



UNIVERSITÀ
DI 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

1	Problem statement	3
2	Equations of motion	3
2.1	Nondimensionalisation	3
2.2	Exact solution	4
3	Numerical algorithm	6
3.1	Residual function	6
3.2	Runge-Kutta	6
3.3	Polynomial interpolation	7
3.4	Root finder	7
4	Numerical solution	8
4.1	Comparison with analytical solution	9
4.2	Time step size	10
4.3	Solutions with friction	11
A	Extra plots	13
B	Code listing	14
B.1	ProjectileMotion/src/main.cpp	14
B.2	Libs/include/exception.hpp	25
B.3	Libs/include/lin_alg.hpp	26
B.4	Libs/include/ode_solver.hpp	27
B.5	Libs/src/ode_solver.cpp	27
B.6	Libs/include/root_finder.hpp	28
B.7	Libs/src/root_finder.cpp	32

List of Figures

1	Shooting and residual plots without friction.	9
2	Absolute difference between numerical and analytical solutions.	10
3	Optimal time step size search.	11
4	Trajectories with friction.	12
5	Comparison with analytical solution (linear interpolation).	13
6	Shooting and residual plots with drag.	13

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:

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

where $g = 9.81 \text{ m/s}^2$.

Since $\vec{v} = (\dot{x}\hat{x} + \dot{y}\hat{y})$, and, by Newton's second law, $\vec{F}_{net} = m\vec{a}$:

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

or, in components:

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

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):

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

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

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

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:

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

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

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

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

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

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 θ and performs the following operations:

1. integrates the equations using a 4th-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

$$\frac{dy}{dt} = f(y, t), \quad y(t_0) = y_0$$

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

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

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
- $k_2 = f(t_n + \Delta t/2, y_n + h k_1/2)$ is the slope at the midpoint of the interval, using y and k_1
- $k_3 = f(t_n + \Delta t/2, y_n + h k_2/2)$ is the slope at the midpoint of the interval, using y and k_2
- $k_4 = f(t_n + \Delta t, y_n + h k_3)$ 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 sub-intervals 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 θ .

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 1.0×10^{-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 2nd 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:

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

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.

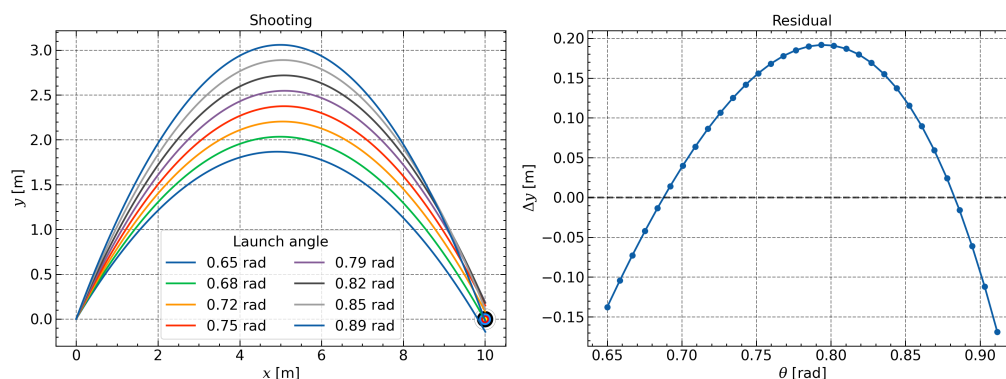


Figure 1: Shooting and residual plots without friction.

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 2nd order polynomial interpolation) solves exactly the problem.

Table 1 shows the obtained results.

	θ_1	θ_2
analytical	0.68777523221	0.88302109458
numerical	0.68777523222	0.88302109459

Table 1: Comparison between analytical and numerical launch angles.

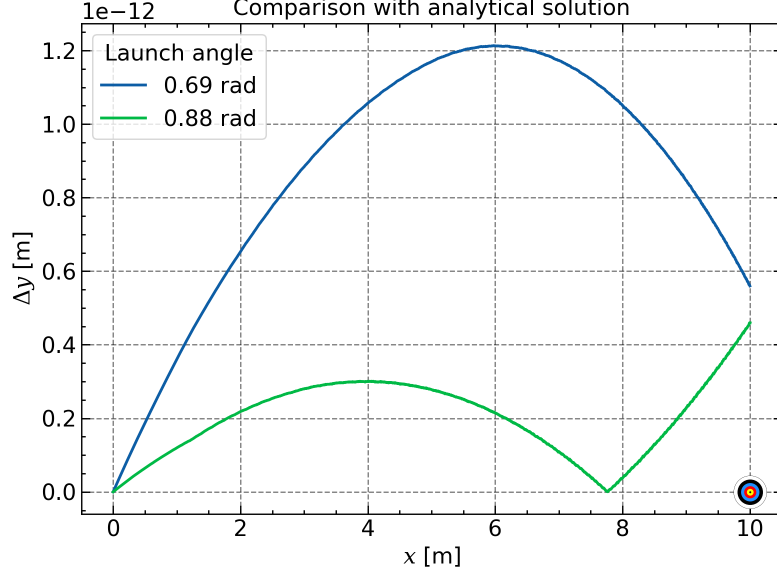


Figure 2: Absolute difference between numerical and analytical solutions.

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 θ is plotted against the time step size, it appears clear that using a time step smaller than about 2×10^{-5} does not yield a better result, and only serves to increase the computational load. Therefore, a time step of 1.0×10^{-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.

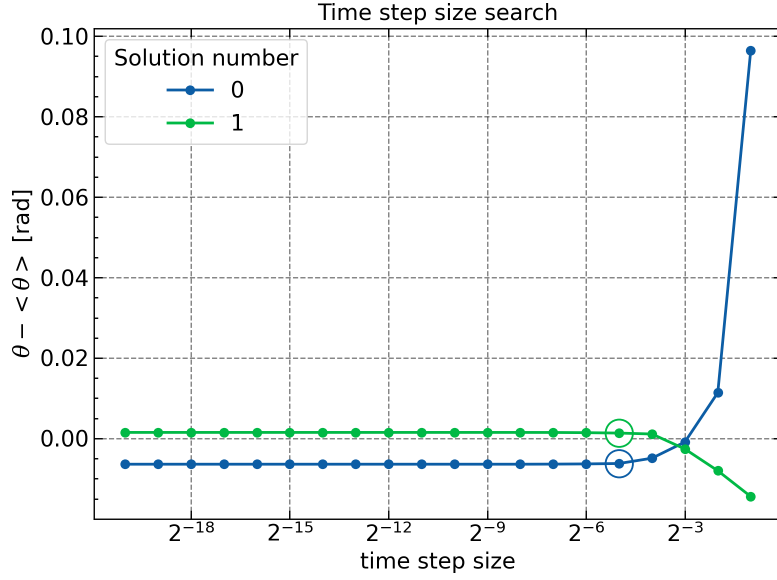


Figure 3: Optimal time step size search.

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 $B = 4.0 \times 10^{-5}$ kg/m, unit mass, and boundary conditions

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

The roots of the residual function found by the algorithm are

Sol. number	θ [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.

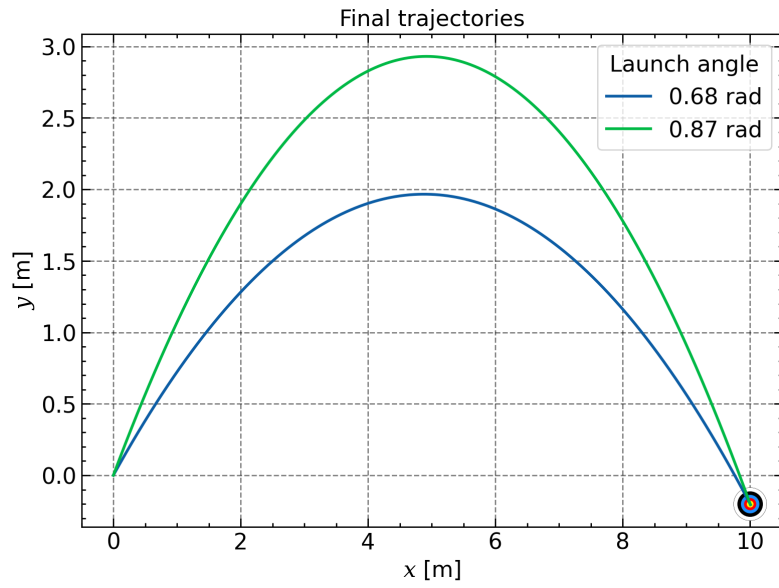


Figure 4: Trajectories with friction.

A Extra plots

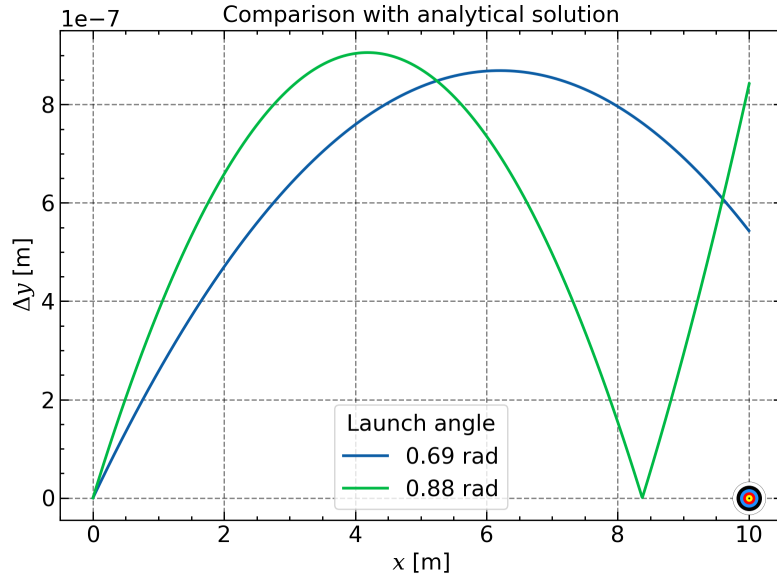


Figure 5: Comparison with analytical solution (linear interpolation).

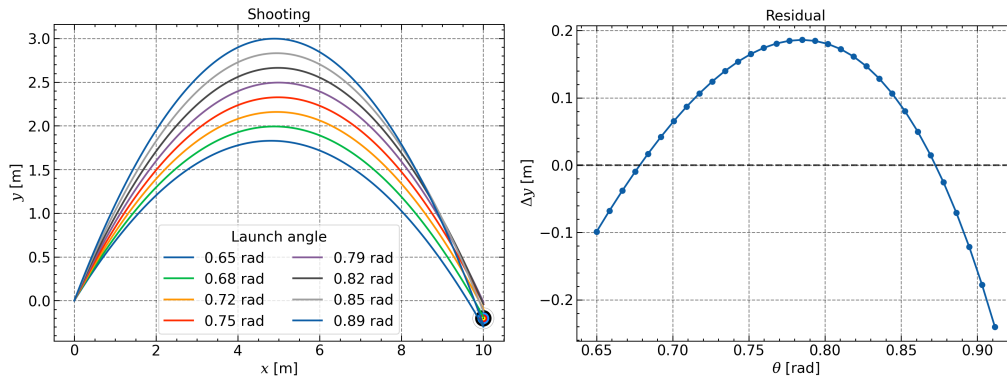


Figure 6: Shooting and residual plots with drag.

B Code listing

B.1 ProjectileMotion/src/main.cpp

```
1  /**
2   * @file main.cpp
3   *
4   * @author Francesco Marchisotti
5   *
6   * @brief Main file for the course's final project.
7   *
8   * @date 2024-05-15
9   */
10
11 #include "../Libs/include/exception.hpp"
12 #include "../Libs/include/lin_alg.hpp"
13 #include "../Libs/include/ode_solver.hpp"
14 #include "../Libs/include/root_finder.hpp"
15
16 #include <cmath>
17 #include <fstream>
18 #include <iostream>
19
20 using std::cerr;
21 using std::cin;
22 using std::cout;
23 using std::endl;
24
25 #define FRICTION 1
26 const static int gOrder = 2;  //!< Selects order of polynomial interpolation
27
28 int numIntegrations = 0;  //!< Number of integrations of the ODEs performed
29
30 double g_dt = 1.0e-5;
31
32 // Problem data
33 #if FRICTION
34 const static double B    = 4.0e-5;  //!< Drag coefficient [kg/m]
35 const static double V0   = 9.90;    //!< Initial velocity [m/s]
36 const static double L    = 10.0;    //!< Target distance [m]
37 const static double YTarg = -0.2;    //!< Target height [m]
38 double gTheta;                //!< Initial launch angle
39 #else
40 const static double B    = 0.0;    //!< Drag coefficient [kg/m]
41 const static double V0   = 10.0;    //!< Initial velocity [m/s]
42 const static double L    = 10.0;    //!< Target distance [m]
43 const static double YTarg = 0.0;    //!< Target height [m]
44 double gTheta;                //!< Initial launch angle
```

```

45 #endif
46
47 // Dimensional factors
48 const static double chi = L;           //!< Space dimensional factor [m]
49 const static double mu = 1.0;          //!< Mass dimensional factor [kg]
50 const static double g = 9.81;          //!< Gravity [m/s^2]
51 const static double tau = sqrt(chi / g); //!< Time dimensional factor [s]
52
53 const static double b = B * chi / mu;  //!< Adimensional friction
54 const static double v0 = V0 * tau / chi; //!< Adimensional speed
55 const static double xTarg = 1.0;        //!< Adimensional target distance
56 const static double yTarg = YTarg / L;  //!< Adimensional target height
57
58 /**
59  * @brief Prints problem data and dimensional constants to file.
60  */
61 void printConstants();
62
63 /**
64  * @brief This function returns the linear interpolation between two points
65  *        evaluated at a certain x.
66  *
67  * @param[in] x The point at which to evaluate the interpolation.
68  * @param[in] x1 The x-coordinate of the first point.
69  * @param[in] y1 The y-coordinate of the first point.
70  * @param[in] x2 The x-coordinate of the second point.
71  * @param[in] y2 The y-coordinate of the second point.
72  *
73  * @return The interpolated line evaluated at x.
74  */
75 double linearInterp(const double &x, const double &x1, const double &y1,
76                    const double &x2, const double &y2);
77
78 /**
79  * @brief This function returns the polinomial interpolation
80  *        between a sufficient number of
81  *        points evaluated at a
82  *        certain x.
83  *
84  * @param[in] x The point at which to evaluate the interpolation.
85  * @param[in] xLast The x-coordinates of the first 'order' points.
86  * @param[in] yLast The y-coordinates of the first 'order' points.
87  * @param[in] xCurrent The x-coordinate of the last point.
88  * @param[in] yCurrent The y-coordinate of the last point.
89  * @param[in] order The order or the polynomial.
90  *
91  * @return The interpolated line evaluated at x.
92  */
93 double polInterp(const double &x, double xLast[], double yLast[],

```



```

93         const double &xCurrent, const double &yCurrent,
94         const int &order);
95
96 /**
97  * @brief      Right Hand Side of the system of ODEs.
98  *
99  * @param[in]  Y  The input values of the variables.
100 * @param[out] R  The output values of the variables.
101 */
102 void RHS(const double &t, double Y[], double R[]);
103
104 /**
105  * @brief      Computes the exact trajectory.
106  *
107  * @param[in]  x          The point at which to evaluate the trajectory.
108  * @param[in]  solNumber  The number of the solution.
109  *                    - 0: the solution with  $\theta < M_{PI} / 4$ 
110  *                    - 1: the solution with  $\theta > M_{PI} / 4$ 
111  *
112  * @return     double
113  */
114 double exact(const double &x, const int &solNumber);
115
116 /**
117  * @brief      Produces the convergence plot.
118  *
119  * @param[in]  dt_0      The first (and largest) value of dt.
120  * @param[in]  nPoints   The number of dts to explore.
121  * @param[in]  factor    The factor that scales dt.
122  */
123 void convergence(const double &dt_0, const int &nPoints,
124                 const double factor = 0.5);
125
126 /**
127  * @brief      Generates data for shooting plot.
128  *
129  * @param[in]  thetaMin  Lower bound for launch angle.
130  * @param[in]  thetaMax  Upper bound for launch angle.
131  * @param[in]  nTheta    Number of launch angles explored.
132  */
133 void shootingPlot(const double &thetaMin, const double &thetaMax,
134                  const double &nTheta);
135
136 /**
137  * @brief      Integration interface.
138  *
139  * @param[in,out] y      Array with the variables.
140  * @param[in]     theta   Launch angle.
141  * @param[out]     xLast  Array with x-position at previous time step.

```

```

142 * @param[out]    yLast    Array with y-position at previous time step.
143 * @param[in]     order    Numer of previous times to save.
144 * @param[in]     outFile  Output file.
145 */
146 void integrate(double y[], const double &theta, double xLast[], double yLast[],
147               const int &order, std::ofstream &outFile);
148
149 /**
150 * @overload
151 *
152 * @brief          Integration interface (without file output).
153 *
154 * @param[in,out]  y        Array with the variables.
155 * @param[in]      theta    Launch angle.
156 * @param[out]     xLast    Array with x-position at previous time step.
157 * @param[out]     yLast    Array with y-position at previous time step.
158 * @param[in]      order    Numer of previous times to save.
159 */
160 void integrate(double y[], const double &theta, double &xLast, double &yLast);
161
162 /**
163 * @overload
164 *
165 * @brief          Integration interface (without past values).
166 *
167 * @param[in,out]  y        Array with the variables.
168 * @param[in]      theta    Launch angle.
169 * @param[in]      outFile  Output file.
170 */
171 void integrate(double y[], const double &theta, std::ofstream &outFile);
172
173 /**
174 * @overload
175 *
176 * @brief          Integration interface (without past values and file output).
177 *
178 * @param[in,out]  y        Array with the variables.
179 * @param[in]      theta    Launch angle.
180 */
181 void integrate(double y[], const double &theta);
182
183 /**
184 * @brief          Function that performs the integration and prints to file.
185 *
186 * @param[in]      RHSFunc   Right Hand Side of the system of ODEs.
187 * @param[in, out] y        Array with the variables. Should be already
188 *                          initialised.
189 * @param[in]      nEq       Number of equations in the system.
190 * @param[in]      t         Starting time of the integration.

```

```

191 * @param[in]      dt          Time step size.
192 * @param[in]      theta       Launch angle.
193 * @param[out]     xLast       Array with x-position at previous time step.
194 * @param[out]     yLast       Array with y-position at previous time step.
195 * @param[in]      order       Numer of previous times to save.
196 * @param[in]      maxStep     Maximum number of integration steps.
197 * @param[in]      outFile     Output file.
198 */
199 void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
200                 double y[], const int &nEq, double t, const double &dt,
201                 const double &theta, double xLast[], double yLast[],
202                 const int &order, const int &maxStep, std::ofstream &outFile);
203
204 /**
205  * @overload
206  *
207  * @brief          Function that performs the integration and prints to file
208  *                  with exact difference.
209  *
210  * @param[in]      RHSFunc     Right Hand Side of the system of ODEs.
211  * @param[in]      exactFunc    Exact trajectory.
212  * @param[in, out] y           Array with the variables. Should be already
213  *                              initialised.
214  * @param[in]      sol_number   Solution number.
215  * @param[in]      theta       Launch angle.
216  * @param[in]      outFile     Output file.
217  */
218 void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
219                 double (*exactFunc)(const double &x, const int &sol_number),
220                 double y[], const int &sol_number, const double &theta,
221                 std::ofstream &outFile);
222
223 /**
224  * @brief          Residual function for the BVP.
225  *
226  * @param[in]      theta       Launch angle.
227  *
228  * @return         y(x = 1) - yTarg
229  */
230 double Residual(const double &theta);
231
232 int main() {
233     convergence(0.5, 20);
234
235     #if FRICTION
236         cout << "==== FRICTION =====" << endl << endl;
237     #else
238         cout << "==== NO FRICTION =====" << endl << endl;
239         for (int i = 0; i < 2; i++) {

```

```

240     double theta =
241         0.5 * (i * M_PI + (i == 0 ? 1.0 : -1.0) * asin(1 / (v0 * v0)));
242     cout.precision(16);
243     cout << "Analytical theta" << i << " = " << theta << endl;
244 }
245 cout << endl;
246 #endif
247
248 printConstants();
249
250 const double thetaMin = 0.65;    // Minimum launch angle
251 const double thetaMax = 0.92;    // Maximum launch angle
252 const int nTheta      = 32;      // Number of launch angles explored
253 const double thetaTol = 1.0e-7;  // Tolerance for root searching
254
255 shootingPlot(thetaMin, thetaMax, nTheta);
256
257 /* +-----+
258  * | Problem solution |
259  * +-----+ */
260 numIntegrations = 0;
261 double roots[4];
262 int nRoots = -1;
263 try {
264     findRoots(Residual, thetaMin, thetaMax, thetaTol, roots, nRoots, 4,
265             "secant");
266
267     cout << "Integrations performed: " << numIntegrations << endl;
268
269     cout.precision(16);
270     cout << "Optimal thetas [rad] (+/- " << thetaTol << "): ";
271     printVector(roots, nRoots);
272 } catch (std::exception &err) {
273     cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
274 } catch (...) {
275     cerr << "Sorry, could not recognise the error." << endl;
276 }
277
278 std::ofstream finTraj;
279 #if FRICTION
280 finTraj.open("data/finTraj.csv");
281 finTraj << "t,x,y,u,v,theta" << endl;
282
283 for (int i = 0; i < nRoots; i++) {
284     double y[4];
285     integrate(y, roots[i], finTraj);
286 }
287 #else
288 finTraj.open("data/noFriction.csv");

```

```

289   finTraj << "t,x,delta_y,u,v,theta" << endl;
290   for (int i = 0; i < nRoots; i++) {
291       double y[4];
292       integration(RHS, exact, y, i, roots[i], finTraj);
293   }
294   #endif
295   finTraj.close();
296
297   return 0;
298 }
299
300 void printConstants() {
301     std::ofstream out;
302     try {
303         out.open("data/constants.csv");
304         if (!out.good()) throw exception("Invalid file.");
305
306         out << "chi,tau,mu,B,b,V0,YTarg" << endl;
307         out << chi << "," << tau << "," << mu << "," << B << "," << b << "," << V0
308             << "," << YTarg << endl;
309
310         out.close();
311     } catch (std::exception &err) {
312         cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
313     } catch (...) {
314         cerr << "Sorry, could not recognise the error." << endl;
315     }
316
317     out.close();
318 }
319
320 double linearInterp(const double &x, const double &x1, const double &y1,
321                    const double &x2, const double &y2) {
322     double m = (y2 - y1) / (x2 - x1);
323     double q = y1 - m * x1;
324     double y = m * x + q;
325     return y;
326 }
327
328 double polInterp(const double &x, double xLast[], double yLast[],
329                 const double &xCurrent, const double &yCurrent,
330                 const int &order) {
331     const int nPoints = order + 1;
332
333     double **M;
334     M = new double *[nPoints];
335     M[0] = new double[nPoints * nPoints];
336     for (int i = 1; i < nPoints; i++) M[i] = M[i - 1] + nPoints;
337

```

```

338 double *v;
339 v = new double[nPoints];
340
341 // Define coefficient matrix
342 for (int i = 0; i < nPoints; i++) {
343     M[i][nPoints - 1] = 1;
344     if (i != nPoints - 1) v[i] = yLast[i];
345     else v[i] = yCurrent;
346     for (int j = nPoints - 2; j >= 0; j--)
347         if (i != nPoints - 1) M[i][j] = M[i][j + 1] * xLast[i];
348         else M[i][j] = M[i][j + 1] * xCurrent;
349 }
350
351 double *coeffs; //<! Array with the coefficients of the polynomial
352 coeffs = new double[nPoints];
353
354 solveLinSystem(M, v, coeffs, nPoints);
355
356 delete[] M[0];
357 delete[] M;
358 delete[] v;
359
360 double value = 0.0;
361 double powerOfX = 1.0;
362 for (int i = nPoints - 1; i >= 0; i--) {
363     value += coeffs[i] * powerOfX;
364     powerOfX *= x;
365 }
366
367 return value;
368 }
369
370 void RHS(const double &t, double Y[], double R[]) {
371     double u = Y[2];
372     double v = Y[3];
373
374     double mod_v = sqrt(u * u + v * v);
375
376     R[0] = u;
377     R[1] = v;
378     R[2] = -b * u * mod_v;
379     R[3] = -1.0 - b * u * mod_v;
380 }
381
382 double exact(const double &x, const int &solNumber) {
383     if (solNumber != 0 && solNumber != 1)
384         throw std::invalid_argument("solNumber must be 0 or 1");
385
386     double theta = 0.5 * (solNumber * M_PI +

```

```

387         (solNumber == 0 ? 1.0 : -1.0) * asin(1 / (v0 * v0)));
388     double u0      = v0 * cos(theta);
389     return (-0.5 * (x / u0) * (x / u0) + tan(theta) * x);
390 }
391
392 void convergence(const double &dt_0, const int &nPoints, const double factor) {
393     double store_g_dt = g_dt;
394     g_dt             = dt_0;
395     std::ofstream conv;
396     conv.open("data/dt_search.csv");
397     conv << "dt,theta1,theta2" << endl;
398     for (int i = 0; i < nPoints; i++) {
399         const double thetaMin = 0.65;    // Minimum launch angle
400         const double thetaMax = 0.92;    // Maximum launch angle
401         const double thetaTol = 1.0e-7;  // Tolerance for root searching
402
403         /* +-----+
404          * | Problem solution |
405          * +-----+ */
406         double roots[4];
407         int nRoots = -1;
408         try {
409             findRoots(Residual, thetaMin, thetaMax, thetaTol, roots, nRoots, 4,
410                     "secant");
411
412             conv.precision(12);
413             conv << g_dt << "," << roots[0] << "," << roots[1] << endl;
414
415             cout << "dt = " << g_dt << " roots = ";
416             printVector(roots, nRoots);
417         } catch (std::exception &err) {
418             cerr << "Caught " << typeid(err).name() << " : " << err.what() << endl;
419         } catch (...) {
420             cerr << "Sorry, could not recognise the error." << endl;
421         }
422
423         g_dt *= factor;
424     }
425     conv.close();
426     g_dt = store_g_dt;
427 }
428
429 void shootingPlot(const double &thetaMin, const double &thetaMax,
430                  const double &nTheta) {
431     const double dTheta = (thetaMax - thetaMin) / nTheta;
432
433     std::ofstream shooting, residual;
434     shooting.open("data/shooting.csv");
435     residual.open("data/residual.csv");

```

```

436 shooting << "t,x,y,u,v,theta" << endl; // Output csv header
437 residual << "theta,res" << endl; // Output csv header
438 for (int i = 0; i < nTheta; i++) {
439     double theta = thetaMin + i * dTheta;
440     double y[4];
441
442     integrate(y, theta, shooting);
443
444     residual << theta << "," << Residual(theta) << endl;
445 }
446 shooting.close();
447 residual.close();
448 }
449
450 void integrate(double y[], const double &theta, double xLast[], double yLast[],
451               const int &order, std::ofstream &outFile) {
452     const double y0[] = {0.0, 0.0, v0 * cos(theta), v0 * sin(theta)};
453     const int nEq =
454         static_cast<int>(sizeof(y0)) / static_cast<int>(sizeof(y0[0]));
455     for (int i = 0; i < nEq; i++) y[i] = y0[i];
456
457     double t0 = 0.0;
458     const double dt = g_dt;
459     const int maxStep = int(2 / dt);
460
461     integration(RHS, y, nEq, t0, dt, theta, xLast, yLast, order, maxStep,
462               outFile);
463 }
464
465 void integrate(double y[], const double &theta, double xLast[], double yLast[],
466               const int &order) {
467     std::ofstream dummyOutfile;
468     integrate(y, theta, xLast, yLast, order, dummyOutfile);
469 }
470
471 void integrate(double y[], const double &theta, std::ofstream &outFile) {
472     double dummyLast[2];
473     integrate(y, theta, dummyLast, dummyLast, 1, outFile);
474 }
475
476 void integrate(double y[], const double &theta) {
477     std::ofstream dummyOutfile;
478     double dummyLast[2];
479     integrate(y, theta, dummyLast, dummyLast, 1, dummyOutfile);
480 }
481
482 void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
483                 double y[], const int &nEq, double t, const double &dt,
484                 const double &theta, double xLast[], double yLast[],

```



```

485         const int &order, const int &maxStep, std::ofstream &outFile) {
486     outFile << t << "," << y[0] << "," << y[1] << "," << y[2] << "," << y[3]
487         << "," << theta << endl;
488
489     numIntegrations++;
490     int stepCounter = 0;
491     bool exitCondition = false;
492     while (stepCounter < maxStep && !exitCondition) {
493         for (int i = 0; i < order - 1; i++) {
494             xLast[i] = xLast[i + 1];
495             yLast[i] = yLast[i + 1];
496         }
497         xLast[order - 1] = y[0];
498         yLast[order - 1] = y[1];
499
500         rk4Step(t, y, RHSFunc, dt, nEq);
501         t += dt;
502         stepCounter++;
503
504         outFile << t << "," << y[0] << "," << y[1] << "," << y[2] << "," << y[3]
505             << "," << theta << endl;
506
507         if (xLast[0] < xTarg && y[0] > xTarg) exitCondition = true;
508     }
509 }
510
511 void integration(void (*RHSFunc)(const double &t, double *Y, double *RHS),
512                 double (*exactFunc)(const double &x, const int &sol_number),
513                 double y[], const int &sol_number, const double &theta,
514                 std::ofstream &outFile) {
515     const double y0[] = {0.0, 0.0, v0 * cos(theta), v0 * sin(theta)};
516     const int nEq =
517         static_cast<int>(sizeof(y0)) / static_cast<int>(sizeof(y0[0]));
518     for (int i = 0; i < nEq; i++) y[i] = y0[i];
519
520     double t = 0.0;
521     const double dt = g_dt;
522     const int maxStep = int(2 / dt);
523
524     outFile << t << "," << y[0] << "," << fabs(y[1] - exactFunc(y[0], sol_number))
525         << "," << y[2] << "," << y[3] << "," << theta << endl;
526
527     numIntegrations++;
528     int stepCounter = 0;
529     bool exitCondition = false;
530     while (stepCounter < maxStep && !exitCondition) {
531         rk4Step(t, y, RHSFunc, dt, nEq);
532         t += dt;
533         stepCounter++;

```

```

534     outFile << t << "," << y[0] << ","
535         << fabs(y[1] - exactFunc(y[0], sol_number)) << "," << y[2] << ","
536         << y[3] << "," << theta << endl;
537
538     if (y[0] > xTarg) exitCondition = true;
539 }
540 }
541
542 double Residual(const double &theta) {
543     double y[4];
544     double xLast[64], yLast[64];
545     if (gOrder > 64) throw exception("gOrder must be at most 64.");
546     integrate(y, theta, xLast, yLast, gOrder);
547
548     double xCurrent = y[0], yCurrent = y[1];
549
550     return polInterp(xTarg, xLast, yLast, xCurrent, yCurrent, gOrder) - yTarg;
551 }
552 }

```

B.2 Libs/include/exception.hpp

```

1  /**
2   * @file    exception.hpp
3   *
4   * @author  Francesco Marchisotti
5   *
6   * @brief   Implements the class exception
7   *
8   * @date    2024-05-08
9   */
10 #pragma once
11
12 #include <iostream>
13
14 class exception : public std::exception {
15 public:
16     /**
17      * @brief Constructor (C++ STL strings).
18      *
19      * @param message The error message.
20      */
21     explicit exception(const std::string& message)
22         : msg_(message) {}
23
24     /**
25      * @brief Destructor.
26      *
27      * Virtual to allow for subclassing.

```

```

28     */
29     virtual ~exception() noexcept {}
30
31     /**
32     * @brief Returns a pointer to the (constant) error description.
33     *
34     * @return A pointer to a const char*. The underlying memory is in possession
35     *         of the exception object. Callers must not attempt to free the
36     *         memory.
37     */
38     virtual const char* what() const noexcept { return msg_.c_str(); }
39
40 protected:
41     std::string msg_;    //!< Error message
42 };

```

B.3 Libs/include/lin_alg.hpp

```

1  /**
2  * @file    lin_alg.hpp
3  *
4  * @brief    This file implements linear algorithms.
5  *
6  * @author    Francesco Marchisotti
7  *
8  * @date      2023-11-24
9  */
10
11 #pragma once
12
13 #include "../include/swap.hpp"
14
15 #include <iomanip>
16 #include <iostream>
17
18 /**
19 * @brief      Prints a vector.
20 *
21 * @param[in]   v      The vector.
22 * @param[in]   nRows   Size of the vector.
23 *
24 * @tparam      T      Type of the elements of the vector.
25 */
26 template <class T>
27 void printVector(T v[], const int& nRows) {
28     std::cout << "{";
29     for (int i = 0; i < nRows; i++) {
30         std::cout << v[i];
31         if (i != nRows - 1) std::cout << ", ";

```

```

32 }
33 std::cout << "}" << std::endl;
34 }

```

B.4 Libs/include/ode_solver.hpp

```

1 /**
2  * @file    ode_solver.hpp
3  *
4  * @brief   Implementation of the ODE Solvers step functions.
5  *
6  * @author  Francesco Marchisotti
7  *
8  * @date    2024-05-01
9  */
10 #pragma once
11
12 #include <iostream>
13
14 /**
15  * @brief   Runge-Kutta 4 method step.
16  *
17  * Takes one step in time (or whatever the independent variable is) using
18  * Runge-Kutta 4 method. The system of first order ODEs is  $dY_i/dt = R_i(t, Y)$ .
19  *
20  * @param[in]    t          The value of time from which to take the step.
21  * @param[in, out] Y        Array containing all the dependent variables.
22  * @param[in]    RHSFunc    Pointer to the function containing all the Right
23  *                           Hand Sides of the system of equations.
24  * @param[in]    dt         The step size.
25  * @param[in]    neq        The number of equations (ie the number of
26  *                           independent variables) in the system.
27  */
28 void rk4Step(const double& t, double Y[],
29             void (*RHSFunc)(const double& t, double Y[], double RHS[]),
30             const double& dt, const int& neq);

```

B.5 Libs/src/ode_solver.cpp

```

1 #include "../include/ode_solver.hpp"
2
3 void rk4Step(const double &t, double Y[],
4             void (*RHSFunc)(const double &t, double Y[], double RHS[]),
5             const double &dt, const int &neq) {
6     if (neq > 64) throw std::invalid_argument("neq must be less than 64");
7
8     double Ystar[64], k1[64], k2[64], k3[64], k4[64];
9
10    RHSFunc(t, Y, k1);

```

```

11
12 for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + 0.5 * dt * k1[i];
13 RHSFunc(t + 0.5 * dt, Ystar, k2);
14
15 for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + 0.5 * dt * k2[i];
16 RHSFunc(t + 0.5 * dt, Ystar, k3);
17
18 for (int i = 0; i < neq; i++) Ystar[i] = Y[i] + dt * k3[i];
19 RHSFunc(t + dt, Ystar, k4);
20
21 for (int i = 0; i < neq; i++)
22     Y[i] += dt / 6.0 * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i]);
23 }

```

B.6 Libs/include/root_finder.hpp

```

1 /**
2  * @file    root_finder.hpp
3  *
4  * @brief    Implementation of the root finder methods.
5  *
6  * @author    Paolino Paperino
7  *
8  * @date      2023-11-19
9  */
10 #pragma once
11
12 #include <iomanip>
13 #include <iostream>
14
15 /**
16  * @brief      Find the roots of a function in a given interval.
17  *
18  * Find the roots of a function f(x) in a given interval [xa, xb]
19  * using the specified method. Works by first bracketing the roots
20  * and then applying the method on every sub-interval.
21  *
22  * @param[in]   f          Pointer to the function.
23  * @param[in]   dfdx       Pointer to the derivative of the function.
24  * @param[in]   xa         Lower bound of the interval.
25  * @param[in]   xb         Upper bound of the interval.
26  * @param[in]   tol        x-tolerance.
27  * @param[out]  roots       Array with the roots of f(x).
28  * @param[out]  nRoots     The number of roots found.
29  * @param[in]   N          The number of sub-intervals.
30  * @param[in]   method     The root finding method. Accepted values are:
31  *                          'bisection', 'falsePosition', 'secant', 'newton'.
32  *
33  * @return      flag

```

```

34 *
35 * @retval    0      Success.
36 * @retval    1      Too many steps.
37 * @retval    2      Initial interval doesn't contain any root.
38 *
39 * @throws    std::invalid_argument  Thrown if 'N' > 128.
40 * @throws    std::invalid_argument  Thrown if 'method' is not among the
41 *                                     accepted values.
42 * @throws    std::runtime_error      Thrown if roots can't be found inside the
43 *                                     interval.
44 * @throws    std::runtime_error      Thrown if one of the root finders exceeded
45 *                                     the maximum number of steps.
46 */
47 int findRoots(double (*f)(const double& x), double (*dfdx)(const double& x),
48               const double& xa, const double& xb, const double& tol,
49               double roots[], int& nRoots, const int N = 128,
50               const std::string method = "newton");
51
52 /**
53 * @overload
54 *
55 * @brief      Find the roots of a function in a given interval (Newton's method
56 *               not available).
57 *
58 * Find the roots of a function f(x) in a given interval [xa, xb]
59 * using the specified method. Works by first bracketing the roots
60 * and then applying the method on every sub-interval.
61 *
62 * @param[in]  f      Pointer to the function.
63 * @param[in]  xa      Lower bound of the interval.
64 * @param[in]  xb      Upper bound of the interval.
65 * @param[in]  tol      x-tolerance.
66 * @param[out] roots   Array with the roots of f(x).
67 * @param[out] nRoots  The number of roots found.
68 * @param[in]  N        The number of sub-intervals.
69 * @param[in]  method   The root finding method. Accepted values are:
70 *                       'bisection', 'falsePosition', 'secant'.
71 *
72 * @return     flag
73 *
74 * @retval     0      Success.
75 * @retval     1      Too many steps.
76 * @retval     2      Initial interval doesn't contain any root.
77 *
78 * @throws     std::invalid_argument  Thrown if 'N' > 128.
79 * @throws     std::invalid_argument  Thrown if 'method' is not among the
80 *                                     accepted values.
81 * @throws     std::runtime_error      Thrown if roots can't be found inside the
82 *                                     interval.

```

```

83 * @throws      std::runtime_error      Thrown if one of the root finders exceeded
84 *                                                    the maximum number of steps.
85 */
86 int findRoots(double (*f)(const double& x), const double& xa, const double& xb,
87             const double& tol, double roots[], int& nRoots, const int N = 128,
88             const std::string method = "bisection");
89
90 /**
91 * @brief        Bracket the roots of a function in a given interval [xa, xb].
92 *
93 * Works by subdividing the interval in a number of sub-intervals
94 * and checking if the function changes sign (an odd number of
95 * times) over this interval. If it does, then the interval contains
96 * (at least) one root.
97 *
98 * @param[in]    f        Pointer to the function.
99 * @param[in]    xa        Lower bound of the interval.
100 * @param[in]    xb        Upper bound of the interval.
101 * @param[out]   xL        Array with the lower bound of the sub-interval containing
102 *                          a root.
103 * @param[out]   xR        Array with the upper bound of the sub-interval containing
104 *                          a root.
105 * @param[in]    N        The number of sub-intervals.
106 * @param[out]   nRoots    The number of roots found.
107 */
108 void bracket(double (*f)(const double& x), const double& xa, const double& xb,
109             double xL[], double xR[], const int& N, int& nRoots);
110
111 /**
112 * @brief        Find the root of a function f(x) in a given interval [xa, xb]
113 *                using secant method.
114 *
115 * @param[in]    f        Pointer to the function.
116 * @param[in]    xa        Lower bound of the interval.
117 * @param[in]    xb        Upper bound of the interval.
118 * @param[in]    xtol      x-tolerance.
119 * @param[in]    ftol      f(x)-tolerance: the values of f(x) that are considered 0.
120 * @param[out]   root      The root of f(x).
121 * @param[out]   ntry      The number of iterations achieved.
122 *
123 * @return       flag
124 *
125 * @retval       0        Success.
126 * @retval       1        Too many steps.
127 *
128 * @throws      std::runtime_error      Thrown if the maximum number of steps is
129 *                                                    exceeded.
130 */
131 int secant(double (*f)(const double& x), double xa, double xb,

```

```

132         const double& xtol, const double& ftol, double& root, int& ntry);
133
134 /**
135  * @overload
136  *
137  * @brief      Find the root of a function f(x) in a given interval [xa, xb]
138  *              using secant method.
139  *
140  * @param[in]  f      Pointer to the function.
141  * @param[in]  xa      Lower bound of the interval.
142  * @param[in]  xb      Upper bound of the interval.
143  * @param[in]  xtol     x-tolerance.
144  * @param[out] root    The root of f(x).
145  *
146  * @return     flag
147  *
148  * @retval     0      Success.
149  * @retval     1      Too many steps.
150  *
151  * @throws     std::runtime_error Thrown if the maximum number of steps is
152  *              exceeded.
153  */
154 int secant(double (*f)(const double& x), double xa, double xb,
155           const double& xtol, double& root);
156
157 /**
158  * @overload
159  *
160  * @brief      Find the root of a function f(x) in a given interval [xa, xb]
161  *              using secant method.
162  *
163  * @param[in]  f      Pointer to the function.
164  * @param[in]  xa      Lower bound of the interval.
165  * @param[in]  xb      Upper bound of the interval.
166  * @param[in]  xtol     x-tolerance.
167  * @param[out] root    The root of f(x).
168  * @param[out] ntry    The number of iterations achieved.
169  *
170  * @return     flag
171  *
172  * @retval     0      Success.
173  * @retval     1      Too many steps.
174  *
175  * @throws     std::runtime_error Thrown if the maximum number of steps is
176  *              exceeded.
177  */
178 int secant(double (*f)(const double& x), double xa, double xb,
179           const double& xtol, double& root, int& ntry);
180

```



```

181 /**
182  * @overload
183  *
184  * @brief      Find the root of a function f(x) in a given interval [xa, xb]
185  *              using secant method.
186  *
187  * @param[in]  f      Pointer to the function.
188  * @param[in]  xa      Lower bound of the interval.
189  * @param[in]  xb      Upper bound of the interval.
190  * @param[in]  xtol     x-tolerance.
191  * @param[in]  ftol     f(x)-tolerance: the values of f(x) that are considered 0.
192  * @param[out] root     The root of f(x).
193  *
194  * @return      flag
195  *
196  * @retval      0      Success.
197  * @retval      1      Too many steps.
198  *
199  * @throws      std::runtime_error  Thrown if the maximum number of steps is
200  *                                  exceeded.
201  */
202 int secant(double (*f)(const double& x), double xa, double xb,
203           const double& xtol, const double& ftol, double& root);

```

B.7 Libs/src/root_finder.cpp

```

1  #include "../include/root_finder.hpp"
2
3  int findRoots(double (*f)(const double &x), double (*dfdx)(const double &x),
4              const double &xa, const double &xb, const double &tol,
5              double roots[], int &nRoots, const int N,
6              const std::string method) {
7      if (N > 128) throw std::invalid_argument("N must be less than 128");
8
9      double xL[128], xR[128];
10
11     bracket(f, xa, xb, xL, xR, N, nRoots);
12
13     if (nRoots == 0) {
14         throw std::runtime_error(
15             "The supplied interval does not contain any roots.");
16     }
17
18     for (int i = 0; i < nRoots; i++)
19         if (method == "bisection") bisection(f, xL[i], xR[i], tol, roots[i]);
20         else if (method == "falsePosition")
21             falsePosition(f, xL[i], xR[i], tol, roots[i]);
22         else if (method == "secant") secant(f, xL[i], xR[i], tol, roots[i]);
23         else if (method == "newton") newton(f, dfdx, xL[i], xR[i], tol, roots[i]);

```

```

24     else throw std::invalid_argument("Invalid method argument.");
25
26     return 0;
27 }
28
29 int findRoots(double (*f)(const double &x), const double &xa, const double &xb,
30             const double &tol, double roots[], int &nRoots, const int N,
31             const std::string method) {
32     if (method == "newton")
33         throw std::invalid_argument(
34             "Newton method isn't available with this prototype.");
35
36     return findRoots(f, nullptr, xa, xb, tol, roots, nRoots, N, method);
37 }
38
39 void bracket(double (*f)(const double &x), const double &xa, const double &xb,
40            double xL[], double xR[], const int &N, int &nRoots) {
41     double dx      = (xb - xa) / N;
42     double xi      = xa;
43     double xi_plus_one = xi + dx;
44     int root_counter = 0;
45
46     double fL = f(xi), fR;
47     for (int i = 0; i < N; i++) {
48         fR = f(xi_plus_one);
49         if (fL == 0.0 ||
50             fL * fR < 0) { // Check if there's a root in [xi, xi_plus_one)
51             xL[root_counter] = xi;
52             xR[root_counter] = xi_plus_one;
53
54             root_counter++;
55         }
56
57         // Shift interval
58         fL = fR;
59         xi = xi_plus_one;
60         xi_plus_one += dx;
61     }
62
63     nRoots = root_counter;
64 }
65
66 // =====
67 // Secant method
68 // =====
69
70 int secant(double (*f)(const double &x), double xa, double xb,
71          const double &xtol, const double &ftol, double &root, int &ntry) {
72     int max_ntry = 64;

```

```

73 double fa    = f(xa);
74 double fb    = f(xb);
75 double dx    = xb - xa;
76
77 // Handle fa, fb = 0
78 if (fa == 0.0) {
79     ntry = 0;
80     root = xa;
81     return 0;
82 } else if (fb == 0.0) {
83     ntry = 0;
84     root = xb;
85     return 0;
86 }
87
88 for (int k = 1; k <= max_ntry; k++) {
89     dx = fb * (xb - xa) / (fb - fa); // Compute increment
90
91     // Shift values
92     xa = xb;
93     fa = fb;
94     xb = xb - dx;
95     fb = f(xb);
96
97     // Check convergence
98     if (fabs(dx) < xtol || fabs(fb) < ftol || fb == 0.0) {
99         ntry = k;
100        root = xb;
101        return 0;
102    }
103 }
104
105 ntry = -1;
106 root = nan("");
107 throw std::runtime_error("Maximum number of steps exceeded.");
108 return 1;
109 }
110
111 int secant(double (*f)(const double &x), double xa, double xb,
112           const double &xtol, double &root) {
113     int n;
114     return secant(f, xa, xb, xtol, -1.0, root, n);
115 }
116
117 int secant(double (*f)(const double &x), double xa, double xb,
118           const double &xtol, double &root, int &ntry) {
119     return secant(f, xa, xb, xtol, -1.0, root, ntry);
120 }
121

```

```
122 int secant(double (*f)(const double &x), double xa, double xb,  
123           const double &xtol, const double &ftol, double &root) {  
124     int n;  
125     return secant(f, xa, xb, xtol, ftol, root, n);  
126 }
```

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