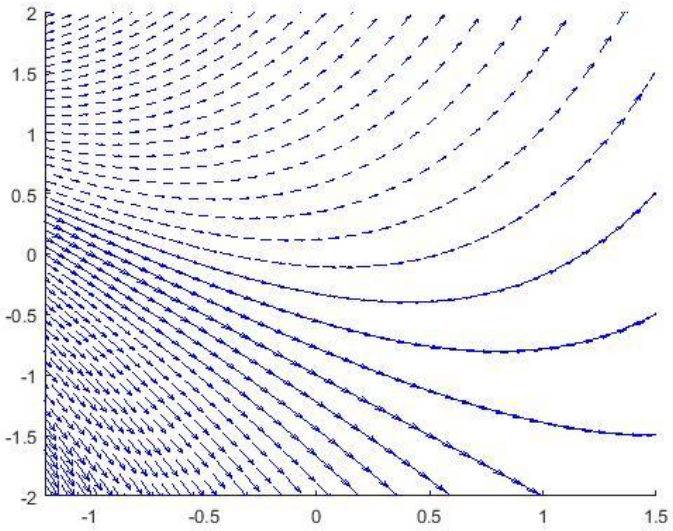
Ordinary Differential Equations - Initial value problems

Differential equations

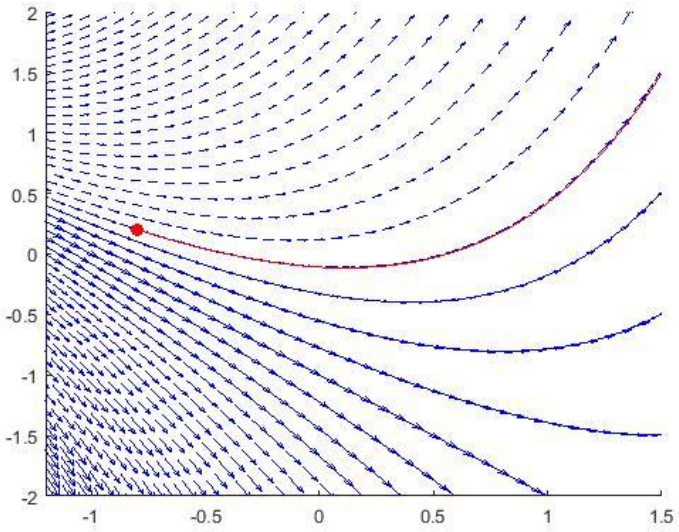
Differential equations are equations containing a function $y = f(x)$ and its derivatives. The ordinary differential equations (ODEs) contain a univariate function and its derivatives. The order of the ODE equals the highest order of derivatives in the equation. As opposed to an ODE, there are partial differential equations (PDEs) as well that contain a multivariate function's partial derivatives.

The solution for a differential equation is not a number but a function. In order for the solution to be unique, some constraints have to be met by the solution function. One constraint can be that the function and its derivative take some value at the beginning of the interval of interest, hence the initial value problem. Another constraint would be that at least one of the values the function or its derivative takes is given at the end of the interval.

Generally, we have a solution set for a given differential equation. For example, take the following ODE: $\frac{dy}{dx} = y + x$. We are looking for a function that solves this equation. All the functions in the figure below solve the equation, so they make up the solution set for the equation. The figure below is also called the trajectory or direction field of the ODE.



If we specify the initial conditions for the function and derivative's value at the beginning of our search interval, e.g. the function has to go through the point $x = -0.8, y = 0.2$, that we have reduced the solution to a specific solution, that is the function given by the red line below:



There are linear ODE's and nonlinear ones. In a linear ODE, the linear expression of the unknown function and its derivatives are given. For example:

$$e^x \cdot \frac{dy}{dx} + a \cdot x^2 + x^4 \cdot y = 0$$ -- This is a linear ODE.

$$\frac{dy}{dx} + a \cdot x \cdot y + b \cdot y^2 = 0$$ -- This is a nonlinear ODE.

In the case of many differential equations and systems (especially if they are nonlinear), the solution function can only be found numerically. The numeric solution give us the data points of the function computed using numeric integration. The aim of these algorithms is to accurately approximate the data points of the solution function using as few steps and function evaluations as possible.

Ordinary Differential Equations - The initial value problem

The general form of a first order ODE with the independent variable t:

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

In the univariate case, we have one independent variable (t) and one dependent variable (y). The derivative of the function y is given by $f(t, y)$. In the case of the initial value problem, we know that the solution goes through the (t_0, y_0) point:

$$y(t_0) = y_0$$

Euler's method

We wish to find the values of the solution function on a given interval using a predefined step size (h). We suppose that the slope of the function (m) on the interval defined by the h step size is constant. If we know the functions value at the beginning of the interval and the value of its derivative (which denotes the slope of the

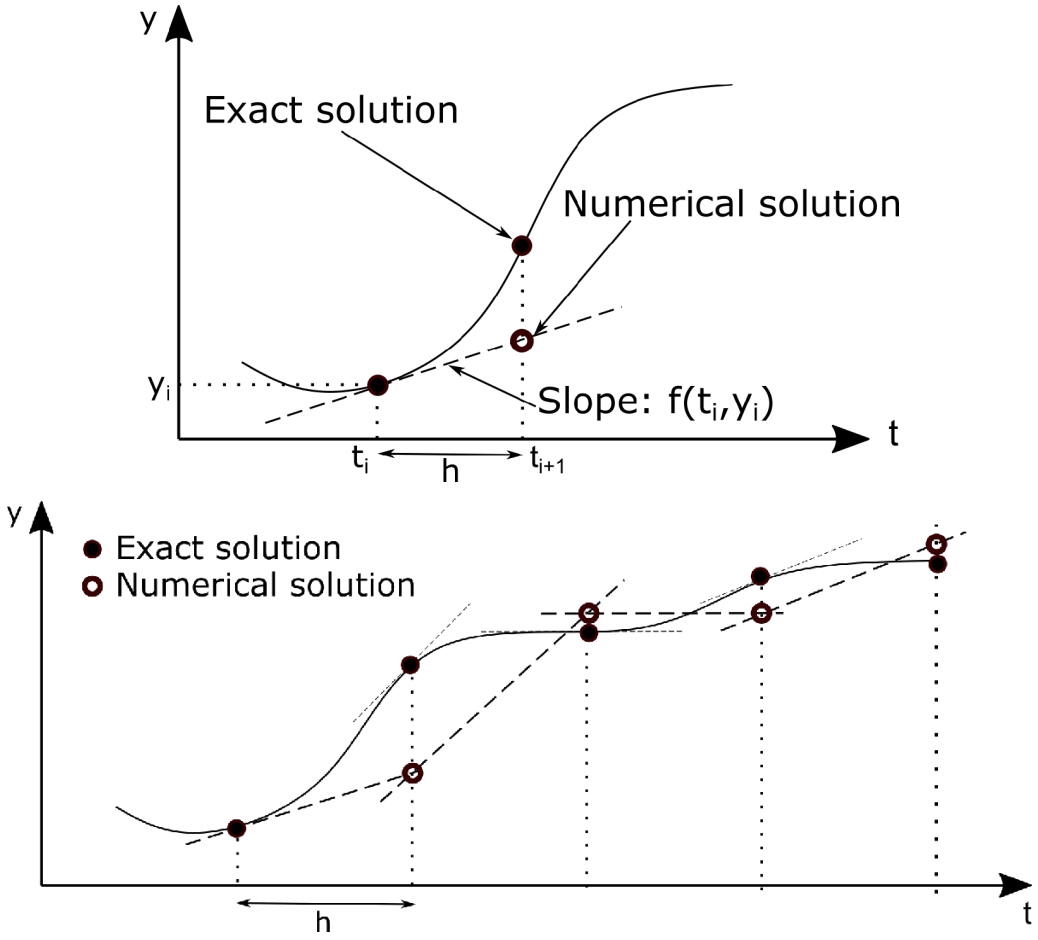
function) than we can approximate the value of the function at the end of the h step size using a line with slope m.

In case of Euler's method, we suppose that the $m = f(t, y)$ value is constant for each subinterval ($h = t_{i+1} - t_i$) where we integrate and its value is given at the beginning of the interval.

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt \approx y_i + f(t_i, y_i) \cdot h = y_i + m_i \cdot h$$

$$t_{i+1} = t_i + h$$

where m is the slope at the beginning of each subinterval and its value is constant on the subinterval. The local error of the method is $\mathrm{O}(h)$, its global error is $\mathrm{O}(h)$ meaning that it is a first-order method.



Let's look at an example custom code that implements Euler's method in MATLAB:

```
function [t,y] = euler (f, y0, a, b, h)
n = round((b - a)/h);
t(1) = a;
y(1) = y0;
for i = 1 : n - 1
    y(i + 1) = y(i) + h*f(t(i), y(i));
    t(i + 1) = t(i) + h;
end
```

The inputs to the function above:

1. f - the handle of the function that defines the right-hand side of the ODE.
2. y0 - the initial value of the solution function at the beginning of the search interval.
3. a - the beginning of the search interval.
4. b - the end of the search interval.
5. h - the step size for the computation.

Solving an ODE using Euler's method

Let's look at the following example. A spherical water tank with radius $R = 10\,\mathrm{m}$ is drained using a hole on its bottom at $h = 0$ height. The hole has a radius $r = 5\,\mathrm{cm}$. The contains approximately 4000 m^3 of water. At the beginning of the draining ($t = 0$), the height of the water inside the tank is 17.44 m. The coefficient of contraption of the hole is $\mu = 0.85$.

The water level inside the tank, measured from the bottom is given by the following first-order ODE:

$$f(t,h) = \frac{dh}{dt} = \frac{-\mu\, r^2 \sqrt{2gh}}{2hR - h^2}$$

where $R = 10\,\mathrm{m}, r = 0.05\,\mathrm{m}, g = 9.81\ m/s^2, \mu = 0.85$. We have two questions:

1. How high will be the water level in the tank after 12 hours?
2. How much time does it take to completely drain the tank?

Solving a differential equation always starts with rearranging the equation into a form where we have the derivative on side and all the other variables and constant on the other. In our case, the ODE is already given in this form, so we can skip this step. The function that is equal to the derivative will be our f function.

We have to define the function in MATLAB and solve it using Euler's method and a step size of 60 seconds. It is possible that the derivative function f itself does not contain the independent variable t. Nevertheless, we still have to define the function in MATLAB using the independent variable t as well.

Definition of the constants:

```
R = 10;
r = 0.05;
g = 9.81;
mu = 0.85;
```

The derivative function:

```
f = @(t, h) -mu*r^2*sqrt(2*g*h)/(2*h*R - h.^2) % Note that the expression is given as a functio
```

```
f = function_handle with value:
    @(t,h)-mu*r^2*sqrt(2*g*h)/(2*h*R-h.^2)
```

Defining the initial value, the interval of the solution and the step size:

```
h0 = 17.44; % initial water level
```

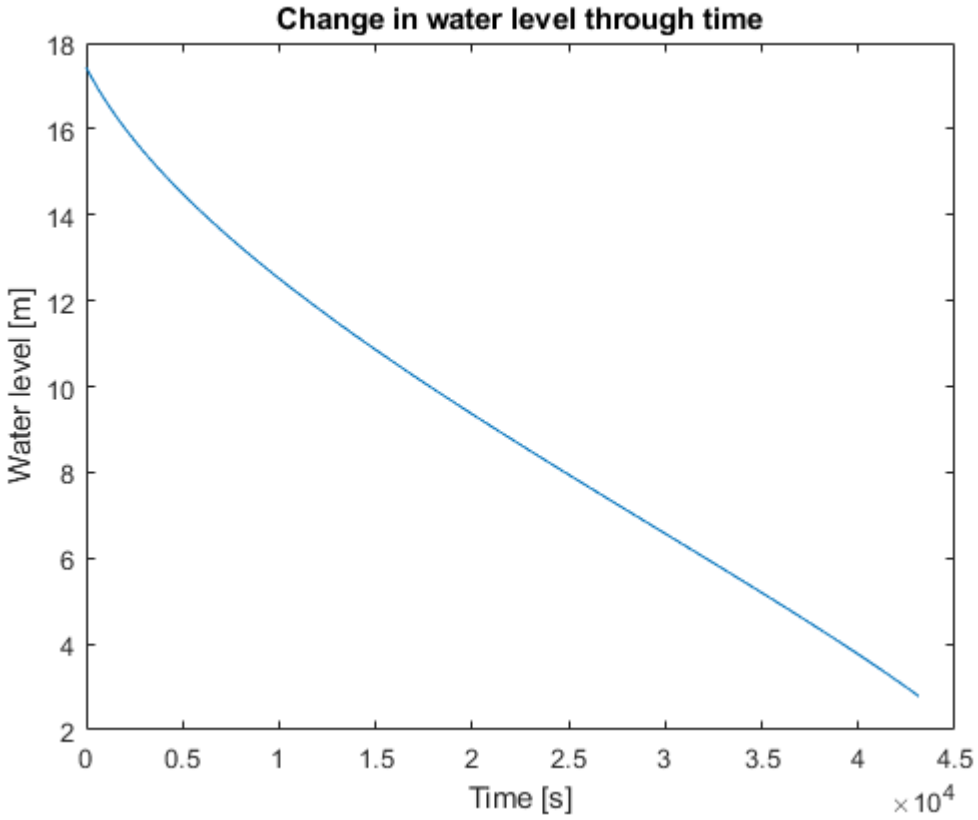
```
t0 = 0; % 0 hours is the beginning of the interval
t1 = 12*3600; % 12 hours is the end of the interval (given in seconds)
s = 60; % step size in seconds
```

The solution using Euler's method:

```
[T, H] = euler(f, h0, t0, t1, s);
```

Plotting the solution:

```
figure(1);
plot(T, H);
xlabel('Time [s]');
ylabel('Water level [m]');
title('Change in water level through time');
```



As the computation was carried out to 12 hours, the last element inside the vector **H** will give us the answer to the first question:

```
H(end)
```

```
ans = 2.7810
```

The water level after 12 hours is ~2.78 m. The method used is a first-order method ($\mathrm{O}(h)$). Before we try to answer the second the question, let's see if we can do better than just a first-order approximation.

Improvements of Euler's method (Heun's method, midpoint method, Runge-Kutta method)

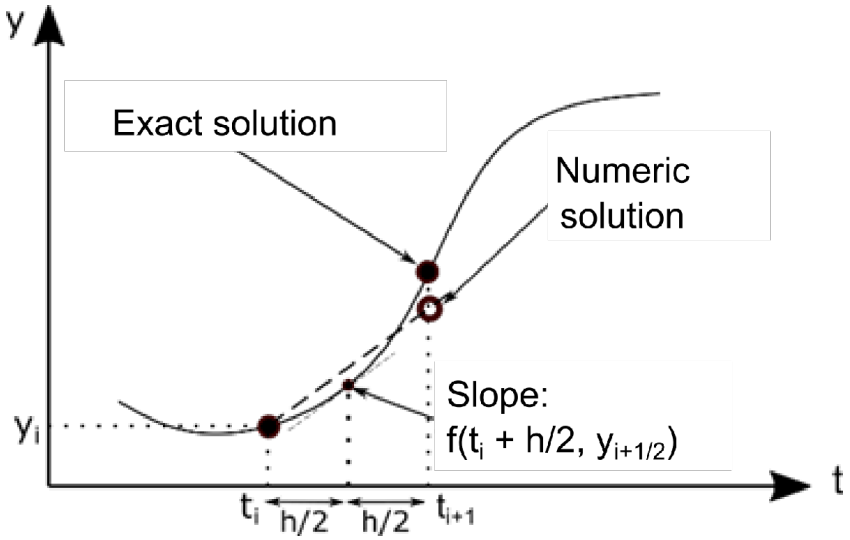
A similar approach is used to estimate the function values in all of the methods mentioned in the section title, the difference lies in the way of approximating the slope of the function.

In Euler's method, the slope is calculated at the beginning of the step.

In Heun's method, the slope is calculated at the beginning (m_i) and at the end of the step interval (m_{i+1}) as well. The final value of the slope that is used is the average of the two computed values. In order to be able to calculate the slope at the end of the step size, we have to know the function value there as $m_{i+1} = f(t_{i+1}, y_{i+1})$. To be able to achieve this, a so-called predictor step is carried out first using Euler's method and the result of the predictor step is used to compute the final slope value:

1. Predictor step (Euler's method): $y_{i+1}^{(0)} = y_i + m_i \cdot h = y_i + \cdot f(t_i, y_i) \cdot h$
2. Corrector step: $t_{i+1} = t_i + h, \quad m_{i+1} = f\left(t_{i+1}, y_{i+1}^{(0)}\right)$
3. Final slope value: $y_{i+1} = y_i + \frac{(m_i + m_{i+1})}{2} \cdot h = y_i + \frac{f(t_i, y_i) + f\left(t_{i+1}, y_{i+1}^{(0)}\right)}{2} \cdot h$

The local error of the method is $\mathrm{O}(h^3)$, its global error is $\mathrm{O}(h^2)$, which means that the method is a second-order method and is a magnitude more accurate than Euler's method.

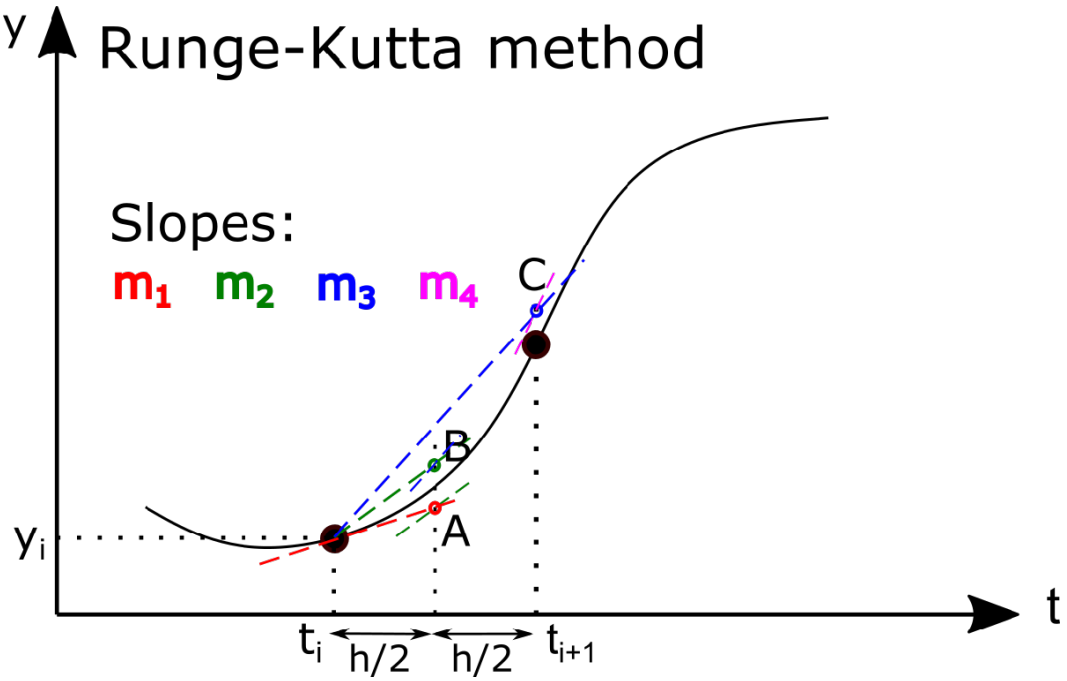


In case of the midpoint method, the derivative is computed in the midpoint of the step and its value will be considered constant for the given step. To achieve this, the preliminary value of the function is computed for the midpoint using Euler's method and then the value of the slope is calculated:

1. Function value at the midpoint (Euler's method): $y_{i+\frac{1}{2}} = y_i + m_i \cdot \frac{h}{2} = y_i + f(t_i, y) \cdot \frac{h}{2}$
2. Slope in the midpoint: $t_{i+\frac{1}{2}} = t_i + \frac{h}{2}, \quad m_{i+\frac{1}{2}} = f\left(t_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}\right)$

3. Function value at the end of the step: $y_{i+1} = y_i + m_{i+\frac{1}{2}} \cdot h$

Euler's method can be further improved if we use even more points to compute the slope and use the weighted average of these values. The most common of these methods is the fourth-order Runge-Kutta method which has a global truncation error of $O(h^4)$. In MATLAB, we can use the built-in **ode45** command.



The function value at t_{i+1} is computed from the formula:

$$y_{i+1} = y_i + \frac{1}{6} \cdot (m_1 + 2m_2 + 2m_3 + m_4) \cdot h$$

where:

- $m_1 = f(t_i, y_i)$ - the slope at the beginning of the step size, point A is computed using this slope,
- $m_2 = f\left(t_i + \frac{h}{2}, y_i + m_1 \cdot \frac{h}{2}\right)$ - the slope in point A, point B is calculated using this slope,
- $m_3 = f\left(t_i + \frac{h}{2}, y_i + m_2 \cdot \frac{h}{2}\right)$ - the slope in point B, point C is computed using this slope,
- $m_4 = f(t_i + h, y_i + m_3 \cdot h)$ - the slope in point C.

The steps of the computation:

1. $m_1 = f(t_i, y_i)$
2. $y_A = y_i + m_1 \cdot \frac{h}{2} \rightarrow m_2 = f\left(t_i + \frac{h}{2}, y_A\right)$
3. $y_B = y_i + m_2 \cdot \frac{h}{2} \rightarrow m_3 = f\left(t_i + \frac{h}{2}, y_B\right)$

4. $y_C = y_i + m_3 \cdot h \rightarrow m_4 = f(t_i + h, y_C)$
5. $y_{i+1} = y_i + \frac{1}{6} \cdot (m_1 + 2m_2 + 2m_3 + m_4) \cdot h$

Solution of a first-order ODE using the Runge-Kutta method

In MATLAB, we can use the fourth-order Runge-Kutta method in the following form:

```
[T, Y] = ode45(odefun, tspan, y0)
```

where `T` and `Y` are the output of the independent variable and the solution, `odefun` is the function handle for the derivative function, `tspan` is the interval of the independent variable and `y0` is the initial value of the function at the start of the interval. The value of `tspan` can be given in two ways:

- as a vector of the start and the end of the interval, in this case, the `ode45` algorithm decides the step size automatically,
- as a vector of the values of the independent variable, in this case, we decide the step size and define it explicitely.

Solution using `ode45`:

```
% Letting the algorithm deicde the step size
[T1, H1] = ode45(f, [0, 12*3600], h0);
H1(end) % water level after 12 hours
```

```
ans = 2.7779
```

```
% Defining the step size explicitely
[T2, H2] = ode45(f, [0:60:12*3600], h0);
H2(end)
```

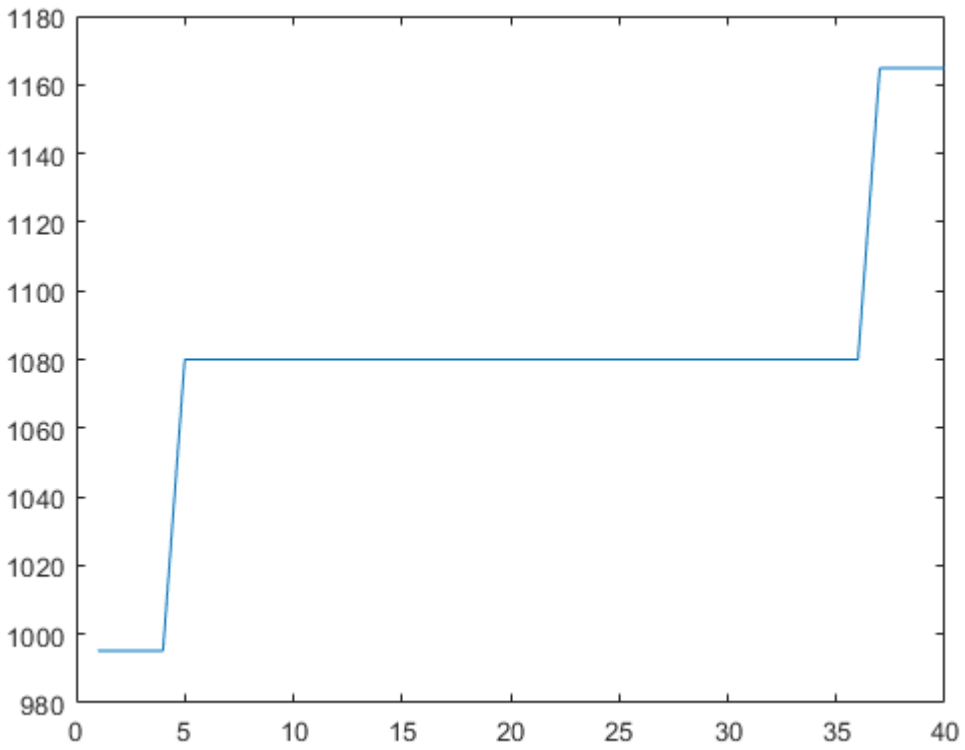
```
ans = 2.7713
```

Plotting the results:

```
figure(1);
hold on;
plot(T1, H1, 'r');
plot(T2, H2, 'g');
```

The difference in the values is very minor, only a couple of millimeters. It is worth noting however, that when the algorithm decided the step size, it wasn't constant throughout the whole interval:

```
figure(2);
plot(diff(T1));
```



The step sized varied between 995 and 1165 seconds. Smaller step sizes can usually make the approximation more accurate, however, it increases the number of function evaluations. The algorithm itself tries to decide when to use a smaller and when to use a bigger step size, depending on the difference between the function values.

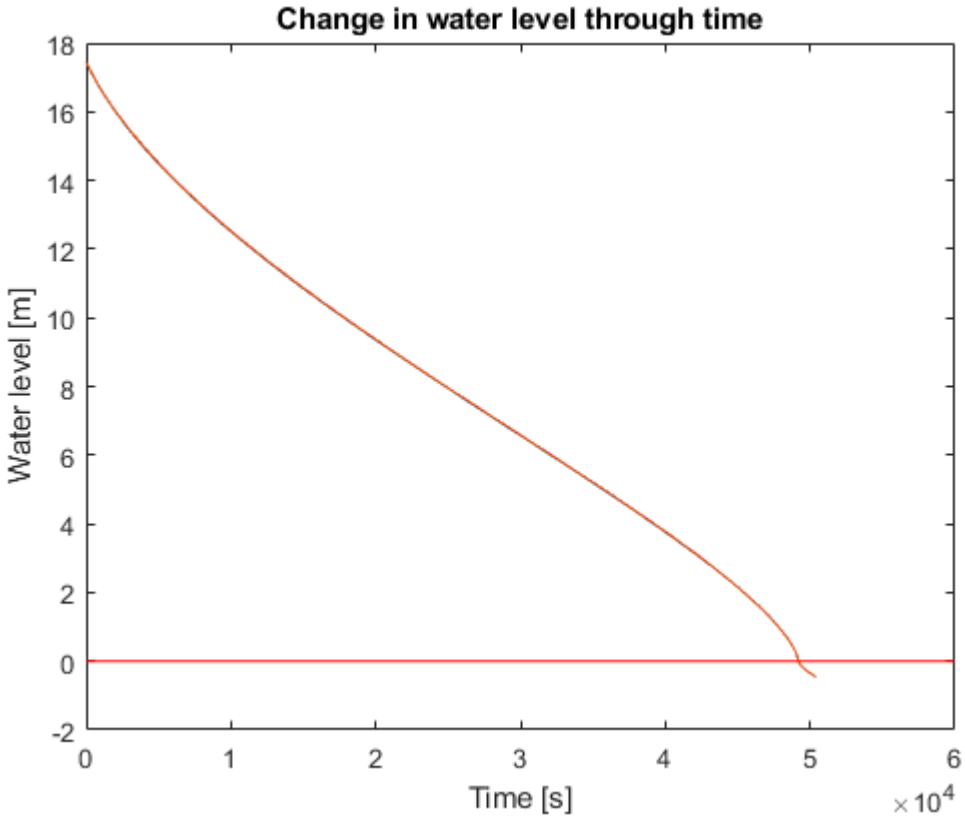
Let's anwer the second question now: how much time is needed to completely drain the tank?

To answer this, we first have to compute the function values using a longer period than 12 hours, as we have seen that after 12 hours, the water level is still higher than 2 meters. Let's use 14 hours (in seconds) for the end of our interval. We have to be careful, because of the square root inside the formula, the results become complex after the $h = 0$ line is passed. We first have to convert the complex into real ones, fit a spline to our solution data and find the intersection of our spline and the horizontal line at $h = 0$:

```
[T3, H3] = ode45(f, 0:60:14*3600, h0);
figure(1);
H3 = real(H3); % ignoring the imaginary parts
plot(T3, H3, 'k');
```

Fitting of a spline to the data:

```
sp = @(t) spline(T3, H3, t);
fplot(sp, [min(T3), max(T3)])
rl = refline(0, 0);
rl.Color = 'r';
```



Findig the intersection of the two lines (solution of $h = 0$ equation), the initial guess is from the figure:

```
h_0 = fzero(sp, 50000)
```

```
h_0 = 4.9192e+04
```

In hours:

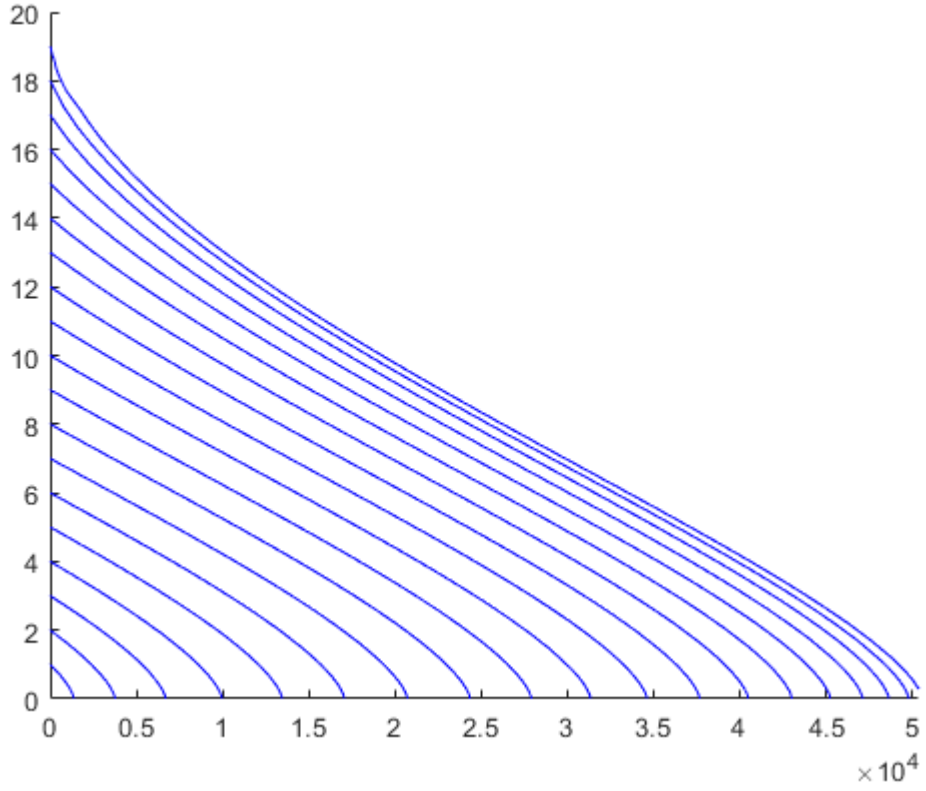
```
h_0 / 3600
```

```
ans = 13.6644
```

What does the draining of the tank looks like if we have different initial water levels? We can visualize this by creating the trajectory or direction field of the ODE by solving it using initial values from 1 m to 20 m for example:

```
figure(3);
hold on;

for i = 20:-1:1 % go from 20 to 19, 18, ... 1
    [T, H] = ode45(f, [0, 14*3600], i);
    plot(T, real(H), 'b');
end
axis([0, 14*3600, 0, 20])
```



Solution of a system of first-order ODEs

In many cases the phenomenon we are investigating is dependent on multiple variables that can alter each other. In such cases we have to solve not a single ODE but a system of ODEs. Let the dependent variables be $y_1, y_2, \cdots, y_n$ and the independent variable be t. In a general form, the system of first-order ODEs can be written as:

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2, \cdots, y_n)$$
$$\frac{dy_2}{dt} = f_2(t, y_1, y_2, \cdots, y_n)$$
$$\vdots$$
$$\frac{dy_n}{dt} = f_n(t, y_1, y_2, \cdots, y_n)$$

The initial values on the interval $[a, b]$:

$$y_1(a) = Y_1, y_2(a) = Y_2, \cdots, y_n(a) = Y_n$$

Some systems of ODEs can be solved using the generalized forms of the explicit methods mentioned before: $t_{i+1} = t_i + h$, in case of Euler's method for example:

$$
\begin{aligned}
y_{1,i+1} &= y_i + f_1(t, y_1, y_2, \cdots, y_n) \cdot h \\
&\vdots \\
y_{n,i+1} &= y_i + f_n(t, y_1, y_2, \cdots, y_n) \cdot h
\end{aligned}
$$

The improved methods can be similarly generalized. Let's see the following simple example. We are looking for the solution in the interval $[0, 1.2]$ using a step size of $h = 0.4$.

$$
\begin{aligned}
f_1 = \frac{dx}{dt} &= x \cdot t - y \qquad & x(0) = 1 \\
f_2 = \frac{dy}{dt} &= y \cdot t + x \qquad & y(0) = 0.5
\end{aligned}
$$

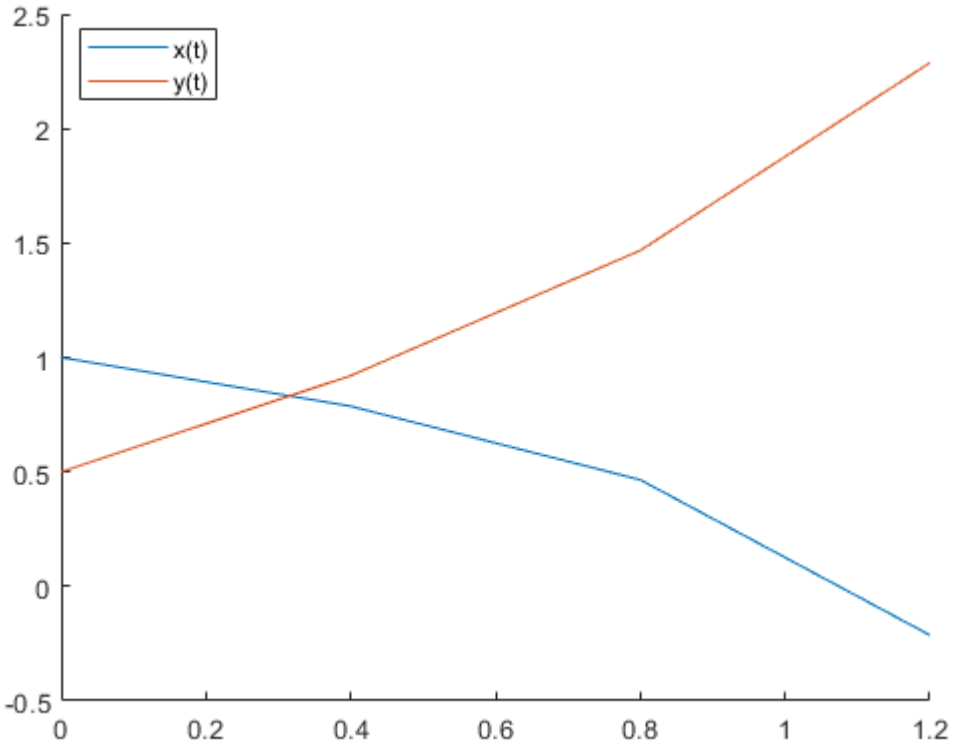
We have two equations, f_1 denotes the derivative of the first variable with respect to t and f_2 denotes the same for the second variable. Let's use the Runge-Kutta method to solve the system. We have to use vector notation when specifying the variables, for example we can use $v = [x, y]$, so that $v(1) = x$ and $v(2) = y$.

If the system is not too complicated, we can define it as an anonymous function in MATLAB:

```
clear all; close all;
odesys = @(t, v) [v(1)*t - v(2); v(2)*t + v(1)];
```

Before we can solve it, we have to define the interval, the step size and the initial values.

```
t = 0:0.4:1.2;
init = [1; 0.5];
[T, Y] = ode45(odesys, t, init);
figure(1);
hold on;
plot(T, Y(:, 1), T, Y(:, 2));
legend('x(t)', 'y(t)', 'Location', 'best');
```



If the system is more complicated, it is better practice to define it in a separate file as a regular function. Let's do this for the above example. We can create a function file called **`sysdiff.m`** and define the system:

```
function dvdt = sysdiff(t, v)
  f1 = v(1)*t - v(2);
  f2 = v(2)*t + v(1);
  dvdt = [f1; f2];
end
```

If our system is defined in a separate function file, we have to put an "@" symbol before its name when referring to it in the solver:

```
[T, Y] = ode45(@sysdiff, t, init)
```

```
T = 4×1
         0
    0.4000
    0.8000
    1.2000
Y = 4×2
    1.0000    0.5000
    0.7868    0.9207
    0.4655    1.4676
   -0.2130    2.2870
```

Second-order ODEs

Second-order ODEs with independent variable t and dependent variable y can be given in the following general form:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right)$$

The equation can be solved on the interval $[a, b]$ if we have two know values. If these values are given at the beginning of the interval, than we call it an initial value problem. The two initial values that have to be given are the value of y and $\frac{dy}{dt}$ at the beginning of the interval. Let these initial values be denoted using A and B:

$$y(a) = A \qquad \text{and} \qquad \frac{dy}{dt}(a) = B$$

This type of second-order ODE can be transformed into a system of two first-order ODEs that can be solved similarly to the previous case. The first step of the solution is to rearrange the equation in such a way that the second derivative is expressed as a function of the other variables and constants $\left(f\left(t, y, \frac{dy}{dt}\right)\right)$. Of course, it can be the case that the second derivative is not dependent on all of the variables. As t is the independent variable, it always has to be specified in MATLAB. We can substitute new variables instead of y and $\frac{dy}{dt}$:

$$w_1 = y \qquad \text{and} \qquad w_2 = \frac{dy}{dt}$$

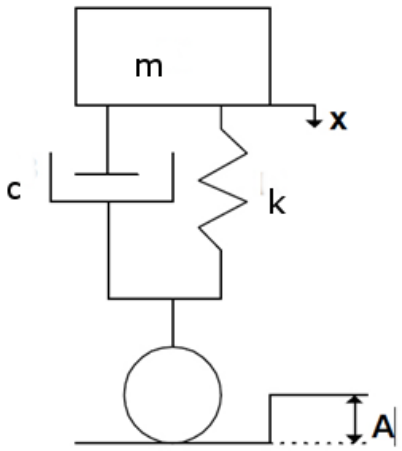
Now, we have to write to equations for the first derivatives of the two new variables and give their initial values:

$$f_1 = \frac{dw_1}{dt} = \frac{dy}{dt} = w_2 \qquad w_1(a) = A$$
$$f_2 = \frac{dw_2}{dt} = \frac{d^2y}{dt^2} = f(t, w_1, w_2) \qquad w_2(a) = B$$

Using these new definitions, the second-order equation is given as a system of two first-order ODEs.

Solution of a second-order ODE in MATLAB

Let's look at the following example. We are simulating the suspension of a vehicle using the following simple model, where the vehicle travels through an obstacle with height A.



In the modell, m is the mass of the vehicle, k is the spring constant (the force in the spring is proportional to the displacement), c is the dampening factor (the dampening force is proportional to the velocity of the mass). The following data is used:

- $m = 1000\,\mathrm{kg}$
- $k = 1000\,\frac{\mathrm{kg}}{s^2}$
- $c = 500\,\frac{\mathrm{kg}}{s}$
- $A = 0.1\,\mathrm{m}$

Summing up the forces in the dampened system, we get the following second-order ODE:

$$m \cdot \ddot{x} + c \cdot \dot{x} + k \cdot (x - A) = 0$$

where x is the vertical position of the vehicle, $\dot{x}$ is the position's first derivative with respect to time (the vertical velocity of the vehicle) and $\ddot{x}$ is the second derivative of the position (the vertical acceleration of the vehicle). Defining the equation in the vehicle's coordinate system:

$$m \cdot \frac{d^2x}{dt^2} + c \cdot \frac{dx}{dt} + k \cdot (x - A) = 0$$

The initial values are the vertical displacement of the vehicle at the begining of our interval and the initial verical velocity at the beginning of the interval, both are zero:

$$x(0) = 0 \qquad \text{and} \qquad \frac{dx}{dt}(0) = 0$$

The first step is to express the second derivative from the equation:

$$\frac{d^2x}{dt^2} = \frac{1}{m} \cdot \left(kA - kx - c \cdot \frac{dx}{dt}\right) = f\left(t, x, \frac{dx}{dt}\right)$$

We have to transform this second-order equation into a system of two first-order ODEs. Before we do this, let's introduce two substitutions:

$$w_1 = x$$
$$w_2 = \frac{dx}{dt}$$

Using these substitutions, we can write two equations and their initial values:

$$f_1 = \frac{dw_1}{dt} = \frac{dx}{dt} = w_2 \qquad w_1(0) = 0$$
$$f_2 = \frac{dw_2}{dt} = \frac{d^2x}{dt^2} = \frac{1}{m}(kA - kw_1 - cw_2) \qquad w_2(0) = 0$$

We can create our system of ODEs in a separate function file **`vehicdiff.m`**. We can use the vector notation $w = [w_1, w_2]$, so that w_1 is the vertical displacement and w_2 is the vertical velocity.

```
function f = vehdiff(t, w)
    % Constants
    m = 1000; k = 1000; A = 0.1; c = 500;
    f1 = w(2);
    f2 = 1/m * (k*A - k*w(1) - c*w(2));
    f = [f1; f2];
end
```

Note that the variable t is given among the input variables, however, it is not usd explicitely in the equations. Let's solve the system using the Runge-Kutta method (**ode45** in MATLAB) specifying an absolute and a relative tolerance of 10^{-4}, on the interval 0-15 seconds.

Optional parameters can be specified for the **ode45** solver similarly to previous solvers, using the **odeset()** function. Some of the most relevant parameters:

- `RelTol` - scalar relative tolerance that is valid for each component of the function y, it measures the error relative to the magnitude of each solution function;
- `AbsTol` - scalar or vector of tolerance values that is valid for all or some of the solution functions, controls the step size of the solver;
- `MaxStep` - the largest acceptable step size,
- `InitialStep` - recommended starting step size.

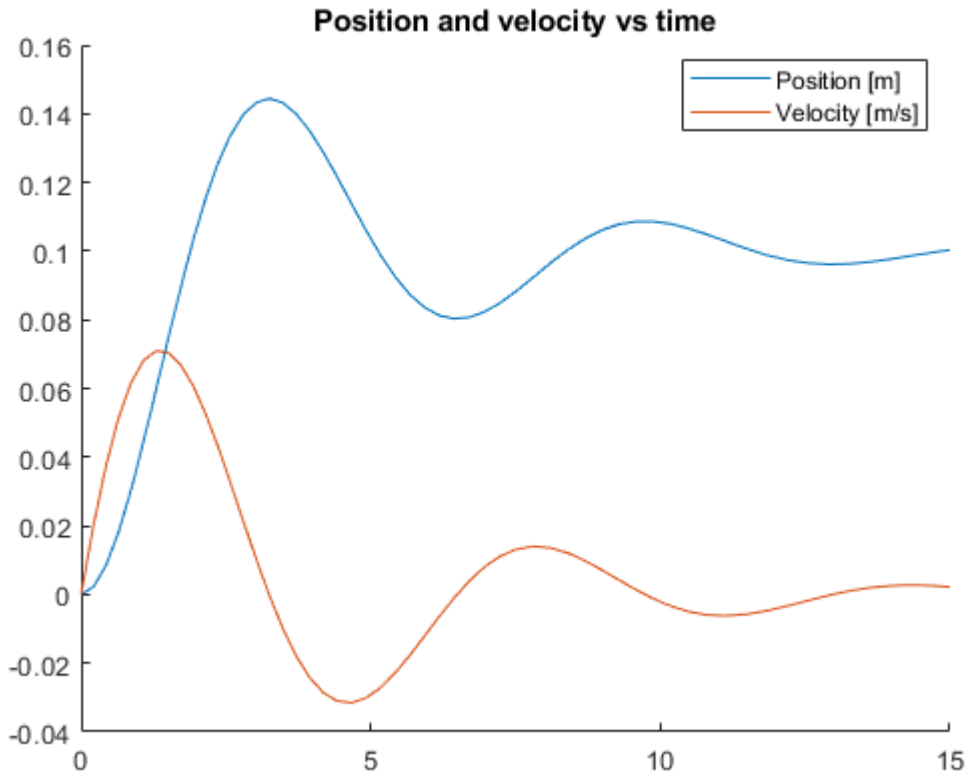
```
clear all; close all;
options = odeset('RelTol', 1e-4, 'AbsTol', [1e-4, 1e-4]);
x0 = 0; % initial position
v0 = 0; % initial vertical velocity
[T, W] = ode45(@vehdiff, [0, 15], [x0; v0], options);
```

The first column of the solution vector W contains the values of the position ($w_1 = x$), the second column contains the values of the vertical velocity $\left(w_2 = \frac{dx}{dt}\right)$.

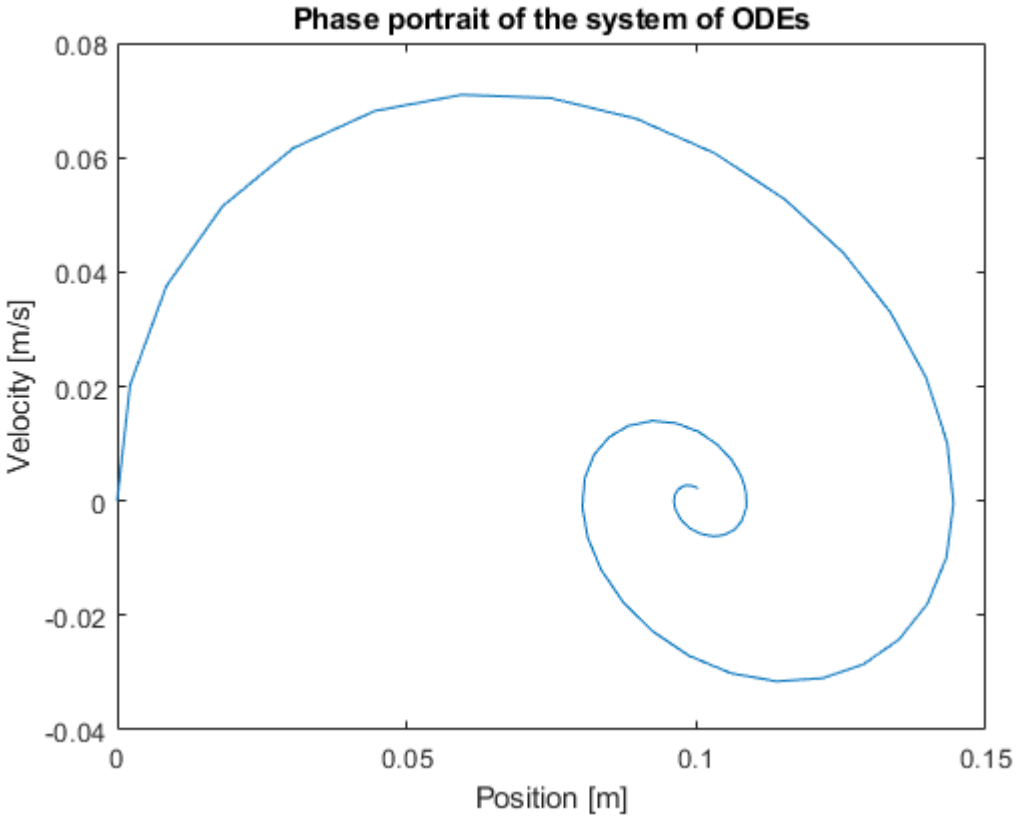
<u>Important remark:</u> always include the independent variable (t in our case) as a variable of the function file that defines the system of ODEs. If the system is defined in a separate file, always include the '@' symbol before its name.

To visualize our results, we can plot the position and the velocity values as functions of time. We can also plot the velocity as a function of the position (where time becomes a parameter of the curve), this is called the phase portrait of the system of ODEs:

```
figure(1);
hold on;
x = W(:, 1);
v = W(:, 2);
plot(T, x, T, v);
legend('Position [m]', 'Velocity [m/s]', 'Location', 'best');
title('Position and velocity vs time');
```




```
figure(2);
plot(x, v);
xlabel('Position [m]');
ylabel('Velocity [m/s]');
title('Phase portrait of the system of ODEs');
```



N-th order ODEs

When dealing with third-order, fourth-order to even higher order ODEs, let's assume an n-th order ODE, the solutions can be reduced to solving a system of n number of first-order ODEs by introducing new variables. In order to solve such an equation, we need n initial values, e.g. in the third-order case 3, in the fourth-order case 4.

The general form of an n-th order ODE:

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2 y}{dt^2}, \cdots, \frac{d^{(n-1)} y}{dt^{(n-1)}}\right), \qquad a \leq t \leq b$$

The initial conditions:

$$y(a) = A_1, \qquad \frac{dy}{dt}(a) = A_2, \qquad \frac{d^2 y}{dt^2}(a) = A_3, \qquad \cdots, \qquad \frac{d^{(n-1)} y}{dt^{(n-1)}}(a) = A_n$$

If we introduce n new variables and their initial values, we can write the following system of ODEs:

$$
\begin{aligned}
y &= w_1 \\
\frac{dw_1}{dt} &= \frac{dy}{dt} = w_2 \\
\frac{\mathrm{dw}_2}{dt} &= \frac{d^2y}{dt^2} = w_3 \\
&\vdots \\
\frac{dw_{n-1}}{dt} &= \frac{d^{(n-1)}y}{dt^{(n-1)}} = w_n \\
\frac{dw_n}{dt} &= \frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2y}{dt^2}, \cdots, \frac{d^{(n-1)}y}{dt^{(n-1)}}\right)
\end{aligned}
\qquad
\begin{aligned}
w_1(a) &= A_1 \\
w_2(a) &= A_2 \\
w_3(a) &= A_3 \\
&\vdots \\
w_n(a) &= A_n
\end{aligned}
$$

As an example, let's solve the following third-order ODE on the interval $[0, 1]$:

$$2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2} - \frac{d^3y}{dx^3} = 0$$

The initial values:

$$
\begin{aligned}
y(0) &= 3 \\
\frac{dy}{dx}(0) &= 2 \\
\frac{d^2y}{dx^2}(0) &= 7
\end{aligned}
$$

The first step is to express the highest order derivative in the equation:

$$\frac{d^3y}{dx^3} = 2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2}$$

Next, we can introduce the new variables and write the system of 3 first-order ODEs:

$$w_1 = y, \qquad w_2 = \frac{dy}{dx}, \qquad w_3 = \frac{d^2y}{dx^3}$$

In the new system of ODEs, we have to define the derivatives of the new variables:

$$
\begin{aligned}
f_1 &= \frac{dw_1}{dx} = \frac{dy}{dx} = w_2 \\
f_2 &= \frac{dw_2}{dx} = \frac{d^2y}{dx^2} = w_3 \\
f_3 &= \frac{dw_3}{dx} = \frac{d^3y}{dx^3} = 2x - 3y + 4 \cdot \frac{dy}{dx} + x \cdot \frac{d^2y}{dx^2}
\end{aligned}
\qquad
\begin{aligned}
w_1(0) &= 3 \\
w_2(0) &= 2 \\
w_3(0) &= 7
\end{aligned}
$$

The system is created in a separate file, **`diff3.m`**:

```
function dwdx = diff3(x, w)
    f1 = w(2);
    f2 = w(3);
    f3 = 2*x - 3*w(1) + 4*w(2) + x*w(3);
```

```
        dwdx = [f1; f2; f3];
    end
```

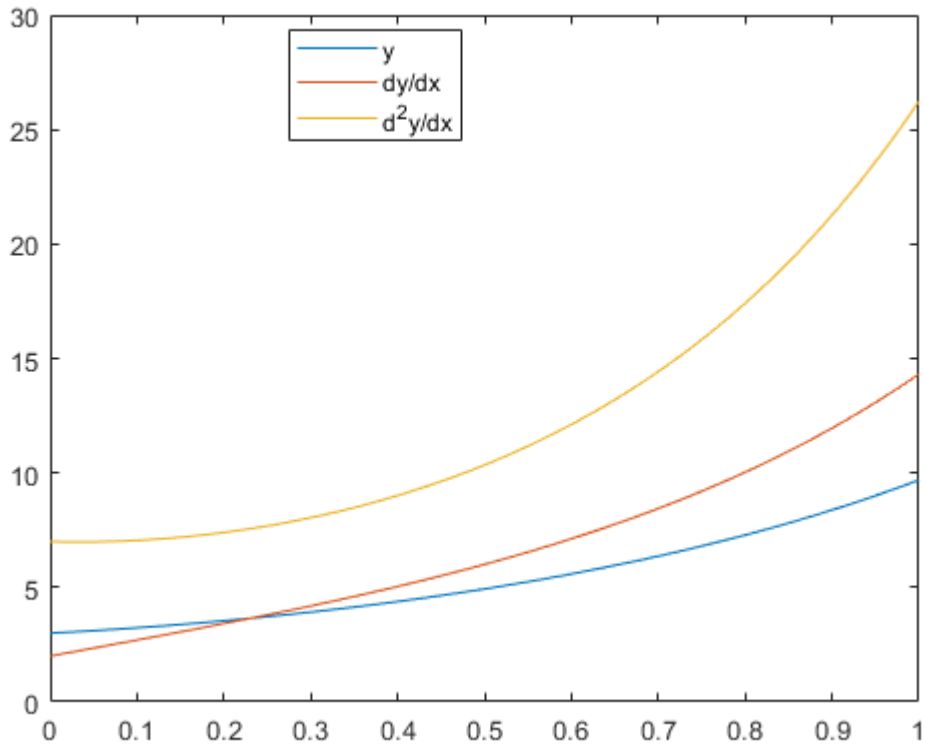
Solution in MATLAB:

```
clear all; close all;

% Initial values
w10 = 3; w20 = 2; w30 = 7;

% Solution
[X, W] = ode45(@diff3, [0, 1], [w10; w20; w30]);

% Plotting the results
figure(1);
plot(X, W(:, 1), X, W(:, 2), X, W(:, 3));
lgd = legend('y', 'dy/dx', 'd^2y/dx', 'Location', 'best');
```



N-th order systems of ODEs

Given an n-th order system of ODEs, the solution can be reduced similarly to the example shown above by introducing new variables. For the sake of example, take the following general form of a second-order system of ODEs:

$$\frac{d^2x}{dt^2} = F_1\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$
$$\frac{d^2y}{dt^2} = F_2\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$

We have to define 4 new variables:

$$w_1 = x, \qquad w_2 = y, \qquad w_3 = \frac{dx}{dt}, \qquad w_4 = \frac{dy}{dt}$$

Using these new variables, we can write a system of four first-order ODEs:

$$f_1 = \frac{dw_1}{dt} = \frac{dx}{dt} = w_3$$
$$f_2 = \frac{dw_2}{dt} = \frac{dy}{dt} = w_4$$
$$f_3 = \frac{dw_3}{dt} = \frac{d^2x}{dt^2} = F_1\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$
$$f_4 = \frac{dw_4}{dt} = \frac{d^2y}{dt^2} = F_2\left(t, x, y, \frac{dx}{dt}, \frac{dy}{dt}\right)$$

The solution is analogous to the previous example.