Department of Engineering Mathematics

EMAT20920: Numerical Methods in MATLAB

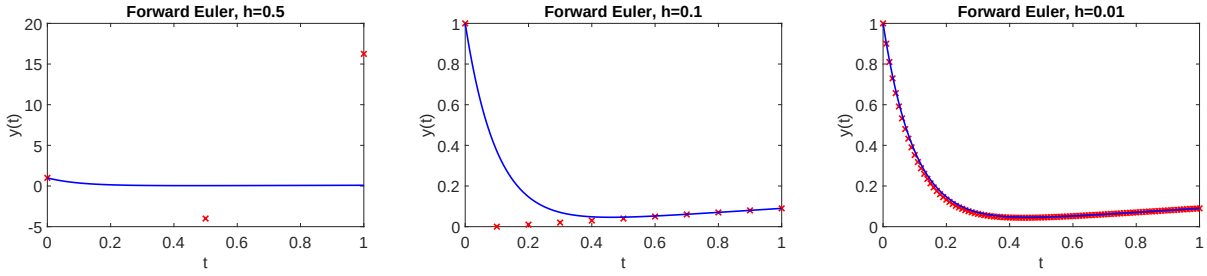
WORKSHEET 6 — SOLUTIONS
NUMERICAL SOLUTION OF ODEs

Question 1: Forward and Backward Euler

(a) Timesteps $h = 0.5$, $h = 0.1$ and $h = 0.01$ correspond to $N = 2$, $N = 10$ and $N = 100$ respectively. Calling `myEuler` with values as in the question

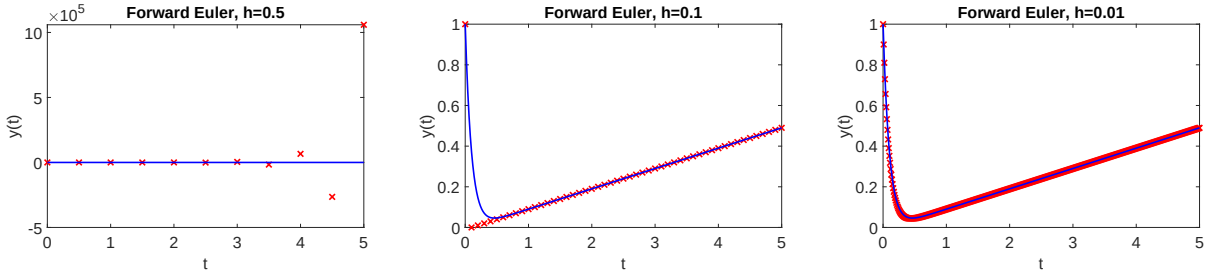
```
[t2,y2]=myEuler(@(t,y) t-10*y,[0 1],1,2);
[t10,y10]=myEuler(@(t,y) t-10*y,[0 1],1,10);
[t100,y100]=myEuler(@(t,y) t-10*y,[0 1],1,100);
```

and plotting the results together with the exact solution leads to the following three graphs:



Only the last, with timestep $h = 0.01$, gets the solution right for all $t = [0, 5]$. The first, with timestep $h = 0.5$ is qualitatively (and badly) wrong — it oscillates and grows (note the very different scale on the vertical axis). This is caused by the (conditional) instability of the forward Euler method.

We see the same effects, more dramatically, by solving up to $t = 5$ instead of $t = 1$ (using $N = 10$, $N = 50$ and $N = 500$, to get the same stepsizes $h = 0.5$, $h = 0.1$ and $h = 0.01$ respectively), yielding the following graphs:



(i) The right-hand-side function in this case is $f(t, y) = t - 10y$. Using this in the backward Euler

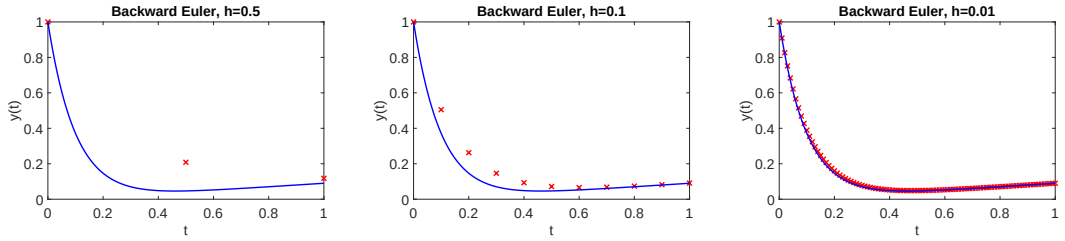
iteration we get

$$\begin{aligned} y_{k+1} &= y_k + hf(t_{k+1}, y_{k+1}) \\ &= y_k + h(t_{k+1} - 10y_{k+1}) \\ &= y_k + ht_{k+1} - 10hy_{k+1} \\ (1+10h)y_{k+1} &= y_k + ht_{k+1} \\ y_{k+1} &= \frac{y_k + ht_{k+1}}{1+10h} \\ &= \frac{y_k + h(t_k + h)}{1+10h} \qquad (\text{using } t_{k+1} = t_k + h) \end{aligned}$$

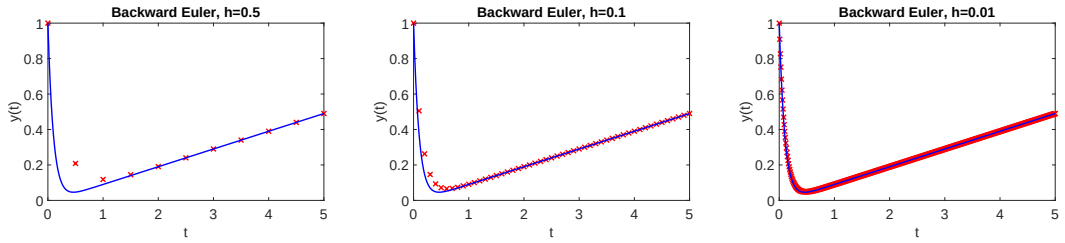
A simple MATLAB code to execute this is the following:

```
function [t,y]=backwardEuler(tspan,N)

x0=1;
t0=tspan(1); tend=tspan(2);
t=linspace(t0,tend,N+1)';
y=zeros(size(t));
h=(tend-t0)/N;

y(1)=x0;
for k=1:N
    y(k+1)=(y(k)+h*(t(k)+h))/(1+10*h);
end

end
```

which gives the following results for $t \in [0, 1]$:



and for $t \in [0, 5]$:



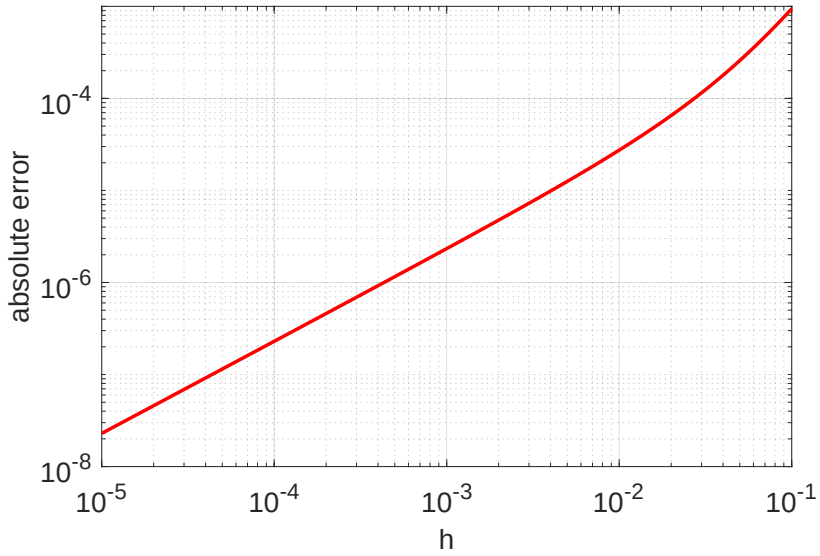
Now all results are qualitatively correct, with improving accuracy as $h \to 0$, as the Backward Euler method is unconditionally stable. Unfortunately, though, it's not often possible to (re)write the iteration in an explicit way; we have been lucky with the function f here.

(ii) The following MATLAB script will find the absolute error between numerical and exact solution at $t = 1$, and plot a graph:

```
t_range = [0 1];
N = floor(logspace(1,5,250));
abserr = zeros(size(N));
y_exact = @(t) (10*t - 1 + 101*exp(-10*t))/100;
for i=1:length(N)
    [t,y]=backwardEuler(t_range,N(i));
    abserr(i)=abs(y_exact(t_range(2))-y(end));
end
loglog((t_range(2)-t_range(1))./N,abserr,'r-','LineWidth',2)
xlabel('h'); ylabel('absolute error');
grid on;
```

Note the use of `floor` in the code above, which rounds numbers down to the nearest integer: the number of timesteps N must be an integer.

The resulting error vs. timestep plot is:



Reading off the graph, the gradient is equal to 1 (in the straight line portion of the graph — remember that, strictly speaking, we have to take the limit $h \to 0$), and so the backward Euler method has global truncation error of $\mathcal{O}(h)$, as expected.

(b) (i) The trapezoid method, from the lecture notes, is given by

$$y_{k+1} = y_k + \frac{h}{2}\Big[f(t_k, y_k) + f(t_{k+1}, y_{k+1})\Big]$$

Again we can plug in the right-hand-side function $f(t, y) = t - 10y$, and rearrange to find y_{k+1}

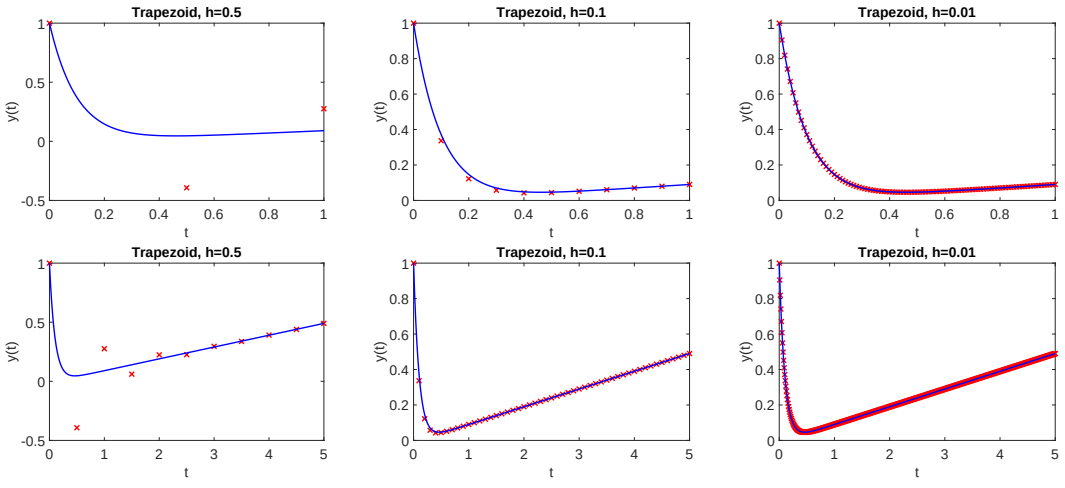
in terms of y_k:

$$\begin{aligned}
y_{k+1} &= y_k + \frac{h}{2}\Big[f(t_k, y_k) + f(t_{k+1}, y_{k+1})\Big] \\
&= y_k + \frac{h}{2}\Big[t_k - 10y_k + t_{k+1} - 10y_{k+1}\Big] \\
&= y_k + \frac{h}{2}(t_k + t_{k+1}) - 5hy_k - 5hy_{k+1} \\
(1+5h)y_{k+1} &= (1-5h)y_k + \frac{h}{2}(t_k + t_{k+1}) \\
y_{k+1} &= \left(\frac{1-5h}{1+5h}\right) y_k + \frac{h(t_k + t_{k+1})}{2(1+5h)} \\
&= \left(\frac{1-5h}{1+5h}\right) y_k + \frac{h(t_k + h/2)}{1+5h} \qquad \text{(using } t_{k+1} = t_k + h\text{)}
\end{aligned}$$

The following code will implement this iteration to solve the ODE:

```
function [t,y]=trapezoid(tspan,N)

x0=1;
t0=tspan(1); tend=tspan(2);
t=linspace(t0,tend,N+1)';
y=zeros(size(t));
h=(tend-t0)/N;
Hm=1-5*h;
Hp=1+5*h;

y(1)=x0;
for k=1:N
    y(k+1)=(Hm/Hp)*y(k)+h*(t(k)+h/2)/Hp;
end

end
```

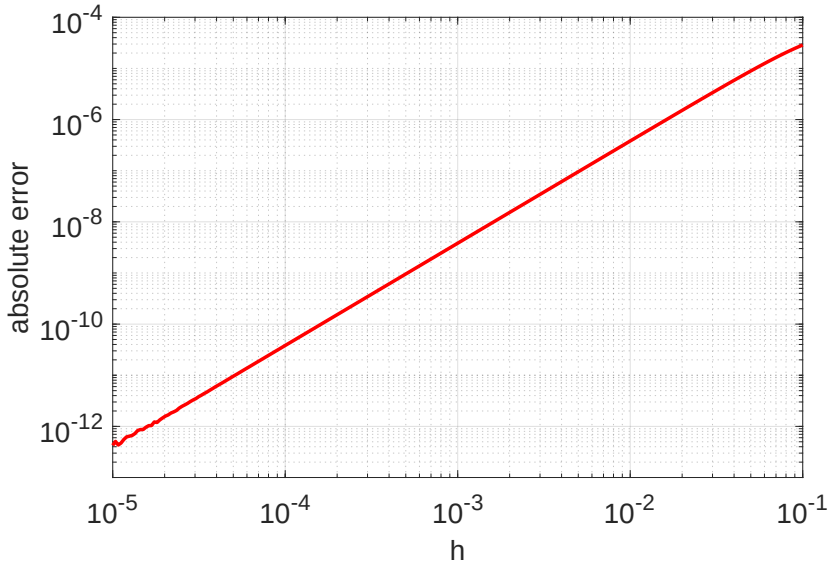
from which we can obtain the following results for $y(t)$, over the intervals $[0, 1]$ and $[0, 5]$:



We see that the trapezoid method is unconditionally stable. There's a hint of instability about the left-hand images, $h = 0.5$, but the iteration recovers to provide the right qualitative behaviour; decaying transient oscillation can still take place in unconditionally stable methods. We can find the global truncation error in exactly the same way as before:

```
t_range = [0 1];
N = floor(logspace(1,5,250));
abserr = zeros(size(N));
y_exact = @(t) (10*t - 1 + 101*exp(-10*t))/100;
for i=1:length(N)
    [t,y]=trapezoid(t_range,N(i));
    abserr(i)=abs(y_exact(t_range(2))-y(end));
end
loglog((t_range(2)-t_range(1))./N,abserr,'r-','LineWidth',2)
xlabel('h'); ylabel('absolute error');
grid on;
```

which gives the error versus stepsize result



from which we can conclude that the slope of the graph is 2, and hence that the global truncation error of the trapezoid method is $\mathcal{O}(h^2)$; i.e., it's a second order scheme.

Question 2: Event detection in `ode45`

(a) (i) We write $y_1 = x$, $y_2 = \dot{x}$ to transform the 2nd order ODE into a system of two 1st order ODEs (for $\boldsymbol{y} = (y_1, y_2)^T$). In MATLAB this can be written as follows:

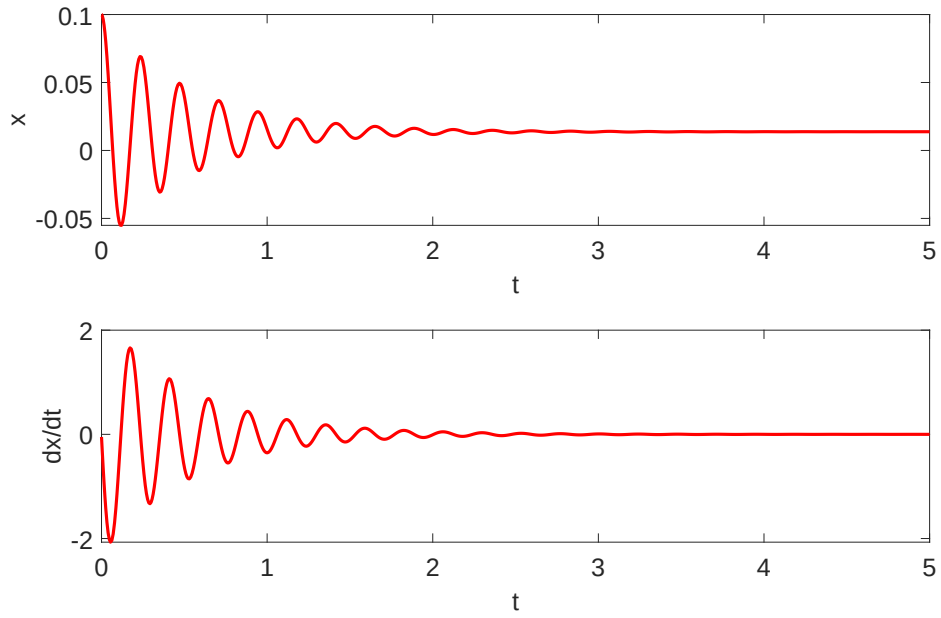
```
function dxdt = rhs_spring_v1(t,y)

m = 0.075;
c = 0.28;
k = 53.4;
g = 9.81;
F = m*g;

dxdt = [ y(2) ; (F-c*y(2)-k*y(1))/m ];

end
```

It's not the best way to write the function, though; it would be preferable to pass the parameters as inputs, particularly as later one of them (m) will vary during the simulation. We'll come back to this idea in part (b).

(ii) We can solve the ODE, and plot graphs of position x and velocity $\dot{x}$ versus time t, by typing:

```
[t,y]=ode45(@rhs_spring_v1,[0 5],[0.1;-0.05]);
subplot(2,1,1);
plot(t,y(:,1),'r-');
xlabel('t'); ylabel('x');
xlim([0 5]);
subplot(2,1,2);
plot(t,y(:,2),'r-');
xlabel('t'); ylabel('dx/dt');
xlim([0 5]);
```

The results look as follows:



The mass oscillates, with constant frequency, and settles to a steady displacement of around $x = 0.01$, with zero velocity. The mass-spring-damper system is underdamped.

(b) (i) In this part, we're going to have to deal with a non-constant mass, so we need to update the ODE right-hand-side function to have the mass as an input:

```
function dxdt = rhs_spring_v2(t,y,m)

c = 0.28;
k = 53.4;
g = 9.81;
F = m*g;

dxdt = [ y(2) ; (F-c*y(2)-k*y(1))/m ];

end
```

Turning to the events function, to detect a maximum of $x(t)$ we need to look for zeros of velocity, $\dot{x} = y_2$, crossing zero from positive to negative velocity (i.e., decreasing). We'll also need to stop the integration so we can change the mass. That's all achieved by the following events function:

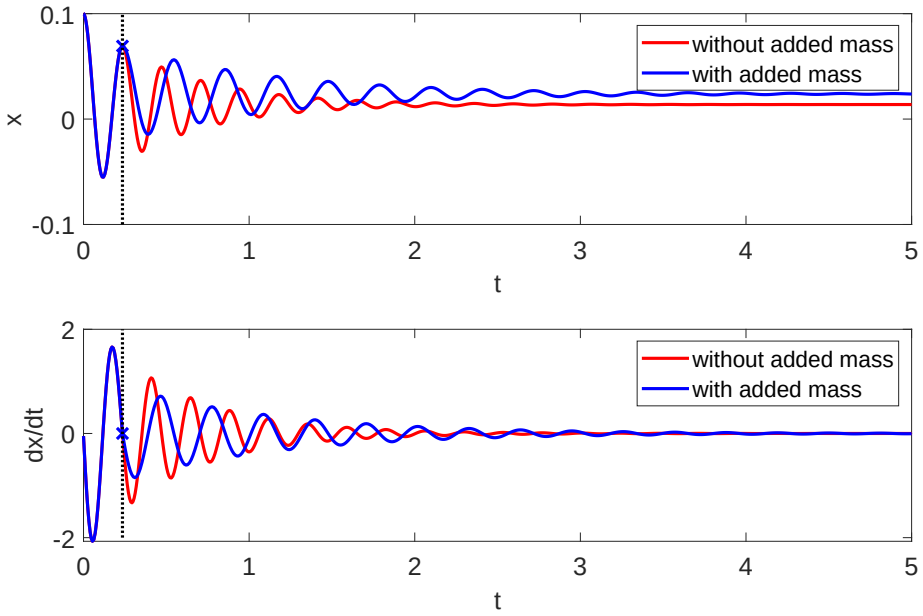
```
function [value,terminal,direction] = springEvent(t,y)
```

```
value=y(2);    % look for y(2)=0
direction=-1;  % with y(2) decreasing
terminal=1;    % and stop the integration

end
```

(ii) The following MATLAB script can be used to run the simulation, and plot graphs of position and velocity versus time, as required.

```
% set parameters
m1 = 0.075;
m2 = 0.055;
% and initial condition
IC = [0.1; -0.05];
% no added mass
m = m1;
[t,y] = ode45(@(t,y) rhs_spring_v2(t,y,m),[0 5],IC);
% with added mass
options = odeset('Events',@springEvent);
m = m1;
[t1,y1,te,ye,ie] = ode45(@(t,y) rhs_spring_v2(t,y,m),[0 5],IC,options);
tstar = te;      % event time
IC = y1(end,:)'; % final state vector [disp; vel]
m = m1+m2;
[t2,y2] = ode45(@(t,y) rhs_spring_v2(t,y,m),[t1(end) 5],IC);
% plot graphs
subplot(2,1,1);
plot(t,y(:,1),'r-',...
    t1,y1(:,1),'g-',t2,y2(:,1),'g-',te,ye(1),'gx');
xlabel('t'); ylabel('x(t)');
xlim([0 5]);
legend('without added mass','with added mass');
subplot(2,1,2);
plot(t,y(:,2),'r-',...
    t1,y1(:,2),'g-',t2,y2(:,2),'g-',te,ye(2),'gx');
xlabel('t'); ylabel('dx/dt(t)');
xlim([0 5]);
legend('without added mass','with added mass');
```

The results are as follows (the time `te` when the mass is added is marked with a dotted line and a cross at the position `ye(1)` and velocity ye(2)):



(iii) The simulation shows that adding the mass causes a change in the equilibrium position (the mass comes to rest at a larger value of x, i.e. lower down), and also in the frequency of motion (more rapid oscillations). The latter effect is perhaps more clearly seen in the velocity versus time graph. The mass-added system takes longer to come to rest, too.