Lecture 4

Time integration : Using SciPy

D. Dickinson
d.dickinson@york.ac.uk

York Plasma Institute, School of PET, University of York

Semester 1

Overview of course

This course provides a brief overview of concepts relating to numerical methods for solving differential equations.

Topics to be covered include:

- Integrating ODEs
  - ~~Explicit techniques~~
  - ~~Implicit techniques~~
  - Using Scipy to integrate ODEs
- Spatial discretisation
  - Finite differencing
  - Spectral methods
  - Finite elements
- Particle In Cell (PIC) approaches
- Continuum techniques

Notes available on the VLE.

Overview this lecture

This lecture will look at

- How to use SciPy to integrate equations.

Using SciPy to integrate

The SciPy module contains a range of sub-modules designed for working with a range of tasks commonly encountered in scientific programming.

Here we'll focus on the integrate submodule which provides tools both for integrating the area under a curve and for integrating ODEs of the standard form in time.

To use the sub-module we can use

```
import scipy.integrate as integrate
```

The specific routine we want to use can be imported using

```
from scipy.integrate import solve_ivp
```

There is also an older `odeint` interface and a class based interface, `ode`, which we won't cover here, that can by used with

```
from scipy.integrate import ode, odeint
```

Using SciPy to integrate

The solve_ivp routine can be used to integrate a system of ODEs defined by a passed function. I recommend you use

```
help(solve_ivp)
```

to explore the options provided by solve_ivp before attempting to use the routine.

[1] In other words the stability doesn't depend on what choice you make for t_eval

Using SciPy to integrate

The solve_ivp routine can be used to integrate a system of ODEs defined by a passed function. I recommend you use

```
help(solve_ivp)
```

to explore the options provided by solve_ivp before attempting to use the routine.
The basic inputs needed to use solve_ivp are

- **fun** The function which returns $d\boldsymbol{y}/dt$.
- **t_span** Two element array/list containing the start and end times of the integration.
- **y0** An array of initial conditions to use.

[1] In other words the stability doesn't depend on what choice you make for t_eval

Using SciPy to integrate

The solve_ivp routine can be used to integrate a system of ODEs defined by a passed function. I recommend you use

```python
help(solve_ivp)
```

to explore the options provided by solve_ivp before attempting to use the routine.
The basic inputs needed to use solve_ivp are

- **fun** The function which returns $d\boldsymbol{y}/dt$.
- **t_span** Two element array/list containing the start and end times of the integration.
- **y0** An array of initial conditions to use.
- **method** A name of a particular integrator to use.
- **t_eval** An array of time points at which we want to know $\boldsymbol{y}(t)$. It's useful to note that this *doesn't* correspond to the actual time steps used by the integrator like in our hand written integrator examples[1].

Look at points along the way to see structure e.g. oscillating, growing, decaying. Shows time history.

[1] In other words the stability doesn't depend on what choice you make for t_eval

Using SciPy to integrate : A simple example

Let's solve the exponential decay problem from the last lecture

```
from scipy.integrate import solve_ivp
from numpy import linspace, exp    give linearly spaced array of values between given numbers
import matplotlib.pyplot as plt
#Function to return dy/dt
def gradientFunc(curTime,curValue):    takes current time and state (from yesterday's function)
    return -10.*curValue    calculate what the F(y, t) is. Doesn't use time here, but takes it as generic form uses it
#Time for outputs
time=linspace(0, 1, 40)    gives 40 spaces between 0 and 1
y0=[10.] #Initial condition
result=solve_ivp(gradientFunc, [time[0],time[-1]], y0, t_eval=time) #Integrate
#Plot
plt.plot(time, result.y[0,:],'x', label='odeint')    numerical result
plt.plot(time, y0*exp(-10.*time), label='Analytic')    analytic solution
plt.legend() ; plt.show()
```

Using SciPy to integrate : A simple example

This is the output we get from the preceding code snippet.

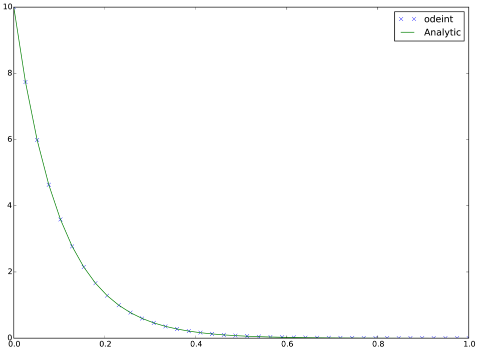


Figure: Comparison of numerical and analytic solution for exponential decay showing good agreement.

The numerical solution matches the analytic one very closely! Indeed solve_ivp can use some smart tricks, such as switching between different methods depending on how stiff it thinks your equation is and adapting the step size to keep errors within some tolerance.

Using SciPy to integrate : Further examples

In the previous example we hard coded the equation so it was always $\dot{y} = ay = -10y$, but what if we wanted to try different values of a?

```
def gradientFunc(curTime,curValue,aVal):
    return aVal*curValue
#Time for outputs
time = linspace(0,1,40)
y0 = [10.] #Initial condition
for aVal in [-10, -5, -2]:
    result = solve_ivp(gradientFunc, [time[0], time[-1]], y0,
                       t_eval = time, args=(aVal,)) #Integrate
    plt.plot(time, result.y[0,:],'x',label='odeint a='+str(aVal))
    plt.plot(time,y0*exp(aVal*time),label='Analytic a='
             +str(aVal))
plt.legend() ; plt.show()
```

pass it extra argument aVal so you don't need to hardcode a value in and need to manually change it throughout

dy/dt = a*y

args needs to be a 'collection' so no error

Here we've used the args keyword of solve_ivp to pass in extra parameters. See the solveivp3.py example on the VLE for another way to achieve this.

This is the output we get from the preceding code snippet.

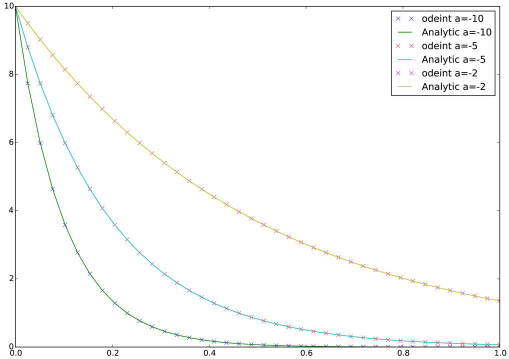


Figure: Comparison of numerical and analytic solution for various exponential decay problems showing good agreement in all cases.

Further examples : Lotka-Volterra

The *Lotka-Volterra* equations are a pair of 1st order differential equations.

Each one's rate over time =

How quickly prey reproduce - how quickly they die from predators

$$\frac{dx}{dt} = x\left(\alpha - \beta y\right), \quad \frac{dy}{dt} = -y\left(\gamma - \delta x\right)$$

How quickly they grow when eating prey - how quickly they die

Note these equations involve multiples of the evolving variables (yx), making the system non-linear.

These are also known as the predator-prey equations as one may consider x to represent the population of prey and y to represent the population of predators. These equations then describe how the populations evolve in time.

It's possible to roughly interpret the model parameters as follows

- α How quickly the prey reproduce
- γ How quickly the predators die
- β How quickly predators kill prey when they meet
- δ How quickly predators grow for every killed prey

Further examples : Lotka-Volterra

First define a function to return the derivatives

Just codes in the functions from previous slide

```
def lv(time, state, alpha, beta, gamma, delta):
    # inputs are system state and
    # the simulation time
    x = state[0] ; y = state[1]
    dxdt = alpha * x - beta*x*y
    dydt = -gamma*y + delta*x*y
    return [dxdt, dydt]
```

x is first thing in state, y is second thing

Note here how the second argument of the function is the state "vector" which contains both x and y values at the current time.

We expect the model parameters α, γ, β and δ to be passed to the derivative functions.

Will have to use args

Further examples : Lotka-Volterra

Now set the initial conditions and use solve_ivp to integrate

```
from scipy.integrate import solve_ivp
from numpy import linspace
initial = [5, 5]
t = linspace(0.0, 20, 200)
result = solve_ivp(lv, [t[0], t[-1]], initial,
                   t_eval = t, args=(1.0, 1.0, 3.0, 1.0))

import matplotlib.pyplot as plt
plt.plot(t, result.y[0,:], label="prey")
plt.plot(t, result.y[1,:], label="predator")
plt.legend() ; plt.show()
```

better to write alpha = 1.0, beta = 1.0 etc, and pass them in args as variable names instead of the numbers, clearer

Further examples : Lotka-Volterra

Running the previous chunks of code gives the following output

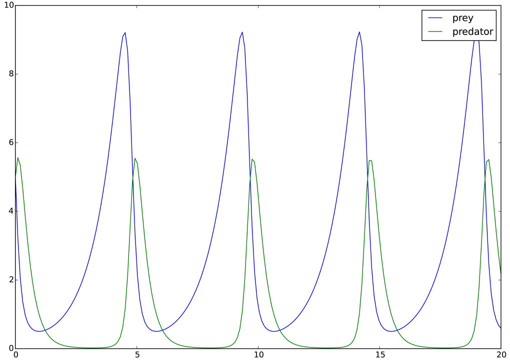


Figure: Population of predators and prey from numerical solution of Lotka-Volterra system.

This shows the periodic population growth of prey, leading later to a peak in the predator population and a resulting crash in the prey numbers.

See the file demo1.py on the VLE for the full Python script.

Further examples : Charged particle motion

Consider the electrostatic Lorentz force law:

$$\frac{d^2\boldsymbol{x}}{dt^2} = \boldsymbol{F}$$

normalised the mass

where we've normalised so we're looking at particles of unit charge and mass and hence $\boldsymbol{F} = \boldsymbol{E}$.

First split this into two 1st order ODEs

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v}, \quad \frac{d\boldsymbol{v}}{dt} = \boldsymbol{F}$$

This equation describes the motion of each charged particle in terms of the electric field, $\boldsymbol{E}$ (which sets $\boldsymbol{F}$). We can find force due to the electric field using Coulomb's law to find the force on a particle due to all others:

$$\boldsymbol{F}\left(\boldsymbol{x}\right) = \sum_{i=1}^{N} \frac{\boldsymbol{x} - \boldsymbol{x}_i}{\left(\boldsymbol{x} - \boldsymbol{x_i}\right)^3}$$

Superimpose force for each individual particle.
How far away are they, in which direction, and
what is their contribution to the force

Further examples : Charged particle motion

This is a relatively complicated example as:

1. It's a 2nd order ODE, so we have two evolving variables, $\boldsymbol{x}$ and $\boldsymbol{v}$.
2. Unless we're in 1D these variables are vectors.
3. We have N particles.

Let's restrict ourselves to 2D, so for each particle we need to evolve x, y, v_x and v_y. For the coded example let's also consider the case with just two particles, though it should be trivial to extend to larger N.

Further examples : Charged particle motion

Here we define a function to return the x and y components of the force.

```
def getForce(x,y):
    #Create array for force values
    N=len(x) ; Fx = zeros(N) ; Fy = zeros(N)
    #Now calculate the force on each particle
    for i in range(N):
        #Loop through all the other particles
        #to calculate their force on this particle
        for j in range(N):
            if i==j: continue #No force on self
            #Calculate distance
            dx = x[i] - x[j] ; dy = y[i] - y[j]
            R=sqrt(dx**2+dy**2)
            #Calculate force
            Fx[i] -= dx/R**3 ; Fy[i] -= dy/R**3
    return Fx, Fy
```

How many particles do we have; array nth long for x and y to later store force for each particle

2 loops (for each particle we have to think about the other ones)

Not very efficient, has to loop individually through each particle, twice!

Further examples : Charged particle motion

Now we can define the function to return the derivatives given the state.

```
def f(time, state):
    #Work out N from state
    N = len(state) // 4

    #Unpack variables
    x = state[0:N] ; y = state[N:(2*N)]
    vx = state[(2*N):3*N] ; vy = state[(3*N):]

    #Find out the force
    Fx, Fy = getForce(x,y)

    #Concatenate takes list of B, N length vectors
    #and returns a single B*N vector
    #Note dx/dt=vx, dy/dt=vy, dvx/dt=Fx and dvy/dt=Fy
    return concatenate([ vx, vy, Fx, Fy ])
```

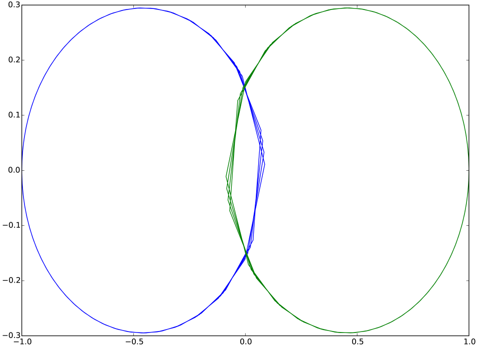
4 because each particle has 4 things: v_x, v_y, x, y (positions and velocities in those directions)
Double slash so result is integer

Further examples : Charged particle motion

Finally we can import the various functions, integrate and plot.

```
from numpy import zeros, sqrt, concatenate, linspace
from scipy.integrate import solve_ivp
#Initial conditions:
#Pcle 1 at (x,y)=(-1,0) with (vx,vy)=(0,0.2)
#Pcle 2 at (x,y)=(1,0) with (vx,vy)=(0,-0.2)
initial = [-1.0, 1.0, 0.0,0.0, 0.0, 0.0, 0.2,-0.2]
#Times of interest
t = linspace(0, 20, 200)
#Integrate
result = solve_ivp(f, [t[0], t[-1]], initial, t_eval = t)
#Plot
import matplotlib.pyplot as plt
plt.plot( result.y[0,:], result.y[2,:])
plt.plot( result.y[1,:], result.y[3,:])
plt.show()
```

Further examples : Charged particle motion



imperfect orbits could be representative of energy loss in a system

Figure: Plot of x-y location of the two particles over all times showing a roughly circular orbit around each other.

Putting the three previous blocks of code together and running generates the above plot. This shows the position of both particles at each of the requested times.

See the file demo2.py on the VLE for the full Python script.

As a quick aside let's look at how to make an animation of the particles motion. We'll modify the proceeding code slightly. First instead of using plt.plot we'll be a little more specific:

```
#Make a figure
fig1 = plt.figure()

#Make some empty lines
line1, = plt.plot([], [], 'ro')
line2, = plt.plot([], [], 'bx')

#Set x and y range
plt.xlim(result.y.min(),result.y.max())
plt.ylim(result.y.min(),result.y.max())
```

Here we make a figure, add some empty lines to it and set the axis ranges.

Next we need to define a function which will draw a single frame of the animation.

```
#Define a function to draw a frame of animation
def update_lines(num, res, l1, l2, ntrail=1):
    #Decide what the minimum index is
    mn=max([0,num-ntrail])

    #Update the line data
    l1.set_data(res[0,mn:num],res[2,mn:num])
    l2.set_data(res[1,mn:num],res[3,mn:num])
    return l1,l2,
```

This function takes the frame index (num), the results data (res), the two lines on the plot (l1 and l2) and updates the line's data values. We've also added an optional variable (ntrail) which will allow us to draw ntrail-1 previous frames as well (to leave a little trail).

Further examples : Charged particle motion

Finally we import the matplotlib.animation submodule and use its function FuncAnimation to draw the animation (passing the updateLines function along with other options).

```
import matplotlib.animation as ani
#Create the animation
line_ani = ani.FuncAnimation(fig1, update_lines, len(t),
           fargs=(result.y, line1, line2, 5),
           interval=5, blit=False, repeat=True)
#Uncomment the below line to get a movie generated to file
#line_ani.save('pcles.mp4')

#Actually show animation
plt.show()
```

See the file demo3.py on the VLE for the full Python script (note saving animation relies on certain external tools, which you may not have).