App.E: Systems of differential equations

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

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Nov 16, 2017

Plan for Friday 17/11

- A class hierarchy of ODE solvers
- Vector ODEs (Systems of ODEs)
- Disease modeling (final project)

Recall the different methods:

Forward Euler

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

4th-order Runge-Kutta

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

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

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)
```

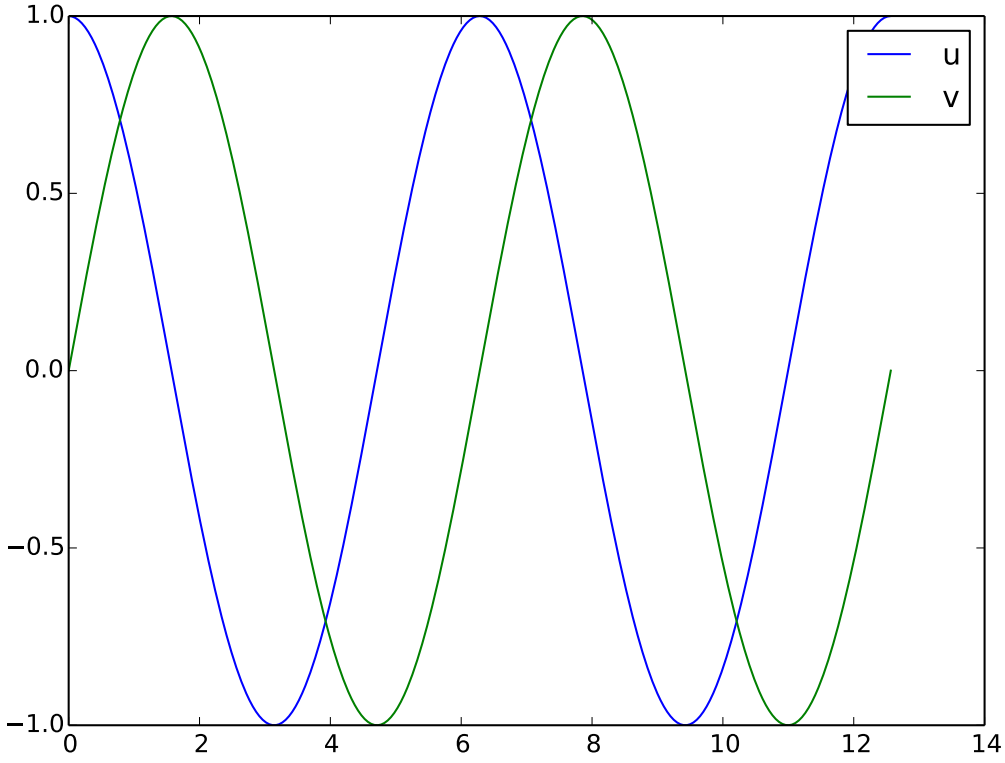
The user should be able to check intermediate solutions and terminate the time stepping

- 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?
```

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 $v(t)$:

$$u'(t) = v(t)$$
$$v'(t) = -u(t)$$

Each unknown must have an initial condition, say

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

In this case, one can derive the exact solution to be

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

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:

$$
\begin{aligned}
S' &= -\beta SI \\
I' &= \beta SI - \nu R \\
R' &= \nu I
\end{aligned}
$$

Initial conditions:

$$
\begin{aligned}
S(0) &= S_0 \\
I(0) &= I_0 \\
R(0) &= 0
\end{aligned}
$$

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

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

in a system of n ODEs:

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

Vector notation for systems of ODEs: vectors

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

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

The adjusted superclass code (part 1)

To make ODESolver work for systems:

- Ensure that `f(u,t)` returns an array.
  This can be done be a general adjustment in the superclass!
- Inspect U_0 to see if it is a number or list/tuple and make corresponding `u` 1-dim or 2-dim array

```
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: ODE model for throwing a ball

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

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

Air resistance is neglected but can easily be added

- 4 ODEs with 4 unknowns:
  - the ball's position $x(t)$, $y(t)$
  - 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:

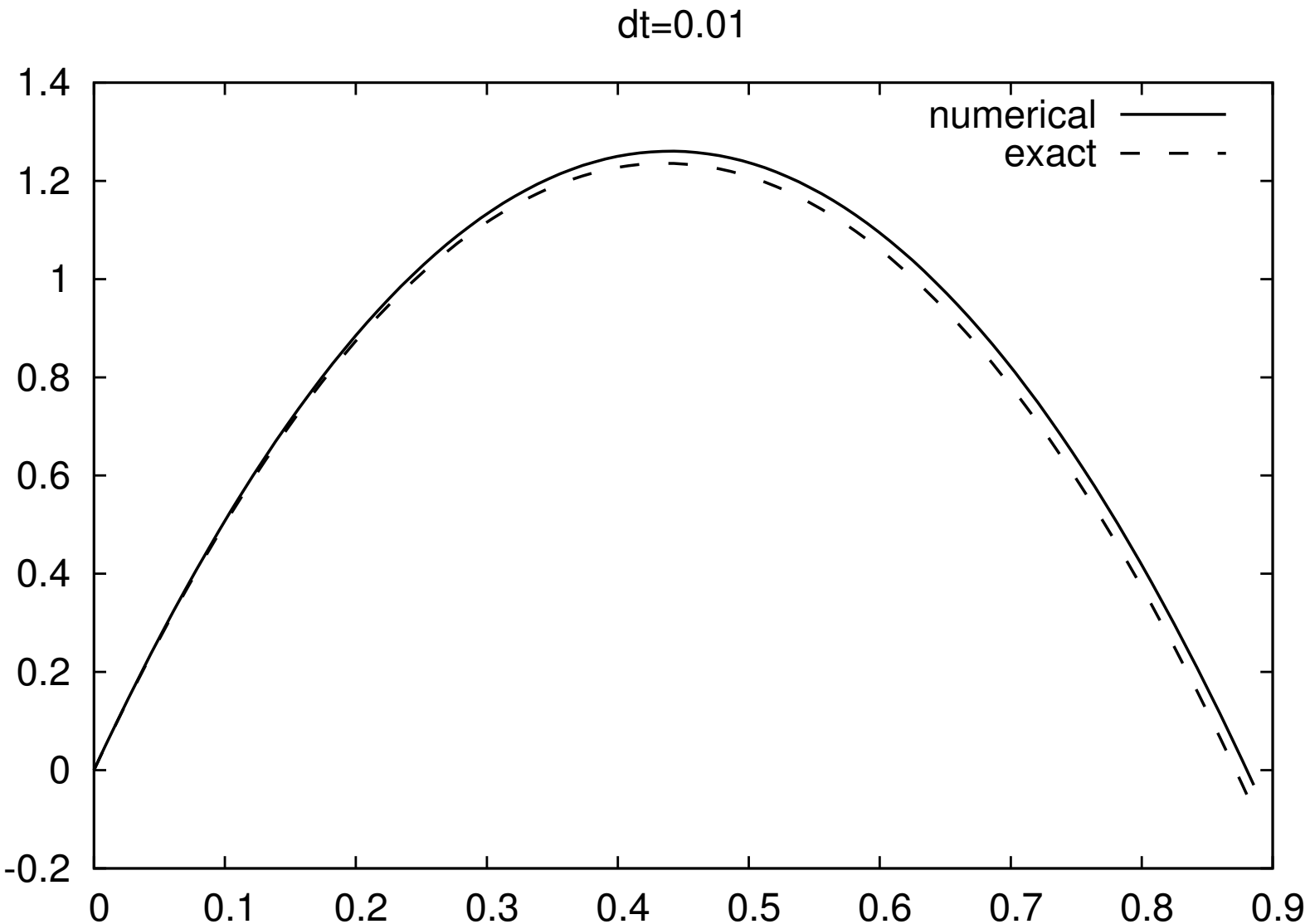
```
from ODESolver import ForwardEuler
# t=0: prescribe x, y, vx, vy
x = y = 0                          # start at the origin
v0 = 5;  theta = 80*pi/180         # velocity magnitude and angle
vx = v0*cos(theta)
vy = v0*sin(theta)
# Initial condition:
U0 = [x, vx, y, vy]

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



Summary

ODE solvers and OOP

- Many different ODE solvers (Euler, Runge-Kutta, ++)
- Most tasks are common to all solvers:
  - Initialization of solution arrays and right hand side
  - Overall `for`-loop for advancing the solution
- Difference; how the solution is advanced from step k to $k + 1$
- OOP implementation:
  - Collect all common code in a base class
  - Implement the different step (`advance`) functions in subclasses

Systems of ODEs

- All solvers and codes are easily extended to systems of ODEs
- Solution at one time step (u_k) is a vector (one-dimensional array), overall solution is a two-dimensional array
- Slightly more book-keeping, but the bulk of the code is identical as for scalar ODEs