

# App.E: Programming of differential equations

Hans Petter Langtangen<sup>1,2</sup>

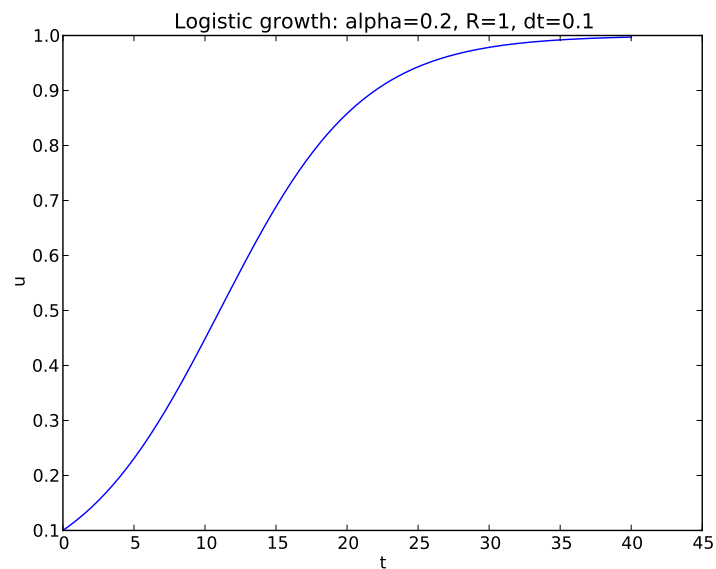
<sup>1</sup>Simula Research Laboratory

<sup>2</sup>University of Oslo, Dept. of Informatics

Oct 20, 2014

## How to solve any ordinary scalar differential equation

$$\begin{aligned}u'(t) &= \alpha u(t)(1 - R^{-1}u(t)) \\ u(0) &= U_0\end{aligned}$$



## Examples on scalar differential equations (ODEs)

### Terminology:

- *Scalar ODE*: a single ODE, one unknown function
- *Vector ODE* or *systems of ODEs*: several ODEs, several unknown functions

### Examples:

$$u' = \alpha u \quad \text{exponential growth}$$

$$u' = \alpha u \left(1 - \frac{u}{R}\right) \quad \text{logistic growth}$$

$$u' = -b|u|u + g \quad \text{body in fluid}$$

**We shall write an ODE in a generic form:**  $u' = f(u, t)$

- Numerical solution methods do not depend on how the ODE looks like
- Method and software aim at *any* ODE
- Therefore we need an abstract notation for an arbitrary ODE

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

The three ODEs on the last slide correspond to

$$f(u, t) = \alpha u, \quad \text{exponential growth}$$

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

$$f(u, t) = -b|u|u + g, \quad \text{body in fluid}$$

Our task: write functions and classes that take  $f$  and produces the solution  $u$

**What is the  $f(u, t)$ ?**

**Problem:** Given an ODE,

$$\sqrt{u}u' - \alpha(t)u^{3/2}\left(1 - \frac{u}{R(t)}\right) = 0,$$

what is the  $f(u, t)$ ?

**Solution:** The target form is  $u' = f(u, t)$ , so we need to isolate  $u'$  on the left-hand side:

$$u' = \underbrace{\alpha(t)u\left(1 - \frac{u}{R(t)}\right)}_{f(u,t)}$$

## How to solve a general ODE numerically by the Forward Euler method

$u' = f(u, t)$ . Assume we have computed  $u$  at discrete time points  $t_0, t_1, \dots, t_n$ . At  $t_k$  we have the ODE

$$u'(t_k) = f(u(t_k), t_k)$$

Approximate  $u'(t_k)$  by a forward finite difference,

$$u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}$$

Insert in the ODE at  $t = t_k$ :

$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} = f(u(t_k), t_k)$$

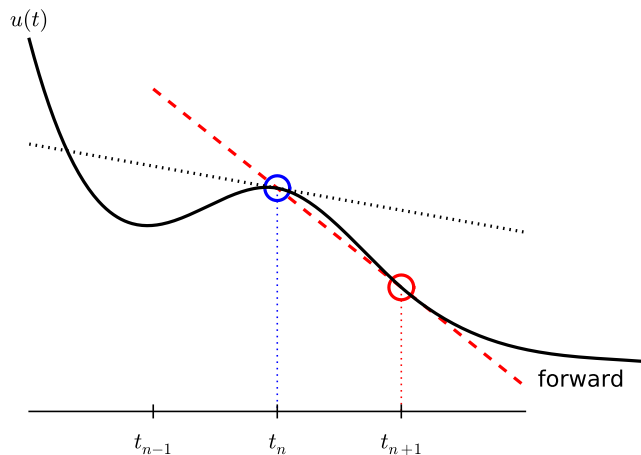
Everything with  $u(t_k)$  is known,  $u(t_{k+1})$  is unknown:

$$u(t_{k+1}) = u(t_k) + \Delta t f(u(t_k), t_k)$$

This is a very simple formula that we can use repeatedly for  $u(t_1)$ ,  $u(t_2)$ ,  $u(t_3)$  and so forth.

The technique is called *Euler's method* or the *Forward Euler method*.

## Illustration of the forward finite difference



**Notation:** We let  $u_k$  denote the numerical approximation to the exact solution  $u(t)$  for  $t = t_k$ .

**Let's apply the method!**

**Problem:** The world's simplest ODE:

$$u' = u, \quad t \in (0, T]$$

Solve  $u$  at discrete time points:  $t_k = k\Delta t$ ,  $k = 0, 1, 2, \dots, t_n$ ,  $t_0 = 0$ ,  $t_n = T$

**Forward Euler method:**

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

**Solution by hand:** What is  $f$ ?  $f(u, t) = u$

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

First step:

$$u_1 = (1 + \Delta t)u_0$$

but what is  $u_0$ ?

**Initial condition:**

- Any ODE  $u' = f(u, t)$  must have an initial condition  $u(0) = U_0$ , with known  $U_0$ , otherwise we cannot start the method!
- In mathematics:  $u(0) = U_0$  must be specified to get a unique solution

**We continue solution by hand**

Say  $U_0 = 2$ :

$$\begin{aligned} u_1 &= (1 + \Delta t)u_0 = (1 + \Delta t)U_0 = (1 + \Delta t)2 \\ u_2 &= (1 + \Delta t)u_1 = (1 + \Delta t)(1 + \Delta t)2 = 2(1 + \Delta t)^2 \\ u_3 &= (1 + \Delta t)u_2 = (1 + \Delta t)2(1 + \Delta t)^2 = 2(1 + \Delta t)^3 \\ u_4 &= (1 + \Delta t)u_3 = (1 + \Delta t)2(1 + \Delta t)^3 = 2(1 + \Delta t)^4 \\ u_5 &= (1 + \Delta t)u_4 = (1 + \Delta t)2(1 + \Delta t)^4 = 2(1 + \Delta t)^5 \\ &\vdots \\ u_k &= 2(1 + \Delta t)^k \end{aligned}$$

Actually, we found a formula for  $u_k$ ! No need to program...

What about the general case  $u' = f(u, t)$ ?

Given any  $U_0$ :

$$\begin{aligned}u_1 &= u_0 + \Delta t f(u_0, t_0) \\u_2 &= u_1 + \Delta t f(u_1, t_1) \\u_3 &= u_2 + \Delta t f(u_2, t_2) \\u_4 &= u_3 + \Delta t f(u_3, t_3) \\&\vdots \\&\vdots\end{aligned}$$

No general formula in this case...

**Rule of thumb:** These hand calculations get very boring, so let's program!

**A special program for the special ODE**  $u' = u$ ,  $u(0) = U_0$

**Algorithm:** Given  $\Delta t$  ( $dt$ ) and  $n$

- Create arrays  $\mathbf{t}$  and  $\mathbf{u}$  of length  $n + 1$
- Create array  $\mathbf{u}$  to hold  $u_k$  and
- Set initial condition:  $\mathbf{u}[0] = U_0$ ,  $\mathbf{t}[0] = 0$
- For  $k = 0, 1, 2, \dots, n - 1$ :
  - $\mathbf{t}[k+1] = \mathbf{t}[k] + dt$
  - $\mathbf{u}[k+1] = (1 + dt) * \mathbf{u}[k]$

**Program:**

```
import numpy as np
import sys

dt = float(sys.argv[1])
U0 = 1
T = 4
n = int(T/dt)

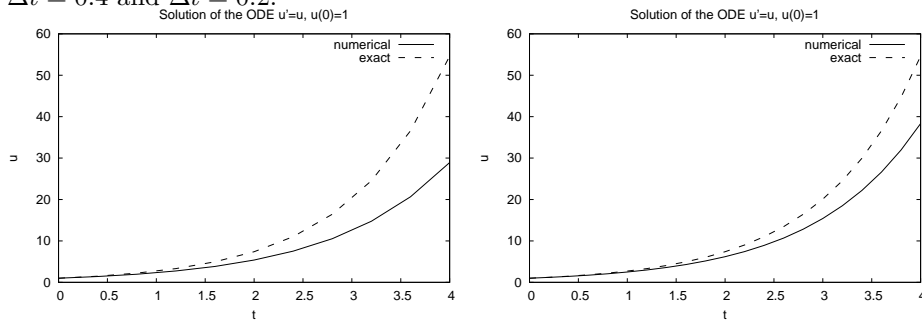
t = np.zeros(n+1)
u = np.zeros(n+1)

t[0] = 0
u[0] = U0
for k in range(n):
    t[k+1] = t[k] + dt
    u[k+1] = (1 + dt)*u[k]

# plot u against t
```

The solution if we plot  $u$  against  $t$

$\Delta t = 0.4$  and  $\Delta t = 0.2$ :



The algorithm for the general ODE  $u' = f(u, t)$

**Algorithm:** Given  $\Delta t$  ( $dt$ ) and  $n$

- Create arrays  $\mathbf{t}$  and  $\mathbf{u}$  of length  $n + 1$
- Create array  $\mathbf{u}$  to hold  $u_k$  and
- Set initial condition:  $\mathbf{u}[0] = U_0$ ,  $\mathbf{t}[0] = 0$
- For  $k = 0, 1, 2, \dots, n - 1$ :
  - $\mathbf{u}[k+1] = \mathbf{u}[k] + dt * f(\mathbf{u}[k], \mathbf{t}[k])$   
textcolorred(the only change!)
  - $\mathbf{t}[k+1] = \mathbf{t}[k] + dt$

Implementation of the general algorithm for  $u' = f(u, t)$

General function:

```
def ForwardEuler(f, U0, T, n):
    """Solve u'=f(u,t), u(0)=U0, with n steps until t=T."""
    import numpy as np
    t = np.zeros(n+1)
    u = np.zeros(n+1) # u[k] is the solution at time t[k]

    u[0] = U0
    t[0] = 0
    dt = T/float(n)

    for k in range(n):
        t[k+1] = t[k] + dt
        u[k+1] = u[k] + dt*f(u[k], t[k])

    return u, t
```

**Magic:** This simple function can solve any ODE (!)

## Example on using the function

**Mathematical problem:** Solve  $u' = u$ ,  $u(0) = 1$ , for  $t \in [0, 4]$ , with  $\Delta t = 0.4$   
Exact solution is  $u(t) = e^t$ .

**Basic code:**

```
def f(u, t):  
    return u  
  
U0 = 1  
T = 3  
n = 30  
u, t = ForwardEuler(f, U0, T, n)
```

**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")
```

## Now you can solve any ODE!

**Recipe:**

- Identify  $f(u, t)$  in your ODE
- Make sure you have an initial condition  $U_0$
- Implement the  $f(u, t)$  formula in a Python function `f(u, t)`
- Choose  $\Delta t$  or no of steps  $n$
- Call `u, t = ForwardEuler(f, U0, T, n)`
- `plot(t, u)`

**Warning:** The Forward Euler method may give very inaccurate solutions if  $\Delta t$  is not sufficiently small. For some problems (like  $u'' + u = 0$ ) other methods should be used.

## A class for solving ODEs

Instead of a function for solving any ODE we now want to make a class and use it like this:

```
method = ForwardEuler(f, dt)  
method.set_initial_condition(U0, t0)  
u, t = method.solve(T)  
plot(t, u)
```

How?

- Store  $f$ ,  $\Delta t$ , and the sequences  $u_k$ ,  $t_k$  as attributes
- Split the steps in the `ForwardEuler` function into three methods:
  - the constructor (`__init__`)
  - `set_initial_condition` for  $u(0) = U_0$
  - `solve` for running the Forward Euler algorithm
  - `advance` for isolating the numerical updating formula  
(new numerical methods just need a different `advance` method, the rest is the same)

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

```
import numpy as np

class ForwardEuler_v1:
    def __init__(self, f, dt):
        self.f, self.dt = f, dt

    def set_initial_condition(self, U0):
        self.U0 = float(U0)
```

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

```
class ForwardEuler_v1:
    ...
    def solve(self, T):
        """Compute solution for 0 <= t <= T."""
        n = int(round(T/self.dt))
        self.u = np.zeros(n+1)
        self.t = np.zeros(n+1)
        self.u[0] = float(self.U0)
        self.t[0] = float(0)

        for k in range(self.n):
            self.k = k
            self.t[k+1] = self.t[k] + self.dt
            self.u[k+1] = self.advance()
        return self.u, self.t

    def advance(self):
        """Advance the solution one time step."""
        u, dt, f, k, t = self.u, self.dt, self.f, self.k, self.t

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



## Alternative class code solving ODEs (part 1)

```
# Idea: drop dt in the constructor.
# Let the user provide all time points (in solve).

class ForwardEuler:
    def __init__(self, f):
        # test that f is a function
        if not callable(f):
            raise TypeError('f is %s, not a function' % type(f))
        self.f = f

    def set_initial_condition(self, U0):
        self.U0 = float(U0)

    def solve(self, time_points):
        ...
```

## Alternative class code for solving ODEs (part 2)

```
class ForwardEuler:
    ...
    def solve(self, time_points):
        """Compute u for t values in time_points list."""
        self.t = np.asarray(time_points)
        self.u = np.zeros(len(time_points))

        self.u[0] = self.U0

        for k in range(len(self.t)-1):
            self.k = k
            self.u[k+1] = self.advance()
        return self.u, self.t

    def advance(self):
        """Advance the solution one time step."""
        u, f, k, t = self.u, self.f, self.k, self.t

        dt = t[k+1] - t[k]
        unew = u[k] + dt*f(u[k], t[k])
        return unew
```

## Verifying the class implementation; mathematics

**Mathematical problem:** Important result: the numerical method (and most others) will exactly reproduce  $u$  if it is linear in  $t$  (!):

$$u(t) = at + b = 0.2t + 3$$

$$h(t) = u(t)$$

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

This  $u$  should be reproduced to machine precision for “any”  $\Delta t$  (not too large).

## Verifying the class implementation; implementation

Code:

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

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

    solver = ForwardEuler(f)
    solver.set_initial_condition(U0=3)
    dt = 0.4; T = 3; n = int(round(float(T)/dt))
    time_points = np.linspace(0, T, n+1)
    u, t = solver.solve(time_points)
    u_exact = h(t)
    diff = np.abs(u_exact - u).max()
    tol = 1E-14
    success = diff < tol
    assert success
```

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  $f(u, t)$ :

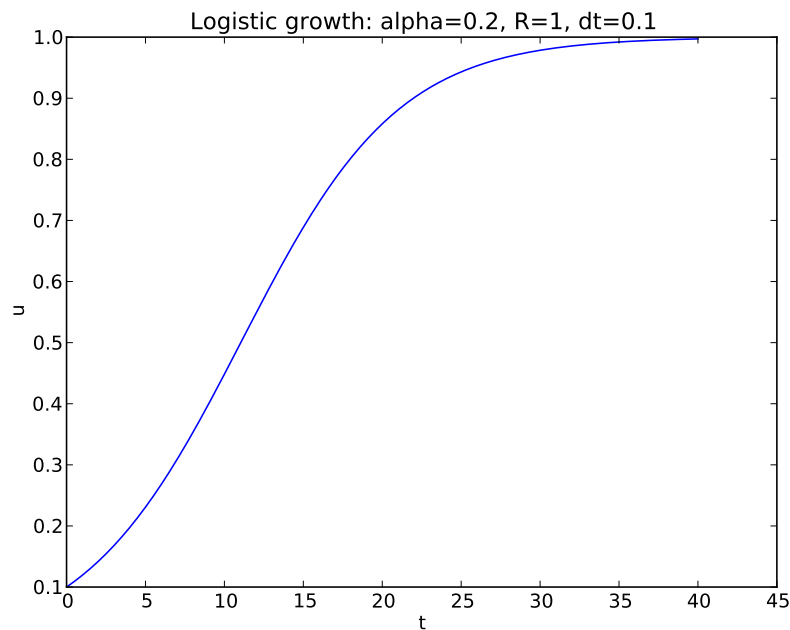
```
class Logistic:
    def __init__(self, alpha, R, U0):
        self.alpha, self.R, self.U0 = alpha, float(R), U0

    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)
time_points = np.linspace(0, 40, 401)
method = ForwardEuler(problem)
method.set_initial_condition(problem.U0)
u, t = method.solve(time_points)
```

**Figure of the solution**



## Numerical methods for ordinary differential equations

### Forward Euler method:

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

### 4th-order Runge-Kutta method:

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

where

$$K_1 = \Delta t f(u_k, t_k)$$

$$K_2 = \Delta t f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t)$$

$$K_3 = \Delta t f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t)$$

$$K_4 = \Delta t f(u_k + K_3, t_k + \Delta t)$$

And lots of other methods! How to program a collection of methods?

## A superclass for ODE methods

### Common tasks for ODE solvers:

- Store the solution  $u_k$  and the corresponding time levels  $t_k, k = 0, 1, 2, \dots, n$
- Store the right-hand side function  $f(u, t)$
- Set and store the initial condition
- Run the loop over all time steps

### Principles:

- Common data and functionality are placed in superclass `ODESolver`
- Isolate the numerical updating formula in a method `advance`
- Subclasses, e.g., `ForwardEuler`, just implement the specific numerical formula in `advance`

### The superclass code

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

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError # implement in subclass

    def set_initial_condition(self, U0):
        self.U0 = float(U0)

    def solve(self, time_points):
        self.t = np.asarray(time_points)
        self.u = np.zeros(len(self.t))
        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
            self.u[k+1] = self.advance()
        return self.u, self.t

    def advance(self):
        raise NotImplementedError # to be impl. in subclasses
```

### Implementation of the Forward Euler method

#### Subclass code:

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t

        dt = t[k+1] - t[k]
        unew = u[k] + dt*f(u[k], t)
        return unew
```

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

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

method = ForwardEuler(test1)
method.set_initial_condition(U0=1)
u, t = method.solve(time_points=np.linspace(0, 3, 31))
plot(t, u)
```

## The implementation of a Runge-Kutta method

Subclass code:

```
class RungeKutta4(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t

        dt = t[k+1] - t[k]
        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 = RungeKutta4(test1)
method.set_initial_condition(U0=1)
u, t = method.solve(time_points=np.linspace(0, 3, 31))
plot(t, u)
```

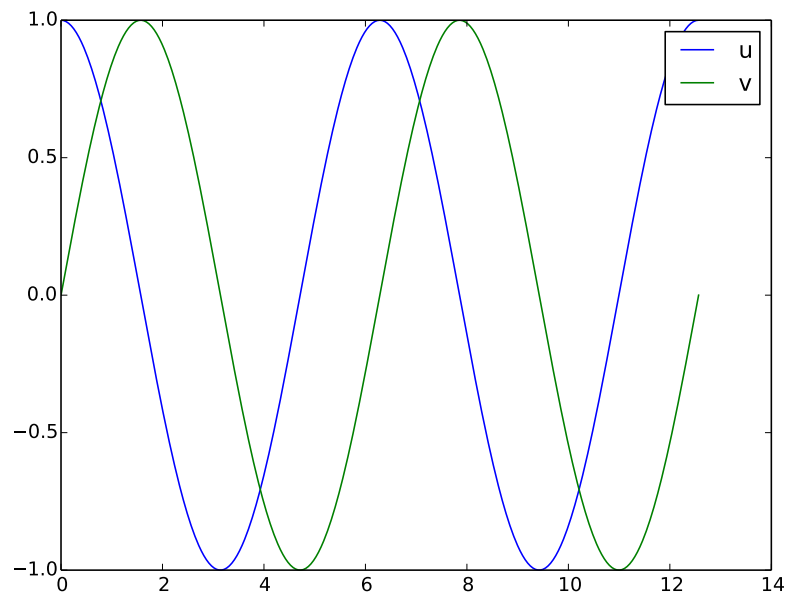
The user should be able to check intermediate solutions and terminate the time stepping

- Sometimes a property of the solution determines when to stop the solution process: e.g., when  $u < 10^{-7} \approx 0$ .
  - Extension `solve(time_points, terminate)`
  - `terminate(u, t, step_no)` is called at every time step, is user-defined, and returns `True` when the time stepping should be terminated
  - Last computed solution is `u[step_no]` at time `t[step_no]`

```
def terminate(u, t, step_no):
    eps = 1.0E-6          # small number
    return abs(u[step_no,0]) < eps  # close enough to zero?
```

## Systems of differential equations (vector ODE)

$$\begin{aligned}u' &= v \\v' &= -u \\u(0) &= 1 \\v(0) &= 0\end{aligned}$$



## Example on a system of ODEs (vector ODE)

Two ODEs with two unknowns  $u(t)$  and  $v(t)$ :

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

Each unknown must have an initial condition, say

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

In this case, one can derive the exact solution

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

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

## The ODE system that is the final project in the course

Model for spreading of a disease in a population:

$$\begin{aligned} S' &= -\beta SI \\ I' &= \beta SI - \nu R \\ R' &= \nu I \\ S(0) &= S_0 \\ I(0) &= I_0 \\ R(0) &= 0 \end{aligned}$$

## Another example on a system of ODEs (vector ODE)

Second-order ordinary differential equation, for a spring-mass system (from Newton's second law):

$$mu'' + \beta u' + ku = 0, \quad u(0) = U_0, \quad u'(0) = 0$$

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

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

The first-order system is then

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

Initial conditions:

$$u^{(0)}(0) = U_0, \quad u^{(1)}(0) = 0$$

## Making a flexible toolbox for solving ODEs

- For scalar ODEs we could make one general class hierarchy to solve “all” problems with a range of methods

- Can we easily extend class hierarchy to systems of ODEs?
- Yes!
- The example here can easily be extended to professional code ([Odespy](#))

## Vector notation for systems of ODEs: unknowns and equations

General software for any vector/scalar ODE demands a general mathematical notation. We introduce  $n$  unknowns

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

in a system of  $n$  ODEs:

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

## Vector notation for systems of ODEs: vectors

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

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

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

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

**The magic of this notation:** Observe that 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 of ODEs

- 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)
- What are the problems?

Forward Euler applied to a system:

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

In Python code:

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

where

- $u$  is a two-dim. array ( $u[k]$  is a row)
- $f$  is a function returning an array (all the right-hand sides  $f^{(0)}, \dots, f^{(n-1)}$ )
- Result: ODESolver will work for systems!
- The only change: ensure that  $f(u, t)$  returns an array  
(This can be done by a general adjustment in the superclass!)

## The adjusted superclass code (part 1)

```
class ODESolver:
    def __init__(self, f):
        # Wrap user's f in a new function that always
        # converts list/tuple to array (or let array be array)
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float, int)): # scalar ODE
            self.neq = 1 # no of equations
            U0 = float(U0)
        else: # system of ODEs
            U0 = np.asarray(U0)
            self.neq = U0.size # no of equations
        self.U0 = U0
```

## The superclass code (part 2)

```
class ODESolver:
    ...
    def solve(self, time_points, terminate=None):
        if terminate is None:
            terminate = lambda u, t, step_no: False

        self.t = np.asarray(time_points)
        n = self.t.size
        if self.neq == 1: # scalar ODEs
            self.u = np.zeros(n)
        else: # systems of ODEs
            self.u = np.zeros((n,self.neq))

        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
            self.u[k+1] = self.advance()
            if terminate(self.u, self.t, self.k+1):
                break # terminate loop over k
        return self.u[:k+2], self.t[:k+2]
```

All subclasses from the scalar ODE works for systems as well

## Example on how to use the general class hierarchy

Spring-mass system formulated as a system of ODEs:

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

$$\begin{aligned} u^{(0)} &= u, \quad u^{(1)} = u' \\ u(t) &= (u^{(0)}(t), u^{(1)}(t)) \\ f(u, t) &= (u^{(1)}(t), -m^{-1}\beta u^{(1)} - m^{-1}ku^{(0)}) \\ u'(t) &= f(u, t) \end{aligned}$$

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]
```

Alternative implementation of the  $f$  function via a class

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)
solver = ForwardEuler(f)
solver.set_initial_condition(U0)

T = 4*pi; dt = pi/20; n = int(round(T/dt))
time_points = np.linspace(0, T, n+1)
u, t = solver.solve(time_points)

# 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-')

```

## Throwing a ball; ODE model

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  $x(t)$ ,  $y(t)$
  - the velocity  $v_x(t)$ ,  $v_y(t)$

## Throwing a ball; code

Define the right-hand side:

```

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

```

### Main program:

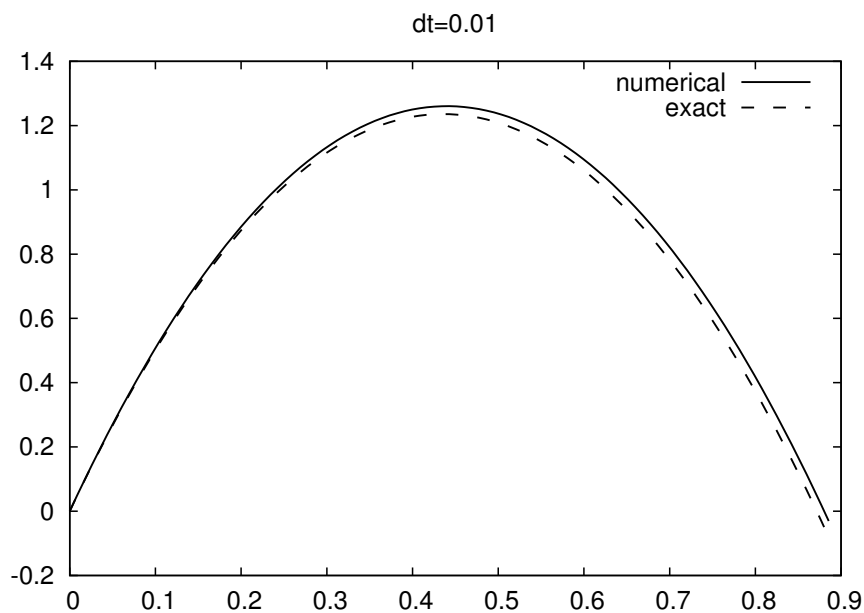
```
from ODESolver import ForwardEuler
# t=0: prescribe x, y, vx, vy
x = y = 0
v0 = 5; theta = 80*pi/180 # start at the origin
vx = v0*cos(theta)         # velocity magnitude and angle
vy = v0*sin(theta)
# Initial condition:
U0 = [x, vx, y, vy]

solver= ForwardEuler(f)
solver.set_initial_condition(u0)
time_points = np.linspace(0, 1.2, 101)
u, t = solver.solve(time_points)

# u is an array of [x,vx,y,vy] arrays, plot y vs x:
x = u[:,0]; y = u[:,2]
plot(x, y)
```

### Throwing a ball; results

Comparison of exact and Forward Euler solutions



### Logistic growth model; ODE and code overview

Model:

$$u' = \alpha u(1 - u/R(t)), \quad u(0) = U_0$$

$R$  is the maximum population size, which can vary with changes in the environment over time

### Implementation features:

- Class Problem holds “all physics”:  $\alpha$ ,  $R(t)$ ,  $U_0$ ,  $T$  (end time),  $f(u, t)$  in ODE
- Class Solver holds “all numerics”:  $\Delta t$ , solution method; solves the problem and plots the solution
- Solve for  $t \in [0, T]$  but terminate when  $|u - R| < \text{tol}$

### Logistic growth model; class Problem ( $f$ )

```
class Problem:
    def __init__(self, alpha, R, U0, T):
        self.alpha, self.R, self.U0, self.T = alpha, R, U0, T

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

    def terminate(self, u, t, step_no):
        """Terminate when u is close to R."""
        tol = self.R*0.01
        return abs(u[step_no] - self.R) < tol

problem = Problem(alpha=0.1, R=500, U0=2, T=130)
```

### Logistic growth model; class Solver

```
class Solver:
    def __init__(self, problem, dt,
                 method=ODESolver.ForwardEuler):
        self.problem, self.dt = problem, dt
        self.method = method

    def solve(self):
        solver = self.method(self.problem)
        solver.set_initial_condition(self.problem.U0)
        n = int(round(self.problem.T/self.dt))
        t_points = np.linspace(0, self.problem.T, n+1)
        self.u, self.t = solver.solve(t_points,
                                     self.problem.terminate)

    def plot(self):
        plot(self.t, self.u)

problem = Problem(alpha=0.1, U0=2, T=130,
                  R=lambda t: 500 if t < 60 else 100)
solver = Solver(problem, dt=1.)
solver.solve()
solver.plot()
print 'max u:', solver.u.max()
```

## Logistic growth model; results

