Projectile and heat equation

Marco Casari

December 11, 2022

1 Introduction

Same notation and formulas shown in file projectileheat_eq.pdf are used in this document. Whenever needed, a tolerance 10^{-7} is chosen.

2 Projectile Motion with Air Resistance

2.1 Initial shooting angles

A preliminary search of the intervals where angles ϕ_0 and ϕ_1 lie is performed by plotting the trajectory at different values of initial angle in the interval $(0°, 90°)$. Boundaries are excluded because the corresponding trajectories can not satisfy the boundary condition $y(L) = 0$: with initial angle $0°$ the projectile should have infinite speed but this is not a physical solution, with initial angle $90°$ the motion is directed only along the y axis. Values greater than $0°$ and less than $90°$ within tolerance are chosen instead.

This procedure provides some plots which show an unexpected behaviour. After an initial parabolic trend, the projectile is supposed to approach an asymptotic trajectory in $x = x_a$ with constant velocity (the terminal velocity v_t) directed only along the negative y axis. For high enough initial angles, trajectories display the asymptotic behaviour because $x_a < L$, but then their slope changes sign and they start to increase again, eventually approaching a linear trend.

The origin of this unexpected behaviour can be traced to the combination of machine precision and the equations governing the system. In fact, the integration method applies the same step to evaluate the functions at successive points x. When the asymptote is crossed, i.e. $x > x_a$, the same equations are evaluated but with $\phi(x) < -\frac{\pi}{2}$, hence $\tan(\phi) > 0$ and both y and v start to increase, up to a point where ϕ' has values below the machine epsilon. At that point, ϕ remains constant for successive iterations, keeping constant also y', resulting in an anomalous linear trajectory.

The initial interval is reduced to avoid plotting trajectories which show this non physical behaviour. An upper bound can be found imposing $x_a > L$ and without resolving explicitly the equation of motion, a conservative choice observed graphically for the actual parameters is $60°$. With this choice, initial shooting angles are included in intervals $[15°, 22.5°]$ and $[52.5°, 60°]$ and their values are

$$\phi_0 = (2.1800469 \times 10^1)° \tag{1}$$

$$\phi_1 = (5.3417848 \times 10^1)° \tag{2}$$

The corresponding trajectories are shown in figure 1.

2.2 Analytical derivation of initial shooting angles in absence of air resistance

The effect of air resistance can be removed by setting $C = 0$. The equations of motion become

$$\begin{cases} \frac{\mathrm{d}^2x}{\mathrm{d}t^2} = 0 \\ \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -g \end{cases} \tag{3}$$

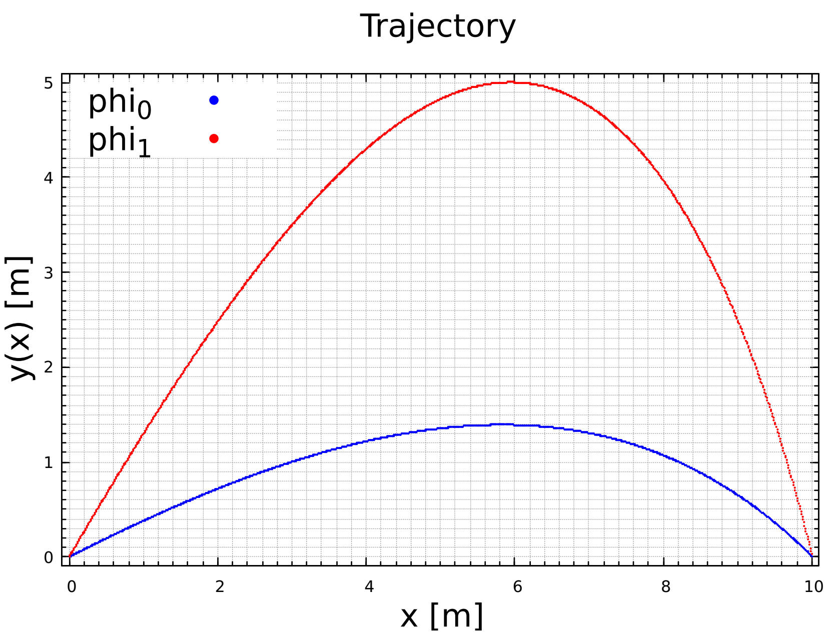

Figure 1: Trajectories having initial shooting angles ϕ_0 and ϕ_1.

where the mass of the projectile is already simplified. Their solutions, with initial values $x(0) = 0$ and $y(0) = 0$, are:

$$\begin{cases} x(t) = v_0\cos(\phi_0)t \\ y(t) = v_0\sin(\phi_0)t - \frac{1}{2}gt^2 \end{cases} . \tag{4}$$

Function $y(t)$ can be rewritten as function of x by inverting $x(t)$ in equation 4:

$$y(x) = \tan(\phi_0)x - \frac{1}{2}\frac{g}{v_0^2\cos(\phi_0)^2}x^2 . \tag{5}$$

Values for the initial shooting angle can be found by applying the boundary condition, which is equivalent to solve equation $y(L) = 0$ for the unknown ϕ_0. The interval of acceptable values for ϕ_0 is discussed in section 2.1 and since $\phi_0 \neq \frac{\pi}{2}$, the resulting values are

$$\phi_0 = \frac{1}{2}\arcsin\left(\frac{gL}{v_0^2}\right) = 8.8122598^\circ \tag{6}$$

$$\phi_1 = \frac{\pi}{2} - \frac{1}{2}\arcsin\left(\frac{gL}{v_0^2}\right) = (8.1187740 \times 10^1)^\circ \tag{7}$$

2.3 Source code

During calculations, angles are expressed in radians. Elements of arrays `Y_0` and `Y_1` are ordered as the equations in file projectileheat_eq.pdf: the first element represents y, the second v and the third ϕ.

```
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>

#define HOMEWORK_NAME "projectile"
#define TOLERANCE 1e-7
#define N_PRECISION 9

/* Global variables */
```

```
static double const global_g = 9.81;
static double const global_C = 0.1;
static double const global_v_0 = 18.0;
static double const global_x_0 = 0.0;
static double const global_L = 10.0;
static double const global_y_0 = 0.0;

/* Function prototypes */
void rhs(double const x, double const Y[], double R[]);
double residual(double phi);

int main() {
    // Setup.
    using namespace std;
    cout << setprecision(N_PRECISION) << scientific;
    int const n_eq = 3;
    int const n_x = 1025;
    double const dx = (global_L - global_x_0) / (n_x - 1);
    double x;
    double phi_0, phi_1;  // Unit of measurement: radian.
    double Y_0[n_eq], Y_1[n_eq];
    ofstream plot_file;

    // Find initial shooting angles.
    phi_0 = bisection(residual, M_PI / 12.0, M_PI / 8.0, TOLERANCE);
    phi_1 = bisection(residual, 7.0 * M_PI / 24.0, M_PI / 3.0, TOLERANCE);
    cout << "phi_0 = " << 180.0 / M_PI * phi_0 << " deg" << endl;
    cout << "phi_1 = " << 180.0 / M_PI * phi_1 << " deg" << endl;

    // Evaluate trajectories and save points to file.
    Y_0[0] = global_y_0;  // Physical quantity y [m]
    Y_0[1] = global_v_0;  // Physical quantity v [m / s]
    Y_0[2] = phi_0;  // Physical quantity phi [rad]
    Y_1[0] = global_y_0;  // Physical quantity y [m]
    Y_1[1] = global_v_0;  // Physical quantity v [m / s]
    Y_1[2] = phi_1;  // Physical quantity phi [rad]
    plot_file.open(HOMEWORK_NAME ".dat");
    plot_file << setprecision(N_PRECISION) << scientific;
    plot_file << "x y_0(x) y_1(x)" << endl;
    for (int i_x = 0; i_x < n_x; i_x++) {
        x = global_x_0 + i_x * dx;
        plot_file << x << ' ' << Y_0[0] << ' ' << Y_1[0] << endl;
        rungekutta4(x, dx, Y_0, rhs, n_eq);
        rungekutta4(x, dx, Y_1, rhs, n_eq);
    }

    // Teardown.
    plot_file.close();
    return 0;
}

void rhs(double const x, double const Y[], double R[]) {
    R[0] = tan(Y[2]);
    R[1] = - global_g / Y[1] * tan(Y[2]) - global_C * Y[1] / cos(Y[2]);
    R[2] = - global_g / (Y[1]*Y[1]);
}

double residual(double phi) {
    int const n_eq = 3;
    int const n_x = 1025;
    double const dx = (global_L - global_x_0) / (n_x - 1);
    double x;
    double Y[n_eq];
    Y[0] = global_y_0;  // Physical quantity y [m]
    Y[1] = global_v_0;  // Physical quantity v [m / s]
    Y[2] = phi;  // Physical quantity phi [rad]
    for (int i_x = 0 ; i_x < n_x; i_x++) {
        x = global_x_0 + i_x * dx;
        rungekutta4(x, dx, Y, rhs, n_eq);
    }
    return Y[0];  // Boundary value is y(L) = 0.
}
```

3 1D Heat Equation

3.1 Tridiagonal matrix derivation

In this section physical quantities are considered dimensionless.

The theta-rule can be rewritten to isolate at each side of the equation the terms of the temperature distribution at time steps n and $n+1$:

$$-\alpha\theta T_{i-1}^{n+1} + (1+2\alpha\theta)T_i^{n+1} - \alpha\theta T_{i+1}^{n+1} = \alpha(1-\theta)T_{i-1}^n + (1-2\alpha(1-\theta))T_i^n + \alpha(1-\theta)T_{i+1}^n \tag{8}$$

For $i = 1, \ldots, N_x - 2$, equation 8 forms a linear system in the unknown values T_i^{n+1} and with constant terms evaluated on the distribution values T_i^n at the previous time step n, which are known. The linear system can be represented using a tridiagonal matrix

$$\begin{bmatrix} c_2 & c_1 & 0 & \cdots & 0 \\ c_1 & c_2 & c_1 & \ddots & \vdots \\ 0 & c_1 & c_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_1 \\ 0 & \cdots & 0 & c_1 & c_2 \end{bmatrix} \begin{pmatrix} T_1^{n+1} \\ T_2^{n+1} \\ T_3^{n+1} \\ \vdots \\ T_{N_x-2}^{n+1} \end{pmatrix} = \begin{pmatrix} b_1^n \\ b_2^n \\ b_3^n \\ \vdots \\ b_{N_x-2}^n \end{pmatrix} \tag{9}$$

with coefficients

$$c_1 = -\alpha\theta = -8.1281250\,, \tag{10}$$

$$c_2 = 1 + 2\alpha\theta = 1.7256250 \times 10^1 \tag{11}$$

and constant terms for $i = 2, \ldots, N_x - 3$ equal to the right hand side of equation 8:

$$\begin{aligned} b_i^n &= \alpha(1-\theta)T_{i-1}^n + (1-2\alpha(1-\theta))T_i^n + \alpha(1-\theta)T_{i+1}^n \\ &= (\alpha + c_1)T_{i-1}^n + (c_2 - 2\alpha)T_i^n + (\alpha + c_1)T_{i+1}^n\,. \end{aligned} \tag{12}$$

Constant terms for $i = 1$ and $i = N_x - 2$ are obtained from equation 8 by applying the boundary conditions to have an explicit expression for T_0^{n+1} and $T_{N_x-1}^{n+1}$ respectively, obtaining:

$$b_1^n = (\alpha + c_1)T_0^n + (c_2 - 2\alpha)T_1^n + (\alpha + c_1)T_2^n - c_1 f_L(t_{n+1}) \tag{13}$$

$$b_{N_x-2}^n = (\alpha + c_1)T_{N_x-3}^n + (c_2 - 2\alpha)T_{N_x-2}^n + (\alpha + c_1)T_{N_x-1}^n - c_1 f_R(t_{n+1}) \tag{14}$$

3.2 Temperature distribution plot

Temperature distribution at intermediate and final time steps is displayed in figure 2.

3.3 Source code

```
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>

#define HOMEWORK_NAME "heat_eq"
#define TOLERANCE 1e-7
#define N_PRECISION 9

/* Function prototypes */
double T_0(double x);
```

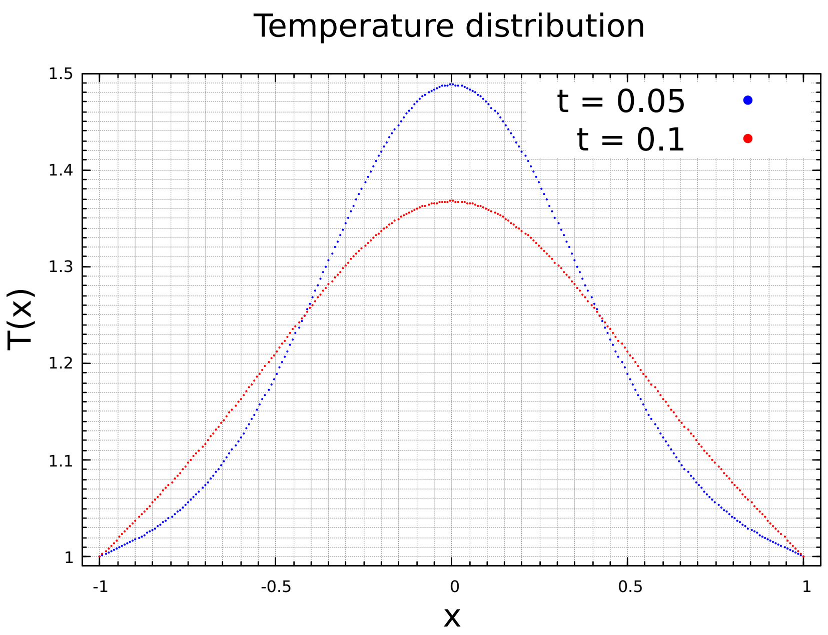

Figure 2: Temperature distribution at intermediate $t = 0.05$ and final $t = 0.1$ time steps, starting from a gaussian temperature distribution.

```cpp
double f_L(double t);
double f_R(double t);

int main() {
    // Set up.
    using namespace std;
    cout << setprecision(N_PRECISION) << scientific;
    // Parameters.
    int const N_x = 256;
    int const N_t = 100;
    double const t_e = 0.1;
    double const x_b = -1.0;
    double const x_e = 1.0;
    double const k = 1.0;
    double const theta = 0.5;
    double const dt = t_e / N_t;
    double const dx = (x_e - x_b) / (N_x - 1);
    double const alpha = k * dt / (dx*dx);
    double const c_1 = - alpha * theta;
    double const c_2 = 1.0 - 2.0 * c_1;
    // Variables.
    double t;
    double x[N_x], d_inf[N_x], d[N_x], d_sup[N_x], b[N_x], T[N_x];
    ofstream plot_file;

    // Set spatial grid and tridiagonal matrix coefficients.
    for (int i = 0; i < N_x; i++) {
        x[i] = x_b + i * dx;
        d_inf[i] = c_1;
        d[i] = c_2;
        d_sup[i] = c_1;
    }

    // Assign initial and boundary values.
    T[0] = f_L(0.0);
    for (int i = 1; i <= N_x - 2; i++) {
        T[i] = T_0(x[i]);
    }
    T[N_x-1] = f_R(0.0);

    // Loop over time to evolve temperature distribution.
    plot_file.open(HOMEWORK_NAME ".dat");
    plot_file << setprecision(N_PRECISION) << scientific;
    for (int n = 0; n <= N_t; n++) {
        t = n * dt;
        // Save solutions to file.
        if (n == N_t / 2 || n == N_t) {
            for (int i = 0; i < N_x; i++) {
                plot_file << x[i] << ' ' << T[i] << endl;
            }
            plot_file << '\n' << endl;
            cout << "Temperature distribution at t = " << t << " saved to file" << endl;
        }
        // Assign values to constant terms of the linear system.
        for (int i = 1; i <= N_x - 2; i++) {
            b[i] = (alpha + c_1) * (T[i-1] + T[i+1]) + (c_2 - 2.0 * alpha) * T[i];
        }
        b[1] -= c_1 * f_L(t + dt);
        b[N_x-2] -= c_1 * f_R(t + dt);
        // Evaluate T(x) at next time step.
        tridiagonal_solver_2(d_inf+1, d+1, d_sup+1, b+1, T+1, N_x-2);
    }

    // Teardown.
    plot_file.close();
    return 0;
}

double T_0(double x) {
    return 1.0 + exp(-x*x * 16.0);
}

double f_L(double t) {
    return 1.0;
}

double f_R(double t) {
    return 1.0;
}
```