4 ODE

Contents

```
% make sure the workspace is clear
if ~exist('____code____','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lectureN.txt
```

LESSON SUMMARY

- Ordinary Differential Equations (ODE) are equations including derivatives with respect to an independent variable. Usually, but by no means always, the independent variable is time. Newton's second law and related equations involve time derivatives and are ODE. So are the equations of evolving chemical reactions and electrical circuits. Mechanical, chemical, and electrical engineering students take note.

- To keep it simple, in this lesson we assume that the independent variable is time. If it is not, you can still *think* of it as time-like.

- Typically, there is more than one ODE for more than one unknown. The number of ODE must match the number of unknowns.

- To solve the ODE in Matlab, you must first create a function that, given values for the independent variable and the unknowns, outputs the derivatives of the unknowns. You should use the ODE inside this function to find these derivatives. More specifics on the needed function: The input values of all the unknowns should be assumed to be stored together in a single input column array. All the found derivatives must be put together in a single *column* array that is the returned output of the function.

- To solve the ODE, you should also be given the values of the unknowns at an initial time. These initial values of the unknowns are called "initial conditions" for obvious reasons. (Warning: do not use the initial conditions in your function above.)

- With your function and the initial conditions, you can solve the ODE using Matlab function `ode45`:

  1. The first input argument of `ode45` must be your function giving the derivatives of the unknowns. Or, if you ended up creating a function that has more than the two input arguments `ode45` wants, the first argument should be an anonymous function with the required two arguments, and based on your function with more of them. Either way, the function should be preceded by an @.

  2. The second input argument of `ode45` must an array of times from the initial time to the final time that you want to have computed.

  3. The third input argument must be the initial values of the unknowns, put in a column array.

- If you specified only two times in the second input argument, start and end, function `ode45` will return the values of the unknowns at times of its own choice from start to end. Otherwise it will return the unknowns at the times you specified. In either case, function `ode45` returns the computed times as a column array and the corresponding unknowns as a 2D array. In the 2D array, the first column contains the values of the first unknown, the second column the values of the second unknown, etcetera. You will typically need to take the separate unknowns out of the array. Use `(:,UNKNOWNNUMBER)` indices to do so.

Key areas of the online book

Before the lecture, in the online book do:

- 4.7 More indexing: PA 4.7.1 only.
- 4.11 Column arrays: all.
- 7.1 2D arrays - Introduction: skip the stuff after PA 7.1.4.

The online book has nothing on ordinary differential equations. If you want more info, look at "Getting Started with Matlab" by Rudra Patrap. This is a well written book, worth buying.

THE EXAMPLE PROBLEM

We want to study Galileo's experiment of dropping iron spheres from the 60 m high leaning Tower of Pisa and seeing how long it takes for them to hit the ground.

If we ignore air resistance, we can easily solve this problem using Physics I only. In particular, let s be the distance that the sphere has traveled down. By definition, the time derivative of the distance traveled is the velocity v. And Newton's second law tells us that the mass m of the sphere times the acceleration (the time derivative of the velocity v), equals the force. That force is the force of gravity mg. So altogether we have:

$$\frac{\mathrm{d}s}{\mathrm{d}t} = v \qquad m\frac{\mathrm{d}v}{\mathrm{d}t} = mg$$

A system of equations like this is called a system of "Ordinary Differential Equations", *(ODE)*, because the equations contain derivatives.

To solve the above ODE, we still need some additional information. In particular, we need "Initial Conditions" valid at the time that Galileo drops the sphere. We will take this time to be zero. At that time, the sphere has not yet traveled any distance so s must be zero at the initial time. In addition, we will assume that Galileo *drops* the sphere, not that he *throws* it down. So we also assume that v is zero at the initial time. So both initial conditions are zero:

$$\text{at the initial time } t = 0: \quad \begin{cases} s = 0 \\ v = 0 \end{cases}$$

Solving the problem analytically.

To solve this system analytically, we can easily integrate the second ODE (i.e. Newton's second law), to give

$$v = gt + C_1$$

where C_1 is a constant. This constant must be zero because of the initial condition that $v = 0$ when $t = 0$. Then we can integrate the first ODE to find the displacement s as:

$$s = \tfrac{1}{2}gt^2 + C_2$$

and C_2 must be zero because of the initial condition that $s = 0$ when $t = 0$. Substituting in the 60 m height of the tower of Pisa for s and 9.81 m/s^2 for g, we find that the time for the sphere to reach the ground is about 3.5 seconds.

SOLVING THE PROBLEM WITH MATLAB INSTEAD

Next we would like to solve the same problem as above, not mathematically but *numerically* with Matlab. For non-tricky systems of ODE like the current example, the usual way to solve it is using the `ode45` function.

Specifying the given ODE

Of course, `ode45` will need information on what ODE we want it to solve. It is not a mindreader. We must provide this information as a *function.* This function must satisfy very specific requirements.

The input arguments should be (1) the independent variable, the time t in our case, or `t` in Matlab; and (2) the unknowns, s and v in our case, combined into a *single* column array. We will name this array `unknowns`.

The output argument of the function must be the derivatives of the unknowns, combined into a single *column array.* We will name this array `unknownsDerivatives`. Inside the function, we should compute the derivatives from the input unknowns using the ODE.

A *minimal* function that satisfies these requirements for our problem is function `Galileo1` shown below:

```
function unknownsDerivatives = Galileo1(t,unknowns)

% take the unknowns out of their array for readability
s=unknowns(1);
v=unknowns(2);

% find the derivative ds/dt now
dsdt=v;
```

```
% acceleration of gravity
g=9.81;

% find the derivative dv/dt now from Newton's second law
dvdt=g;

% return the derivatives as a *column* array
unknownsDerivatives=[dsdt dvdt]';

end
```

The ode45 call

Next we can find the solution of the ODE problem using `ode45`. The needed command takes the general form:

```
    [tValues unknownsValues] = ...
        ode45(ODEFUN, tDesired, unknownsInitialValues)
```

Let's look at this in more detail. For the first input parameter of `ode45`,

```
    ODEFUN
```

we must specify our function `Galileo1` that computes the derivatives of the unknowns using the ODE. The name should be preceded by an @. Check lesson 2 for why.

For the second input parameter of `ode45`,

```
    tDesired
```

we must specify the times for which we want `ode45` to find the unknowns. We want `ode45` to compute the solution from time 0 to the 3.5 seconds it takes the sphere to hit the ground. If we were only interested in the values of the unknowns at the final time 3.5, we could specify `tDesired` as [0 3.5]. Then `ode45` will return the unknowns at the initial time 0, some intermediate times of its own choosing, and the final time 3.5. However, we also want to plot the intermediate times, and `ode45` might not return enough times to do that accurately. To be safe, it is better to fully specify the times to find the unknowns at. That can be done by specifying tDesired as `linspace(0,3.5,n)` with `n` say 50 to get the unknowns at 50 equally spaced times.

For the third and last input parameter of `ode45`,

```
    unknownsInitialValues
```

we must specify the initial values for the unknowns. In our example, initially both s and v are zero, so we need to specify two zeros here. Note that this should be a column array so specify it either as [0; 0] or as [0 0]'. (The quote turns the [0 0] row into a column.)

As far as the output *produced* by the

```
    [tValues unknownsValues] = ...
        ode45(ODEFUN, tDesired, unknownsInitialValues)
```

`ode45` call is concerned:

- `tValues` is a column array that contains the values of the time t at which `ode45` has computed the unknowns s and v for us.
- `unknownsValues` are the values of the unknowns at these times, as a two-dimensional array.

Let's try it:

```
% the given initial values s=0, v=0, as a column array
unknownsInitialValues=[0 0]';

% request the solution at 50 times from 0 to 3.5
tDesired=linspace(0,3.5,50);

% call ode45 to find the solution for those times
[tValues unknownsValues] = ...
    ode45(@Galileo1,tDesired,unknownsInitialValues);
```

Function `ode45` has now computed the unknowns s and v at the times in tValues. However, it has dumped all the computed values in the single array `unknownsValues`. In particular:

1. `unknownsValues(:,1)` are the values of the first unknown, s in our example, for all computed times in `tValues`.
2. `unknownsValues(:,2)` are the values of the second unknown, v in our example, for all computed times in `tValues`.

Note that the second index is always the number of the unknown. The first index is the number of the computed time; specifing this as a colon means "all computed times".

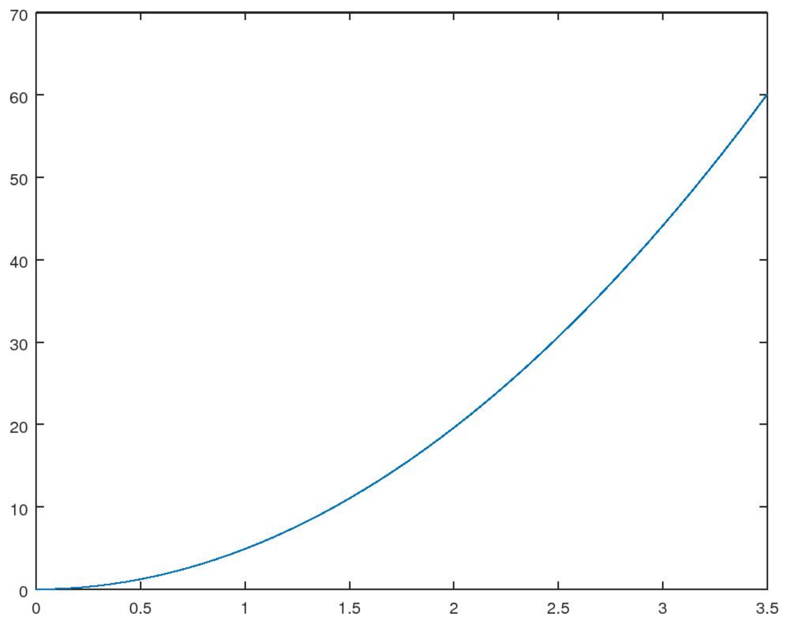
To avoid code that is hard to read and error prone, we should take the individual unknowns out of array `unknownsValues` and give them easily understandable names. In this case we only need the s values.

```
% take the s values computed by ode45 out of the 2D array
sValues=unknownsValues(:,1);
```

We can now do the things we want:

```
% plot the s values against the times in tValues
plot(tValues,sValues)

% print out the final distance using ode45
fprintf('Without air resistance, the ode45 distance\n')
fprintf('after 3.5 seconds is: %7.4f m\n',sValues(end))

% print out the final distance using the analytic formula
fprintf('Without air resistance, the exact distance\n')
fprintf('after 3.5 seconds is: %7.4f m\n',0.5*9.81*3.5^2)
```

```
Without air resistance, the ode45 distance
after 3.5 seconds is: 60.0863 m
Without air resistance, the exact distance
after 3.5 seconds is: 60.0862 m
```



USING ANONYMOUS FUNCTIONS

For more complicated problems, we may need to use anonymous functions, just like we needed to do in lessons 2 and 3.

The problem with air resistance

For an example problem that will require an anonymous function, consider the case where we include air resistance. Air resistance will slightly slow down even a heavy sphere, and can slow down a light sphere quite a lot.

Air resistance makes solving the motion analytically a lot more difficult. Fortunately, with Matlab we can still solve it easily numerically. The only thing we need to do is add the correct air resistance force. In particular, the two equations become

$$\frac{\mathrm{d}s}{\mathrm{d}t} = v \qquad m\frac{\mathrm{d}v}{\mathrm{d}t} = mg - F_{\mathrm{air}}$$

Until you reach Thermal Fluids 1, you will need to google what the expression for the air resistance of a sphere is. It turns out to be

$$F_{\mathrm{air}} = C_D \tfrac{1}{2} \rho_{\mathrm{air}} v^2 A$$

where C_D is called the "drag coefficient" of the sphere, ρ_{air} is the density of air, and A is the "frontal area" (area seen from the front) of the sphere,

$$A = \pi r^2$$

where r is the radius of the sphere.

Also note that when we divide Newton's equation by the mass of the sphere m, as we need to do to get the derivative of v, we end up with a term F_{air}/m. So the mass no longer drops out. That means that we now also need to know the mass of the iron sphere. That is simply the density of iron times the volume of the sphere,

$$m = \rho_{\mathrm{iron}} \frac{4\pi}{3} r^3$$

Approximate values for the various constants we need are

$$C_D \approx 0.5 \qquad \rho_{\mathrm{air}} \approx 1.225 \text{ kg/m}^3 \qquad \rho_{\mathrm{iron}} \approx 7{,}860 \text{ kg/m}^3$$

Note that in reality the drag coefficient depends on the velocity (see Wikipedia). The value 0.5 is a reasonable ballpark average value.

We can put the above equations and constants in a new function called `Galileo`. This function then includes the effect of air resistance. However, we should not put any particular value for r in the function, as we want to try out a number of different values of r. It would be messy and error-prone to change the function for every individual value of r. So our only reasonable option is to add r to the input arguments of function `Galileo`. Then the final function becomes:

```
function unknownsDerivatives = Galileo(t,unknowns,r)

%
% Function that describes the ordinary differential
% equations governing Gallileo's falling iron spheres.
%
%       unknownsDerivatives = Galileo(t,unknowns,r)
%
% Input: t: the time since the start of the fall.
%        unknowns: array with two components:
%            unknowns(1): the distance 's' that the
%                         sphere has traveled down.
%            unknowns(2): the downward velocity 'v' of
%                         the sphere.
%        r: radius of the iron sphere.
%
% Output: unknownsDerivatives: the time derivatives of
%         the unknowns:
%            unknownsDerivatives(1) = ds/dt = v
%            unknownsDerivatives(2) = dv/dt = ...
%                                     (FGravity - FAir)/m
%         where FGravity is the force of gravity, FAir
%         the force of air resistance, and m the mass of
%         the iron sphere.
%
% This function assumes certain reasonable values for the
% acceleration of gravity, the densities of iron and air,
% and the average drag coefficient of the sphere.  These
% values may need to be adjusted depending on conditions.
%

% take the unknowns out of their array for readability
s=unknowns(1);
v=unknowns(2);

% find the derivative ds/dt now
dsdt=v;

% acceleration of gravity
g=9.81;

% density of iron
rhoIron=7860;

% density of air at sea level
```

```
rhoAir=1.225;

% approximate drag coefficient of a normal size sphere
CD=0.5;

% frontal area of the iron sphere
A=pi*r^2;

% mass of the iron sphere
m=(4/3)*pi*r^3*rhoIron;

% force of gravity
FGravity=m*g;

% force of air resistance
FAir=CD*0.5*rhoAir*v^2*A;

% find the derivative dv/dt now from Newton's second law
dvdt=(FGravity-FAir)/m;

% return the derivatives as a *column* array
unknownsDerivatives=[dsdt dvdt]';

end
```

The needed anonymous function

The additional parameter r in function `Galileo` is a problem because `ode45` will not accommodate it. As far as `ode45` is concerned, the function ODEFUN must have exactly two input parameters; time and the array of unknowns.

The solution for this problem is much like the earlier one for `fzero` in lesson 2. We must define an anonymous function that has the two arguments that `ode45` needs and that uses `Galileo` to get its values. In short,we need the anonymous function

```
    (t,unknowns) Galileo(t,unknowns,r)
```

That then is the last thing needed in getting the case with air resistance to work.

Creating a script for solving the problem

Since we want to find the solution for more than one value of the sphere radius r, it would be messy to keep retyping the same code.

Instead what we can do is create a script file `SolveGalileo.m` that finds *one* solution assuming the value of r has already been set. Then we can run this same script using different values for r.

The contents of script `SolveGalileo.m` are:

```
% the given initial values s=0, v=0, as a column array
unknownsInitialValues=[0 0]';

% request the solution at 50 times from 0 to 3.5
tDesired=linspace(0,3.5,50);

% call ode45 to find the solution to the final time
[tValues unknownsValues] = ...
    ode45(@(t,unknowns) Galileo(t,unknowns,r),...
          tDesired,unknownsInitialValues);

% take the s values computed by ode45 out of the 2D array
sValues=unknownsValues(:,1);

% print out the sphere diameter and final distance
fprintf('For D = %4.2f m, the distance is: %5.2f m\n',...
  2*r,sValues(end))

% plot the distance traveled s versus time t
plot(tValues,sValues)
```

Computing a few different cases

Now we are ready to study the effect of the sphere radius!

```
% try a 20 cm radius
r=0.2;
% run script SolveGalileo to find the distance
SolveGalileo

% put a hold on the graph so that we can add more curves
hold on

% try a 5 cm radius
r=0.05;
% run script SolveGalileo to find the distance
SolveGalileo

% try a 2.5 cm radius
r=0.025;
```

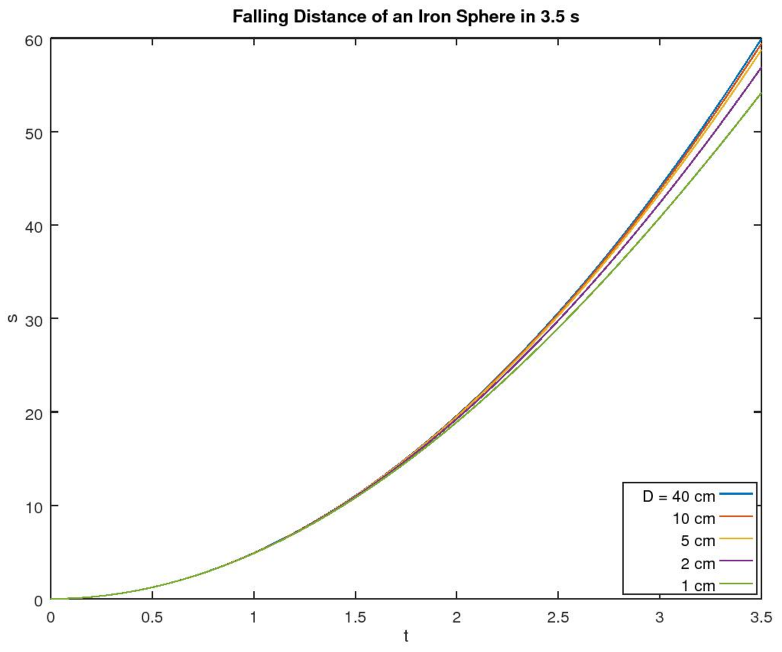
```
% run script SolveGalileo to find the distance
SolveGalileo

% try a 1 cm radius
r=0.01;
% run script SolveGalileo to find the distance
SolveGalileo

% try a 0.5 cm radius
r=0.005;
% run script SolveGalileo to find the distance
SolveGalileo

% add title, labels, and a legend to the graph
title('Falling Distance of an Iron Sphere in 3.5 s')
xlabel('t')
ylabel('s')
legend('D = 40 cm',...
       '       10 cm',...
       '        5 cm',...
       '        2 cm',...
       '        1 cm',...
       'location','southeast')
% put the legend text to the left of the line segments
legend('left')

% allow any other figures to be made
hold off
```

```
For D = 0.40 m, the distance is: 59.91 m
For D = 0.10 m, the distance is: 59.40 m
For D = 0.05 m, the distance is: 58.73 m
For D = 0.02 m, the distance is: 56.87 m
For D = 0.01 m, the distance is: 54.14 m
```

Note that the 10 cm sphere should be about half a meter in front of the 5 cm one when they reach the ground. However, a dishonest Galileo could easily compensate for that by dropping the bigger sphere about 0.02 s later than the smaller one. That is an imperceivably small delay. Even a honest Galileo might do it.

ADDITIONAL REMARKS

Make sure that you use suitable units when solving ODE with `ode45`. For example, do not try to solve Galileo's problem in units of microns and centuries. If your expected solution consists of very large or very small numbers, at the



very least you will need to change an option like `AbsTol` to an appropriate value. See the Matlab documentation.

If the system of first order differential equations describes, say, a set of chemical reactions, there may be a problem with using `ode45`. Typically, some reactions proceed very quickly and others much more slowly. The slow reactions imply that you have to solve the evolution for a relatively long time. But `ode45` must compute accurately over the shortest time scales in order not to get the fast reactions all wrong. Having to compute accurately over very many short time intervals is a problem for `ode45`; the computation may take excessive computational time.

Such a problem, and any other problem where there is a very large spread in typical time scales, is called "stiff". For stiff problems you want to use a solver dedicated to such problems. One basic one provided by Matlab is `ode15s`. It can be used just like `ode45`.