5mm.

Programming of Differential Equations (Appendix E)

Hans Petter Langtangen

Simula Research Laboratory

University of Oslo, Dept. of Informatics

Differential equations

- A differential equation (ODE) written in generic form:

$$u'(t) = f(u(t), t)$$

  The solution of this equation is a function $u(t)$
- To obtain a unique solution $u(t)$, the ODE must have an initial condition: $u(0) = u_0$
- Different choices of $f(u, t)$ give different ODEs:

$$f(u,t) = \alpha u, \quad u' = \alpha u \quad \text{exponential growth}$$

$$f(u,t) = \alpha u\left(1 - \frac{u}{R}\right), \quad u' = \alpha u\left(1 - \frac{u}{R}\right) \quad \text{logistic growth}$$

$$f(u,t) = -b|u|u + g, \quad u' = -b|u|u + g \quad \text{body in fluid}$$

Our task: solve any ODE $u' = f(u, t)$ by programming

How to solve a general ODE numerically

- Given $u' = f(u, t)$ and $u(0) = u_0$, the Forward Euler method generates a sequence of $u_1, u_2, u_3, \ldots$ values for u at times $t_1, t_2, t_3, \ldots$:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)$$

  where $t_k = k\Delta t$
- This is a simple stepping-forward-in-time formula
- Algorithm using growing lists for u_k and t_k:
  - Create empty lists `u` and `t` to hold u_k and t_k for $k = 0, 1, 2, \ldots$
  - Set initial condition: `u[0]` = u_0
  - For $k = 0, 1, 2, \ldots, n-1$:
    `unew = u[k] + dt*f(u[k], t[k])`
    append `unew` to `u`
    append `tnew = t[k] + dt` to `t`

Implementation as a function

```
def ForwardEuler(f, dt, u0, T):
    """Solve u'=f(u,t), u(0)=u0, in steps of dt until t=T."""
    u = []; t = []   # u[k] is the solution at time t[k]
    u.append(u0)
    t.append(0)
    n = int(round(T/dt))
    for k in range(n):
        unew = u[k] + dt*f(u[k], t[k])

        u.append(unew)
        tnew = t[-1] + dt
        t.append(tnew)
    return numpy.array(u), numpy.array(t)
```

This simple function can solve any ODE (!)

Example on using the function

Mathematical problem:
Solve $u' = u$, $u(0) = 1$, for $t \in [0, 3]$, with $\Delta t = 0.1$
Basic code:

```
def f(u, t):
    return u

u0 = 1
T = 3
dt = 0.1
u, t = ForwardEuler(f, dt, u0, T)
```

Compare exact and numerical solution:

```
from scitools.std import plot, exp
u_exact = exp(t)
plot(t, u, 'r-', t, u_exact, 'b-',
     xlabel='t', ylabel='u', legend=('numerical', 'exact'),
     title="Solution of the ODE u'=u, u(0)=1")
```


A class for solving ODEs

- Instead of a function for solving any ODE we now want to make a class
- Usage of the class:

```
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, t0)
u, t = method.solve(T)
```

- Store f, Δt, and the sequences u_k, t_k as attributes
- Split the steps in the `ForwardEuler` function into three methods

The code for a class for solving ODEs (part 1)

```
class ForwardEuler:
    def __init__(self, f, dt):
        self.f, self.dt = f, dt

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process

        self.u.append(float(u0))
        self.t.append(float(t0))
        self.k = 0  # time level counter
```


The code for a class for solving ODEs (part 2)

```
class ForwardEuler:
    ...
    def solve(self, T):
        """Advance solution in time until t <= T."""
        t = 0
        while t < T:
            unew = self.advance()   # numerical formula
            self.u.append(unew)
            t = self.t[-1] + self.dt
            self.t.append(t)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)

    def advance(self):
        """Advance the solution one time step."""
        # avoid "self." to get more readable formula:
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        unew = u[k] + dt*f(u[k], t)   # Forward Euler scheme
        return unew
```

Verifying the class implementation

Mathematical problem:

$$u' = 0.2 + (u - h(t))^4, \quad u(0) = 3, \quad t \in [0, 3]$$

$$u(t) = h(t) = 0.2t + 3 \quad \text{(exact solution)}$$

The Forward Euler method will reproduce such a linear u exactly!
Code:

```
def f(u, t):
    return 0.2 + (u - h(t))**4

def h(t):
    return 0.2*t + 3

u0 = 3; dt = 0.4; T = 3
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, 0)
u, t = method.solve(T)
u_exact = h(t)
print 'Numerical: %s\nExact: %s' % (u, u_exact)
```


Using a class to hold the right-hand side $f(u,t)$

Mathematical problem:

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = u_0, \quad t \in [0, 40]$$

Class for right-hand side $f(u,t)$:

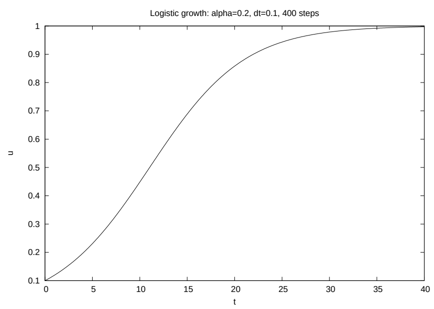
```
class Logistic:
    def __init__(self, alpha, R, u0):
        self.alpha, self.R, self.u0 = alpha, float(R), u0

    def __call__(self, u, t):   # f(u,t)
        return self.alpha*u*(1 - u/self.R)
```

Main program:

```
problem = Logistic(0.2, 1, 0.1)
T = 40;  dt = 0.1
method = ForwardEuler(problem, dt)
method.set_initial_condition(problem.u0, 0)
u, t = method.solve(T)
```


Figure of the solution



Ordinary differential equations

Mathematical problem:

$$u'(t) = f(u, t)$$

Initial condition:

$$u(0) = u_0$$

Possible applications:

- Exponential growth of money or populations: $f(u,t) = \alpha u$, $\alpha =$ const
- Logistic growth of a population under limited resources:

  $$f(u,t) = \alpha u\left(1 - \frac{u}{R}\right)$$

  where R is the maximum possible value of u
- Radioactive decay of a substance: $f(u,t) = -\alpha u$, $\alpha =$ const

Numerical solution of ordinary differential equations

- Numerous methods for $u'(t) = f(u,t)$, $u(0) = u_0$
- The Forward Euler method:

  $$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)$$

- The 4th-order Runge-Kutta method:

  $$u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$$

  $$\begin{aligned} K_1 &= \Delta t\, f(u_k, t_k), \\ K_2 &= \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t), \\ K_3 &= \Delta t\, f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t), \\ K_4 &= \Delta t\, f(u_k + K3, t_k + \Delta t) \end{aligned}$$

- There is a jungle of different methods – how to program?

A superclass for ODE methods

Common tasks for ODE solvers:

- Store the solution u_k and the corresponding time levels t_k, $k = 0, 1, 2, \ldots, N$
- Store the right-hand side function $f(u,t)$
- Store the time step Δt and last time step number k
- Set the initial condition
- Implement the loop over all time steps
- Code for the steps above are common to all classes and hence placed in superclass `ODESolver`
- Subclasses, e.g., `ForwardEuler`, just implement the specific stepping formula in a method `advance`

The superclass code

```
class ODESolver:
    def __init__(self, f, dt):
        self.f = f
        self.dt = dt

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process
        self.u.append(u0)
        self.t.append(t0)
        self.k = 0     # time level counter

    def solve(self, T):
        t = 0
        while t < T:
            unew = self.advance()    # the numerical formula
            self.u.append(unew)

            t = self.t[-1] + self.dt
            self.t.append(t)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)

    def advance(self):
        raise NotImplementedError
```


Implementation of the Forward Euler method

Subclass code:

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        unew = u[k] + dt*f(u[k], t)
        return unew
```

Application code for $u' = u$, $u(0) = 1$, $t \in [0, 3]$, $\Delta t = 0.1$:

```
from ODESolver import ForwardEuler
def test1(u, t):
    return u

method = ForwardEuler(test1, dt=0.1)
method.set_initial_condition(u0=1)
u, t = method.solve(T=3)
plot(t, u)
```

The implementation of a Runge-Kutta method

Subclass code:

```
class RungeKutta4(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]
        dt2 = dt/2.0
        K1 = dt*f(u[k], t)
        K2 = dt*f(u[k] + 0.5*K1, t + dt2)
        K3 = dt*f(u[k] + 0.5*K2, t + dt2)
        K4 = dt*f(u[k] + K3, t + dt)
        unew = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return unew
```

Application code (same as for ForwardEuler):

```
from ODESolver import RungeKutta4
def test1(u, t):
    return u

method = ForwardEuler(test1, dt=0.1)
method.set_initial_condition(u0=1)
u, t = method.solve(T=3)
plot(t, u)
```


Making a flexible toolbox for solving ODEs

- We can continue to implement formulas for different numerical methods for ODEs – a new method just requires the formula, not the rest of the code needed to set initial conditions and loop in time
- The OO approach saves typing – no code duplication
- Challenge: you need to understand exactly which "slots" in subclases you have to fill in – the overall code is an interplay of the superclass and the subclass
- Warning: more sophisticated methods for ODEs do not fit straight into our simple superclass – a more sophisticated superclass is needed, but the basic ideas of using OO remain the same
- Believe our conclusion: ODE methods are best implemented in a class hierarchy!

Example on a system of ODEs

- Several coupled ODEs make up a *system of ODEs*
- A simple example:

$$\begin{aligned} u'(t) &= v(t), \\ v'(t) &= -u(t) \end{aligned}$$

  Two ODEs with two unknowms $u(t)$ and $v(t)$
- Each unknown must have an initial condition, say

$$u(0) = 0, \quad v(0) = 1$$

- One can then derive the exact solution

$$u(t) = \sin(t), \quad v(t) = \cos(t)$$

- Systems of ODEs appear frequently in physics, biology, finance, ...

Another example on a system of ODEs

- Second-order ordinary differential equation, for a spring-mass system,

$$mu'' + \beta u' + ku = 0, \quad u(0) = u_0, \ u'(0) = 0$$

- We can rewrite this as a system of two *first-order* equations
- Introduce two new unknowns

$$u^{(0)}(t) \equiv u(t), \quad u^{(1)}(t) \equiv u'(t)$$

- The first-order system is then

$$\begin{aligned} \frac{d}{dt}u^{(0)}(t) &= u^{(1)}(t), \\ \frac{d}{dt}u^{(1)}(t) &= -m^{-1}\beta u^{(1)} - m^{-1}ku^{(0)} \end{aligned}$$

$$u^{(0)}(0) = u_0, \quad u^{(1)}(0) = 0$$

Vector notation for systems of ODEs (part 1)

- In general we have n unknowns

$$u^{(0)}(t), u^{(1)}(t), \ldots, u^{(n-1)}(t)$$

  in a system of n ODEs:

$$\begin{aligned} \frac{d}{dt}u^{(0)} &= f^{(0)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \\ \frac{d}{dt}u^{(1)} &= f^{(1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \\ \cdots &= \cdots \\ \frac{d}{dt}u^{(n-1)} &= f^{(n-1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \end{aligned}$$

Vector notation for systems of ODEs (part 2)

- We can collect the $u^{(i)}(t)$ functions and right-hand side functions $f^{(i)}$ in vectors:

$$u = (u^{(0)}, u^{(1)}, \ldots, u^{(n-1)})$$

$$f = (f^{(0)}, f^{(1)}, \ldots, f^{(n-1)})$$

- The first-order system can then be written

$$u' = f(u, t), \quad u(0) = u_0$$

  where u and f are vectors and u_0 is a vector of initial conditions
- Why is this notation useful? The notation make a scalar ODE and a system look the same, and we can easily make Python code that can handle both cases within the same lines of code (!)

How to make class ODESolver work for systems

- Recall: ODESolver was written for a scalar ODE
- Now we want it to work for a system $u' = f$, $u(0) = u_0$, where u, f and u_0 are vectors (arrays)
- Forward Euler for a system:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)$$

  (vector = vector + scalar $\times$ vector)
- In Python code:
  ```
  unew = u[k] + dt*f(u[k], t)
  ```
  where `u` is a list of arrays (`u[k]` is an array) and `f` is a function returning an array (all the right-hand sides $f^{(0)}, \ldots, f^{(n-1)}$)
- Result: ODESolver will work for systems!
- The only change: ensure that `f(u,t)` returns an array (This can be done be a general adjustment in the superclass!)

of: go through the code and check that it will work for a system as

```
    def set_initial_condition(self, u0, t0=0):
        self.u = []                  # list of arrays
        self.t = []
        self.u.append(u0)            # append array u0
        self.t.append(t0)
        self.k = 0

    def solve(self, T):
        t = 0
        while t < T:
            unew = self.advance()   # unew is array
            self.u.append(unew)     # append array

            t = self.t[-1] + self.dt
            self.t.append(t)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)

# in class Forward Euler:
    def advance(self):
        unew = u[k] + dt*f(u[k], t)  # ok if f returns array
        return unew
```

Smart trick

- Potential problem: `f(u,t)` may return a list, not array
- Solution: ODESolver can make a wrapper around the user's `f` function:

```
self.f = lambda u, t: numpy.asarray(f(u, t), float)
```

- Now the user can return right-hand side of the ODE as list, tuple or array - all existing method classes will work for systems of ODEs!

Back to implementing a system (part 1)

Spring-mass system formulated as a system of ODEs:

$$mu'' + \beta u' + ku = 0, \quad u(0),\ u'(0) \text{ known}$$

$$u^{(0)} = u, \quad u^{(1)} = u'$$

$$u(t) = (u^{(0)}(t), u^{(1)}(t))$$

$$f(u,t) = (u^{(1)}(t), -m^{-1}\beta u^{(1)} - m^{-1}ku^{(0)})$$

$$u'(t) = f(u,t)$$

Code defining the right-hand side:

```
def myf(u, t):
    # u is array with two components u[0] and u[1]:
    return [u[1],
            -beta*u[1]/m - k*u[0]/m]
```


Back to implementing a system (part 2)

Better (no global variables):

```
class MyF:
    def __init__(self, m, k, beta):
        self.m, self.k, self.beta = m, k, beta

    def __call__(self, u, t):
        m, k, beta = self.m, self.k, self.beta
        return [u[1], -beta*u[1]/m - k*u[0]/m]
```

Main program:

```
from ODESolver import ForwardEuler
# initial condition:
u0 = [1.0, 0]
f = MyF(1.0, 1.0, 0.0)   # u'' + u = 0 => u(t)=cos(t)
T = 4*pi; dt = pi/20
method = ForwardEuler(f, dt)
method.set_initial_condition(u0)
u, t = method.solve(T)

# u is an array of [u0,u1] arrays, plot all u0 values:
u0_values = u[:,0]
u0_exact = cos(t)
plot(t, u0_values, 'r-',  t, u0_exact, 'b-')
```


Application: throwing a ball (part 1)

- Newton's 2nd law for a ball's trajectory through air leads to

$$\begin{aligned}\frac{dx}{dt} &= v_x\\ \frac{dv_x}{dt} &= 0\\ \frac{dy}{dt} &= v_y\\ \frac{dv_y}{dt} &= -g\end{aligned}$$

  Air resistance is neglected but can easily be added!
- 4 ODEs with 4 unknowns: the ball's position $x(t)$, $y(t)$ and the velocity $v_x(t)$, $v_y(t)$

Application: throwing a ball (part 2)

Define the right-hand side:

```
def f(u, t):
    x, vx, y, vy = u
    g = 9.81
    return [vx, 0, vy, -g]
```

Main program:

```
from ODESolver import ForwardEuler
# t=0: prescribe velocity magnitude and angle
v0 = 5;  theta = 80*pi/180
# initial condition:
u0 = [0, v0*cos(theta), 0, v0*sin(theta)]

T = 1.2; dt = 0.01
method = ForwardEuler(f, dt)
method.set_initial_condition(u0)
u, t = method.solve(T)
# u is an array of [x,vx,y,vy] arrays, plot y vs x:
plot(u[:,0], u[:,1])
```


Application: throwing a ball (part 3)

Comparison of exact and Forward Euler solutions