[4];
    integration(RHS, exact, y, i, roots[i], finTraj);
  }
#endif
  finTraj.close();

  return 0;
}

void printConstants() {
  std::ofstream out;
  try {
    out.open("data/constants.csv");
    if (!out.good()) throw exception("Invalid file.");

    out << "chi,tau,mu,B,b,V0,YTarg" << endl;
    out << chi << "," << tau << "," << mu << "," << B << "," << b << "," << V0
        << "," << YTarg << endl;

    out.close();
  } catch (std::exception &err) {
    cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
  } catch (...) {
    cerr << "Sorry, could not recognise the error." << endl;
  }

  out.close();
}

double linearInterp(const double &x, const double &x1, const double &y1,
                    const double &x2, const double &y2) {
  double m = (y2 - y1) / (x2 - x1);
  double q = y1 - m * x1;
  double y = m * x + q;
  return y;
}

double polInterp(const double &x, double xLast[], double yLast[],
                 const double &xCurrent, const double &yCurrent,
                 const int &order) {
  const int nPoints = order + 1;

  double **M;
  M    = new double *[nPoints];
  M[0] = new double[nPoints * nPoints];
  for (int i = 1; i < nPoints; i++) M[i] = M[i - 1] + nPoints;

```

```
  double *v;
  v = new double[nPoints];

  // Define coefficient matrix
  for (int i = 0; i < nPoints; i++) {
    M[i][nPoints - 1] = 1;
    if (i != nPoints - 1) v[i] = yLast[i];
    else v[i] = yCurrent;
    for (int j = nPoints - 2; j >= 0; j--)
      if (i != nPoints - 1) M[i][j] = M[i][j + 1] * xLast[i];
      else M[i][j] = M[i][j + 1] * xCurrent;
  }

  double *coeffs;  //<! Array with the coefficients of the polynomial
  coeffs = new double[nPoints];

  solveLinSystem(M, v, coeffs, nPoints);

  delete[] M[0];
  delete[] M;
  delete[] v;

  double value    = 0.0;
  double powerOfX = 1.0;
  for (int i = nPoints - 1; i >= 0; i--) {
    value += coeffs[i] * powerOfX;
    powerOfX *= x;
  }

  return value;
}

void RHS(const double &t, double Y[], double R[]) {
  double u = Y[2];
  double v = Y[3];

  double mod_v = sqrt(u * u + v * v);

  R[0] = u;
  R[1] = v;
  R[2] = -b * u * mod_v;
  R[3] = -1.0 - b * u * mod_v;
}

double exact(const double &x, const int &solNumber) {
  if (solNumber != 0 && solNumber != 1)
    throw std::invalid_argument("solNumber must be 0 or 1");

  double theta = 0.5 * (solNumber * M_PI +
```

```
                            (solNumber == 0 ? 1.0 : -1.0) * asin(1 / (v0 * v0)));
  double u0    = v0 * cos(theta);
  return (-0.5 * (x / u0) * (x / u0) + tan(theta) * x);
}

void convergence(const double &dt_0, const int &nPoints, const double factor) {
  double store_g_dt = g_dt;
  g_dt              = dt_0;
  std::ofstream conv;
  conv.open("data/dt_search.csv");
  conv << "dt,theta1,theta2" << endl;
  for (int i = 0; i < nPoints; i++) {
    const double thetaMin = 0.65;    // Minimum launch angle
    const double thetaMax = 0.92;    // Maximum launch angle
    const double thetaTol = 1.0e-7;  // Tolerance for root searching

    /* +------------------+
     * | Problem solution |
     * +------------------+ */
    double roots[4];
    int nRoots = -1;
    try {
      findRoots(Residual, thetaMin, thetaMax, thetaTol, roots, nRoots, 4,
                "secant");

      conv.precision(12);
      conv << g_dt << "," << roots[0] << "," << roots[1] << endl;

      cout << "dt = " << g_dt << " roots = ";
      printVector(roots, nRoots);
    } catch (std::exception &err) {
      cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
    } catch (...) {
      cerr << "Sorry, could not recognise the error." << endl;
    }

    g_dt *= factor;
  }
  conv.close();
  g_dt = store_g_dt;
}

void shootingPlot(const double &thetaMin, const double &thetaMax,
                  const double &nTheta) {
  const double dTheta = (thetaMax - thetaMin) / nTheta;

  std::ofstream shooting, residual;
  shooting.open("data/shooting.csv");
  residual.open("data/residual.csv");
```

```
  shooting << "t,x,y,u,v,theta" << endl;  // Output csv header
  residual << "theta,res" << endl;        // Output csv header
  for (int i = 0; i < nTheta; i++) {
    double theta = thetaMin + i * dTheta;
    double y[4];

    integrate(y, theta, shooting);

    residual << theta << "," << Residual(theta) << endl;
  }
  shooting.close();
  residual.close();
}

void integrate(double y[], const double &theta, double xLast[], double yLast[],
               const int &order, std::ofstream &outFile) {
  const double y0[] = {0.0, 0.0, v0 * cos(theta), v0 * sin(theta)};
  const int nEq =
    static_cast<int>(sizeof(y0)) / static_cast<int>(sizeof(y0[0]));
  for (int i = 0; i < nEq; i++) y[i] = y0[i];

  double t0         = 0.0;
  const double dt   = g_dt;
  const int maxStep = int(2 / dt);

  integration(RHS, y, nEq, t0, dt, theta, xLast, yLast, order, maxStep,
              outFile);
}

void integrate(double y[], const double &theta, double xLast[], double yLast[],
               const int &order) {
  std::ofstream dummyOutfile;
  integrate(y, theta, xLast, yLast, order, dummyOutfile);
}

void integrate(double y[], const double &theta, std::ofstream &outFile) {
  double dummyLast[2];
  integrate(y, theta, dummyLast, dummyLast, 1, outFile);
}

void integrate(double y[], const double &theta) {
  std::ofstream dummyOutfile;
  double dummyLast[2];
  integrate(y, theta, dummyLast, dummyLast, 1, dummyOutfile);
}

void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
                 double y[], const int &nEq, double t, const double &dt,
                 const double &theta, double xLast[], double yLast[],
```

```
                 const int &order, const int &maxStep, std::ofstream &outFile) {
  outFile << t << "," << y[0] << "," << y[1] << "," << y[2] << "," << y[3]
          << "," << theta << endl;

  numIntegrations++;
  int stepCounter    = 0;
  bool exitCondition = false;
  while (stepCounter < maxStep && !exitCondition) {
    for (int i = 0; i < order - 1; i++) {
      xLast[i] = xLast[i + 1];
      yLast[i] = yLast[i + 1];
    }
    xLast[order - 1] = y[0];
    yLast[order - 1] = y[1];

    rk4Step(t, y, RHSFunc, dt, nEq);
    t += dt;
    stepCounter++;

    outFile << t << "," << y[0] << "," << y[1] << "," << y[2] << "," << y[3]
            << "," << theta << endl;

    if (xLast[0] < xTarg && y[0] > xTarg) exitCondition = true;
  }
}

void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
                 double (*exactFunc)(const double &x, const int &sol_number),
                 double y[], const int &sol_number, const double &theta,
                 std::ofstream &outFile) {
  const double y0[] = {0.0, 0.0, v0 * cos(theta), v0 * sin(theta)};
  const int nEq =
    static_cast<int>(sizeof(y0)) / static_cast<int>(sizeof(y0[0]));
  for (int i = 0; i < nEq; i++) y[i] = y0[i];

  double t         = 0.0;
  const double dt   = g_dt;
  const int maxStep = int(2 / dt);

  outFile << t << "," << y[0] << "," << fabs(y[1] - exactFunc(y[0], sol_number))
          << "," << y[2] << "," << y[3] << "," << theta << endl;

  numIntegrations++;
  int stepCounter    = 0;
  bool exitCondition = false;
  while (stepCounter < maxStep && !exitCondition) {
    rk4Step(t, y, RHSFunc, dt, nEq);
    t += dt;
    stepCounter++;
```

```
    outFile << t << "," << y[0] << ","
            << fabs(y[1] - exactFunc(y[0], sol_number)) << "," << y[2] << ","
            << y[3] << "," << theta << endl;

    if (y[0] > xTarg) exitCondition = true;
  }
}

double Residual(const double &theta) {
  double y[4];
  double xLast[64], yLast[64];
  if (gOrder > 64) throw exception("gOrder must be at most 64.");
  integrate(y, theta, xLast, yLast, gOrder);

  double xCurrent = y[0], yCurrent = y[1];

  return polInterp(xTarg, xLast, yLast, xCurrent, yCurrent, gOrder) - yTarg;
}
```

## B.2 Libs/include/exception.hpp

```
/**
 * @file    exception.hpp
 *
 * @author  Francesco Marchisotti
 *
 * @brief   Implements the class exception
 *
 * @date    2024-05-08
 */
#pragma once

#include <iostream>

class exception : public std::exception {
public:
  /**
   * @brief Constructor (C++ STL strings).
   *
   * @param message The error message.
   */
  explicit exception(const std::string& message)
    : msg_(message) {}

  /**
   * @brief Destructor.
   *
   * Virtual to allow for subclassing.
```

```
   */
  virtual ~exception() noexcept {}

  /**
   * @brief Returns a pointer to the (constant) error description.
   *
   * @return A pointer to a const char*. The underlying memory is in posession
   *         of the exception object. Callers must not attempt to free the
   *         memory.
   */
  virtual const char* what() const noexcept { return msg_.c_str(); }

protected:
  std::string msg_;  //!< Error message
};
```

## B.3 Libs/include/lin_alg.hpp

```
/**
 * @file    lin_alg.hpp
 *
 * @brief   This file implements linear algorithms.
 *
 * @author  Francesco Marchisotti
 *
 * @date    2023-11-24
 */

#pragma once

#include "../include/swap.hpp"

#include <iomanip>
#include <iostream>

/**
 * @brief      Prints a vector.
 *
 * @param[in]  v      The vector.
 * @param[in]  nRows  Size of the vector.
 *
 * @tparam     T      Type of the elements of the vector.
 */
template <class T>
void printVector(T v[], const int& nRows) {
  std::cout << "{";
  for (int i = 0; i < nRows; i++) {
    std::cout << v[i];
    if (i != nRows - 1) std::cout << ", ";
```

```
  }
  std::cout << "}" << std::endl;
}
```

## B.4 Libs/include/ode_solver.hpp

```
/**
 * @file    ode_solver.hpp
 *
 * @brief   Implementation of the ODE Solvers step functions.
 *
 * @author  Francesco Marchisotti
 *
 * @date    2024-05-01
 */
#pragma once

#include <iostream>

/**
 * @brief          Runge-Kutta 4 method step.
 *
 * Takes one step in time (or whatever the independent variable is) using
 * Runge-Kutta 4 method. The system of first order ODEs is dY_i/dt = R_i(t, Y).
 *
 * @param[in]      t        The value of time from which to take the step.
 * @param[in, out] Y        Array containing all the dependent variables.
 * @param[in]      RHSFunc  Pointer to the function containing all the Right
 *                          Hand Sides of the system of equations.
 * @param[in]      dt       The step size.
 * @param[in]      neq      The number of equations (ie the number of
 *                          independent variables) in the system.
 */
void rk4Step(const double& t, double Y[],
             void (*RHSFunc)(const double& t, double Y[], double RHS[]),
             const double& dt, const int& neq);
```

## B.5 Libs/src/ode_solver.cpp

```
#include "../include/ode_solver.hpp"

void rk4Step(const double &t, double Y[],
             void (*RHSFunc)(const double &t, double Y[], double RHS[]),
             const double &dt, const int &neq) {
  if (neq > 64) throw std::invalid_argument("neq must be less than 64");

  double Ystar[64], k1[64], k2[64], k3[64], k4[64];

  RHSFunc(t, Y, k1);
```

```

  for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + 0.5 * dt * k1[i];
  RHSFunc(t + 0.5 * dt, Ystar, k2);

  for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + 0.5 * dt * k2[i];
  RHSFunc(t + 0.5 * dt, Ystar, k3);

  for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + dt * k3[i];
  RHSFunc(t + dt, Ystar, k4);

  for (int i = 0; i < neq; i++)
    Y[i] += dt / 6.0 * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]);
}
```

## B.6 Libs/include/root_finder.hpp

```
/**
 * @file    root_finder.hpp
 *
 * @brief   Implementation of the root finder methods.
 *
 * @author  Paolino Paperino
 *
 * @date    2023-11-19
 */
#pragma once

#include <iomanip>
#include <iostream>

/**
 * @brief      Find the roots of a function in a given interval.
 *
 * Find the roots of a function f(x) in a given interval [xa, xb]
 * using the specified method. Works by first bracketing the roots
 * and then applying the method on every sub-interval.
 *
 * @param[in]  f       Pointer to the function.
 * @param[in]  dfdx    Pointer to the derivative of the function.
 * @param[in]  xa      Lower bound of the interval.
 * @param[in]  xb      Upper bound of the interval.
 * @param[in]  tol     x-tolerance.
 * @param[out] roots   Array with the roots of f(x).
 * @param[out] nRoots  The number of roots found.
 * @param[in]  N       The number of sub-intervals.
 * @param[in]  method  The root finding method. Accepted values are:
 *                     `bisection`, `falsePosition`, `secant`, `newton`.
 *
 * @return     flag
```

```
 *
 * @retval     0       Success.
 * @retval     1       Too many steps.
 * @retval     2       Initial interval doesn't contain any root.
 *
 * @throws     std::invalid_argument  Thrown if `N` > 128.
 * @throws     std::invalid_argument  Thrown if `method` is not among the
 *                                    accepted values.
 * @throws     std::runtime_error     Thrown if roots can't be found inside the
 *                                    interval.
 * @throws     std::runtime_error     Thrown if one of the root finders exceeded
 *                                    the maximum number of steps.
 */
int findRoots(double (*f)(const double& x), double (*dfdx)(const double& x),
              const double& xa, const double& xb, const double& tol,
              double roots[], int& nRoots, const int N = 128,
              const std::string method = "newton");

/**
 * @overload
 *
 * @brief      Find the roots of a function in a given interval (Newton's method
 *             not available).
 *
 * Find the roots of a function f(x) in a given interval [xa, xb]
 * using the specified method. Works by first bracketing the roots
 * and then applying the method on every sub-interval.
 *
 * @param[in]  f       Pointer to the function.
 * @param[in]  xa      Lower bound of the interval.
 * @param[in]  xb      Upper bound of the interval.
 * @param[in]  tol     x-tolerance.
 * @param[out] roots   Array with the roots of f(x).
 * @param[out] nRoots  The number of roots found.
 * @param[in]  N       The number of sub-intervals.
 * @param[in]  method  The root finding method. Accepted values are:
 *                     `bisection`, `falsePosition`, `secant`.
 *
 * @return     flag
 *
 * @retval     0       Success.
 * @retval     1       Too many steps.
 * @retval     2       Initial interval doesn't contain any root.
 *
 * @throws     std::invalid_argument  Thrown if `N` > 128.
 * @throws     std::invalid_argument  Thrown if `method` is not among the
 * accepted values.
 * @throws     std::runtime_error     Thrown if roots can't be found inside the
 *                                    interval.
```

```
 * @throws     std::runtime_error     Thrown if one of the root finders exceeded
 *                                   the maximum number of steps.
 */
int findRoots(double (*f)(const double& x), const double& xa, const double& xb,
              const double& tol, double roots[], int& nRoots, const int N = 128,
              const std::string method = "bisection");

/**
 * @brief      Bracket the roots of a function in a given interval [xa, xb].
 *
 * Works by subdividing the interval in a number of sub-intervals
 * and checking if the function changes sign (an odd number of
 * times) over this interval. If it does, then the interval contains
 * (at least) one root.
 *
 * @param[in]  f       Pointer to the function.
 * @param[in]  xa      Lower bound of the interval.
 * @param[in]  xb      Upper bound of the interval.
 * @param[out] xL      Array with the lower bound of the sub-interval containing
 *                     a root.
 * @param[out] xR      Array with the upper bound of the sub-interval containing
 *                     a root.
 * @param[in]  N       The number of sub-intervals.
 * @param[out] nRoots  The number of roots found.
 */
void bracket(double (*f)(const double& x), const double& xa, const double& xb,
             double xL[], double xR[], const int& N, int& nRoots);

/**
 * @brief      Find the root of a function f(x) in a given interval [xa, xb]
 *             using secant method.
 *
 * @param[in]  f     Pointer to the function.
 * @param[in]  xa    Lower bound of the interval.
 * @param[in]  xb    Upper bound of the interval.
 * @param[in]  xtol  x-tolerance.
 * @param[in]  ftol  f(x)-tolerance: the values of f(x) that are considered 0.
 * @param[out] root  The root of f(x).
 * @param[out] ntry  The number of iterations achieved.
 *
 * @return     flag
 *
 * @retval     0     Success.
 * @retval     1     Too many steps.
 *
 * @throws     std::runtime_error  Thrown if the maximum number of steps is
 *                                 exceeded.
 */
int secant(double (*f)(const double& x), double xa, double xb,
```

```
           const double& xtol, const double& ftol, double& root, int& ntry);

/**
 * @overload
 *
 * @brief      Find the root of a function f(x) in a given interval [xa, xb]
 *             using secant method.
 *
 * @param[in]  f     Pointer to the function.
 * @param[in]  xa    Lower bound of the interval.
 * @param[in]  xb    Upper bound of the interval.
 * @param[in]  xtol  x-tolerance.
 * @param[out] root  The root of f(x).
 *
 * @return     flag
 *
 * @retval     0     Success.
 * @retval     1     Too many steps.
 *
 * @throws     std::runtime_error  Thrown if the maximum number of steps is
 *                                 exceeded.
 */
int secant(double (*f)(const double& x), double xa, double xb,
           const double& xtol, double& root);

/**
 * @overload
 *
 * @brief      Find the root of a function f(x) in a given interval [xa, xb]
 *             using secant method.
 *
 * @param[in]  f     Pointer to the function.
 * @param[in]  xa    Lower bound of the interval.
 * @param[in]  xb    Upper bound of the interval.
 * @param[in]  xtol  x-tolerance.
 * @param[out] root  The root of f(x).
 * @param[out] ntry  The number of iterations achieved.
 *
 * @return     flag
 *
 * @retval     0     Success.
 * @retval     1     Too many steps.
 *
 * @throws     std::runtime_error  Thrown if the maximum number of steps is
 *                                 exceeded.
 */
int secant(double (*f)(const double& x), double xa, double xb,
           const double& xtol, double& root, int& ntry);

```

```
/**
 * @overload
 *
 * @brief      Find the root of a function f(x) in a given interval [xa, xb]
 *             using secant method.
 *
 * @param[in]  f     Pointer to the function.
 * @param[in]  xa    Lower bound of the interval.
 * @param[in]  xb    Upper bound of the interval.
 * @param[in]  xtol  x-tolerance.
 * @param[in]  ftol  f(x)-tolerance: the values of f(x) that are considered 0.
 * @param[out] root  The root of f(x).
 *
 * @return     flag
 *
 * @retval     0     Success.
 * @retval     1     Too many steps.
 *
 * @throws     std::runtime_error  Thrown if the maximum number of steps is
 *                                 exceeded.
 */
int secant(double (*f)(const double& x), double xa, double xb,
           const double& xtol, const double& ftol, double& root);
```

## B.7 Libs/src/root_finder.cpp

```
#include "../include/root_finder.hpp"

int findRoots(double (*f)(const double &x), double (*dfdx)(const double &x),
              const double &xa, const double &xb, const double &tol,
              double roots[], int &nRoots, const int N,
              const std::string method) {
  if (N > 128) throw std::invalid_argument("N must be less than 128");

  double xL[128], xR[128];

  bracket(f, xa, xb, xL, xR, N, nRoots);

  if (nRoots == 0) {
    throw std::runtime_error(
      "The supplied interval does not contain any roots.");
  }

  for (int i = 0; i < nRoots; i++)
    if (method == "bisection") bisection(f, xL[i], xR[i], tol, roots[i]);
    else if (method == "falsePosition")
      falsePosition(f, xL[i], xR[i], tol, roots[i]);
    else if (method == "secant") secant(f, xL[i], xR[i], tol, roots[i]);
    else if (method == "newton") newton(f, dfdx, xL[i], xR[i], tol, roots[i]);
```

```
    else throw std::invalid_argument("Invalid method argument.");

  return 0;
}

int findRoots(double (*f)(const double &x), const double &xa, const double &xb,
              const double &tol, double roots[], int &nRoots, const int N,
              const std::string method) {
  if (method == "newton")
    throw std::invalid_argument(
      "Newton method isn't available with this prototype.");

  return findRoots(f, nullptr, xa, xb, tol, roots, nRoots, N, method);
}

void bracket(double (*f)(const double &x), const double &xa, const double &xb,
             double xL[], double xR[], const int &N, int &nRoots) {
  double dx          = (xb - xa) / N;
  double xi          = xa;
  double xi_plus_one = xi + dx;
  int root_counter   = 0;

  double fL = f(xi), fR;
  for (int i = 0; i < N; i++) {
    fR = f(xi_plus_one);
    if (fL == 0.0 ||
        fL * fR < 0) {  // Check if there's a root in [xi, xi_plus_one)
      xL[root_counter] = xi;
      xR[root_counter] = xi_plus_one;

      root_counter++;
    }

    // Shift interval
    fL = fR;
    xi = xi_plus_one;
    xi_plus_one += dx;
  }

  nRoots = root_counter;
}

// ============================================================================
// Secant method
// ============================================================================

int secant(double (*f)(const double &x), double xa, double xb,
           const double &xtol, const double &ftol, double &root, int &ntry) {
  int max_ntry = 64;
```

```
  double fa    = f(xa);
  double fb    = f(xb);
  double dx    = xb - xa;

  // Handle fa, fb = 0
  if (fa == 0.0) {
    ntry = 0;
    root = xa;
    return 0;
  } else if (fb == 0.0) {
    ntry = 0;
    root = xb;
    return 0;
  }

  for (int k = 1; k <= max_ntry; k++) {
    dx = fb * (xb - xa) / (fb - fa);  // Compute increment

    // Shift values
    xa = xb;
    fa = fb;
    xb = xb - dx;
    fb = f(xb);

    // Check convergence
    if (fabs(dx) < xtol || fabs(fb) < ftol || fb == 0.0) {
      ntry = k;
      root = xb;
      return 0;
    }
  }

  ntry = -1;
  root = nan("");
  throw std::runtime_error("Maximum number of steps exceeded.");
  return 1;
}

int secant(double (*f)(const double &x), double xa, double xb,
           const double &xtol, double &root) {
  int n;
  return secant(f, xa, xb, xtol, -1.0, root, n);
}

int secant(double (*f)(const double &x), double xa, double xb,
           const double &xtol, double &root, int &ntry) {
  return secant(f, xa, xb, xtol, -1.0, root, ntry);
}

```

```
int secant(double (*f)(const double &x), double xa, double xb,
           const double &xtol, const double &ftol, double &root) {
  int n;
  return secant(f, xa, xb, xtol, ftol, root, n);
}
```

GitHub repository: https://github.com/marchfra/Algoritmi.git