12.010 Computational Methods of Scientific Programming 2021

Lecture 7: ODE solutions

Summary

- Ordinary differential equation (ODE) solutions:
- Methods from scipy
- Methods from scratch.

ODE solvers

- ODE solvers like odeint() are powerful for solving all sorts of differential equations involving gradients with respect to a single variable.
- Interesting examples include
  - Lorenz equations – a toy model for thinking about atmospheric predictability
  - Lotka-Volterra equations – a toy model for thinking about predator-prey cycles in ecosystems
  - Ballistic equations – the trajectory of golf ball or rocket

- We can solve these systems ourselves using simple methods (e.g., Euler forward), but in general, an ODE solver will have smart techniques to automatically preserve accuracy as well as it can.

ODE solvers – DIY approach

- To illustrate ODE solver concept we first code a simple solver explicitly by hand for the Lorenz63 equations

$$\frac{dx}{dt} = \sigma(x - y)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

This set of equations was devised by Ed Lorenz (https://en.wikipedia.org/wiki/Edward_Norton_Lorenz) at MIT in the 1960s, with computational help from Ellen Fetter (https://en.wikipedia.org/wiki/Ellen_Fetter).

The equations were devised as a simple model for reasoning about chaotic phenomena in atmospheric dynamics.

three state variables x, y and z that vary in time t with parameters rho, sigma and beta being used to explore model behavior.

ODE solvers – DIY approach

- We can define a simple function for L63 equations
  - Note use of docstring as a reminder for our future selves!

```
def lorenz63( x, y, z, σ=10., ρ=28., β=8./3. ):
    """
    Function to evaluate the Lorenz 63 time derivative equation for
    state and parameters.
    Arguments:
    x, y, z: x,y,z values.
    σ, ρ, β: static parameters α, ρ and β.
    """
    dxdt=σ*(y-x)
    dydt=x*ρ-x*z-y
    dzdt=x*y-β*z
    return dxdt,dydt,dzdt
?lorenz63
```

```
Signature: lorenz63(x, y, z, σ=10.0, ρ=28.0, β=2.6666666666666665)
Docstring:
Function to evaluate the Lorenz 63 time derivative equation for a gi
state and parameters.
Arguments:
x, y, z: x,y,z values.
σ, ρ, β: static parameters α, ρ and β.
```

ODE solvers – DIY approach

- Now lets define a loop that can timestep forward the equations using an "Euler forward" scheme

$$\phi^{n+1} = \phi^n + \Delta t f(\phi^n)$$

Here ϕ is a vector of L63 model state [x,y,z], and $f(\phi^n)$ is our `lorenz63()` function.

```
import numpy as np
x0,y0,z0=0.,1.,1.05; Δt=0.01 # Initial conditions and p
nsteps=10000;
x=np.zeros(nsteps+1);x[0]=x0
y=np.zeros(nsteps+1);y[0]=y0
z=np.zeros(nsteps+1);z[0]=z0
for i in range(nsteps):
        dxdt,dydt,dzdt=lorenz63( x[i], y[i], z[i] )
        x[i+1]=x[i]+dxdt*Δt
        y[i+1]=y[i]+dydt*Δt
        z[i+1]=z[i]+dzdt*Δt
```

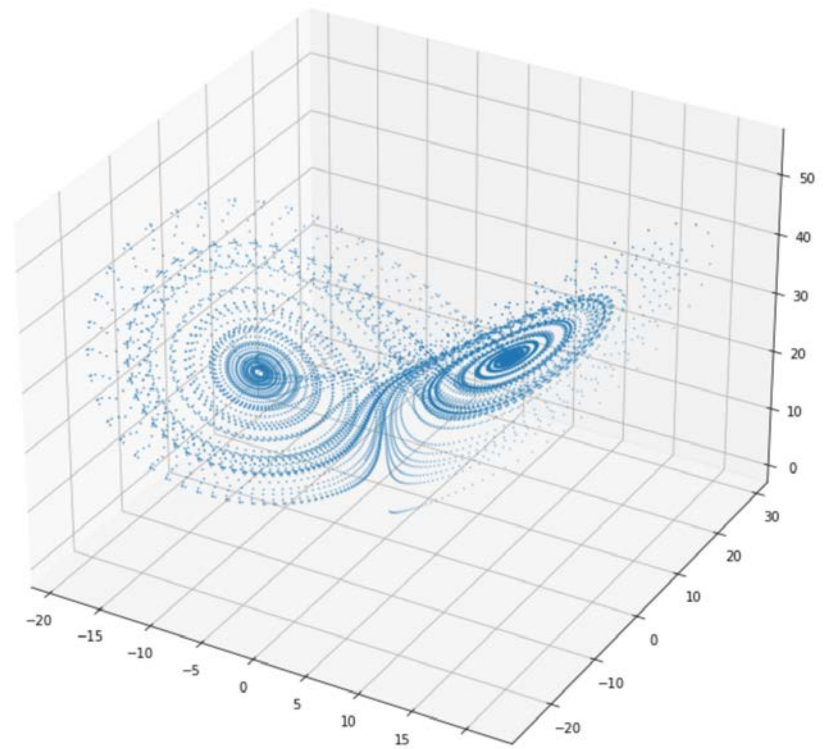
The superscript ^{n} denotes discrete time-levels, separated by time-step Δt.
The loop computes time series of values for x,y and z.

ODE solvers – DIY approach

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
plt.figure(figsize=(16, 12))
ax=plt.axes(projection='3d')
ax.scatter3D(x,y,z,s=0.5);
```

- From the time series of values of x, y,z we can plot the solution

- Note – for some reason there is no plt.scatter3D() function, so we have to use a function tied to the axes object.



ODE solvers – DIY approach

- To make this more like “odeint” we can create a “stepper” function that can operate on any function that return derivatives.
- In this case, the stepper function evaluates the Euler forward loop.

```
def euler_forward_stepper(f, u0,nt,dt, params={}):
    """
    Function euler_forward_stepper steps forward for n steps
```

```
    # Set up initial state
    nf=len(u0)
    u=np.zeros( (nt+1,nf) )
    dudt=np.zeros( nf )
    for i in range(nf):
        u[0,i]=u0[i]

    # Step forward
    for n in range(nt):
        dudt=f(*u[n,:],**params)
        u[n+1,:]=u[n,:]+np.array(dudt)*dt

    # return result
    return u
```

ODE solvers – DIY approach

- Now we can pass the `lorenz63()` function into the `euler_forward_stepper()` "black-box"
- We can sun the model twice, for slightly different initial conditions

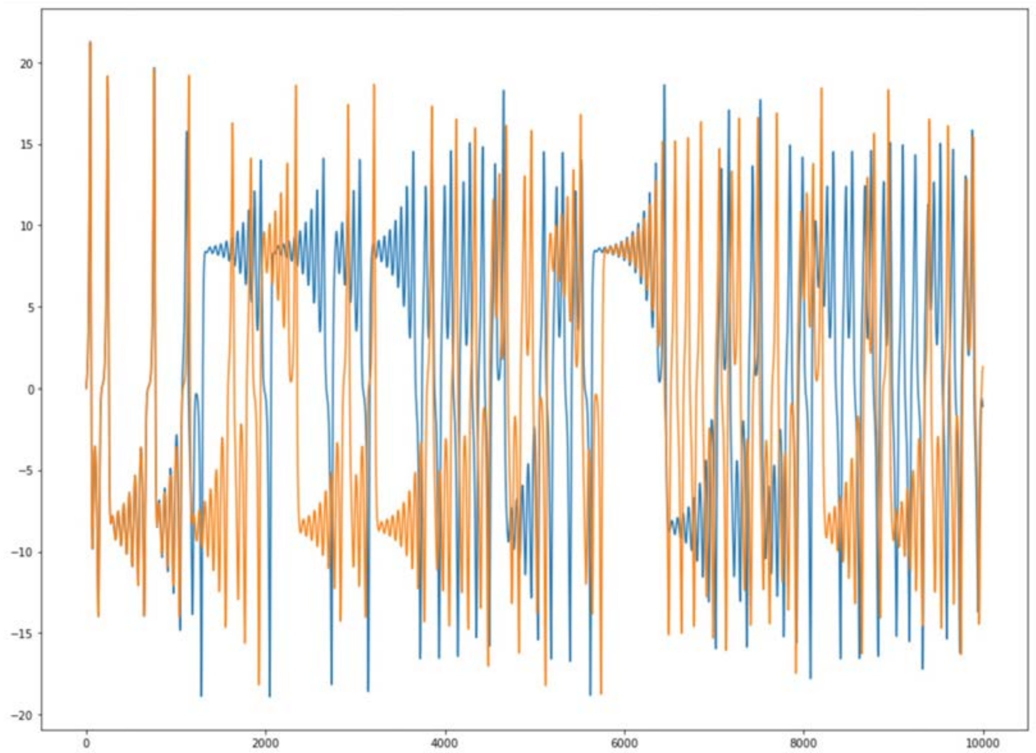
- This will step forward the ODE a specified number of steps using an Euler forward method from some initial state. The DIY stepper has a fixed timestep and a fixed method.
- odeint is similar but it has more internal smarts to select timesteps and methods.

```
nsteps=10000;
dt=0.01;
eps=1.e-4;
u0=euler_forward_stepper(lorenz63, np.array([0.,1.,1.05]), nsteps, dt);
u1=euler_forward_stepper(lorenz63, np.array([0.+eps,1.,1.05]), nsteps, dt);
```

ODE solvers – DIY approach

- Finally we can plot the solution for two very close sets of initial conditions.
- The solution tracks closely for some time but then diverges significantly.
- This is a major reason why forecasting the weather (especially beyond 14 days) is hard!

```
plt.figure(figsize=(16, 12))
plt.plot(u0[:,0]);plt.plot(u1[:,0]);
```

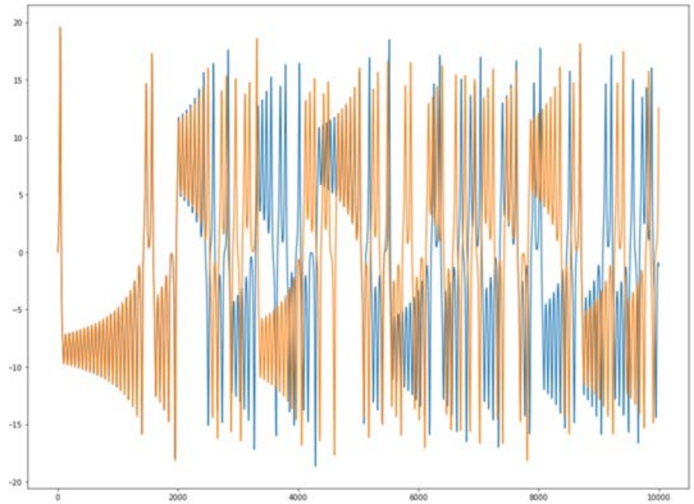


ODE solvers – using Scipy

- DIY ODE solver is OK, but -
- May be unstable for longer timesteps.
- Only has one method
- The method is not very accurate, so it requires a small timestep
- `scipy.integrate.odeint()` generalizes to allow more advanced methods, automated step size…

```
from scipy.integrate import odeint
def dfdy(y,t):
    return lorenz63( y[0], y[1], y[2] )
ys0=odeint(dfdy, np.array([0., 1., 1.05]), np.array([*range(0,10000)])*0.01)
ys1=odeint(dfdy, np.array([0.+eps, 1., 1.05]), np.array([*range(0,10000)])*0.01)

plt.figure(figsize=(16, 12))
plt.plot(ys0[:,0])
plt.plot(ys1[:,0])
```

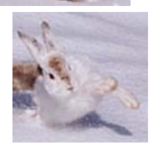


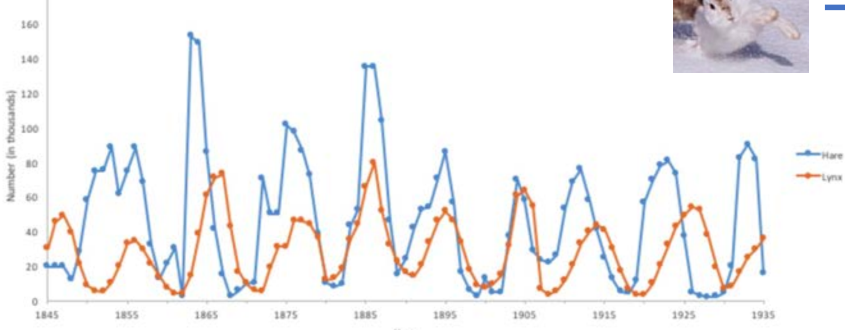
Note – the solution differs from the Euler forward DIY setup!

ODE solvers – using Scipy

Now we can play with any ODE equations numerically

e.g predator-prey model (Lotka and Volterra – 1925, 1926)





$$\frac{dh}{dt} = \alpha h \; - \beta h l$$

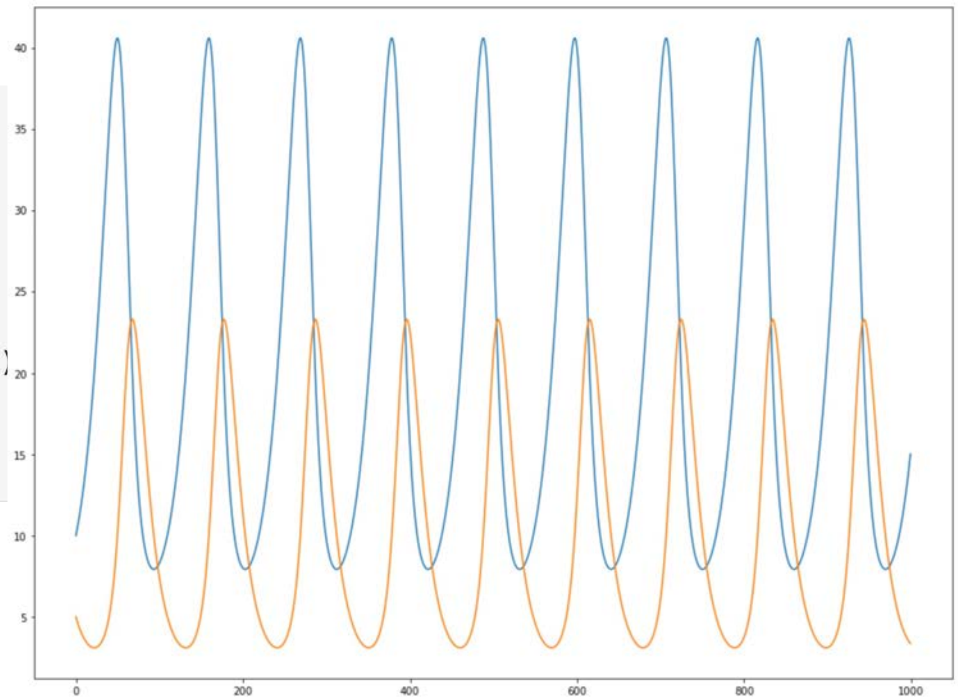
$$\frac{dl}{dt} = -\gamma l + \epsilon \beta h l$$

h – hare population
l – lynx population
α – hare growth rate
γ – lynx death rate
ϵ – lynx growth per hare killed

ODE solvers – using Scipy

Lotka Volterra code

```
# Lotka-Volterra
def lv(y,t,α=1.,β=0.1,γ=1.5,ϵ=0.75):
    u_prey=y[0]
    v_pred=y[1]
    dudt =  α*u_prey - β*u_prey*v_pred
    dvdt = -γ*v_pred + ϵ*β*u_prey*v_pred
    return dudt, dvdt
ys=odeint(lv, np.array([10, 5]), np.linspace(0, 50, 1000) )
plt.figure(figsize=(16, 12))
plt.plot(ys[:,0])
plt.plot(ys[:,1]);
```

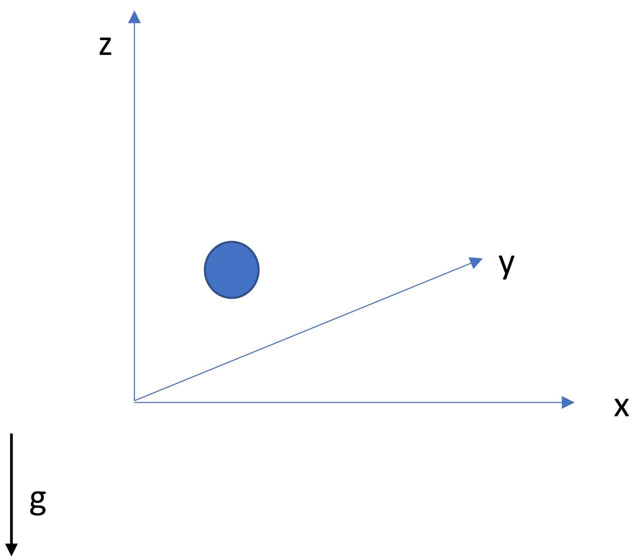


ODE solvers – using Scipy

- Try with notebook *Lec08-ode.ipynb*

ODE solvers – starting from scratch

- Consider throwing a ball



$$\frac{d^2z}{dt^2} = -g$$

1. No friction (in a vaccum!).

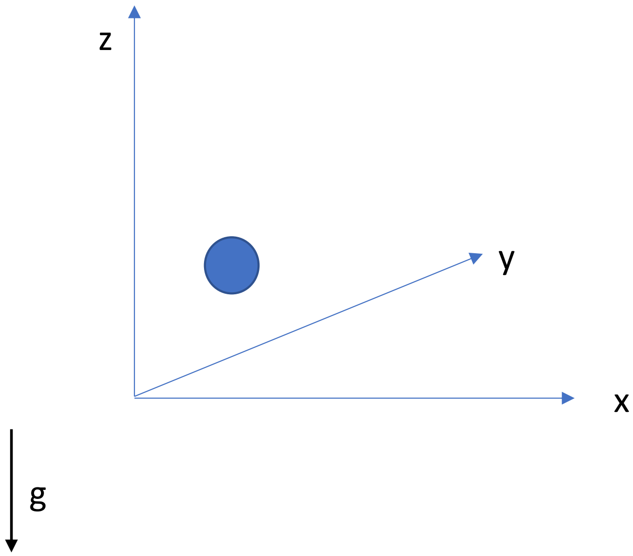
$$\frac{d^2y}{dt^2} = 0$$

$$\frac{d^2x}{dt^2} = 0$$

Lets try and write code to calculate the trajectory.

ODE solvers – starting from scratch

- Consider throwing a ball



2. With friction for F = (F_x, F_y, F_z)

$$\frac{d^2z}{dt^2} = -g - F_z$$

$$\frac{d^2y}{dt^2} = 0 - F_y$$

$$\frac{d^2x}{dt^2} = 0 - F_x$$

Lets try and write code to calculate the trajectory.

Summary

- Ordinary differential equation (ODE) solutions:
  - Introduce new variables for second—and higher-order derivatives. For example, for an acceleration equation, add velocity as a variable.
- Methods from scipy
- Methods from scratch.