Module 4.1: Euler's Method

A NumFys Module

http://www.numfys.net

NTNU – Trondheim
Norwegian University of
Science and Technology

Fall 2012

p.1 Matlab Module

First Step:
Save the corresponding Matlab codes onto your computer and open them in Matlab.

p.2 Euler's Method

Question:
How can we solve a first-order differential equation of the form

$$\frac{d}{dt}x(t) = g(x(t), t), \tag{1}$$

with the initial condition $x(t_0) = x_0$, if we cannot solve it analytically?

Example 1:
We want to solve the ordinary differential equation (ODE)

$$\frac{d}{dt}x(t) = \cos(x(t)) + \sin(t) \tag{2}$$

with $x(0) = 0$, i.e. we need to find the right function $x(t)$ that fulfills the ODE and the initial condition (IC).

p.3 Euler's Method

Given the initial condition $x(0) = 0$, we want to know $x(t)$ for $t > 0$. We will now find an approximate numerical solution of the exact solution by computing the values of the function only at discrete values of t.

To do so, we define a discrete set of t-values, called grid points, by

$$t_n = t_0 + n * h \quad \text{with} \quad n = 0, 1, 2, 3, ..., N. \tag{3}$$

The distance between two adjacent grid points is h. The largest value is $t_N = t_0 + N * h$. Depending on the problem, t_N might be given and h is then determined by how many grid points N we choose

$$h = \frac{t_N - t_0}{N - 1}. \tag{4}$$

p.4 Euler's Method

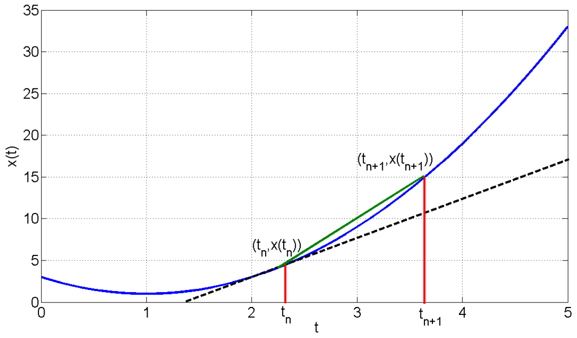
The key is now to approximate the derivative of $x(t)$ at a point t_n by

$$\left.\frac{dx}{dt}\right|_{t=t_n} \approx \frac{x(t_{n+1}) - x(t_n)}{h}, \quad h > 0. \tag{5}$$

We know that this relation is exact in the limit $h \to 0$, since $x(t)$ is differentiable according to equation (2). For $h > 0$, however, equation (5) is only an approximation that takes into account the current value of $x(t)$ and the value at the next (forward) grid point. Hence, this method is called a forward difference approximation.

p.5 Euler's Method

In equation (5), we approximate the slope of the tangent line at t_n ("the derivative") by the slope of the chord that connects the point $(t_n, x(t_n))$ with the point $(t_{n+1}, x(t_{n+1}))$. This is illustrated in this figure: blue - graph; dotted - tangent line; green - chord.



p.6 Euler's Method

Substituting the approximation (5) into (2), we obtain

$$\frac{x(t_{n+1}) - x(t_n)}{h} \approx \cos(x(t_n)) + \sin(t_n). \tag{6}$$

Rearranging the equation, using the notation $x_n = x(t_n)$ and writing this as an equality (rather than an approximation) yields

$$x_{n+1} = x_n + h\left[\cos(x_n) + \sin(t_n)\right]. \tag{7}$$

This describes an iterative method to compute the values of the function successively at all grid points t_n (with $t_n > 0$), starting at $t_0 = 0$ and $x_0 = 0$ in our case. This is called Euler's method.

p.7 Euler's Method

For example, the value of x at the next grid point, $t_1 = h$, after the starting point is

$$\begin{aligned} x_1 &= x_0 + h\left[\cos(x_0) + \sin(t_0)\right] \quad (8) \\ &= 0 + h\left[\cos(0) + \sin(0)\right] \quad (9) \\ &= h. \quad (10) \end{aligned}$$

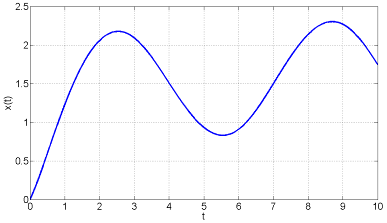
Similarly, we find at $t_2 = 2h$

$$\begin{aligned} x_2 &= x_1 + h\left[\cos(x_1) + \sin(t_1)\right] \quad (11) \\ &= h + h\left[\cos(h) + \sin(h)\right]. \quad (12) \end{aligned}$$

It is now a matter of what value to choose for h.

p.8 Euler's Method

In the corresponding Matlab code, we choose $h = 0.001$ and $N = 10000$, and so $t_N = 10$. Here is a plot of $x(t)$, where the discrete points have been connected by straight lines.



Run the code yourself!
What happens to x_N when we decrease h by a factor of 10? (Remember to increase N simultaneously by a factor of 10 so as to obtain the same value for t_N.)

p.9 Euler's Method

Accuracy:
We see that the value of x_N depends on the step size h. In theory, a higher accuracy of the numerical solution in comparison to the exact solution can be achieved by decreasing h since our approximation of the derivative $\frac{d}{dt}x(t)$ becomes more accurate.

However, we cannot decrease h indefinitely since, eventually, we are hitting the limits set by the machine precision. Also, lowering h requires more steps and, hence, more computational time.

p.10 Euler's Method

For Euler's method, it turns out that the **global error (error at a given t) is proportional to the step size h** while the **local error (error per step) is proportional to h^2**. This is called a first-order method.

p.11 Euler's Method

We can now summarize Euler's method.

Given the ODE

$$\frac{d}{dt}x(t) = g(x(t),t) \quad \text{with} \quad x(t_0) = x_0, \tag{13}$$

we can approximate the solution numerically in the following way:

1. Choose a step size h.
2. Define grid points: $t_n = t_0 + n * h \ \text{with} \ n = 0,1,2,3,...,N$.
3. Compute iteratively the values of the function at these grid points: $x_{n+1} = x_n + h * g(x_n, t_n)$. Start with $n = 0$.

p.12 Euler's Method

Instability:
Apart from its fairly poor accuracy, the main problem with Euler's method is that it can be unstable, i.e. the numerical solution can start to deviate from the exact solution in dramatic ways. Usually, this happens when the numerical solution grows large in magnitude while the exact solution remains small.

A popular example to demonstrate this feature is the ODE

$$\frac{dx}{dt} = -x \quad \text{with} \quad x(0) = 1. \tag{14}$$

The exact solution is simply $x(t) = e^{-t}$. It fulfills the ODE and the initial condition.

p.13 Euler's Method

On the other hand, our Euler method reads

$$x_{n+1} = x_n + h * (-x_n) = (1 - h)x_n. \tag{15}$$

Clearly, if $h > 1$, $x(t_n)$ will oscillate between negative and positive numbers and grow without bounds in magnitude as t_n increases. We know that this is incorrect since we know the exact solution in this case.

On the other hand, when $0 < h < 1$, the numerical solution approaches zero as t_n increases, reflecting the behavior of the exact solution.

Therefore, we need to make sure that the step size of the Euler method is sufficiently small so as to avoid such instabilities.

p.14 Euler’s Method

Second-order ODEs:
We will now demonstrate how Euler’s method can be applied to second-order ODEs.

In physics, we often need to solve Newton’s law which relates the change in momentum of an object to the forces acting upon it. Assuming constant mass, it usually has the form

$$m\frac{d^2}{dt^2}x(t) = F(v(t), x(t), t), \tag{16}$$

where we restrict our analysis to one dimension. (The following ideas can be extended to two and three dimensions in a straightforward manner.)

p.15 Euler's Method

Dividing by the mass, we find

$$\frac{d^2}{dt^2}x(t) = G(v(t), x(t), t), \tag{17}$$

with $G(v, x, t) := F(v, x, t)/m$. We can re-write this second-order ODE as two coupled, first-order ODEs. By definition, we have $v(t) = \frac{d}{dt}x(t)$. Hence, we obtain

$$\frac{dx}{dt} = v, \tag{18}$$

$$\frac{dv}{dt} = G(v, x, t). \tag{19}$$

Now, we only need to specify the initial conditions $x_0 = x(t_0)$ and $v_0 = v(t_0)$ to have a well-defined problem.

p.16 Euler's Method

Using the same discretization of time as previously, we can apply the ideas of Euler's method also to this first-order system. It yields

$$x_{n+1} = x_n + h * v_n, \tag{20}$$

$$v_{n+1} = v_n + h * G(v_n, x_n, t_n), \tag{21}$$

where (at $n = 0$) x_0 and v_0 are the initial conditions at $t = t_0$.

p.17 Euler's Method

Example 2:
Let us consider a particle of mass m that is in free fall towards the center of a planet of mass M. Let us assume that the atmosphere exerts a force

$$F_{drag} = Dv^2 \tag{22}$$

onto the particle which is proportional to the square of the velocity. Here, D is the drag coefficient. Note that the x-axis is pointing away from the planet. Hence, we only consider $v \leq 0$.

The particle motion is described by the following governing equation (G: gravitational constant)

$$m\frac{d^2x}{dt^2} = Dv^2 - \frac{GmM}{x^2}. \tag{23}$$

p.18 Euler's Method

Dividing each side by m gives

$$\frac{d^2x}{dt^2} = \frac{D}{m}v^2 - \frac{GM}{x^2}. \tag{24}$$

Following our recipe above, we re-cast this as two first-order ODEs

$$\frac{dx}{dt} = v, \tag{25}$$

$$\frac{dv}{dt} = \frac{D}{m}v^2 - \frac{GM}{x^2}. \tag{26}$$

We choose $D = 0.0025\,\mathrm{kg\,m^{-1}}$, $m = 1\,\mathrm{kg}$ and $M = M_{Earth}$, i.e. the mass of the Earth.

p.19 Euler's Method

Accordingly, our algorithm now reads

$$x_{n+1} = x_n + h * v_n, \tag{27}$$

$$v_{n+1} = v_n + h * \left[\frac{D}{m} v_n^2 - \frac{GM}{x_n^2} \right]. \tag{28}$$

Let us specify the following initial conditions and step size:

$$t_0 = 0, \;\; x(t_0) = x_0 = 7000.0\,\mathrm{km}, \;\; v(t_0) = v_0 = 0\,\mathrm{m/s}, \;\; h = 0.001\,\mathrm{s}. \tag{29}$$

We could now iterate the above equations until the particle hits the ground, i.e. until $x = R_{Earth}$, where R_{Earth} is the radius of Earth. This occurs in finite time both in reality and in our code.

p.20 Euler's Method

Moreover, the particle would also reach $x = 0$ in finite time, given the above equations, while the speed grows to infinity. However, the code would crash well before x approaches zero due to the speed reaching very large values.

Note: The governing equation actually changes when $|x| < R$.

p.21 Euler’s Method

Therefore, we need to be careful with our numerical solution procedure. This goes to show that it is **often very useful to understand the physical problem under consideration when solving its governing equations numerically.**

Let us integrate until $t_N = 100\,\mathrm{s}$, equivalent to $N = 10^5$ time steps. As it turns out, the particle does not reach the ground by $t = t_N$.

Again, run the corresponding Matlab code yourself!
What do you observe?
What happens when you choose $x_0 = 10000.0\,\mathrm{km}$ as your initial height?

p.22 Euler’s Method

Summary:
We have learned how to use Euler’s method to solve first-order and second-order ODEs numerically. It is the simplest method in this context and very easy to implement.
However:

1. There are more precise methods for solving ODEs.
2. There are more stable methods for solving ODEs.

Notwithstanding these issues, Euler’s method is often useful: it is easy to use (the coding is less error prone); it can provide a helpful first impression of the solution; modern-day computer power makes computational expense less of an issue.
Simply put, sometimes it is sufficient.