Explicit methods for solving ODEs: review

- Last time we considered the simplest approach to solving ordinary differential equations (ODEs) – stepping forward in time using finite-differencing of the derivatives.
- The lowest (1^{st}) order approximation is Euler's method – simple to implement, but too low precision to be used in practice.
- Higher order methods (e.g. Runge-Kutta of order 4, RK4) are much better and are widely used as 'workhorse' methods. For error control the RK45 (and similar) methods are used
- This class of methods are referred to as explicit methods. Often they are all we need, but there are classes of problems for which they fail → stability.

Stability of ODEs

- One issue with explicit methods is that they are not particularly stable.
- As an illustration of what this means, consider the equation:

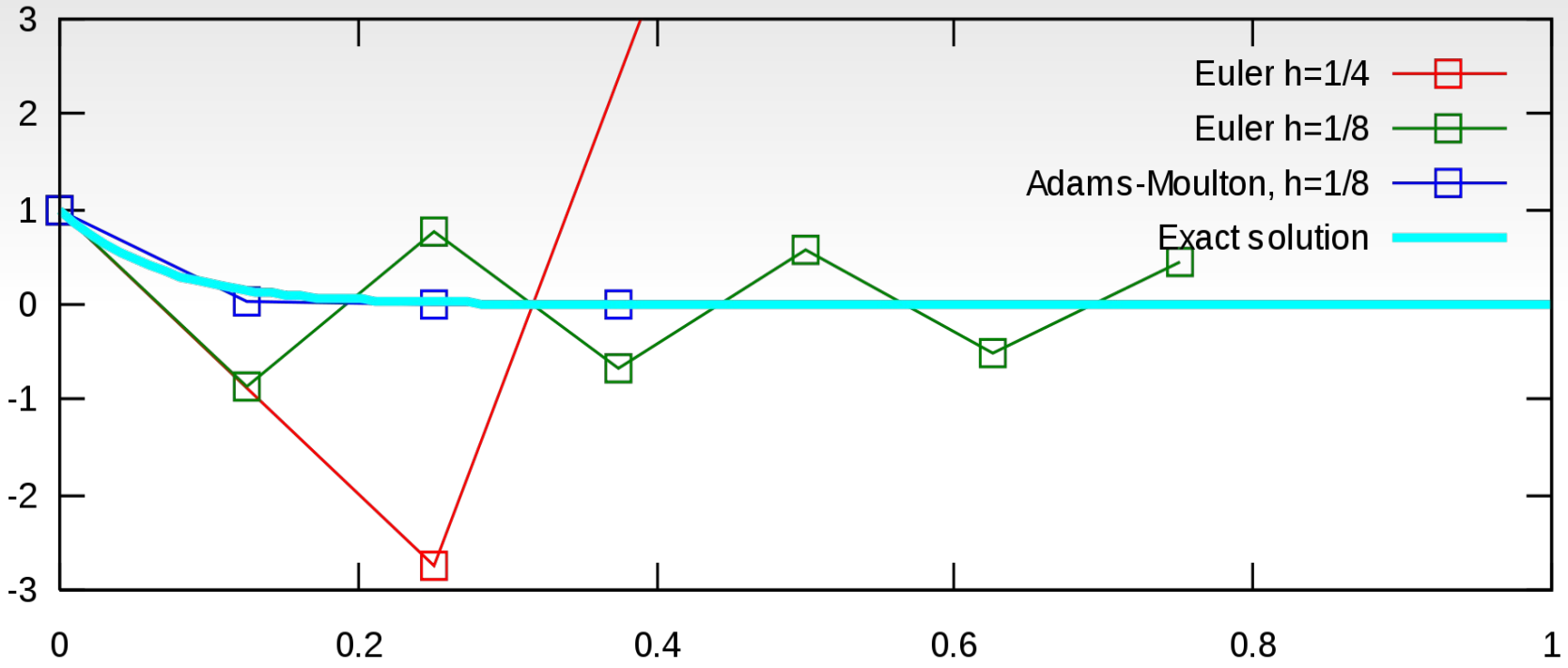
$$y' = -\lambda y \qquad y(0) = \beta$$

- Its solution is clearly $y(x) = \beta e^{-\lambda x}$
- Solving this by Euler's method, we find:

$$y(x+h) = y(x) + hy'(x) = (1 - \lambda h)\, y(x)$$

- Thus, if $|1 - \lambda h| > 1$ this method is unstable (why?).

Stability of ODEs: illustration



Clearly, when the stability condition is violated the Euler method is unusable → need either small steps (h), or an implicit method.

Implicit methods: backwards Euler method

Consider again the equation y'=f(x,y). Instead of approximating the derivative with forward finite difference, as in standard Euler method, we can use the backward one:

$$y' \approx \frac{y(x) - y(x-h)}{h} = f(x,y)$$

$$y_1 \approx y(x_0) + hy'(x_1) = y_0 + hf(x_1, y_1)$$

$$y_{n+1} \approx y(x_n) + hy'(x_{n+1}) = y_n + hf(x_{n+1}, y_{n+1})$$

Implicit methods: backwards Euler method

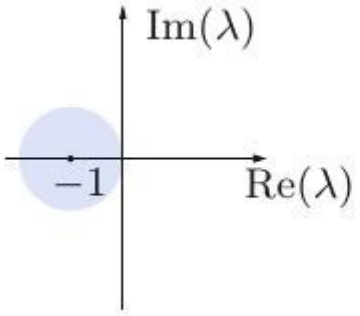
$$y_{n+1} \approx y(x_n) + hy'(x_{n+1}) = y_n + hf(x_{n+1}, y_{n+1})$$

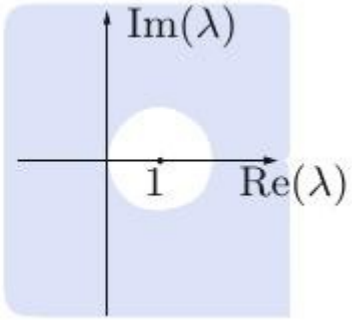
Implicit equations for the successive solution points - we need to employ an algebraic equation solver (Newton-Raphson, secant).

Why would we bother to complicate things like this, compared to the simple forward Euler method?

The implicit method's stability region is much larger:

Forward (explicit) Euler

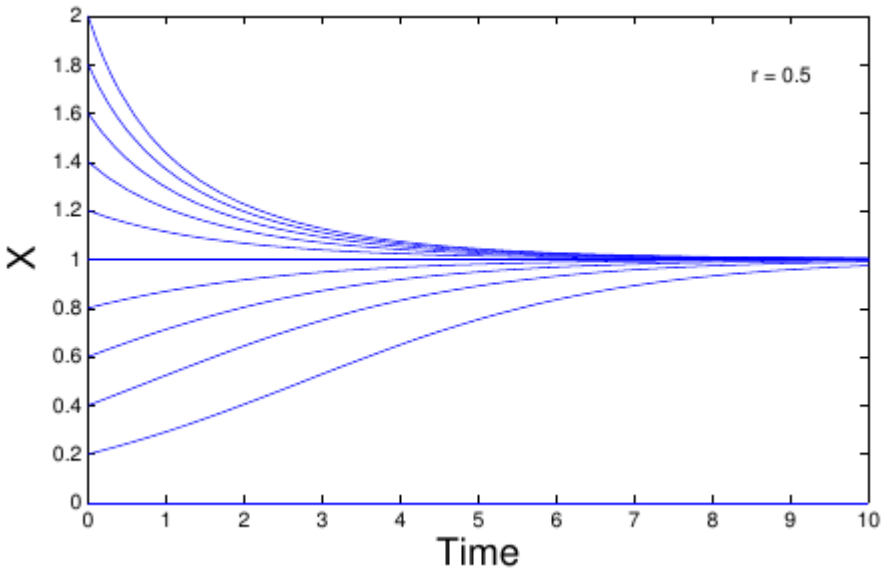


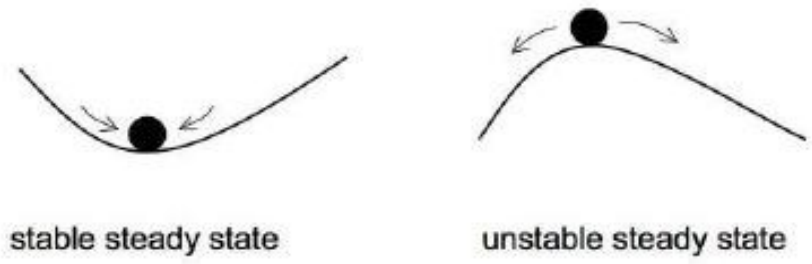


Backward (implicit) Euler

Stable vs. unstable equilibrium

- Solutions of differential equations often have equilibrium points, but some of those are stable and some are not. Example: logistics equation again: dy/dt=y(1-y)





Illustrations by D. Gonze

Implicit methods: backwards Euler method

$$y_1 \approx y(x_0) + hy'(x_1) = y_0 + hf(x_1, y_1)$$

$$y_{n+1} \approx y(x_n) + hy'(x_{n+1}) = y_n + hf(x_{n+1}, y_{n+1})$$

Precision of the forward and backward methods is similar – backwards Euler is still just a first order method. However, the solution is much more stable, even for relatively large h (which is definitely not the case for the forward Euler). Important for efficient numerical solution of stiff equations and systems.

Stiff equations

- There is no very strict mathematical definition of stiffness for differential equations. Indications are:
  - Problem contains widely differing time scales.
  - Normal solution methods work only very slowly, or not at all.
  - Step size is dictated by stability requirements rather than accuracy ones,etc.
- Example: $y'' + 1001y' + 1000y = 0$
- Solution: $\mathbf{y}(x) = \sum_i C_i \mathbf{v}_i \exp(-\lambda_i x)$

  where $\lambda_1 = 1$ and $\lambda_2 = 1000$

  Second term requires very small timestep for stability, which is in fact not needed for accuracy (why?)

Backwards Euler: implementation

```
def implicit_euler(f_fcn, x0, ts, p):
    """ implicit euler with fixed stepsizes, using Newton's method to solve
    the occurring implicit system of nonlinear equations
    """

    def F_fcn(x_new, x, t_new, t, p):
        """ implicit function to solve:  0 = F(x_new, x, t_new, t_old)"""
        return (t_new - t) * f_fcn(t_new, x_new, p) - x_new + x

    def J_fcn(x_new, x, t_new, t, p):
        """ computes the Jacobian of F_fcn
        all inputs are double arrays
        """
        y = UTPM(numpy.zeros((D,N,N)))
        y.data[0,:]   = x_new
        y.data[1,:,:] = numpy.eye(N)
        F = F_fcn(y, x, t_new, t, p)
        return F.data[1,:,:].T

    x = x0.copy()
    D,P,N = x.data.shape
    x_new = x.copy()
    x_list = [x.data.copy() ]
```

From http://packages.python.org/algopy/examples/ode_solvers.html

Backwards Euler: implementation

```
  for nts in range(ts.size-1):
      h = ts[nts+1] - ts[nts]
      x_new.data[0,...] = x.data[0,...]
      x_new.data[1:,...] = 0
      # compute the Jacobian at x
      J = J_fcn(x_new.data[0,0], x.data[0,0], ts[nts+1],ts[nts], p.data[0,0])
      # d=0: apply Newton's method to solve 0 = F_fcn(x_new, x, t_new, t)
      step = numpy.inf
      while step > 10**-10:
          delta_x = numpy.linalg.solve(J, F_fcn(x_new.data[0,0], x.data[0,0], ts[nts+1],
ts[nts], p.data[0,0]))
          x_new.data[0,0] -= delta_x
          step = numpy.linalg.norm(delta_x)
      # d>0: compute higher order coefficients
      J = J_fcn(x_new.data[0,0], x.data[0,0], ts[nts+1], ts[nts], p.data[0,0])
      for d in range(1,D):
          F = F_fcn(x_new, x, ts[nts+1], ts[nts], p)
          x_new.data[d,0] = -numpy.linalg.solve(J, F.data[d,0])
      x.data[...] = x_new.data[...]
      x_list.append(x.data.copy())
 return numpy.array(x_list)
```

Implicit methods: higher order

Also, higher order implicit methods can be constructed in a similar way, an example of such method is:

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2h}{3}f(x_{n+1}, y_{n+1})$$

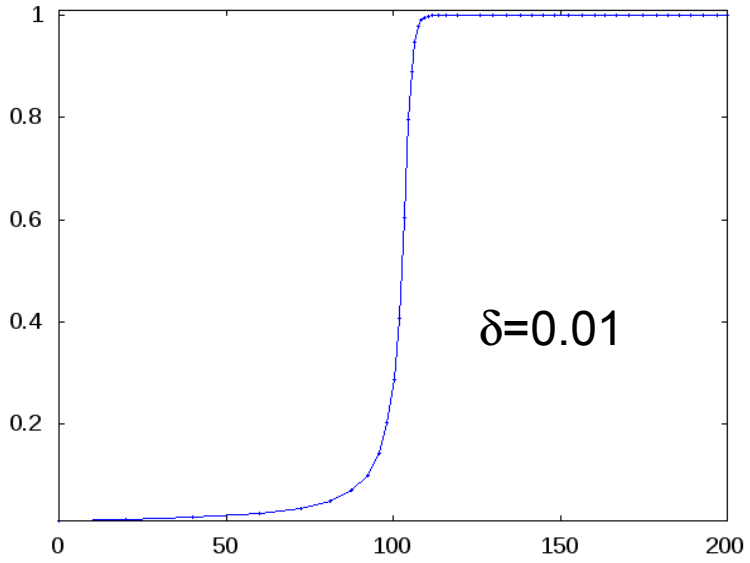
This and other implicit methods have same advantages (stability, handling of stiffness) as the backwards Euler method, but faster convergence. Implicit Runge-Kutta methods are standard 'workhorse' methods for such equations.

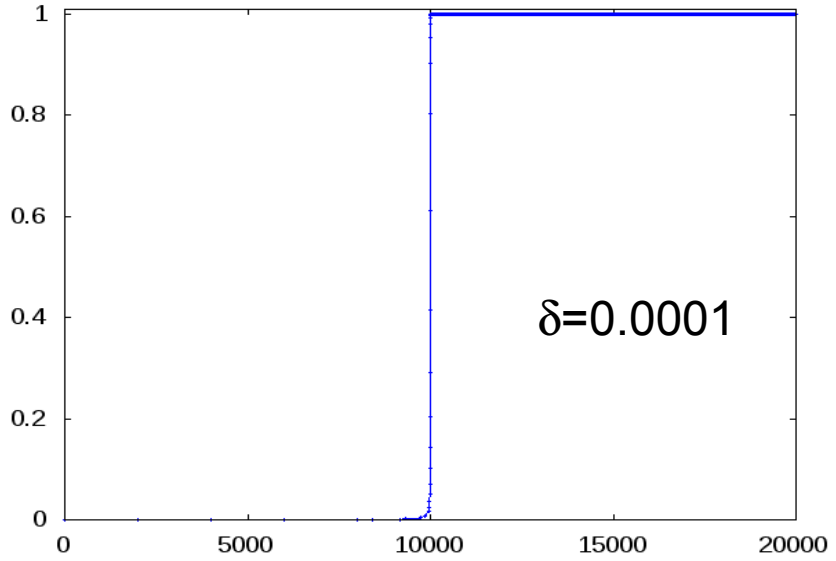
Stiff equation: example

```
def f(y,t):
    return y**2-y**3
delta=0.01
x=np.linspace(0,2./delta,100)
z=scipy.integrate.odeint(f,delta,x)
```

$$\dot{y} = y^2 - y^3,$$
$$y(0) = \delta,$$
$$0 \leq t \leq 2/\delta.$$

Model of flame propagation by L. Shampine





Python (Scipy) commands

scipy.integrate.odepack: Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems of first order ODEs:

dy/dt = func(y,t0,...)

where y can be a vector.

scipy.integrate.ode: Interface to a number of ODE solvers, e.g.

Dopri5: an explicit Runge-Kutta method of order (4)5 due to Dormand & Prince (with stepsize control).

In either of the above cases one can also specify options (like tolerances) and extra parameters to pass to differential equation function, look at help!

Remarks

Similarly to the integration routines, it is better to aim for a given tolerance, rather than a fixed number of steps N by controlling the step-size h adaptively.

Example strategy: step doubling, where difference between 2 steps with h and 1 step with 2*h is compared.

Stiff equations: multiple, very different scales in problem so that steps size needs to be ~ small scale for stability even if that scale is irrelevant for the solution – implicit methods are typically used in such cases.

Professional-grade solvers (like the ones build into Scipy) do error control and stiff/non-stiff method switching automatically.