App.E: Programming of differential equations

Hans Petter Langtangen[1,2] Joakim Sundnes[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Nov 10, 2017

Plan for the rest of the fall (1)

- Friday November 10:
  - Short quiz
  - Exer 9.4, 9.6 (inheritance, OOP)
  - How to solve *any* scalar ODE
- Wednesday November 15:
  - Exer E.21, E.22, 8.x
  - Vector ODEs (Systems of ODEs)
  - Random numbers and games
- Friday November 17:
  - More on vector ODEs
  - Disease modeling (final project)

Plan for the rest of the fall (2)

- November 20 - November 27:
  - Final project on disease modeling
  - No ordinary lectures
  - Time for questions about the project ("orakel") will be announced
  - Lectures "on demand" Nov 22 and Nov 24 (project relevant)
- November 27 - Exam:
  - Repetition lectures ("on demand")

Quiz (special methods)

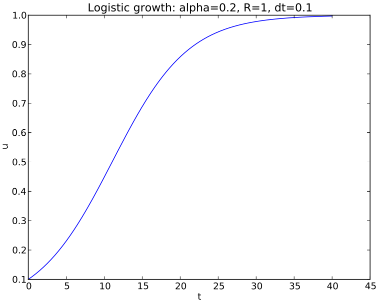
What is printed by the following code? Why?

```
from numpy import *

class MyList:
    def __init__(self,values):
        self.values = values

    def __add__(self,other):
        result = []
        for i in range(len(self.values)):
        result.append(str(self.values[i])+'+' \
                                      +str(other.values[i]))
        return result

l1 = [2,3,4]; l2 = [5,6,1]
a1 = array(l1); a2 = array(l2)
m1 = MyList(l1); m2 = MyList(l2)
print(l1+l2)
print(a1+a2)
print(m1+m2)
```

1. How to solve any ordinary scalar differential equation

How to solve any ordinary scalar differential equation

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



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:

$$
\begin{aligned}
u' &= \alpha u \quad \text{exponential growth} \\
u' &= \alpha u \left(1 - \frac{u}{R}\right) \quad \text{logistic growth} \\
u' + b|u|u &= g \quad \text{falling body in fluid}
\end{aligned}
$$

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

- Our methods and software should be applicable to *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

$$\begin{aligned}
f(u,t) &= \alpha u, \quad \text{exponential growth}\\
f(u,t) &= \alpha u\left(1 - \frac{u}{R}\right), \quad \text{logistic growth}\\
f(u,t) &= -b|u|u + g, \quad \text{body in fluid}
\end{aligned}$$

Our task: write functions and classes that take f as input and produce u as output

Such abstract f functions are widely used in mathematics

We can make generic software for:

- Numerical differentiation: $f'(x)$
- Numerical integration: $\int_a^b f(x)dx$
- Numerical solution of algebraic equations: $f(x) = 0$

Applications:

1. $\frac{d}{dx}x^a \sin(wx)$: $f(x) = x^a \sin(wx)$
2. $\int_{-1}^{1}(x^2 \tanh^{-1} x - (1+x^2)^{-1})dx$:
   $f(x) = x^2 \tanh^{-1} x - (1+x^2)^{-1}$, $a = -1$, $b = 1$
3. Solve $x^4 \sin x = \tan x$: $f(x) = x^4 \sin x - \tan x$

We use finite difference approximations to derivatives to turn an ODE into a difference equation

$u' = f(u, t)$

Assume we have computed u at discrete time points $t_0, t_1, \ldots, t_k$. 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,

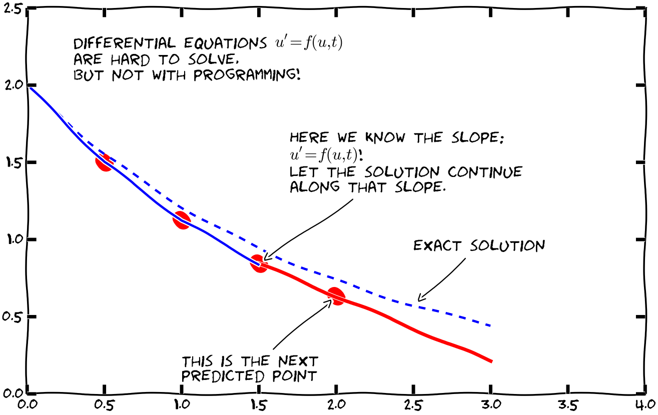
$$u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}$$

Insert in the ODE at $t = t_k$:

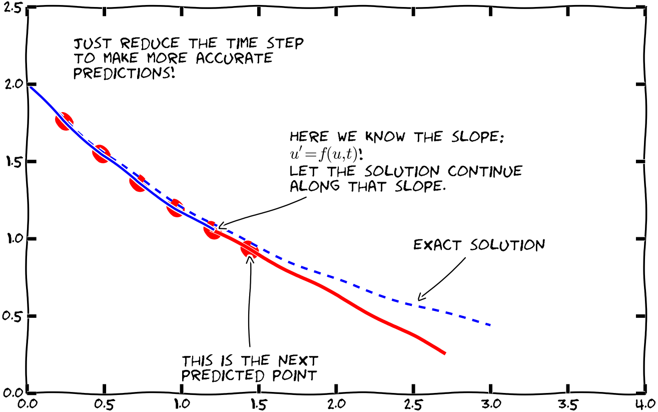
$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} = f(u(t_k), t_k)$$

Terms with $u(t_k)$ are known, and this is an algebraic (difference) equation for $u(t_{k+1})$

The Forward Euler (or Euler's) method; idea



The Forward Euler (or Euler's) method; idea



The Forward Euler (or Euler's) method; mathematics

Solving with respect to $u(t_{k+1})$

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

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

Difference equation notation:

Let u_k denote the numerical approximation to the exact solution $u(t)$ at $t = t_k$.

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

This is an ordinary difference equation we can solve!

Let's apply the method!

Problem: The world's simplest ODE

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

Solve for u at $t = t_k = k\Delta t$, $k = 0, 1, 2, \ldots, t_n$, $t_0 = 0$, $t_n = T$

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?

An ODE needs an initial condition: $u(0) = U_0$

Numerics:

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!

Mathematics:

In mathematics: $u(0) = U_0$ must be specified to get a unique solution.

Example:

$$u' = u$$

Solution: $u = Ce^t$ for any constant C. Say $u(0) = U_0$: $u = U_0 e^t$.

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

Given any U_0:

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

We start with a specialized program for $u' = u$, $u(0) = U_0$

Algorithm:

Given Δt (`dt`) and n

- Create arrays `t` and `u` of length $n + 1$
- Set initial condition: `u[0]` $= U_0$, `t[0]=0`
- For $k = 0, 1, 2, \ldots, n - 1$:
  - `t[k+1] = t[k] + dt`
  - `u[k+1] = (1 + dt)*u[k]`

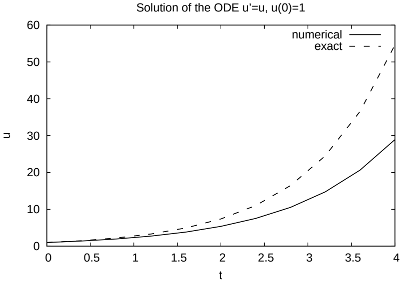
We start with a specialized program for $u' = u$, $u(0) = U_0$

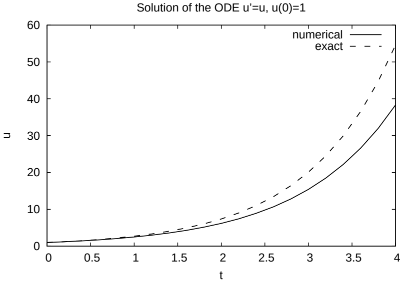
Program:

```
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 Δt (`dt`) and n

- Create arrays `t` and `u` of length $n+1$
- Create array `u` to hold u_k and
- Set initial condition: `u[0]` $= U_0$, `t[0]=0`
- For $k = 0, 1, 2, \ldots, n-1$:
  - `u[k+1] = u[k] + dt*f(u[k], t[k])` (the only change!)
  - `t[k+1] = 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: $u(t) = e^t$.

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 Δt 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 Δt is not sufficiently small. For some problems (like $u'' + u = 0$) other methods should be used.

Let us make a class instead of a function for solving ODEs

Usage of the class:

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

How?

- Store f, Δt, and the sequences u_k, t_k as attributes
- Split the steps in the `ForwardEuler` function into four methods:
  - the constructor (`__init__`)
  - `set_initial_condition` for $u(0) = U_0$
  - `solve` for running the numerical time stepping
  - `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))  # no of intervals
        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."""
        # Create local variables to get rid of "self." in
        # the numerical formula
        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
```

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

Mathematical problem:

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = U_0, \quad t \in [0, 40]$$

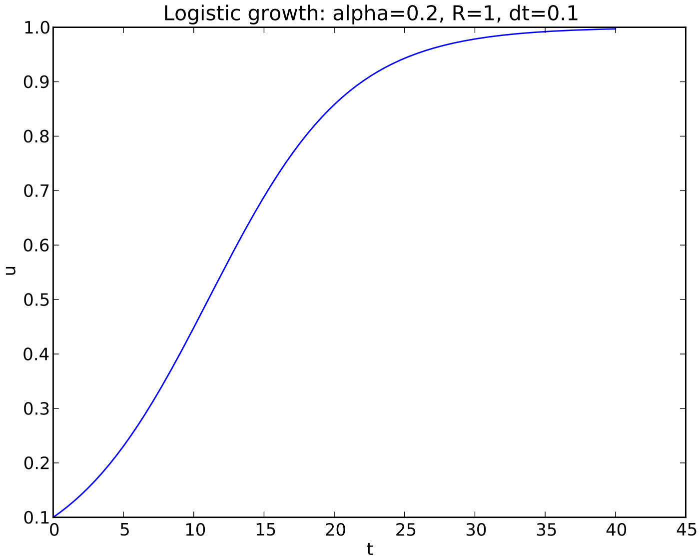
Class for right-hand side $f(u, t)$:

```
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:

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

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

And lots of other methods! How to program a wide collection of methods? Use object-oriented programming!

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 $f(u, t)$
- 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`

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 $f(u, t)$
- 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`

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 $f(u, t)$
- 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 NotImplemtedError # 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?
```