# 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]

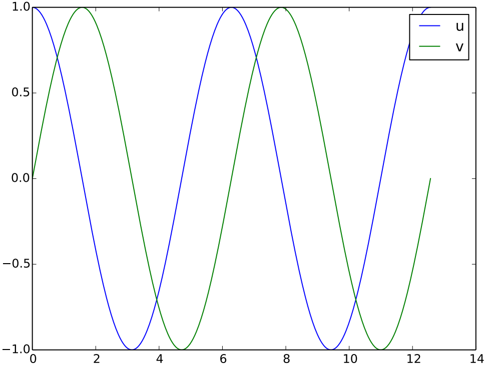
Nov 5, 2018

# Plan for Thursday 1/11 (adjusted)

- Exercise 9.1, 9.3, 9.4
- More on ODE solvers:
    - The forward Euler method as a class
    - Alternative ODE solvers
    - Class hierarchies for ODE solvers
    - Vector ODEs (Systems of ODEs)

# 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 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:

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

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!

# 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:

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

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

# Example - scalar ODE vs system of two

**Scalar ODE**

```
t = [0.  0.4 0.8 1.2 (...) ]

u = [ 1.0 1.4  1.96 2.744  (...)]

u[0] = 1.0
u[1] = 1.4

(...)
```

**System of two ODEs**

```
u = [[1.0 0.8][1.4 1.1] [1.9 2.7] (...)]

u[0] = [1.0 0.8]
u[1] = [1.4 1.1]

(...)
```

# 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:

```
# Initial condition, start at the origin:
x = 0; y = 0
# velocity magnitude and angle:
v0 = 5; theta = 80*np.pi/180
vx = v0*np.cos(theta); vy = v0*np.sin(theta)

U0 = [x, vx, y, vy]

solver= ForwardEuler(f)
solver.set_initial_condition(U0)
time_points = np.linspace(0, 1.0, 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]

plt.plot(x, y)
plt.show()
```

# 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