4 ODE

Contents

Initialization

```
% reduce needless whitespace
format compact
% reduce irritations
more off
% start a diary
%diary lectureN.txt
```

THE PROBLEM WE WANT TO SOLVE

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.
Nowadays we can easily solve this problem. 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 of the sphere m times the acceleration (the time derivative of the velocity v), equals the force. That force is the force of gravity mg. So we have:

$$\begin{aligned}\frac{\mathrm{d}s}{\mathrm{d}t} &= v \\ m\frac{\mathrm{d}v}{\mathrm{d}t} &= mg\end{aligned}$$

A system of equations like this is called a system of "Ordinary Differential Equations", *(ODE)*, because the equations contain derivatives. We can easily solve it.
But to do so, we need some additional information, we need "Initial Conditions". In particular, we will take the time to be 0 when Gallileo releases the sphere. At that time, the sphere has not yet traveled any distance so s must be zero at time zero. In addition, we will assume that gallileo *drops* the sphere, not that he *throws* it down. So we also assume that v is zero at time zero. So the initial conditions are:

$$s = 0 \qquad \text{and} \qquad v = 0 \qquad \text{at} \qquad t = 0$$

We can easily integrate the second ODE (i.e. Newton's second law), to give

$$v = gt + C_1$$

and C_1 must be zero because of the initial condition on v. With that 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. 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 of math

Now we would like to solve the same problem as above, but not mathematically but *numerically* with Matlab's `ode45` function. With `ode45`, systems of ordinary differential equations can be solved as

```
[tValues, unknownsValues] = ...
      ode45(ODE, tRange, unknownsInitialValues)
```

Here the "unknowns" will be our two unknowns s and v. Completely to the right in the `ode45` call, for

```
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 *must* 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.)
For the middle parameter of `ode45`,

```
tRange
```

we must specify the time range that we want `ode45` to evaluate. We want ODE to compute the solution from time 0 to the 3.5 seconds it takes the sphere to hit the ground. The simplest is to specify [0,3.5]. More generally, we can specify

```
linspace(0,3.5,n)
```

where `n` is at least two. The advantage of specifying the parameter this way is that if we specify a bigger `n`, `ode45` will find the unknowns at more intermediate times between 0 and 3.5. That may be desirable for plotting or interpolating the found unknowns.
For the first parameter of `ode45`,

```
ODE
```

we must specify the ordinary differential equations that we want `ode45` to solve. The only systems of differential equations that `ode45` will solve are of the form

$$\begin{aligned} \frac{\mathrm{d}u_1}{\mathrm{d}t} &= \ldots \\ \frac{\mathrm{d}u_2}{\mathrm{d}t} &= \ldots \\ &\vdots \end{aligned}$$

where u_1 and u_2 are the unknowns (s and v in our example). So we must divide Newton's second law in our equations by m in order to get rid of the m in the left hand side.
Note next that the ODE must be specified in terms of a *function*. This function must take in values of the unknowns, (values that `ode45` will provide). The function must return the corresponding values of the *derivatives* of the unknowns. A *minimal* function that will do that for our problem is function `Galileo1` shown below:

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

% take s and v out of unknowns for readability
s=unknowns(1);
v=unknowns(2);

% acceleration of gravity
g=9.81;

% derivative ds/dt
dsdt=v;
```

```
% derivative dvdt
dvdt=g;

% return them as a *column* vector
unknownsDerivatives=[dsdt dvdt]';

end
```

As far as the output *produced* by the

```
[tValues, unknownsValues] = ...
     ode45(ODE,tRange,unknownsInitialValues)
```

`ode45` call is concerned:

- `tValues` are the values of the time t at which `ode45` has computed s and v for us. These values will at least include the time values in the `tRange` we specified.
- `unknownsValues` are the values of the unknowns s and v at these times.

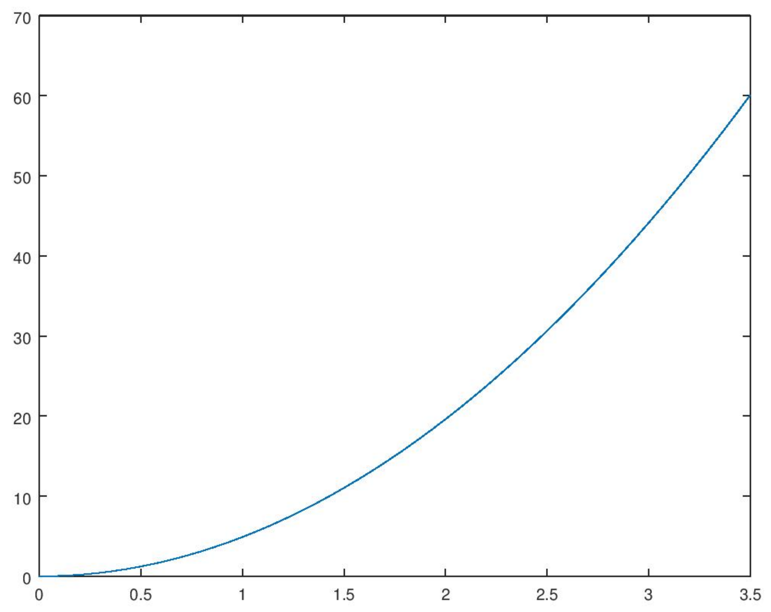
In particular,

1. `unknownsValues(:,1)` are the values of s for the computed times in `tValues`.
2. `unknownsValues(:,2)` are the values of v for the computed times in `tValues`.

```
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45('Galileo1',linspace(0,3.5,50),[0 0]');

% plot the $s$ values found by ode45 against time
plot(tValues,unknownsValues(:,1))
% note that we got the s values out of "multidimensional"
% array unknownsValues(...,...) by specifing the second
% "index" as 1 (meaning just s values, no v values) and
% the first "index" as a colon (meaning *all* s values)
```

INCLUDING AIR RESISTANCE

So far we have looked only at the ideal case that the sphere can drop through the air without any resistance. But the air 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 include the correct air resistance. In particular, the two equations become

$$\begin{aligned}\frac{\mathrm{d}s}{\mathrm{d}t} &= v\\ m\frac{\mathrm{d}v}{\mathrm{d}t} &= mg - F_{\text{air}}\end{aligned}$$

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

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

where C_D is the drag coefficient of the sphere, ρ_{air} the density of air, and A 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 for `ode45`, we end up with a term F_{air}/m. So we need to know

the mass of the iron sphere. That is simply the density of iron times the volume of the sphere,

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

Approximate values for the various constants are

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

We can put the various expressioms in a function `Galileo` which then includes the effect of air resistance. However, we should not put any value of r in that file, as we want to try out many values of r. We cannot and should not change the function for each individual value of r. So we must add r to the input arguments of function `Galileo`. So the final function becomes:

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

% Function that describes the ordinary differential
% equations governing Gallileo's falling iron spheres.
%
% Input: t: the time since the start of the fall.
%        unknowns: vector 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, to be used by function ode45:
%             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.

% take s and v out of unknowns for readability
s=unknowns(1);
v=unknowns(2);

% density of iron
rhoIron=7860;

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

```

% acceleration of gravity
g=9.81;

% force of gravity
FGravity=m*g;

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

% density of air at sea level
rhoAir=1.225;

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

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

% derivative ds/dt
dsdt=v;

% derivative dvdt
dvdt=(FGravity-FAir)/m;

% return them as a *column* vector
unknownsDerivatives=[dsdt dvdt]';

end
```

The additional parameter r in function `Galileo` is a problem because `ode45` will not accomodate it. As far as `ode45` is concerned, the function ODE must have exactly two parameters; time and the vector of unknowns.
The solution for this problem is much like the earlier one for `fzero`. We must define an anonymous function that has the two arguments that `ode45` needs and that uses `Galileo` to get its values. That then is the last thing needed in getting the case with air resistance to work.

Computing a few different cases

```
% the initial values of s and v at time t = 0
unknownsIV=[0 0]';

% we want the solution at at least 50 times from 0 to 3.5
tRange=linspace(0,3.5,50);

```

```
% try a 20 cm radius
r=0.2;
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45(@(t,y) Galileo(t,y,r),tRange,unknownsIV);
% print out the final distance
fprintf('For r = %4.2f, the distance is: %5.2f\n',...
  r,unknownsValues(end,1))
% plot the distance traveled s versus time t
plot(tValues,unknownsValues(:,1))

% put a hold on plot to keep all plotted curve
hold on

% try a 10 cm radius
r=0.1;
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45(@(t,y) Galileo(t,y,r),tRange,unknownsIV);
% print out the final distance
fprintf('For r = %4.2f, the distance is: %5.2f\n',...
  r,unknownsValues(end,1))
% plot the distance traveled s versus time t
plot(tValues,unknownsValues(:,1))

% try a 5 cm radius
r=0.05;
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45(@(t,y) Galileo(t,y,r),tRange,unknownsIV);
% print out the final distance
fprintf('For r = %4.2f, the distance is: %5.2f\n',...
  r,unknownsValues(end,1))
% plot the distance traveled s versus time t
plot(tValues,unknownsValues(:,1))

% try a 2 cm radius
r=0.02;
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45(@(t,y) Galileo(t,y,r),tRange,unknownsIV);
% print out the final distance
fprintf('For r = %4.2f, the distance is: %5.2f\n',...
  r,unknownsValues(end,1))
% plot the distance traveled s versus time t
plot(tValues,unknownsValues(:,1))
```

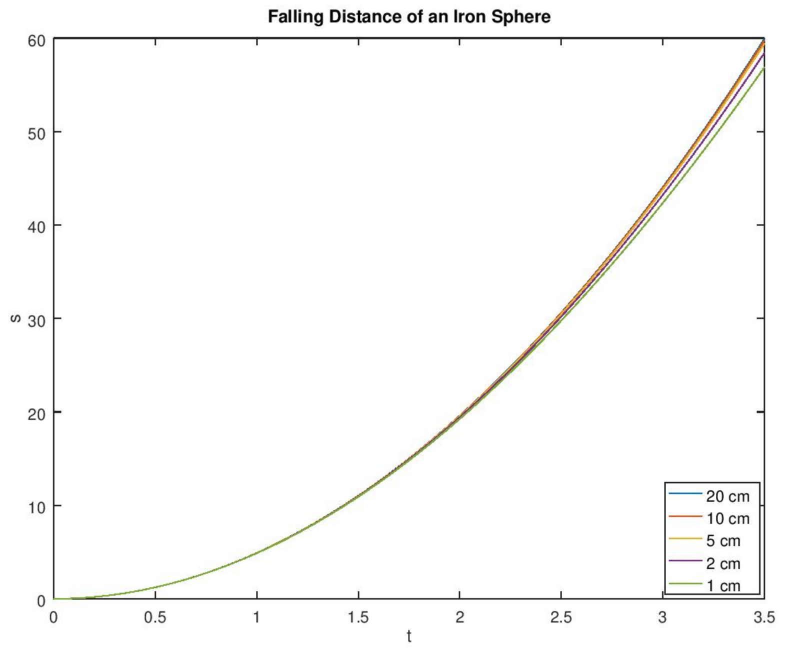
```
% try a 1 cm radius
r=0.01;
% call ode45 to find the solution to t=3.5
[tValues, unknownsValues] = ...
    ode45(@(t,y) Galileo(t,y,r),tRange,unknownsIV);
% print out the final distance
fprintf('For r = %4.2f, the distance is: %5.2f\n',...
  r,unknownsValues(end,1))
% plot the distance traveled s versus time t
plot(tValues,unknownsValues(:,1))

title('Falling Distance of an Iron Sphere')
xlabel('t')
ylabel('s')
legend('20 cm','10 cm','5 cm','2 cm','1 cm')
legend('location','southeast')
```

```
For r = 0.20, the distance is: 59.91
For r = 0.10, the distance is: 59.74
For r = 0.05, the distance is: 59.40
For r = 0.02, the distance is: 58.41
For r = 0.01, the distance is: 56.87
```

ADDITIONAL REMARKS

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 ode; 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.

End lesson 4


Falling Distance of an Iron Sphere
s
t
0
0.5
1
1.5
2
2.5
3
3.5
0
10
20
30
40
50
60
20 cm
10 cm
5 cm
2 cm
1 cm