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:
- \( f(u, t) = au \)
  - exponential growth
- \( f(u, t) = a(u - b)u \)
  - logistic growth
- \( f(u, t) = -b(u + g) \)
  - 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 \( u_1, u_2, \ldots \) values for \( u \) at times \( t_1, t_2, \ldots \):
  \[ u_{k+1} = u_k + \Delta t f(u_k, t_k) \]
  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 \( k = 0, 1, 2, \ldots, n-1 \):
  \[ \text{unew} = u[k] + \Delta t f(u[k], t[k]) \]
  \[ \text{append} \text{unew} \text{to} u \]
  \[ \text{append} \text{tnew} = t[k] + \Delta t \text{to} t \]

#### Mathematical problem:

**Solve** \( u' = u(t), u(0) = 1 \), for \( t \in [0, 3] \), with \( \Delta t = 0.1 \)

**Basic code:**
```
def f(u, t):
    return u
u = 1
T = 3
dt = 0.1
u, t = ForwardEuler(f, dt, u, 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')
```

#### The code for a class solving ODEs (part 1)
```
class ForwardEuler:
    def __init__(self, f, dt):
        self.f = f
        self.dt = dt
        self.u = []
        self.t = []

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

    def advance(self):
        # Advance solution one time step
        t = self.t[-1]
        u = self.u[-1]
        t = t + self.dt
        u = self.f(u, t)
        self.u.append(u)
        self.t.append(t)

    def solve(self, T):
        # Solve u' = f(u, t), u(0) = u0, in steps of dt until t <= T
        t = 0
        u = []
        self.t = []
        for t in range(int(round(T/dt))):
            u.append(u)
            t.append(t + self.dt)
        return numpy.array(u), numpy.array(t)
```

#### A class for solving ODEs (part 2)
```
class ForwardEuler:
    def __init__(self, f, dt):
        self.f = f
        self.dt = dt
        self.u = []
        self.t = []

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

    def advance(self):
        # Advance solution one time step
        t = self.t[-1]
        u = self.u[-1]
        t = t + self.dt
        u = self.f(u, t)
        self.u.append(u)
        self.t.append(t)

    def solve(self, T):
        # Solve u' = f(u, t), u(0) = u0, in steps of dt until t <= T
        t = 0
        u = []
        self.t = []
        for t in range(int(round(T/dt))):
            u.append(u)
            t.append(t + self.dt)
        return numpy.array(u), numpy.array(t)
```
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:
  \[
  \begin{align*}
  K_1 & = \Delta t f(u_k, t_k) \\
  K_2 & = \Delta t f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t) \\
  K_3 & = \Delta t f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t) \\
  K_4 & = \Delta t f(u_k + K_3, t_k + \Delta t)
  \end{align*}
  \]
  \[
  u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)
  \]
- 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 \( \Delta t \) 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

Implementation of the Forward Euler method

Subclass code:

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

def solve(self, T):
    t = 0
    while t < T:
        unew = self.advance()  # the numerical formula
        t = t + self.dt
    return unew
```

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

```python
from ODESolver import ForwardEuler
def test1(t):
    return 0.2 + (u - h(t))**4

method = ForwardEuler(test1, dt=0.1)
```

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

Mathematical problem:

\[
 u'(t) = \alpha f(u, t), \quad u(0) = u_0, \quad t \in [0, 40]
\]

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

```python
class Logistic:
    def __init__(self, alpha=0.2, dt=0.1):
        self.alpha = alpha
        self.dt = dt

    def set_initial_condition(self, u0):
        self.u0 = u0

    def solve(self, T):
        t = 0
        while t < T:
            unew = self.advance()  # the numerical formula
            t = t + self.dt
        return unew
```

A superclass for ODE methods

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

def set_initial_condition(self, u0, t0=0):
    self.u0 = u0
    self.t0 = t0

    def advance(self):
        return self.f(self.u0, self.t0)

    def solve(self, T):
        return advance()
```

Numerical solution of ordinary differential equations

- Numerical method 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:
  \[
  \begin{align*}
  K_1 & = \Delta t f(u_k, t_k) \\
  K_2 & = \Delta t f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t) \\
  K_3 & = \Delta t f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t) \\
  K_4 & = \Delta t f(u_k + K_3, t_k + \Delta t)
  \end{align*}
  \]
  \[
  u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)
  \]
- There is a jungle of different methods – how to program?
The implementation of a Runge-Kutta method

Subclass code:

class RungeKutta4(ODESolver):
def advance(self):
u = copy(self.u)
self.u = self.f(u, self.t, self.k, self.dt)
dt2 = self.dt / 2.0
K1 = self.f(u, self.t)
K2 = self.f(u + 0.5*K1, self.t + 0.5*dt2)
K3 = self.f(u + 0.5*K2, self.t + 0.5*dt2)
K4 = self.f(u + K2, self.t + dt2)
unew = u + K1/6.0 + K2/3.0 + 2*K3/3.0 + K4/6.0
return unew

Application code (same as for ForwardEuler):

from ODESolver import RungeKutta4
def test1(t):
return u
method = RungeKutta4(test1, dt=0.1)
method.set_initial_condition(u0)
result = method.solve(T)
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 subclasses 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:

\[ u'(t) = v(t), \]

\[ v'(t) = -u(t) \]

Two ODEs with two unknowns \( u(t) \) 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, ...

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:

\[ \frac{d}{dt} u^{(0)} = f^{(0)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \]

\[ \frac{d}{dt} u^{(1)} = f^{(1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \]

\[ \cdots \]

\[ \frac{d}{dt} u^{(n-1)} = f^{(n-1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t) \]

Vector notation for systems of ODEs (part 2)

We can collect the \( f^{(i)}(t) \) functions and right-hand side functions

\[ f^{(i)} \]

in vectors:

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

The first-order system can then be written

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

where \( u \) and \( f \) are vectors and \( u_0 \) is a vector of initial conditions.

Why is this notation useful? The notation makes 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 \( u = f(t, u(0) = u_0 \) where \( u \) 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:

\[ \text{unew} = u[1] + \text{dt} \cdot \text{f(u[1], t)} \]

where \( u \) is a list of arrays \( \{u[k] \mid k \in \text{array} \} \) and \( t \) 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 \( t(u_k, t) \) returns an array (This can be done be a general adjustment in the superclass!)

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

class VectorODESolver(ODESolver):
def __init__(self, u, T=0):
self.u = [u] # list of arrays
self.t = [0]
sel.f.append(f)
sel.{k} = 0

def solve(self, T):
t = 0
while t < T:
\text{unew = self.advance()} # unew is array
self.u.append(unew) # append array
self.t += self.{k} + self.{k} + self.t
return numpy.array(self.u), numpy.array(self.t)

# in class ForwardEuler:
def advance(self):
\text{unew = self} + \text{dt} \cdot \text{f(u[k], t)}
\text{# ok if f returns array}
return unew

Another example on a system of ODEs

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

\[ m\ddot{u} + \beta \dot{u} + ku = 0, \quad u(0) = u_0, \quad \dot{u}(0) = 0 \]

We can rewrite this as a system of two first-order equations.

Introduce two new unknowns

\[ u^{(0)}(t) = u(t), \quad u^{(1)}(t) = \dot{u}(t) \]

The first-order system is then

\[ \frac{d}{dt} u^{(0)} = u^{(1)}, \]

\[ \frac{d}{dt} u^{(1)} = -m^{-1} \beta u^{(0)} - m^{-1} ku^{(0)} \]

\[ u^{(0)}(0) = u_0, \quad u^{(1)}(0) = 0 \]
**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:
  ```python
  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:

\[
mu'' + \beta u' + ku = 0,\]

\( u(0), u'(0) \) known

\[
u(0) = u_0, \quad u'(0) = u_0'
\]

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

Code defining the right-hand side:

```python
def myf(u, t):
    return [u[1], -beta*u[1]/m - k*u[0]/m]
```

**Back to implementing a system (part 2)**

Better (no global variables):

```python
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:

```python
from ODESolver import ForwardEuler
# initial condition:
u0 = [1.0, 0]
T = 4*pi; dt = pi/20
method = ForwardEuler(f, dt)
method.set_initial_condition(u0)
u, t = method.solve(T)
```

**Application: throwing a ball (part 1)**

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

\[
\frac{dv_x}{dt} = 0
\]

\[
\frac{dv_y}{dt} = -g
\]

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:

```python
from ODESolver import ForwardEuler
# t=0: prescribe velocity magnitude and angle
dt = 0.01; 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)
```

**Application: throwing a ball (part 3)**

Comparison of exact and Forward Euler solutions