

Background

Many systems are governed by Ordinary Differential Equations (ODEs), the oscillations of a pendulum, the trajectory of a satellite or the deflection of a string under a load, for instance.

The simplest non-trivial ODE is $\dot{y} \equiv dy/dt = ay$ where a is a constant. Its solution is $y(t) = Ce^{at}$, where C is an as-yet undetermined constant.

The harmonic oscillator equation $\ddot{x} + x = 0$ is a 2nd order ODE, because it involves a 2nd derivative. The general solution of the harmonic oscillator is $x(t) = C_1 \cos t + C_2 \sin t$, check it! Now we have 2 undetermined constants, C_1 and C_2 .

▷ This pattern holds for any order: an N th order ODE would have N undetermined constants, or more precisely, an N th order ode requires N extra pieces of information, or “conditions,” to fully determine the solution.

- If these N conditions are given at the same time, t_0 say, then we have an *initial value problem*. We know everything at t_0 and the equation tells us how things change as a function of t so we can just *march forward*.

- If some of these N conditions, $k < N$ say, are given at $t = a$ and the $N - k$ remaining conditions are given at $b > a$, then we have a *boundary value problem*. This is a harder case because we do not have enough information at $t = a$ to start marching forward. We have to solve for all the points between a and b simultaneously. You solved a boundary value problem in Problem Set 4, problem 1 which was basically a discrete version of $-\ddot{x} = f(t)$ with boundary conditions $x(0) = x(24L) = 0$. We will not consider differential boundary value problems in NEEP271.

General form of an **initial value problem**:

$$\frac{dy}{dt} = f(t, y) \quad \text{with} \quad y(t_0) = y_0 \quad (1)$$

where t is a scalar (often corresponding to time, but not necessarily) and $y(t)$ could be a vector. Note that we can write this problem in integral form:

$$y(t) = y_0 + \int_{t_0}^t f(s, y(s)) ds. \quad (2)$$

In fact the simplest ODE correspond to cases where f does not depend on y . Then the problem is just to calculate an integral of $f(t)$.

Euler’s method This is the basic idea and the simplest method one can imagine. You can see it as approximating the derivative dy/dt by the forward finite difference formula:

$$\frac{dy}{dt} \approx \frac{y(t+h) - y(t)}{h}$$

we know from PS2, problem 1, that this is only 1st order accurate, *i.e.* the error is $O(h)$. Substituting this difference approximation in (1) gives the difference equation:

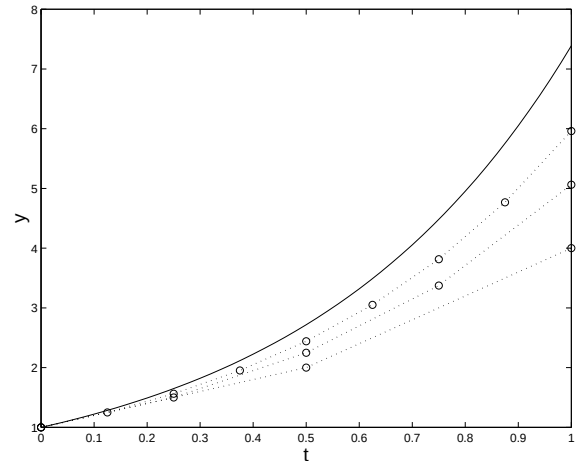
$$y(t+h) = y(t) + h f(t, y(t)).$$

This directly leads to the scheme:

$$y_{n+1} = y_n + h f(t_n, y_n) \quad \text{with} \quad y_0 \text{ given.} \quad (3)$$

You can also see this scheme as a Riemann sum approximation to (2). Euler’s method is a fine method but it is only 1st order accurate meaning that the difference between $y(t_1)$, the exact solution at t_1 , and its Euler approximation y_n is $O(h)$: $|y(t_1) - y_n| = O(h)$. Note that we must compare the solutions at the same times to make any sense, so we must have $t_1 = t_0 + nh$.

Euler's method for $\dot{y} = 2y$ with $y(0) = 1$ and $N = 2, 4, 8$.
Solid line is exact solution $y(t) = \exp(2t)$.



A general function called **ode1** with this exponential test case built-in (“**exptest**”) is posted on the lectures web page. Download it! then type `>> help ode1` or `>> type ode1` to see what it's about. To use it

```
>> t0=0; t1=1; y0=1; N=8; % sets parameters
>> [t,y]=ode1('exptest',[t0 t1],y0,N); % computes solution by Euler
>> plot(t,y,'-o')
```

You can use **ode1** for whatever problem you want, even systems of several equations. You just need to define the $f(t, y)$ you want in the format of the function **exptest** written at the bottom of the **ode1** file.

We can learn more from Euler's method. In particular consider $\dot{y} = -|a|y$ whose solution $y(t) = y_0 \exp(-|a|t)$ decays exponentially. Euler's method for this is $y_{n+1} = y_n - |a|hy_n = (1 - |a|h)y_n$. This is a simple recurrence formula that we can solve exactly for this ultra-simple $f(t, y)$. The exact solution is $y_n = (1 - |a|h)^n y_0$. So for this case we can easily prove convergence to the exact solution because with $t = nh$, $y_n = (1 - |a|t/n)^n y_0$ and we know from Calculus that

$$\lim_{n \rightarrow \infty} \left(1 - \frac{|a|t}{n}\right)^n = e^{-|a|t}.$$

So we can come as close as we want to the exact solution provided we take enough steps. Note that this ignores roundoff error, in practice there is an optimum step h (cf. PS2).

So we know that Euler works (i.e. $\lim_{n \rightarrow \infty} |y(t) - y_n| = 0$ with $t = nh$), we have said and could show using Taylor's formula that $|y(t) - y_n| = O(h)$ which is useful because we get an idea of how the error behaves with the step size h . Now from the special test case $\dot{y} = -|a|y$ whose Euler solution is $y_n = (1 - |a|h)^n y_0$, we can say something extra: if $|a|h > 2$ then the solution is totally crazy! instead of decaying exponentially as it should, it flips back and forth across zero and *grows*! This is an example of a *numerical instability*. This is a big issue for systems of equations and for partial differential equations (PDEs).

In conclusion: discrete methods like Euler introduce an extra parameter, the step size h , which should be small. “Small” is a relative term. For $\dot{y} = ay$, a is an inverse time. It is a that prescribes the time scale. So h small means $ah \ll 1$. And you better not take h such that $|a|h > 2!$...

In practice, we can get something much more accurate than Euler without too much work, in fact with much less work for a given accuracy (Note this assumes the functions are smooth!). We discussed and motivated several in class. Those were

Midpoint method: 2nd order accurate: $|y(t) - y_n| = O(h^2)$

$$\begin{aligned} y_{n+1/2} &= y_n + \frac{h}{2} f(t_n, y_n) \\ y_{n+1} &= y_n + h f(t_{n+1/2}, y_{n+1/2}) \end{aligned} \quad (4)$$

Heun's method: 2nd order accurate

$$\begin{aligned} \tilde{y}_{n+1} &= y_n + h f(t_n, y_n) \\ y_{n+1} &= y_n + \frac{h}{2} (f(t_n, y_n) + f(t_{n+1}, \tilde{y}_{n+1})) \end{aligned} \quad (5)$$

General 2nd order Runge-Kutta (RK2): with $C_1 + C_2 = 1$ and $\alpha C_2 = 1/2$

$$\begin{aligned} k_1 &= f(t_n, y_n), \quad k_2 = f(t_{n+\alpha}, y_n + \alpha h k_1) \\ y_{n+1} &= y_n + h (C_1 k_1 + C_2 k_2) \end{aligned} \quad (6)$$

4th order Runge-Kutta (RK4):

$$\begin{aligned} k_1 &= f(t_n, y_n), \quad k_2 = f(t_{n+1/2}, y_n + h k_1/2) \\ k_3 &= f(t_{n+1/2}, y_n + h k_2/2), \quad k_4 = f(t_{n+1}, y_n + h k_3) \\ y_{n+1} &= y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4). \end{aligned} \quad (7)$$

Note: when $f(t, y)$ does not depend on y : Midpoint = midpoint rule for integration, Heun's = trapezoidal rule, RK4 = Simpson's rule.

These are all one-step schemes. Once we've done one step we can easily change the step size. This is good for adaptive methods.

Matlab has two built-in ode solvers: **ode23** and **ode45**. **ode23** uses both a 2nd order and a 3rd order Runge-Kutta method and compares the two solutions at each step to determine whether the steps is not too big. **ode45** uses 4th and 5th order RK methods. So both **ode23** and **ode45** are adaptive. They are both state-of-the-art packages, efficient, accurate and with lots of built-in bells and whistles.

Higher order IVP: Any higher order IVP can be reduced to a first order system.

Example: $d^3x/dt^3 + 2d^2x/dt^2 + 3dx/dt + 5x = g(t)$, let $x = y_1$, $dx/dt = y_2$, $d^2x/dt^2 = y_3$ then

$$\dot{y}_1 = y_2, \quad \dot{y}_2 = y_3, \quad \dot{y}_3 = -2y_3 - 3y_2 - 5y_1 + g(t),$$

which is a first order system for the vector $y(t) = [y_1; y_2; y_3]$. See the **orbit** function on the "lectures" web page.