Chapter 1

Ordinary Differential equations

1.1 Introduction to Ordinary Differential equation

In mathematics, an ordinary differential equation (ODE) is an equation which contains functions of only one variable and its derivatives.

An ordinary differential equation of order n has the following form

$$y^{(n)} = f(x, y, y', y'', \cdots, y^{(n-1)}) \tag{1.1}$$

Where
x is the independent variable,
$y^{(i)} = \frac{d^i y}{dx^i}$ is the ith derivative of $y(x)$,
f is the force term.

Example

Arguably, the most famous ODE of the second order is Newton's second law connecting the acceleration of the body (a) to the force (F) acting on it:

$$a(t) = \frac{F}{m}$$

where m is the mass of the body and t is time. For simplicity, we talk here only about 'y' component of the position. Recall that acceleration is the second derivative of the position, i.e. $a(t) = y''(t)$. The force function F might depend on time, the body position, and its velocity (i.e. the first derivative of position y'), so F should be written as $F(t, y, y')$. So, we rewrite Newton's second law as the 2nd order ODE:

$$y'' = \frac{F(t, y, y')}{m} = f(t, y, y') \tag{1.2}$$

As you can see, time serves as independent variable. We can obtain the canonical eq. (1.1) by simple relabeling $t \to x$.

Any nth order ODE (such as eq. (1.1)) can be transformed to a system of the first order ODEs.

Transformation of the n$_{th}$ order ODE to the system of the first order ODEs

We define the following variables

$$y_1 = y, y_2 = y', y_3 = y'', \cdots, y_n = y^{(n-1)} \tag{1.3}$$

Then we can write the following system of equations

$$\begin{pmatrix} y_1' \\ y_2' \\ y_3' \\ \vdots \\ y_{n-1}' \\ y_n' \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} = \begin{pmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \\ f(x, y_1, y_2, y_3, \cdots y_n) \end{pmatrix} \tag{1.4}$$

We can rewrite eq. (1.4) in a much more compact vector form

The canonical form of the ODEs system

$$\vec{y}\,' = \vec{f}(x, \vec{y}\,) \tag{1.5}$$

Example

Let's convert Newton's second law (eq. (1.2)) to the system of the 1st order ODEs. The acceleration of a body is the first derivative of velocity with respect to time and is equal to the force divided by mass

$$\frac{dv}{dt} = v'(t) = a(t) = \frac{F}{m}$$

Also, we recall the velocity itself is the derivative of the position with respect to time.

$$\frac{dy}{dt} = y'(t) = v(t)$$

Combining the above, we rewrite eq. (1.2) as

$$\begin{pmatrix} y' \\ v' \end{pmatrix} = \begin{pmatrix} v \\ f(t, y, v) \end{pmatrix} \tag{1.6}$$

We do the following variables relabeling $t \to x$, $y \to y_1$, and $v \to y_2$ and rewrite our equation in the canonical form resembling eq. (1.4):

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} y_2 \\ f(x, y_1, y_2) \end{pmatrix} \tag{1.7}$$

1.2 Boundary conditions

The system of n ODEs requires n constraints to be fully defined. This is done by providing the *boundary conditions*. There are several alternative ways to do it. The most intuitive way is by specifying the full set $\vec{y}$ components at some starting position x_0, i.e. $\vec{y}(x_0) = \vec{y_0}$. This is called the *initial value problem*.

The alternative way is to specify some components of $\vec{y}$ at the starting value x_0 and the rest at the final value x_f. The problem specified this way is called the *two-point boundary value problem*.

In this chapter, we will consider only the initial value problem and its solutions.

The initial value problem boundary conditions

We need to specify all components of the $\vec{y}$ at the initial position x_0.

$$\begin{pmatrix} y_1(x_0) \\ y_2(x_0) \\ y_3(x_0) \\ \vdots \\ y_n(x_0) \end{pmatrix} = \begin{pmatrix} y_{1_0} \\ y_{2_0} \\ y_{3_0} \\ \vdots \\ y_{n_0} \end{pmatrix} = \begin{pmatrix} y_0 \\ y'_0 \\ y''_0 \\ \vdots \\ y_0^{(n-1)} \end{pmatrix}$$

For the Newton's second law example, which we considered above, the boundary condition requires us to specify initial position and velocity of the object in addition to eq. (1.7). Then, the system is fully defined and has only one possible solution.

1.3 Numerical method to solve ODEs

1.3.1 Euler's method

Let's consider the simplest case: a first order ODE (notice the lack of the vector notation)

$$y' = f(x, y)$$

There is an exact way to write the solution

$$y(x_f) = y(x_0) + \int_{x_0}^{x_f} f(x, y)dx$$

The problem with above formula is that the $f(x, y)$ depends on the y itself. However, on a small enough interval $[x, x+h]$, we can assume that $f(x, y)$ does not change, i.e. it is constant. In this case, we can use the familiar rectangles integration formula (see section 9.2).

$$y(x + h) = y(x) + f(x, y)h$$

When applied to an ODE, this process is called Euler's method.

We need to split our $[x_0, x_f]$ interval into a bunch of steps of the size h and leap frog from the x_0 to the $x_0 + h$, then to the $x_0 + 2h$, and so on.

Now, we can make an easy transformation to the vector case (i.e. the n_{th} order ODE):

Euler's method (error $\mathcal{O}(h^2)$)

$$\vec{y}(x + h) = \vec{y}(x) + \vec{f}(x, y)h$$

The MATLAB implementation of Euler's method is shown below

Listing 1.1: odeeuler.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_ode/code/odeeuler.m)

```
function [x,y]= odeeuler(fvec, xspan, y0, N)
    %% Solves a system of ordinary differential equations with the Euler
       method
    % x — column vector of x positions
    % y — solution array values of y, each row corresponds to particular row
        of x.
    %     each column corresponds, to a given derivative of y,
    %     including y(:,1) with no derivative
    % fvec  — handle to a function f(x,y) returning forces column vector
    % xspan — vector with initial and final x coordinates i.e. [x0, xf]
    % y0    — initial conditions for y, should be row vector
    % N     — number of points in the x column (N>=2),
    %         i.e. we do N—1 steps during the calculation

    x0=xspan(1);         % start position
    xf=xspan(2);         % final position

    h=(xf—x0)/(N—1);     % step size
    x=linspace(x0,xf,N); % values of x where y will be evaluated

    odeorder=length(y0);
    y=zeros(N,odeorder); % initialization
    x(1)=x0; y(1,:)=y0;  % initial conditions

    for i=2:N % number of steps is less by 1 then number of points since we
       know x0,y0
        xprev=x(i—1);
        yprev=y(i—1,:);
        % Matlab somehow always send column vector for 'y' to the forces
           calculation code
        % transposing yprev to make this method compatible with Matlab.
        % Note the dot in .' this avoid complex conjugate transpose
        f=fvec(xprev, yprev.');
        % we receive f as a column vector, thus, we need to transpose again
        f=f.';
        ynext=yprev+f*h;  % vector of new values of y: y(x+h)=y(x)+f*h
        y(i,:)=ynext;
    end
end
```

Similarly to the rectangle integration method, which is inferior in comparison to more advance methods (for example, the trapezoidal and Simpson's), the Euler method is less precise for a given h. There are better algorithms, which we discuss below.

1.3.2 The second-order Runge-Kutta method (RK2)

Using multivariable calculus and the Taylor expansion, as shown for example in [1], we can write

$$\begin{aligned}\vec{y}(x_{i+1}) = \vec{y}(x_i + h) = \\ &= \vec{y}(x_i) + C_0\vec{f}(x_i, \vec{y}_i)h + C_1\vec{f}(x_i + ph, \vec{y}_i + qh\vec{f}(x_i, \vec{y}_i))h + \mathcal{O}(h^3)\end{aligned}$$

where C_0, C_1, p and q are some constant satisfying the following set of constraints

$$C_0 + C_1 = 1 \tag{1.8}$$

$$C_1 p = 1/2 \tag{1.9}$$

$$C_1 q = 1/2 \tag{1.10}$$

It is clear that the system is under-constrained since we have only 3 equations for 4 constants. There are a lot of possible choices of parameters C_0, C_1, p, and q. One choice has no advantage over another.

But there is one "intuitive" choice: $C_0 = 0$, $C_1 = 1$, $p = 1/2$, and $q = 1/2$. It provides the following recipe for how to find $\vec{y}$ at the next position after the step h.

Modified Euler's method or midpoint method (error $\mathcal{O}(h^3)$)

$$\begin{aligned}\vec{k}_1 &= h\vec{f}(x_i, \vec{y}_i) \\ \vec{k}_2 &= h\vec{f}(x_i + \frac{h}{2}, \vec{y}_i + \frac{1}{2}\vec{k}_1) \\ \vec{y}(x_i + h) &= \vec{y}_i + \vec{k}_2\end{aligned}$$

As the name suggest, we calculate what $\vec{y}(x + h)$ could be with the Euler-like method by calculating $\vec{k}_1$, but then we do only a half step in that direction and calculate the updated force vector in the midpoint. Finally, we use this force vector to find the improved value of $\vec{y}$ at $x + h$.

1.3.3 The fourth-order Runge-Kutta method (RK4)

A higher order expansion of the $\vec{y}(x + h)$ also allows multiple choices of possible expansion coefficients (see [1]). One of the "canonical" choices (see [2]) is spelled out in

The fourth-order Runge-Kutta method with truncation error $\mathcal{O}(h^5)$

$$\begin{aligned}
\vec{k}_1 &= h\vec{f}(x_i, \vec{y}_i)\\
\vec{k}_2 &= h\vec{f}(x_i + \frac{h}{2}, \vec{y}_i + \frac{1}{2}\vec{k}_1)\\
\vec{k}_3 &= h\vec{f}(x_i + \frac{h}{2}, \vec{y}_i + \frac{1}{2}\vec{k}_2)\\
\vec{k}_4 &= h\vec{f}(x_i + h, \vec{y}_i + \vec{k}_3)\\
\vec{y}(x_i + h) &= \vec{y}_i + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)
\end{aligned}$$

1.3.4 Other numerical solvers

By no means have we covered all methods of solving ODEs. So far, we only talked about fixed step *explicit* methods. When the force term is changing slowly, it is reasonable to increase the step size h, or decrease it when the force term is quickly varying at the given interval. This leads to a slew of adaptive methods. There are also *implicit* methods, where one solves for $\vec{y}(x_i + h)$ satisfying the following equation

$$\vec{y}(x_i) = \vec{y}(x_i + h) - f(x, \vec{y}(x_i + h))h \tag{1.11}$$

Such implicit methods are more robust, but they are computationally more demanding. Several other ODEs solving algorithms are covered for example in [1, 2].

1.4 Stiff ODEs and stability issues of the numerical solution

Let's have a look at the first order ODE

$$y' = 3y - 4e^{-x} \tag{1.12}$$

It has the following analytical solution

$$y = Ce^{3x} + e^{-x} \tag{1.13}$$

where C is a constant.

If the initial condition is $y(0) = 1$, then the solution is

$$y(x) = e^{-x}$$

The `ode_unstable_example.m` script (shown below) calculates and plots the numerical and the analytical solutions

Listing 1.2: `ode_unstable_example.m` (available at `http://physics.wm.edu/programming_with_MATLAB_book/ch_ode/code/ode_unstable_example.m`)

```
%% we are solving y'=3*y-4*exp(-x) with y(0)=1
y0=[1]; %y(0)=1
xspan=[0,2];

fvec=@(x,y) 3*y(1)-4*exp(-x);
% the fvec is scalar, there is no need to transpose it to make a column
   vector

Npoints=100;
[x,y] = odeeuler(fvec, xspan, y0, Npoints);

% general analytical solution is
% y(x)= C*epx(3*x)+exp(-x), where C is some constant
% from y(0)=1  follows C=0
yanalytical=exp(-x);


plot(x, y(:,1), '-', x, yanalytical, 'r.-');
set(gca,'fontsize',24);
legend('numerical','analytical');
xlabel('x');
ylabel('y');
title('y vs. x');
```

As we can see in fig. 1.1, the numerical solution diverges from the analytical one. Initially, we might think that this is due to a large step, h, or the use of the inferior Euler method. However, even if we decrease h (by increasing `Npoints`) or change the ODE solving algorithm, the growing discrepancy from the analytical solution will show up.

This discrepancy is a result of accumulated round-off errors (see section 1.5). From a computer's point of view, due to accumulated errors at some point, the numerically calculated $y(x)$ deviates from the analytical solution. This is equivalent to saying that we follow the track where initial condition is $y(0) = 1 + \delta$. The δ is small, but it forces $C \neq 0$, thus, the numerical solution picks up the diverging $\exp(3x)$ term from eq. (1.13). We might think that the decrease of h should help, at least, this clearly push the deviation point to the right. This idea forces us to pick smaller and smaller h (thus, increasing the calculation time) for otherwise seemingly smooth evolution of y and its derivative. These kinds of equations are called *stiff*. Note that due to the round-off errors, we cannot decrease h indefinitely. The

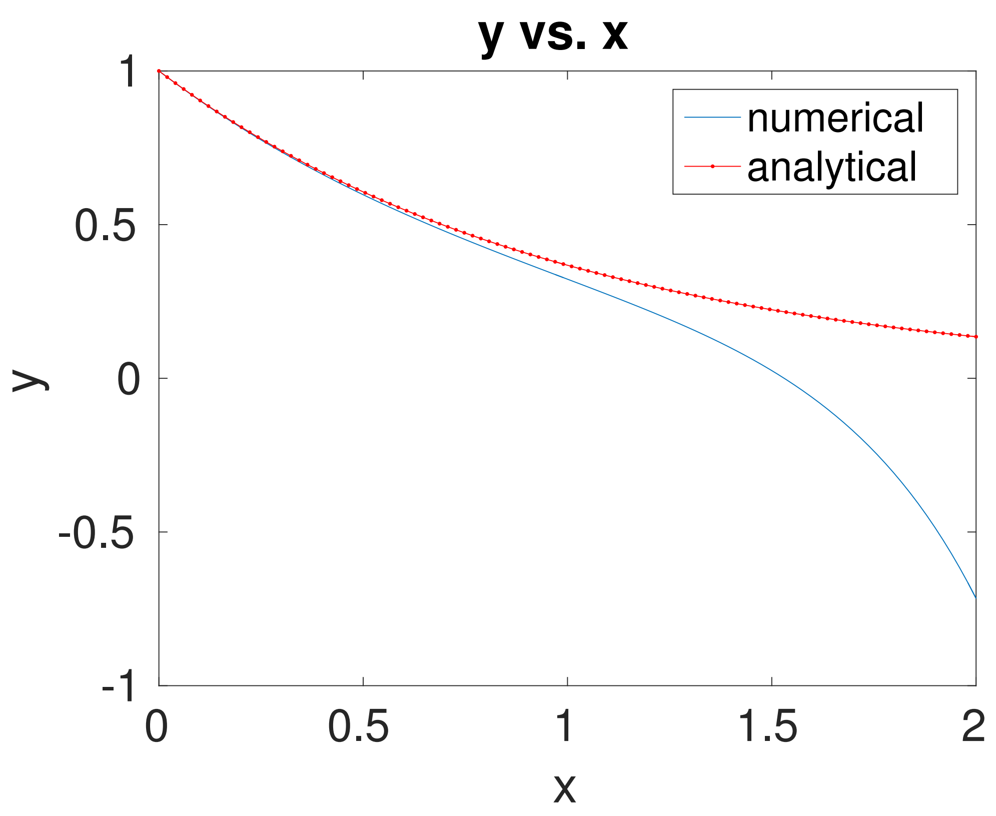

Figure 1.1: Comparison of numerical and analytical solutions of eq. (1.12).

implicit algorithm (which are only briefly mentioned in section 1.3.4) usually are more stable in such settings.

Words of wisdom

Do not trust the numerical solutions (regardless of the method) without proper consideration.

1.5 MATLAB's built-in ODEs solvers

Have a look in the help files for ODEs. In particular, pay attention to

- `ode45` uses adaptive explicit 4th order Runge-Kutta method (good default method)
- `ode23` uses adaptive explicit 2nd order Runge-Kutta method
- `ode113` suitable for "stiff" problems

“Adaptive” means you do not need to choose the step size h. The algorithm does it by itself. However, remember the rule about not trusting a computer’s choice.

Run the built-in `odeexamples` command to see some of the demos for ODEs solvers.

1.6 ODEs examples

In this section, we will cover several examples of physical systems involving ODEs. With any ODE, the main challenge is to transform a compact human notation to the canonical ODE form (see eq. (1.5)) for which we have multiple numerical methods to produce the solution.

1.6.1 Free fall example

Let’s consider a body which falls in Earth’s gravitational field only in the vertical direction (‘y’). For simplicity, we assume that there is no air resistance and the only force acting on the body is due to the gravitational pull of Earth. We also assume that everything happens at sea level, so the gravitational force is height independent. In this case, we can rewrite Newton’s second law as

$$y'' = F_g/m = -g \tag{1.14}$$

where y is the vertical coordinate of the body, F_g is the force due to gravity, m is the mass of the body, and $g = 9.8$ m/s^2 is the constant (under our assumptions) acceleration due to gravity. The gravitational force is directed opposite to the y coordinate axis which points up, thus we put the negative sign in front of g. The y'' is the second derivative with respect to time (the independent variable in this problem).

The above equation is a second order ODE, which we need to convert to a system of two first order ODEs. We note that the velocity component (v) along the ‘y’ axis is equal to the first derivative of the ‘y’ position. The derivative of the velocity v is the acceleration y''. So we can rewrite the above second order ODEs as

$$\begin{pmatrix} y' \\ v' \end{pmatrix} = \begin{pmatrix} v \\ -g \end{pmatrix} \tag{1.15}$$

Finally, we transform the above system to the canonical form of eq. (1.5) by the following relabeling $t \to x$, $y \to y_1$, and $v \to y_2$:

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} y_2 \\ -g \end{pmatrix} \tag{1.16}$$

To use an ODE numerical solver, we need to program the ODE force term calculation function, which is in charge of the right hand side of the above system of equations. This is done in the listing shown below

Listing 1.3: free_fall_forces.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_ode/code/free_fall_forces.m)

```
function fvec=free_fall_forces(x,y)
        % free fall forces example
        % notice that physical meaning of the  independent variable 'x' is
           time
        % we are solving y''(x)=-g, so the transformation to the canonical
           form is
        % y1=y; y2=y'
        % f=(y2,-g);

        g=9.8; % magnitude of the acceleration due to the free fall in m/s^2

        fvec(1)=y(2);
        fvec(2)=-g;
        % if we want to be compatible with Matlab solvers, fvec should be a
           column
        fvec=fvec.'; % Note the dot in .' This avoids complex conjugate
           transpose
end
```

Now, we are ready to numerically solve the ODEs. We do it with the algorithm implemented in `odeeuler.m` shown in listing 1.1. MATLAB's built-ins are also perfectly suitable for this task, but we need to omit the number of points in this case, since the step size h is chosen by the algorithm. See how it is done in the code below

Listing 1.4: ode_free_fall_example.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_ode/code/ode_free_fall_example.m)

```
%% we are solving y''=-g,  i.e  free fall motion

% Initial conditions
y0=[500,15];  % we start from the hight of 500 m  and our initial velocity
    is 15 m/s

% independent variable 'x' has the meaning of time in our case
timespan=[0,13]; % free fall for duration of  13 seconds



Npoints=20;

%% Solve the ODE
[time,y] = odeeuler(@free_fall_forces, timespan, y0, Npoints);
% We can use MATLAB's built-ins, for example ode45.
% In this case, we should  omit Npoints. See the line below.
%[time,y] = ode45(@free_fall_forces, timespan, y0);


%% Calculating the analytical solution
g=9.8;
yanalytical=y0(1) + y0(2)*time - g/2*time.^2;
vanalytical=y0(2) - g*time;

%% Plot the results
subplot(2,1,1);
plot(time, y(:,1), '-', time, yanalytical, 'r-');
set(gca,'fontsize',20);
legend('numerical','analytical');
xlabel('Time, S');
ylabel('y-position, m');
title('Position vs. time');
grid on;

subplot(2,1,2);
plot(time, y(:,2), '-', time, vanalytical, 'r-');
set(gca,'fontsize',20);
legend('numerical','analytical');
xlabel('Time, S');
ylabel('y-velocity, m/s');
title('Velocity vs. time');
grid on;
```

For such a simple problem, there is an exact analytical solution

$$\begin{cases} y(t) = y_0 + v_0 t - gt^2/2 \\ v(t) = v_0 - gt \end{cases} \tag{1.17}$$

The listed above code calculates and plots both the numerical and the analytical solutions to compare them against each other. The results are shown in fig. 1.2. As we can see, both solutions are almost on top of each other, i.e. they are almost the same. Try to increase the number of points (`Npoints`), i.e. decrease the step size h, to see how the numerical solution converges to the true analytical solution.

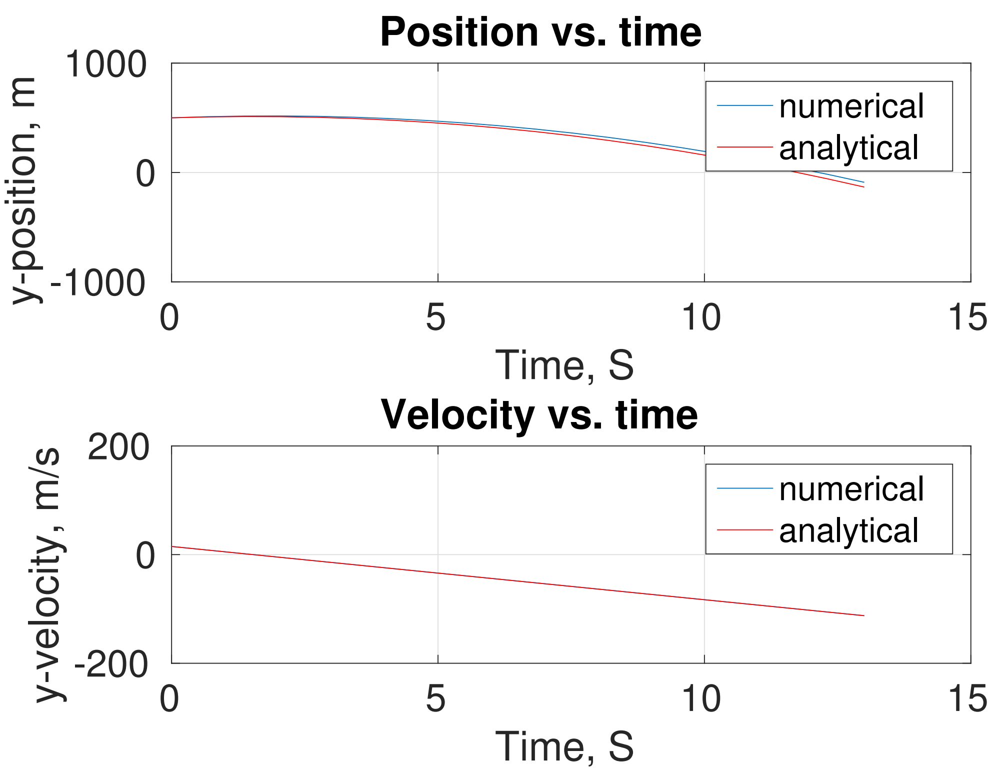

Figure 1.2: The free fall problem analytical and numerical solutions.

1.6.2 Motion with the air drag

The above example is very simple. Let's solve a much more elaborate problem: the motion of a projectile influenced by air drag. We will consider two dimensional motion in 'x-y' plane near the Earth surface, so acceleration due to gravity (g) is constant. This time, we have to take in account the air drag force ($\vec{F}_d$) which is directed opposite to the body's velocity and proportional to the velocity ($\vec{v}$) squared (note the $v\vec{v}$ term below)

$$\vec{F}_d = -\frac{1}{2}\rho C_d A v\vec{v} \tag{1.18}$$

Here C_d is the drag coefficient which depends on the projectile shape, A is the cross-sectional area of the projectile, and ρ is the density of the air. For simplicity, we will assume that the air density is constant within the range of reachable positions.

Newton's second equation in this case can be written as

$$m\vec{r}\,'' = \vec{F}_g + \vec{F}_d \tag{1.19}$$

where $\vec{r}$ is the radius vector tracking the position of the projectile, $\vec{F}_g = m\vec{g}$ is the force of the gravitational pull on the projectile with the mass m. The above equation is second order ODE. We transform it to the system of the first order ODEs similarly to the previous example:

$$\begin{pmatrix} \vec{r}\,' \\ \vec{v}\,' \end{pmatrix} = \begin{pmatrix} \vec{v} \\ \vec{F}_g/m + \vec{F}_d/m \end{pmatrix} \tag{1.20}$$

We should pay attention to the vector form of the above equations, which reminds us that each term has 'x' and 'y' components. We spell it out in the following equation

$$\begin{pmatrix} x\,' \\ v_x\,' \\ y\,' \\ v_y\,' \end{pmatrix} = \begin{pmatrix} v_x \\ F_{g_x}/m + F_{d_x}/m \\ v_y \\ F_{g_y}/m + F_{d_y}/m \end{pmatrix} \tag{1.21}$$

We can simplify the above by noticing that $F_{g_x} = 0$ since the gravity is directed vertically. Also, we note that $F_{d_x} = -F_d v_x/v$ and $F_{d_y} = -F_d v_y/v$, where the magnitude of the air drag force is $F_d = C_d A v^2/2$. The simplified equation looks like

$$\begin{pmatrix} x\,' \\ v_x\,' \\ y\,' \\ v_y\,' \end{pmatrix} = \begin{pmatrix} v_x \\ -F_d v_x/(vm) \\ v_y \\ -g - F_d v_y/(vm) \end{pmatrix} \tag{1.22}$$

Finally, we bring it to the canonical form with the following relabeling $x \to y_1$, $v_x \to y_2$, $y \to y_3$, $v_y \to y_4$, and $t \to x$.

The key to success is to adhere to the above transformation and alternate between it and the human (physical) notation during problem implementation. See how it is done in the code below

Listing 1.5: ode_projectile_with_air_drag_model.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_ode/code/ode_projectile_with_air_drag_model.m)

```
function [t, x, y, vx, vy] = ode_projectile_with_air_drag_model()
    %% Solves the equation of motions for a projectile with air drag
       included
    % r''= F = Fg+ Fd
    % where
    % r is the radius vector, Fg is the gravity pull force, and Fd is the
       air drag force.
    % The above equation can be decomposed to x and y projections
    % x'' = Fd_x
    % y'' = -g + Fd_y
    % Fd =  1/2 * rho * v^2 * Cd * A is the drag force magnitude
    % where v is speed.
    % The drag force directed against the velocity vector
    % Fd_x= − Fd ∗ v_x/v  ; % vx/v takes care of the proper sign of the drag
        projection
    % Fd_y= − Fd ∗ v_y/v  ; % vy/v takes care of the proper sign of the drag
        projection
    % where vx and vy are the velocity projections

    % at the first look it does not look like ODE but since x and y depends
       only on t
    % it is actually a system of ODEs

    % transform system to the canonical form
    % x  −> y1
    % vx −> y2
    % y  −> y3
    % vy −> y4
    % t  −> x
    %
    % f1 −> y2
    % f2 −> Fd_x
    % f3 −> y4
    % f4 −> −g + Fd_y

    % some constants
    rho=1.2;  % the density of air kg/m^3
    Cd=.5;    % an arbitrary choice of the drag coefficient
    m=0.01;   % the mass of the projectile  in kg
    g=9.8;    % the acceleration due to gravity
    A=.25e−4; % the area of the projectile in m^2, a typical bullet is 5mm x
        5mm

    function fvec = projectile_forces(x,y)
        % it is crucial to move from the ODE notation to the human notation
        vx=y(2);
        vy=y(4);
        v=sqrt(vx^2+vy^2); % the speed value

        Fd=1/2 ∗ rho ∗ v^2 ∗ Cd ∗ A;

        fvec(1) = y(2);
        fvec(2) = −Fd∗vx/v/m;
        fvec(3) = y(4);
        fvec(4) = -g -Fd*vy/v/m;

        % To make matlab happy we need to return a column vector.
        % So, we transpose (note the dot in .')
        fvec=fvec.';
    end

    %% Problem parameters setup:
    %  We will set initial conditions similar to a bullet fired from
    %  a rifle at 45 degree to the horizon.
    tspan=[0, 80];       % time interval of interest
    theta=pi/4;          % the shooting angle above the horizon
    v0 = 800;            % the initial projectile speed in m/s
    y0(1)=0;             % the initial x position
    y0(2)=v0*cos(theta); % the initial vx velocity projection
    y0(3)=0;             % the initial y position
    y0(4)=v0*sin(theta); % the initial vy velocity projection

    % We are using matlab solver
    [t,ysol] = ode45(@projectile_forces, tspan, y0);

    % Assigning the human readable variable names
    x =  ysol(:,1);
    vx = ysol(:,2);
    y =  ysol(:,3);
    vy = ysol(:,4);
    v=sqrt(vx.^2+vy.^2); % speed

    % The analytical drag-free motion solution.
    % We should not be surprised by the projectile deviation from this
       trajectory
    x_analytical = y0(1) + y0(2)*t;
    y_analytical = y0(3) + y0(4)*t -g/2*t.^2;
    v_analytical= sqrt(y0(2).^2 + (y0(4) - g*t).^2); % speed

    ax(1)=subplot(2,1,1);
    plot(x,y, 'r-', x_analytical, y_analytical, 'b-');
    set(gca,'fontsize',14);
    xlabel('Position x component, m');
    ylabel('Position y component, m');
    title ('Trajectory');
    legend('with drag', 'no drag', 'Location','SouthEast');
```

```
    ax(2)=subplot(2,1,2);
    plot(x,v, 'r-', x_analytical, v_analytical, 'b-');
    set(gca,'fontsize',14);
    xlabel('Position x component, m');
    ylabel('Speed');
    title ('Speed vs. the x position component');
    legend('with drag', 'no drag', 'Location','SouthEast');

    linkaxes(ax,'x'); % very handy for related subplots
end
```

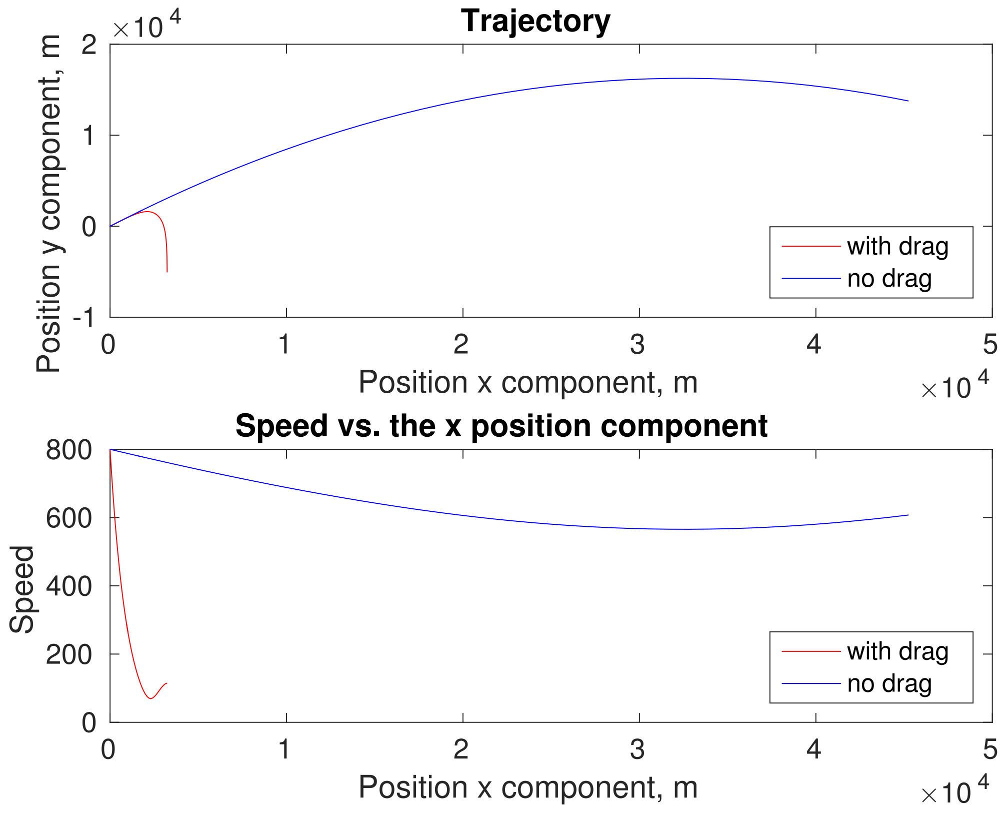

Figure 1.3: The projectile trajectory and its speed vs. 'x' position calculated without and with the air drag force taken in account.

The code shows two trajectories of the same projectile: one for the case when we take in account the air drag and the other without it (see fig. 1.3). In the latter case, we provide the analytical solution as well as the numerical solution, in similar manner to the previous example. However, once we account for drag, we cannot easily determine the analytical solution and must solve numerically. Finally, we justified the use of numerical methods[1]. The downside to solving the problem numerically is that we cannot easily check the result of numerical calculations in the same way that we could with the analytical solution. However, we still can do some checks as we know that the drag slows down the projectile, so it should travel less distance. Indeed, as we can see in fig. 1.3, the bullet travels only about 3 km when influenced by the drag, while drag-free bullets can fly much further than 40 km[2]. Have a look at the trajectory of the bullet with the drag. Closer to the end, it drops straight down. This is because the drag reduces 'x' component of the velocity to zero, but gravity is still affecting the bullet, so it picks up the non zero vertical component value. If we look at the speed plot, it is reasonable to expect that speed decreases as the bullet travels. Why then does the speed increase at the end of the trajectory? We can see that this happens as the bullet reaches the highest point, so the potential energy converts to kinetic after this point, and the speed grows. The same effect is observed in the plot for the drag-free motion. The above sanity checks allow us to conclude that the numerical calculation seems to be reasonable.

1.7 Self-Study

Problem 1: Here is a model for a more realistic pendulum. Numerically solve (using the built-in `ode45` solver) the following physical problem of a pendulum's motions

$$\theta''(t) = -\frac{g}{L}\sin(\theta)$$

where g is acceleration due to gravity (g=9.8 m/s^2), $L = 1$ is the length of the pendulum, and θ is the angular deviation of the pendulum from the vertical.

Assuming that the initial angular velocity (β) is zero, i.e. $\beta(0) = \theta'(0) = 0$, solve this problem (i.e. plot $\theta(t)$ and $\beta(t)$) for two values of the initial deflection $\theta(0) = \pi/10$ and $\theta(0) = \pi/3$. Both solutions must be presented on the same plot. Make the final time large enough to include at least 10 periods. Show that the period of the pendulum depends on the initial deflection. Does it takes longer to make one swing with a larger or smaller initial deflection?

Problem 2: Implement the fourth-order Runge-Kutta method (RK4) according to the recipe outlined in section 1.3.3. It should be input compatible to the home-made Euler's implementation listing 1.1. Compare the solution of the above problem with your own RK4 implementation to the built-in `ode45` solver.

[1]You might have noticed that the author provided an analytical solution, whenever it is possible, to be used as the test case for the numerical solution.

[2]The air not only let us breathe, but also allows to live much closer to the fire ranges.

Bibliography

[1] Holistic numerical methods. http://mathforcollege.com/nm/. Accessed: 2016-11-09.

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.