Assignment

Projectile Motion with Air Resistance and 1D Heat Equation

Elena Scibona

December 8, 2022

1 Projectile Motion

Numerical solution with air resistance

A projectile is fired at the origin of the coordinates with velocity $v_0 = 18$ m/s. Its motion can be modelled with the system of equations

$$\begin{cases} y' = \tan\phi \\ v' = -\frac{g}{v}\tan\phi - Cv\sec\phi \\ \phi' = -\frac{g}{v^2} \end{cases}$$

where y is the vertical spatial coordinate, v indicates the velocity of the projectile and ϕ represents the angle of the trajectory with the horizontal x-axis. The gravitational acceleration is $g = 9.81$ m/s^2 and $C = 0.1$ m^{-1}. Given the further condition that the second intersection of the trajectory with the x-axis is at $L = 10$ m, the system takes the form of a standard Boundary Value Problem. In order to evaluate the two possible initial angles ϕ_0 and ϕ_1, the so called "Shooting Method" is applied.

At first, trial values for the initial angle were guessed in the range $[1, 89]$ degrees. The corresponding plots of $y(x)$ correctly showed the existence of two acceptable values below $\pi/3$ radians, while a little over this threshold they displayed an unusual behaviour. This led to the detection of additional unphysical solutions, which arise when in the range $[0, 10]$ the function $\phi(x)$ reaches and exceeds the value $-\pi/2$, a singularity for $\tan\phi$: since $y' \propto \tan\phi$, $y(x)$ is not differentiable in this domain. For large initial angles the algorithm fails to correctly model the system because finite-precision computing struggles with asymptotic behaviour. In order to avoid such problem, in the final version of the code the search of ϕ_0 and ϕ_1 is confined in the range $[\pi/12, \pi/3]$.

Using the previously implemented function `Bracketing(...)`, two smaller intervals are selected, each containing one and only one solution. At this point, the root-finding problem is solved by applying the bisection method to `Residual(...)`, a function that returns the value of the vertical coordinate in L given an initial ϕ.

The tolerance required of `Bisection(...)` is 10^{-8}, while the number of fourth-order Runge-Kutta steps in `Residual(...)` has been chosen *a posteriori*: larger values would return the same angles (considering only significant figures), but with more computing.

The output of the code is:

```
Initial angles:
phi_0 = 21.742704 degrees
phi_1 = 53.487076 degrees
```

Analytic solution without air resistance

The equations of motion for this problem are

$$\begin{cases} x = vt\cos\phi \\ y = vt\sin\phi - \frac{1}{2}gt^2 \end{cases}$$

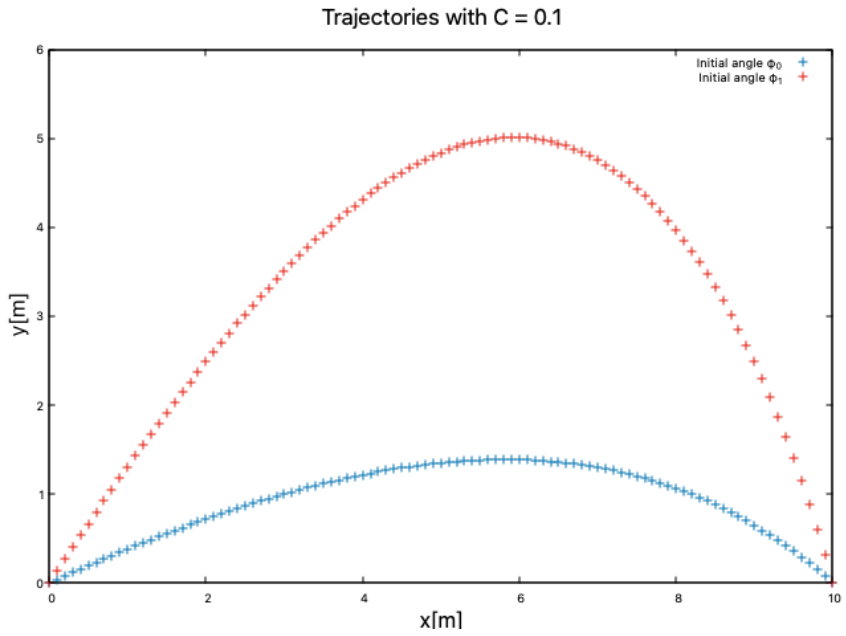

Figure 1: Gnuplot plot from data in "data.dat". The number of Runge-Kutta steps used to produce the plot is the same used in `Residual(...)`.

with v the magnitude of the initial velocity and ϕ the angle at $t = 0$. The system can be rewritten expressing t as a function of x, $t = x/(v\cos\phi)$, and inserting this expression in the second equation:

$$\begin{aligned} y &= x\tan\phi - \frac{g}{2}\frac{x^2}{v^2\cos^2\phi} \\ &= x\tan\phi - \frac{g}{2}\frac{x^2}{v^2}(1+\tan^2\phi) \quad . \end{aligned}$$

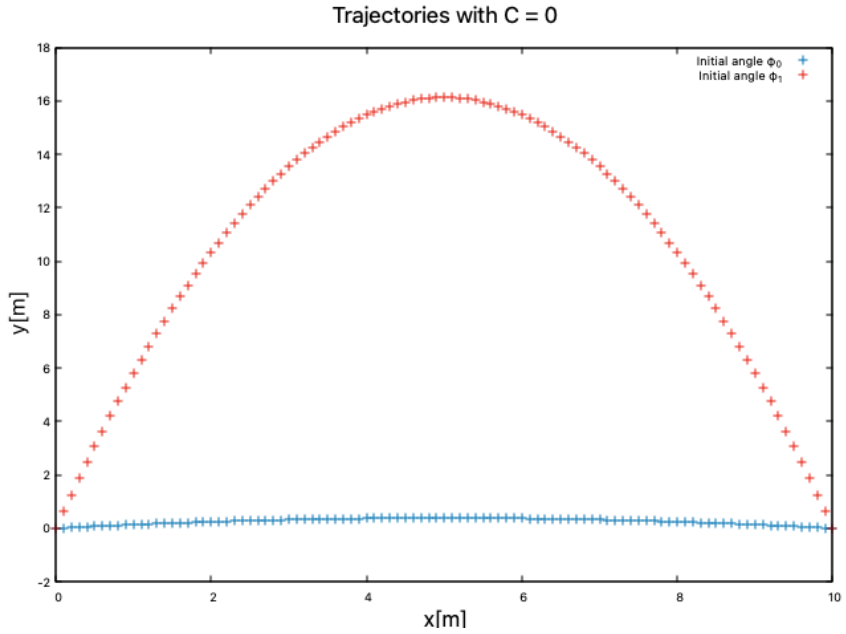

Figure 2: When $C = 0$ the asymmetry due to air resistance is removed and $\phi_0 + \phi_1 = \pi/2$.

Since $P = (0, 10)$ is a point of the trajectory, we can replace $x \to 10$ and $y \to 0$, insert the given values for v and g and solve the second order equation

$$\frac{10^2 \times 9.81}{2 \times 18^2} a^2 - 10a + \frac{10^2 \times 9.81}{2 \times 18^2} = 0 \qquad \text{(with a} = \tan\phi\text{)}$$

to calculate the initial angles.

The first solution, $a \approx 0.155$, gives $\phi \approx 8.81$ degrees, while the second one, $a \approx 6.450$, corresponds to $\phi \approx 81.18$ degrees[1].

Code

The constants C and g, together with the chosen tolerance, are not defined as variables: their value is given by preprocessor directives.
Angles are expressed in radians in order to use the goniometric functions of the standard library `cmath`.

```
#include "lib.h"

using namespace std;

#define CONST_C 0.1
#define CONST_G 9.81
#define TOL 1e-8

void RHS_Func(double, double*, double*);
double Residual(double);

////////////////////////////////////////////////////////////////////////////
int main(){

    ofstream out;
    out.open("data.dat");
    if(out.fail()){
        cerr << "\nError: stream to \"dati.dat\" failed.";
        exit(1);
    }

    const double x_b = 0., x_e = 10.;
    const int nsteps = 100;
    const int dim = 3;
    double* Y = new double[dim];
    double dx = fabs(x_e - x_b) / (double)nsteps;

    const int N = 100;
    const double phi_l = M_PI / 12., phi_r = M_PI / 3.;
    double h = fabs(phi_r - phi_l) / (double)N;
    double* x_0 = new double[N];

    int roots = Bracket(Residual, phi_l, phi_r, N, x_0);
    double* angle = new double[roots];

    cout << "\n" << "Initial angles:";

    for(int i = 0; i < 2; i++){
        Bisection(Residual, x_0[i], x_0[i] + h, TOL, angle[i]);
        cout << "\nphi_" << i << " = ";
        cout << setprecision(8) << angle[i] * 180. / M_PI << " degrees";
        double x = x_b;
        int count = 1;
        Y[0] = 0.;
        Y[1] = 18.;
        Y[2] = angle[i];
        out << "\n" << setw(8) << x << setw(15) << Y[0];
```

[1] Air resistance is taken into account by the term $\propto C$ in the equations used to model the system and can be easily removed. If the code is slightly modified ($C = 0$, `phi_l` $= \pi/38$ and `phi_r` $= \pi/2$), the evaluation of ϕ_0 and ϕ_1 returns values consistent with the analytical ones.

```
48
49          do{
50              RK4_Step(x, Y, dx, dim, RHS_Func);
51              x = x_b + dx * count;
52              out << "\n" << setw(8) << x << setw(15) << Y[0];
53              count++;
54          }while(x < x_e);
55
56          out << "\n\n";
57      }
58
59      out.close();
60
61      delete[] Y;
62      delete[] x_0;
63      delete[] angle;
64
65      return 0;
66  }
67  ////////////////////////////////////////////////////////////////////////////
68  void RHS_Func(double x, double* Y, double* R){
69      double y = Y[0];
70      double v = Y[1];
71      double phi = Y[2];
72
73      R[0] = tan(phi);
74      R[1] = - (double)CONST_G * R[0] / v - CONST_C * v / cos(phi);
75      R[2] = - CONST_G / v / v;
76  }
77
78  double Residual(double phi){
79      static double x_b = 0., x_e = 10.;
80      static int nsteps = 100;
81      static int dim = 3;
82      double dx = fabs(x_e - x_b) / (double)nsteps;
83      double* Y = new double[dim];
84
85      double x = x_b;
86      int count = 1;
87      Y[0] = 0.;
88      Y[1] = 18.;
89      Y[2] = phi;
90
91      do{
92          RK4_Step(x, Y, dx, dim, RHS_Func);
93          x = x_b + dx * count;
94          count++;
95      }while(x < x_e);
96
97      double residual = Y[0];
98
99      delete[] Y;
100     return residual;
101 }
```

2 1D Heat Equation

2.1 Preliminary analysis

Given the initial temperature distribution

$$T(x,0) = 1 + e^{-x^2/a^2} \qquad (\text{with } a = 0.25)$$

where $x \in [-1,1]$ and the boundary conditions

$$T(-1,t) = T(1,t) = 1 \quad ,$$

the 1D heat equation

$$\frac{\partial T(x,t)}{\partial t} = k\frac{\partial^2 T(x,t)}{\partial^2 x} \qquad (\text{with } k=1)$$

can be used to compute the values of the temperature in a given position after a certain time. The strategy to study this system consists in discretizing the problem ($x \to x_i$ with $i = 0, 1, ...N_x - 1$ and $t \to t_n$ with $n = 0, 1, ..., N_t$) and using an iterative procedure that at every iteration computes the value of the temperature in the domain at the successive time step.

The linear combination of forward and backward Euler scheme, obtained combining the forward/backward approximation to time derivative with a central approximation to space derivative, gives the so called "Theta Rule", which can be written explicitly as:

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

$$\Updownarrow$$

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

where $\alpha = k\Delta t/\Delta x^2$.
In a simpler notation:

$$\epsilon T_{i-1}^{n+1} + \eta T_i^{n+1} + \epsilon T_{i+1}^{n+1} = F_i^n \qquad \text{with} \qquad F_i^n = (\alpha - \alpha\theta)(T_{i-1}^n + T_{i+1}^n) + (1 - 2\alpha + 2\alpha\theta)T_i^n.$$

Since the the value of T_i^0 is given $\forall i$ by the initial conditions and $T(x_0, t) = T_0^n$ as well as $T(x_{N_x-1}, t) = T_{N_x-1}^n$ are constant for the boundary conditions, the right side of the equation is a constant term and the system is in tridiagonal form. Indeed, it is equal to:

$$\begin{pmatrix} \eta & \epsilon & 0 & 0 & 0 & 0 \\ \epsilon & \eta & \epsilon & 0 & 0 & 0 \\ 0 & \epsilon & \eta & \epsilon & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & \epsilon & \eta & \epsilon \\ 0 & 0 & 0 & 0 & \epsilon & \eta \end{pmatrix} \begin{pmatrix} T_1^1 \\ T_2^1 \\ T_3^1 \\ \vdots \\ T_{N_x-3}^1 \\ T_{N_x-2}^1 \end{pmatrix} = \begin{pmatrix} F_1^0 - \epsilon T_0^1 \\ F_2^0 \\ F_3^0 \\ \vdots \\ F_{N_x-3}^0 \\ F_{N_x-2}^0 - \epsilon T_{N_x-1}^1 \end{pmatrix}.$$

Therefore T_i^1 can be computed using the tridiagonal matrix solver algorithm and a new vector of constant terms F_i^1 is calculated. The process can be iterated in order to get the temperature distribution at successive time steps.

Code

Preprocessor directives are used in the code to set the boundary conditions and the values of parameters fixed by the assignment. Moreover, at line 11 and 12 two macros are introduced for a clearer management of indexes.
Even if the tridiagonal solver affects only the points inside the perimeter of the grid, all arrays in the code are created with the same dimension N_x. This improves code readability, but must be taken into account when calling the function `Tridiagonal(...)`: the `double*` current parameters use pointer arithmetic to avoid the first meaningless element of the array and the `int` parameter discards the last one as well.
The function `U_0_X(...)` computes the specific initial conditions of this problem, then it is declared and defined in the main file. Instead, `Tridiagonal(...)` is implemented in the local library.
Unnecessary variables were created to store the values of the coefficients of the tridiagonal matrices, with the purpose of making the code clearer.

```
1 #include "lib.h"
2
3 using namespace std;
4
5 #define THETA 0.5
6 #define CONST_K 1.
```

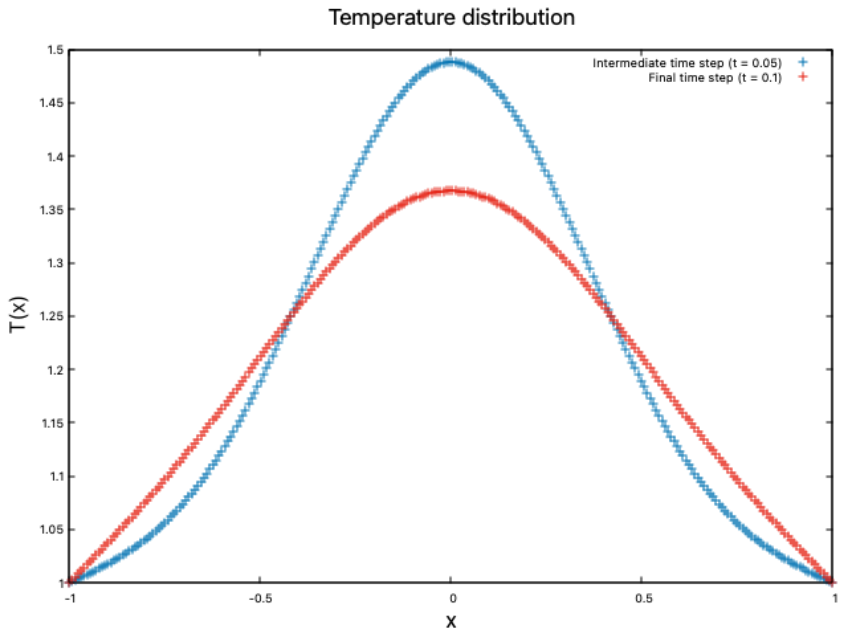

Figure 3: Gnuplot plot from data in "data.dat". The overall look of the graphs is in accordance with the expectations: the distribution is symmetrical since both the initial and boundary conditions are symmetrical in the domain $[-1, 1]$ and the comparison between different time steps shows that it widens and lowers with increasing time.

```
#define REVERSE_CONST_A 1. / 0.25
#define N_X 256
#define N_T 100
#define B_C 1.
#define IBEG_X 1
#define IEND_X N_X - 2


double U_0_X(double);

////////////////////////////////////////////////////////////////////////

int main(){

    ofstream out;
    out.open("data.dat");
    if(out.fail()){
        cerr << "\nError: stream to \"data.dat\" failed.";
        exit(1);
    }

    const double x_b = -1., x_e = 1.;
    const double t_b = 0., t_e = 0.1;

    double dx = fabs(x_e - x_b) / (double)(N_X - 1.);
    double dt = fabs(t_e - t_b) / (double)N_T;
    unsigned int dim = N_X;

    double* x = new double[dim];
    for(int i = 0; i < N_X; i++)
        x[i] = x_b + dx * i;

    double* u = new double[dim];
    double* const_terms = new double[dim];
    double* a = new double[dim];
    double* b = new double[dim];
    double* c = new double[dim];
    double cfl = CONST_K * dt / dx / dx;
```

```
    double coeff_a = - cfl * THETA;
    double coeff_b =  1. + 2. * cfl * THETA;
    double coeff_c = coeff_a;

//Initialization of the arrays
   for(int i = IBEG_X + 1; i < IEND_X; i++){
        const_terms[i] = cfl * (1. - THETA) * (U_0_X(x[i-1]) + U_0_X(x[i+1])) + (1. - 2. * cfl * (1. - THETA)) * U_0_X(x[i]);
        a[i] = coeff_a;
        b[i] = coeff_b;
        c[i] = coeff_c;
    }

    const_terms[IBEG_X] = cfl * (1. - THETA) * (B_C + U_0_X(x[IBEG_X+1])) + (1. - 2. * cfl * (1. - THETA)) * U_0_X(x[IBEG_X]) - B_C * coeff_a;
    const_terms[IEND_X] = cfl * (1. - THETA) * (U_0_X(x[IEND_X-1]) + B_C) + (1. - 2. * cfl * (1. - THETA)) * U_0_X(x[IEND_X]) - B_C * coeff_c;
    a[IBEG_X] = 0.; a[IEND_X] = coeff_a;
    b[IBEG_X] = coeff_b; b[IEND_X] = coeff_b;
    c[IBEG_X] = coeff_c; c[IEND_X] = 0.;

    Tridiagonal(a + 1, b + 1, c + 1, const_terms + 1, u + 1, dim - 2);

//Iteration over time
    for(int j = 1; j < N_T; j++){

        for(int i = IBEG_X + 1; i < IEND_X; i++)
            const_terms[i] = cfl * (1. - THETA) * (u[i-1] + u[i+1]) + (1. - 2. * cfl * (1. - THETA)) * u[i];

        const_terms[IBEG_X] = cfl * (1. - THETA) * (B_C + u[IBEG_X+1]) + (1. - 2. * cfl * (1. - THETA)) * u[IBEG_X] - B_C * coeff_a;
        const_terms[IEND_X] = cfl * (1. - THETA) * (u[IEND_X-1] + B_C) + (1. - 2. * cfl * (1. - THETA)) * u[IEND_X] - B_C * coeff_c;

        Tridiagonal(a + 1, b + 1, c + 1, const_terms + 1, u + 1, dim - 2);
        if(j == N_T * 0.5 - 1 || j == N_T - 1){
            u[0] = B_C;
            u[dim-1] = B_C;

            for(int i = 0; i < dim; i++)
                out << "\n" << setw(15) << x[i] << setw(15) << u[i];

            out << "\n\n";
        }
    }

    out.close();

    delete[] x;
    delete[] const_terms;
    delete[] u;
    delete[] a;
    delete[] b;
    delete[] c;

    return 0;
}

//////////////////////////////////////////////////////////////////////

double U_0_X(double x){
    return 1. + exp(-x * x * REVERSE_CONST_A * REVERSE_CONST_A);
}
```