MAS212 Scientific Computing and Simulation

Dr. Sam Dolan

School of Mathematics and Statistics,
University of Sheffield

Autumn 2017

http://sam-dolan.staff.shef.ac.uk/mas212/

G18 Hicks Building
s.dolan@sheffield.ac.uk

Today's lecture

- Numerical methods for solving ODEs
- Methods:
  1. Euler's method
  2. The midpoint method
  3. Runge-Kutta methods
- Concepts:
  - **Truncation error** (local and global)
  - **Order** of accuracy
  - **Stability** and **convergence**

Euler's method

- Suppose we have some first-order ODE describing an initial value problem:

$$\boxed{\frac{dx}{dt} = f(x,t) \qquad\qquad x(t_0) = x_0}$$

- **How** do we solve the ODE numerically?

How do we …

… find a sequence of numerical estimates $[x_0, x_1, x_2, \ldots x_n]$ at times $[t_0, t_1, t_2 \ldots t_n]$, where $t_0 < t_1 < t_2 < \ldots$, that are reasonable estimates of the exact sequence $[x(t_0), x(t_1), x(t_2), \ldots, x(t_n)]$ where $x(t)$ is the **exact** solution of the ODE (which may be unknown).

- We take small steps …

Euler's method

$$\boxed{\frac{dx}{dt} = f(x,t) \qquad\qquad x(t_0) = x_0}$$

- Suppose we know $x(t_k)$ and we want to estimate $x(t_{k+1})$, where $t_{k+1} = t_k + h$ and h is small : $0 < h \ll 1/f'$.
- Assume $x(t)$ is locally 'smooth enough' in the vicinity of $t = t_k$ for a **Taylor series** expansion:

$$x(t_{k+1}) = x(t_k+h) = x(t_k)+h\,\left.\frac{dx}{dt}\right|_{t_k} + \frac{1}{2}h^2\,\left.\frac{d^2x}{dt^2}\right|_{t_k} + \ldots + \frac{1}{m!}h^m\,\left.\frac{d^mx}{dt^m}\right|_{t_k} + \ldots$$

- **Euler's method**: Truncate the Taylor series after just **two** terms.

$$\begin{aligned} x(t_{k+1}) &\approx x(t_k) + h\,\left.\frac{dx}{dt}\right|_{t_k} \\ &\approx x(t_k) + h\,f\,(x(t_k), t_k)\,. \end{aligned}$$

$$\Rightarrow \quad x_{k+1} = x_k + hf\,(x_k, t_k)$$

- i.e. use the derivative function f evaluated at the initial point to take one small step forward to $t_{k+1} = t_k + h$.

$$\boxed{\frac{dx}{dt} = f(x,t) \qquad\qquad x(t_0) = x_0}$$

Euler's method

$$\boxed{x_{k+1} = x_k + h\,f\,(x_k, t_k)}$$

Algorithm:

- Divide the domain $[t_0, t_f]$ into n equally-spaced intervals:

$$t_k = t_0 + kh, \qquad k \in \{0, 1, \ldots, n\}, \qquad h \equiv \frac{t_f - t_0}{n}.$$

- Start with the initial condition x_0 at t_0
- Apply $x_{k+1} = x_k + h\,f\,(x_k, t_k)$ **once** to get x_1 from x_0
- Now iterate (repeat) ...

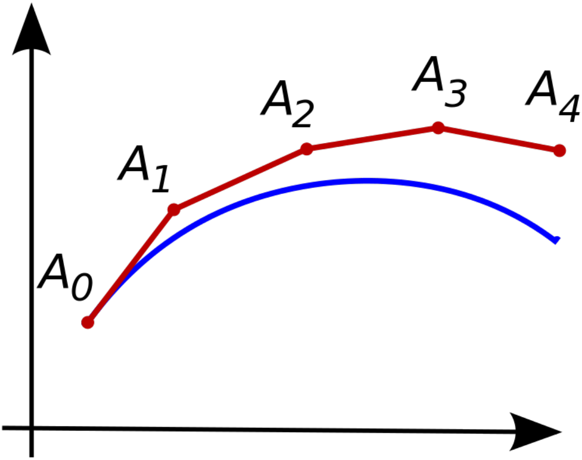


Image from:
http://en.wikipedia.org/wiki/Euler_method

Example

Solve the initial value problem $\dfrac{dx}{dt} = x, \quad x(0) = 1,$
on the domain $t \in [0, 4]$ analytically and numerically

(a) Using separation of variables to find the **exact** solution

(b) Using Euler's method to find a **numerical** solution

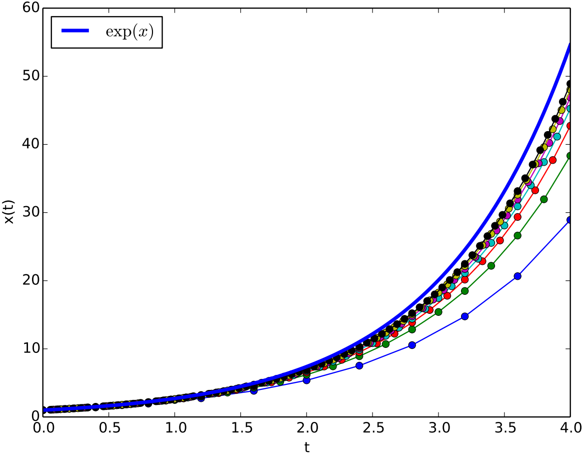
(a)

$$
\begin{aligned}
\frac{dx}{dt} &= x \\
\Rightarrow \quad \int \frac{dx}{x} &= \int dt \\
\Rightarrow \quad \ln|x| &= t + c \\
\Rightarrow \quad x(t) &= Ae^{t} \qquad x(0) = 1 \quad \Rightarrow A = 1
\end{aligned}
$$

$$\boxed{x(t) = e^{t}}$$

(b) Euler's method

- The sequence of numerical estimates $x_0, x_1 \ldots, x_n$ depends on the step size h.



- As $h \to 0$, the numerical solution converges towards the exact solution ... **but slowly!**

(b) Euler's method (example code)

```python
def Euler(n=10, tstart=0.0, tend=4.0):
    """ Apply Euler's method to ODE:  dx/dt = x.
     n is the number of intervals in the domain [tstart, tend]."""
    ts = np.linspace(tstart, tend, n+1)
    h = (tend - tstart) / n
    xs = np.zeros(n+1)
    xs[0] = 1.0    # initial condition
    for k in range(n):
        xs[k+1] = xs[k] + h*xs[k]
    return ts, xs
```

(b) Euler's method (example code)

```
# Plot several curves, for various "n"
plt.xlabel('t'); plt.ylabel('x(t)')
for k in range(1,8):
    ts, xs = Euler(n = 10*k) # use 10, 20, ... 80 intervals
    plt.plot(ts, xs, '-o')
# Plot also the exact solution
ts = np.linspace(0.0, 4.0, 100)
xs = np.exp(ts)
plt.plot(ts, xs, lw=3, label='$\exp(x)$')
plt.legend(loc='upper left')
plt.show()
```

Error

- For $h > 0$ the Euler method gives an approximation of the exact solution $x(t)$
- i.e. a sequence of estimates x_k at times t_k that (we hope) converge to $x(t_k)$ as $h \to 0$
- How **accurate** are these estimates?
- Define **error** ϵ_k as difference between exact value $\bar{x}(t_k)$ and numerical estimate x_k:

$$\epsilon_k \equiv x_k - \bar{x}(t_k)$$

- Clearly, ϵ_k depends on grid point k and grid spacing h.
- Most often, the exact solution $\bar{x}(t)$ is **unknown**, but nevertheless we need to estimate the error.

Local truncation error

- In deriving Euler's method we **truncated** the Taylor series
- We may keep track of the order of the neglected terms using big-O notation.
- Let's take care to distinguish between

x_k : the numerical estimate at $t = t_k$
$\bar{x}(t_k)$: the exact solution at $t = t_k$

$$
\begin{aligned}
x_{k+1} &= x_k + h\,f(x_k, t_k) \\
\Rightarrow\; x_{k+1} - \bar{x}(t_{k+1}) &= x_k + h\,f(x_k, t_k) - \bar{x}(t_{k+1}) \\
\Rightarrow\; \epsilon_{k+1} &= x_k + h\,f(x_k, t_k) - \left(\bar{x}(t_k) + h\; f(x,t)\Big|_{t_k} + \mathcal{O}(h^2)\right) \\
&= x_k - \bar{x}(t_k) + h\{f(x_k, t_k) - f(\bar{x}(t_k), t_k)\} + \mathcal{O}(h^2) \\
\Rightarrow\; \epsilon_{k+1} &= \epsilon_k + h\{f(x_k, t_k) - f(\bar{x}(t_k), t_k)\} + \mathcal{O}(h^2)
\end{aligned}
$$

- If we assume perfect accuracy at previous step ($x_k = \bar{x}(t_k) \Rightarrow \epsilon_k = 0$) then the first two terms vanish, and

$$\epsilon_{k+1} = \mathcal{O}(h^2)$$

- We say: The **local truncation error** is $\mathcal{O}(h^2)$, or 2nd order.

Global truncation error

$$\epsilon_{k+1} = \epsilon_k + h\{f(x_k, t_k) - f(\bar{x}(t_k), t_k)\} + \mathcal{O}(h^2)$$

- Consider the middle term more carefully:

$$\begin{aligned}
f(x_k, t_k) - f(\bar{x}(t_k), t_k) &= f(x_k, t_k) - f(x_k - \epsilon_k, t_k) \\
&= f(x_k, t_k) - \left(f(x_k, t_k) - \epsilon_k \left.\frac{\partial f}{\partial x}\right|_{x_k, t_k} + \mathcal{O}\left(\epsilon_k^2\right) \right) \\
&= \epsilon_k \left.\frac{\partial f}{\partial x}\right|_{x_k, t_k} + \mathcal{O}\left(\epsilon_k^2\right)
\end{aligned}$$

- As ϵ_k is $\mathcal{O}(h)$ or higher, it follows that the middle term is at $\mathcal{O}(h^2)$ or higher, and so

$$\epsilon_{k+1} = \epsilon_k + \mathcal{O}\left(h^2\right)$$

- What is the error after n steps of Euler method?

$$\epsilon_n = n \times \mathcal{O}\left(h^2\right) \qquad (\text{as} \quad \epsilon_0 = 0)\,.$$

- Total number of intervals n on fixed domain is inversely proportional to h

$$n = \frac{t_f - t_0}{h} \quad \Rightarrow \quad \boxed{\epsilon_n = \mathcal{O}\left(h\right)}$$

For the Euler method:

- The **local** truncation error (LTE) is $\mathcal{O}(h^2)$ i.e. 2nd order
- The **global** truncation error (GTE) is $\mathcal{O}(h)$ i.e. 1st order
- We say that Euler's method is a **first-order method**.
- This 'explains' why the convergence was slow.
- Much better methods are available!

Stability

- There is a more insidious problem than slow convergence
- ... **numerical instability** !

What is a numerical instability?

Generally, a spurious feature in a numerical solution, not present in the exact solution, that grows with time and dominates over the real, physical solution.

Exponentially-growing instabilities are the most common type.

- The Euler method is **unstable** in two cases
  - (i) $\frac{\partial f}{\partial x} > 0$
  - (ii) $h > 2/\left|\frac{\partial f}{\partial x}\right|$
- In case (i), the exponential growth of the instability is usually obscured by the growth of the physical solution.
- In case (ii), the instability is readily apparent.

Stability

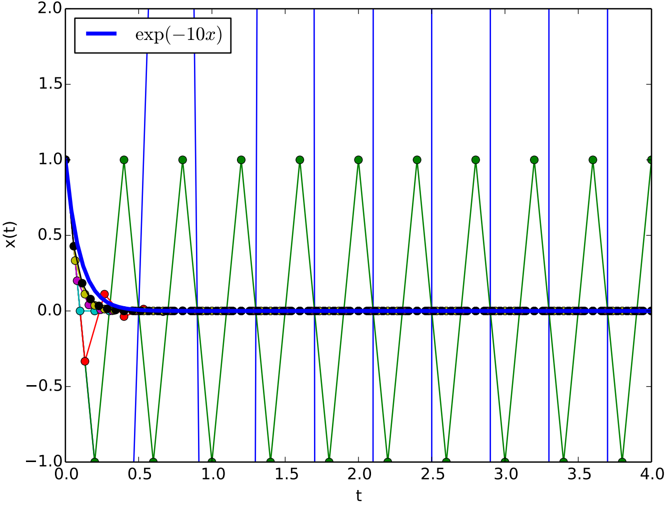
Example:

Solve the initial value problem $\dfrac{dx}{dt} = -10x, \quad x(0) = 1,$

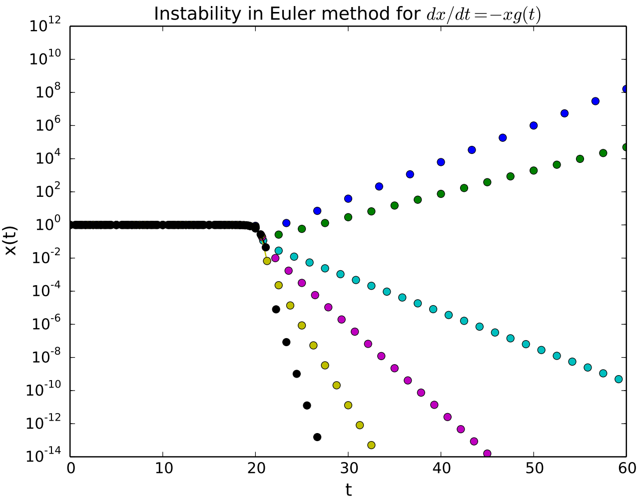
for various step sizes h

```
tmax = 4.0
def Euler(n=10):
    ts = np.linspace(0.0, tmax, n+1)
    h = tmax / n
    xs = np.zeros(n+1)
    xs[0] = 1.0     # initial condition
    for k in range(n):
        xs[k+1] = xs[k] - 10.0*h*xs[k]
    return ts, xs

plt.xlabel('t'); plt.ylabel('x(t)')
plt.ylim(-1,2)
for k in range(1,8):
    ts, xs = Euler(n = 10*k)
    plt.plot(ts, xs, '-o')
ts = np.linspace(0.0, tmax, 100)
xs = np.exp(-10.0*ts)
plt.plot(ts, xs, lw=3)
plt.legend(loc='upper left')
plt.show()
```



- For large intervals *h*, the numerical solution blows up (and oscillates)



- Another example: $\dfrac{dx}{dt} = -xg(t)$ where $g(t) = 1 + \tanh(t - 20)$.
- Note logarithmic scale on y-axis
- All OK ... until gradient becomes large, around $t = 20$.

Stability of Euler method

$$x_{k+1} = x_k + h\, f\,(x_k, t_k) \qquad\qquad (^*)$$

- Suppose that x_k differs from a solution to the *difference equation* $(^*)$ by small amount, δx_k.
- δx_k could be due to finite accuracy of the computer (i.e. rounding error $\sim 10^{-14}$)
- What happens to this error at the next step?

$$\begin{aligned} x_{k+1} + \delta x_{k+1} &= x_k + \delta x_k + hf\,(x_k + \delta x_k, t_k) \\ &= x_k + \delta x_k + h\left(f\,(x_k, t_k) + \delta x_k \frac{\partial f}{\partial x} + \dots \right) \end{aligned}$$

- Subtracting the difference equation $(^*)$,

$$\begin{aligned} \delta x_{k+1} &\approx \delta x_k + h\,\delta x_k \frac{\partial f}{\partial x} \\ &\approx g\,\delta x_k \qquad \text{where} \quad g \equiv 1 + h\frac{\partial f}{\partial x}. \end{aligned}$$

- If $|g| > 1$ then δx_k will **grow exponentially** with k

Stability of Euler method

- We found that the error δx_k obeys its own difference equation:

$$\delta x_{k+1} \approx g\,\delta x_k \qquad \text{where} \quad g \equiv 1 + h\frac{\partial f}{\partial x}.$$

- If $|g| > 1$ then δx_k will **grow exponentially** with k
- Instability in two cases:

  (i) $g > 1 \quad \Leftrightarrow \quad \frac{\partial f}{\partial x} > 0$

  (ii) $g < -1 \quad \Leftrightarrow \quad h > 2/\left|\frac{\partial f}{\partial x}\right|$

- $\Rightarrow$ Euler method is badly flawed and **should not be used!**

The midpoint method for $\frac{dx}{dt} = f(x,t)$

- One drawback of the Euler method is that it is **asymmetrical**
- It uses the derivative at the **start** of the interval.
- Better to use derivative at the **centre** of the interval ...
- ...i.e. at $t_{k+1/2} = t_k + \frac{1}{2}h$
- ...but how do we find $x_{k+1/2}$ in the centre?
- **Q.** How to estimate x at midpoint? **A.** Use an Euler step :

$$\begin{aligned} x_{k+1/2} &= x_k + \tfrac{1}{2}h\,f(x_k, t_k) \\ x_{k+1} &= x_k + h\,f(x_{k+1/2}, t_{k+1/2}). \end{aligned}$$

- Or, written in one line,

$$\boxed{x_{k+1} = x_k + h\,f\left(x_k + \tfrac{1}{2}h\,f(x_k, t_k),\ t_k + \tfrac{1}{2}h\right)}$$

Example

Use the midpoint method

$$x_{k+1} = x_k + h\,f\left(x_k + \tfrac{1}{2}h\,f(x_k, t_k),\ t_k + \tfrac{1}{2}h\right)$$

to solve the equation of simple harmonic motion

$$\ddot{x} + x = 0, \qquad x(0) = 1, \quad \dot{x}(0) = 0$$

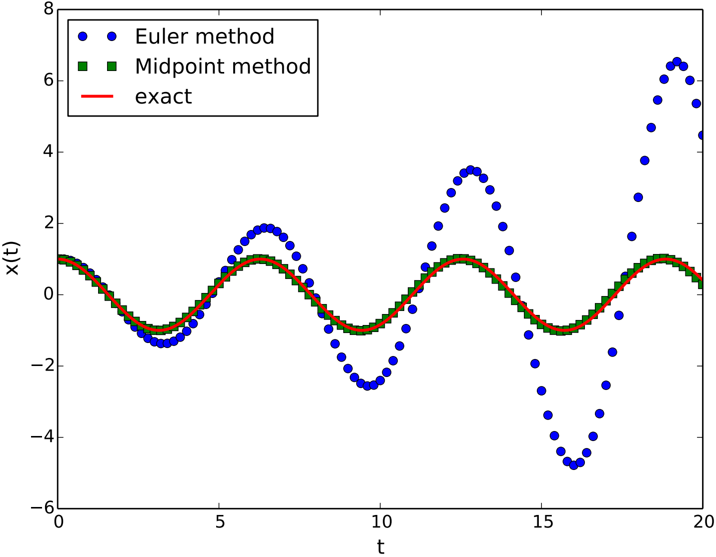
- The second-order SHM equation is equivalent to **two** first-order ODEs:

$$\dot{x} = y, \quad \dot{y} = -x, \quad x(0) = 1, \quad y(0) = 0.$$

- We may apply the method with any number of dependent variables $x, y, \ldots$.

Code example:

```
def midpoint(tmin=0.0, tmax=20.0, n=100):
    """The midpoint method applied to the SHM equation.
     dx/dt = y,   dy/dt = -x"""
    ts = np.linspace(tmin, tmax, n+1)
    h = (tmax - tmin)/n
    xs = np.zeros(n+1); ys = np.zeros(n+1)
    xs[0] = 1.0; ys[0] = 0.0
    for k in range(n):
        x1 = xs[k] + 0.5*h*ys[k]  # midpoint estimate
        y1 = ys[k] - 0.5*h*xs[k]
        xs[k+1] = xs[k] + h*y1
        ys[k+1] = ys[k] - h*x1
    return ts, xs, ys
```


Euler method
Midpoint method
exact
x(t)
t
8
6
4
2
0
−2
−4
−6
0
5
10
15
20

Exercise

Show that the midpoint method is 2nd-order accurate.

That is: show that the global truncation error is $\mathcal{O}(h^2)$.

Exercise

Determine whether the midpoint method is stable.

Runge-Kutta methods

Explicit Runge-Kutta methods

Recurrence relation:

$$x_{j+1} = x_j + \sum_{i=1} b_i k_i$$

where

$$\begin{aligned}
k_1 &= h\,f(x_j, t_j) \\
k_2 &= h\,f(x_j + a_{21}k_1, t_j + c_2 h) \\
k_3 &= h\,f(x_j + a_{31}k_1 + a_{32}k_2, t_j + c_3 h) \\
&\dots \\
k_s &= h\,f(x_j + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1}, t_j + c_s h)
\end{aligned}$$

- The midpoint method is a member of a **family of methods** that use intermediate estimates in a systematic way.
- s is the **order** of the method
- A method is specified by the coefficients b_i, c_i and a_{ij}.

Runge-Kutta methods

- A method is specified by the coefficients b_i, c_i and a_{ij}.
- Butcher tableau are used to give these coefficients:

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\dots & \dots & \dots & \dots & & \\
c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \dots & b_{s-1} & b_s
\end{array}
$$

- **Example:** Midpoint method

$$
\begin{array}{c|cc}
0 & & \\
1/2 & 1/2 & \\
\hline
 & 0 & 1
\end{array}
$$

Runge-Kutta methods

- The most well-used version is the **classical Runge-Kutta method** or **RK4 method** :

$$
\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

- This is a fourth-order accurate method.
- `odeint` implements this method.