Problem Class 1: Using solve_ivp to integrate ODEs

David Dickinson d.dickinson@york.ac.uk

Semester 1

1 Recap of solve_ivp basics

In lecture 4 we looked at some examples of using the solve_ivp routine provided by the scipy integrate sub-module. To get started the first thing is to import this routine using a line like:

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

To see how to use solve_ivp you can do help(solve_ivp). The basic inputs needed to use solve_ivp are

- fun The function which provides returns $d\boldsymbol{y}/dt$. Must take at least, the current state, $\boldsymbol{y}$, and the current time, t.
- t_span Two element array/list containing the start and end times of the integration.
- y0 An array of initial conditions to use (one for each element of $\boldsymbol{y}$).

Consider the following simple ODE

$$\frac{df}{dt} = -af \tag{1}$$

which has the solution $y = y_0 \exp(-at)$. We can solve this numerically using solve_ivp as follows:

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, exp
import matplotlib.pyplot as plt

#Here we define our function which returns df/dt = -a*f
#Note we've assumed that a=10
def dfdt(curT,curF):
    #We don't do anything with curT
    return -10*curF

#Now define the times at which we want to know the result
time = linspace(0,1,40)

#Set the initial condition
f0 = [10.]

#Now we can solve the equation
result = solve_ivp(dfdt,[time[0], time[-1]], f0, t_eval = time)

#Plot
plt.plot(time, result.y[0,:],'x', label='odeint')
plt.plot(time, f0*exp(-10.*time), label='analytic')
plt.xlabel('Time'); plt.ylabel("f")
plt.legend() ; plt.show()
```

This should produce the plot shown in fig. 1

Note that here we have assumed that $a = 10$. We can make this code more general by making use of the args argument to solve_ivp. This allows us to pass extra data to the function that solve_ivp is calling and is very useful. The following demonstrates how this can be used to allow the reuse of the function dfdt for different values of a.

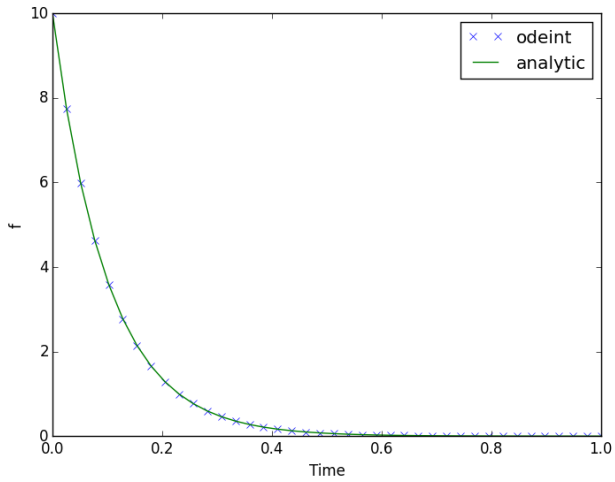

Figure 1: Output of the simple exponential decay example.

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, exp
import matplotlib.pyplot as plt

#Here we define our function which returns df/dt = -a*f
#Note we can pass in a, but it defaults to 10
def dfdt(curT,curF,a=10):
    #We don't do anything with curT
    return -a*curF

#Now define the times at which we want to know the result
time=linspace(0,1,40)

#Set the initial condition
f0=[10.]

#Which a values do we want to use?
avals=[0,1,10,30,-1]

for a in avals:
    #Now we can solve the equation for this a
    #Note we need a comma after the a here to
    #make sure args is a *tuple*
    result = solve_ivp(dfdt,[time[0], time[-1]], f0,
                       t_eval = time, args=(a,))

    plt.plot(time,result.y[0,:],label='a = '+str(a))

plt.xlabel('Time'); plt.ylabel("f")
plt.legend(loc='best') ; plt.show()
```

This should produce the plot shown in fig. 2

Optional exercise:

1. Implement the modified example above and check you can reproduce the figure.
2. Modify this code to solve the following up to $t = 10$ in 400 steps for $b = 50$ and $a = 30$. The output is shown in fig. 3.
$$\frac{df}{dt} = -a\sin(f) - b\cos(t)$$
3. Modify the function dfdt to take both a and b as arguments.

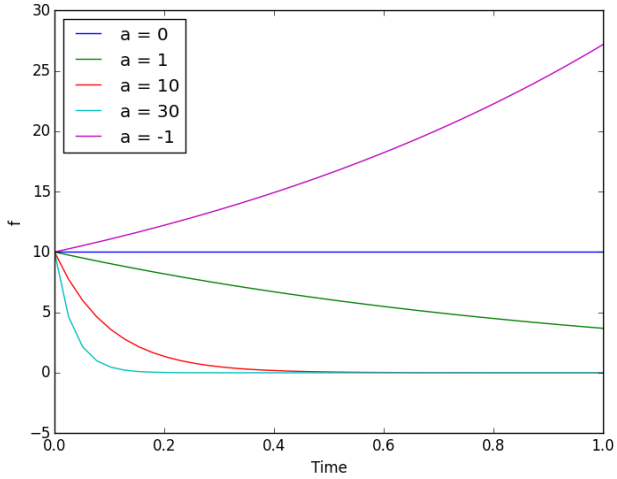

Figure 2: Output of the modified exponential decay example.

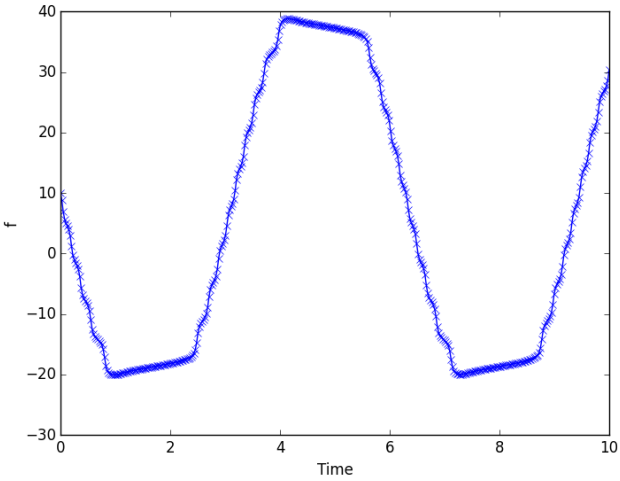

Figure 3: Solution of $df/dt = -30\sin(f) - 50\cos(t)$

2 Exercise: Mass on a spring, 1D system

The force exerted by a spring on a mass m attached to the end is given by Hooke's law:

$$F(t) = m\frac{d^2x(t)}{dt^2} = -kx(t) \tag{2}$$

where k is the spring constant and x measures the compression/stretch of the spring through the position of the spring's free end. In order to solve eq. (2) using solve_ivp for $x(t)$ we must first rewrite it in the form $df/dt = F(t)$. To do this let's define

$$\frac{dx}{dt} = v$$

so that we have

$$\frac{dv}{dt} = -k\frac{x}{m} = \frac{d^2x}{dt^2}$$

This is now in a form that solve_ivp can deal with, we have converted our second order ODE into two first order ODEs.

Tasks:

1. Write a Python script to solve eq. (2) and plot $x(t)$ and $v(t)$. Solve when $m = 0.01$ and $k = 0.5$.
2. Calculate the kinetic and potential energy of the system as a function of time (see the Wikipedia age on springs) and plot these as a function of time. How does the total energy vary with time?

3. Consider some damping to the system. This is represented by adding $-cv\,(t)$ to the right hand side of eq. (2), where c is the damping coefficient. Derive the new set of equations, modify your previous code to include this and solve with $c = 0.005$.
4. How does this modify the energetics of the system?
5. *Optional*: Try to write a python program to solve this system without using solve_ivp (e.g. using the simple explicit Euler method from the lectures). Hint: The Euler update is $y\,(t + \delta t) = y + \delta t dy/dt$ and you can reuse most of your existing code. What happens to the energy plot without damping, how does this vary with the time step δt?

An example solution is given in section 5.1 if you want some tips or if you get stuck.

3 Exercise: Simple pendulum

The equation of motion for a pendulum of lenth l is given by

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta) \tag{3}$$

where g represent the acceleration by gravity and θ is the angular displacement. This is often solved using the "small angle approximation" where $\theta \ll 1$ so $\sin(\theta) \approx \theta$.

Tasks:

1. Write a Python script to solve eq. (3) both with and without the small angle approximation (not at the same time). Plot θ as a function of time for different starting displacements. How well does the small angle approximation work?

An example solution is given in section 5.2 if you want some tips or if you get stuck. There's also an example of how to animate the results.

4 Exercise: Double pendulum

So far the examples have been concerned with the motion of a single object. Now we're going to look at something which gives some slightly more interesting results, the double pendulum. Perhaps the easiest way to get the equations of motion of the double pendulum is through the Lagrangian approach, which provides equations for the time derivative of the two angular displacements, θ_1 and θ_2, in terms of the current displacement and the momenta, p_1 and p_2. The system is closed with equations for the time derivatives of p_1 and p_2. The full set of equations, assuming equal mass and length pendulums, are thus

$$\frac{d\theta_1}{dt} = \frac{6}{ml^2}\frac{2p_1 - 3p_2\cos(\theta_1 - \theta_2)}{16 - 9\cos^2(\theta_1 - \theta_2)} \tag{4}$$

$$\frac{d\theta_2}{dt} = \frac{6}{ml^2}\frac{8p_2 - 3p_1\cos(\theta_1 - \theta_2)}{16 - 9\cos^2(\theta_1 - \theta_2)} \tag{5}$$

$$\frac{dp_1}{dt} = -\frac{ml^2}{2}\left[\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}\sin(\theta_1 - \theta_2) + 3\frac{g}{l}\sin(\theta_1)\right] \tag{6}$$

$$\frac{dp_2}{dt} = \frac{ml^2}{2}\left[\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}\sin(\theta_1 - \theta_2) - \frac{g}{l}\sin(\theta_2)\right] \tag{7}$$

Tasks:

1. Write a Python script to solve the double pendulum system.
2. How does the result change when you change the initial conditions slightly?
3. Look at the documentation for solve_ivp. What do the parameters rtol and atol control? How sensitive are the results of the double pendulum simulation to these parameters? Go back and see how sensitive your previous results are to these parameters.

An example solution is given in section 5.3.

5 Example solutions

5.1 Hooke's Law

The below is a sample code to solve eq. (2), the output is given in fig. 4.

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, exp
import matplotlib.pyplot as plt

#Here we define our function which returns d[x,v]/dt = [v,-kx/m]
def dfdt(curT,curF,springK=1,mass=1,damping=0.01):
    #Unpack values
    x = curF[0] ; v = curF[1]
    #Calculate time derivative of two terms
    dxdt = v  #dx/dt=v
    dvdt = -(springK*x/mass)-(damping*v/mass) # dv/dt = -(k*x/m)-(c*v/m)
    return [dxdt,dvdt]

#Now define the times at which we want to know the result
time=linspace(0,10,400)

#Set the initial conditions
x0=0.4 ; v0=0. ; f0=[x0,v0]

#Set the spring and mass parameters
mass = 1.0e-2 ; k = 0.5 ; damping = 0.005

#Now we can solve the equation
result = solve_ivp(dfdt,[time[0], time[-1]], f0,
                t_eval = time, args=(k,mass,damping))

#Extract x and v
x_of_t = result.y[0,:] ; v_of_t = result.y[1,:]

##Plot
plt.plot(time,x_of_t,label='x(t)') ; plt.plot(time,v_of_t,label='v(t)')
plt.xlabel('Time'); plt.ylabel("x and v") ; plt.legend(loc='best') ; plt.show()

#Energy plot -- Uncomment the below to produce plot (comment out above plots)
##Define a function that returns the kinetic, potential and total energy
#def getEnergy(x,v,k,mass):
#    kE=0.5*mass*v*v ; pE=0.5*k*x*x ; tE=kE+pE
#    return kE,pE,tE

#ke,pe,te=getEnergy(x_of_t,v_of_t,k,mass)
#plt.plot(time,ke,label="Kinetic energy")
#plt.plot(time,pe,label="Potential energy")
#plt.plot(time,te,label="Total energy")
#plt.xlabel('Time'); plt.ylabel("Energy")
#plt.legend(loc='best') ; plt.show()
```

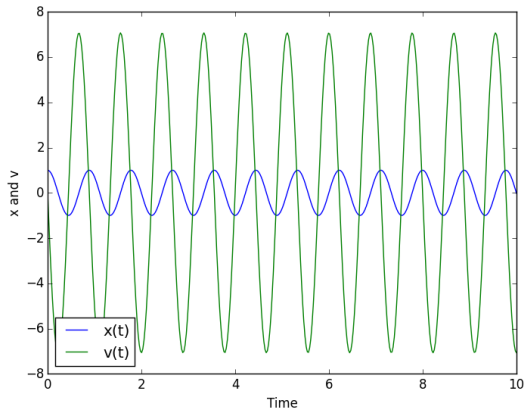

Figure 4: Solution of Hooke's law, eq. (2), without damping.

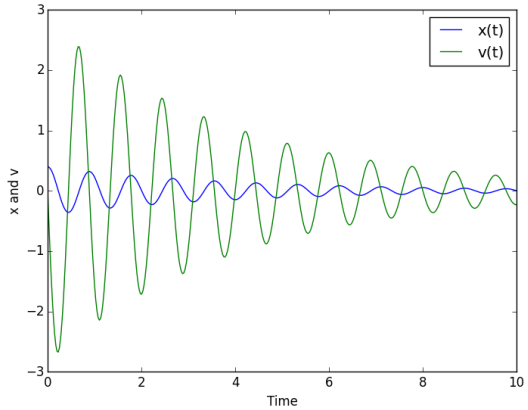

Figure 5: Solution of Hooke's law, eq. (2), with damping.

5.2 Simple pendulum

The below is a sample code to solve eq. (3) with and without the small angle approximation, the output is given in fig. 8.

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, sin
import matplotlib.pyplot as plt

#Here we define our function which returns d[theta,omega]/dt = [omega,-g theta/l]
def dfdt(curT,curF,length=1.0,g=9.81,useSmall=False):
    #Unpack values
    theta = curF[0] ; omega = curF[1]

    #Calculate time derivative of two terms
    dthetadt = omega
    if useSmall:
        domegadt = -g*theta/length
    else:
        domegadt = -g*sin(theta)/length
    return [dthetadt,domegadt]

#Now define the times at which we want to know the result
time=linspace(0,10,40)

#Set the initial conditions, note theta is in radians
theta0=0.5 ; omega0=0. ; f0=[theta0,omega0]

#Set the parameters
g = 9.81 ; length = 2
```

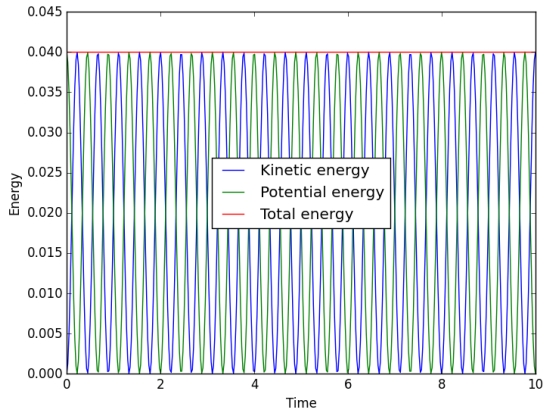

Figure 6: Energy breakdown in solution of Hooke's law, eq. (2), without damping. Total energy is conserved when there is no damping.

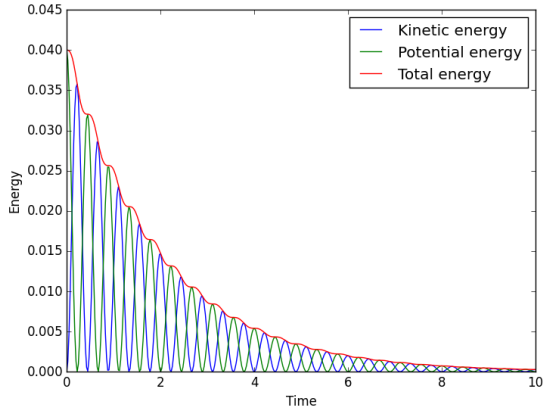

Figure 7: Energy breakdown in solution of Hooke's law, eq. (2), with damping. Total energy is not conserved when there is damping.

```
#Now we can solve the equation
result = solve_ivp(dfdt, [time[0], time[-1]], f0,
                   t_eval = time, args=(length,g))
plt.plot(time,result.y[0,:],label=r'$\theta(t)$ -- Full')
plt.plot(time,result.y[1,:],label=r'$\omega(t)$ -- Full')

result = solve_ivp(dfdt, [time[0], time[-1]], f0,
                   t_eval = time, args=(length,g,True))
plt.plot(time,result.y[0,:],'--x',label=r'$\theta(t)$ -- Small angle')
plt.plot(time,result.y[1,:],'--x',label=r'$\omega(t)$ -- Small angle')
plt.xlabel('Time'); plt.ylabel(r"$\theta$ and $\omega$")
plt.legend(loc='best') ; plt.show()
```

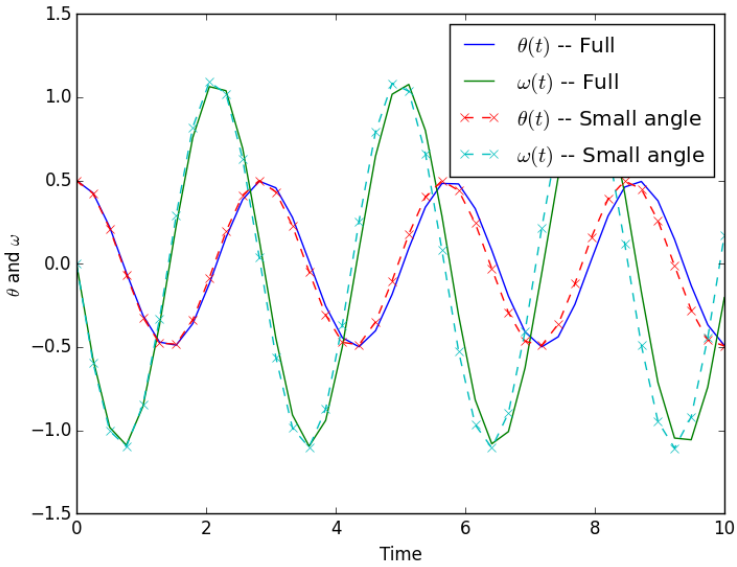

Figure 8: Motion of simple pendulum defined by eq. (3), with and without the small angle approximation.

The following gives a demonstration of how to do an animation using the result of the integration (try making this use the small angle approximation!).

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, sin, cos
import matplotlib.pyplot as plt
import matplotlib.animation as ani

#Here we define our function which returns d[theta,omega]/dt = [omega,-g theta/l]
def dfdt(curT,curF,length=1.0,g=9.81,useSmall=False):
    #Unpack values
    theta = curF[0] ; omega = curF[1]

    #Calculate time derivative of two terms
    dthetadt = omega
    if useSmall:
        domegadt = -g*theta/length
    else:
        domegadt = -g*sin(theta)/length
    return [dthetadt,domegadt]

#Now define the times at which we want to know the result
time=linspace(0,10,400)

#Set the initial conditions
theta0=3.5; omega0=0.; f0=[theta0,omega0]

#Set the parameters
g = 9.81 ; length = 2

#Now we can solve the equation
result = solve_ivp(dfdt, [time[0], time[-1]], f0,
                   t_eval = time, args=(length,g))

#Plot
fig1 = plt.figure(); plt.xlabel(r"$x$"); plt.ylabel(r"$y$")
line1 = plt.plot([],[],'-') #The line connecting origin and bob
line2 = plt.plot([],[],'go') #Pendulum bob

#Calculate positions and set the plot range
plt.xlim(-length,length); plt.ylim(-length,length)
```

```
fig1.gca().set_aspect('equal')

#Function to get pendulum x-y position given theta and length
def pendPos(theta,length):
    x = length*sin(theta) ; y = length*cos(theta)
    return x,y
#Function to draw the pendulum
def drawPendulum(num,res,l1,l2,length=1):
    x,y = pendPos(res[0,num],-length)
    l1.set_data([0,x],[0,y]) ; l2.set_data(x,y)
    return l1,l2,
#Make animation and then show it
line_ani = ani.FuncAnimation(fig1,drawPendulum,len(time),
    fargs=(result.y,line1[0],line2[0],length),
    interval=50, blit=False, repeat=False)
plt.show()
```

5.3 Double pendulum

```
#Import useful routines from modules
from scipy.integrate import solve_ivp
from numpy import linspace, sin, cos, array
import matplotlib.pyplot as plt
import matplotlib.animation as ani

#Here we return the time derivatives of theta1,theta2,p1,p2
def dfdt(curT,curF,mass=1.0,length=1.0,g=9.81):
    #Unpack values
    theta1 = curF[0] ; theta2=curF[1] ; p1 = curF[2] ; p2 = curF[3]
    cDif = cos(theta1-theta2) ; sDif = sin(theta1-theta2)
    dth1 = (6/(mass*length*length))*(2*p1-3*p2*cDif)/(16-9*cDif*cDif)
    dth2 = (6/(mass*length*length))*(8*p2-3*p1*cDif)/(16-9*cDif*cDif)
    dp1 = -0.5*mass*length*length*(dth1*dth2*sDif+3*g*sin(theta1)/length)
    dp2 = 0.5*mass*length*length*(dth1*dth2*sDif-g*sin(theta2)/length)
    return [dth1,dth2,dp1,dp2]

#Define times, initial conditions and parameters
time=linspace(0,10,400)
theta1_0=1.5 ; theta2_0=3.5 ; f0=[theta1_0,theta2_0,0,0]
mass = 1; g = 9.81 ; length = 2
result = solve_ivp(dfdt, [time[0], time[-1]], f0,
                   t_eval = time, args=(mass,length,g))

#Plot
fig1 = plt.figure(); plt.xlabel(r"$x$"); plt.ylabel(r"$y$")
line1 = plt.plot([],[],'-') #The line connecting origin and bob1
line2 = plt.plot([],[],'go') #Pendulum bob1
line3 = plt.plot([],[],'-') #The line connecting bob1 and bob2
line4 = plt.plot([],[],'ro') #Pendulum bob2

#Calculate positions and set the plot range
plt.xlim(-2*length,2*length); plt.ylim(-2*length,2*length)
fig1.gca().set_aspect('equal')

#Function to get pendulum x-y position given theta and length
def pendPos(theta,length,origin=[0,0]):
    x = array(origin[0]+length*sin(theta)) ; y = array(origin[1]+length*cos(theta))
    return x,y

#Function to draw the pendulum
def drawPendulum(num,res,l1,l2,l3,l4,length=1,ntrail=1):
    mn=max([0,num-ntrail])
    x,y = pendPos(res[0,mn:num+1],-length)
    l1.set_data([0,x[-1]],[0,y[-1]]) ; l2.set_data(x,y)
    x2,y2 = pendPos(res[1,mn:num+1],-length,[x,y])
    l3.set_data([x[-1],x2[-1]],[y[-1],y2[-1]]) ; l4.set_data(x2,y2)
    return l1,l2,l3,l4,

#Make animation and then show it
trailLen = 0# len(time) #Make this len(time) to keep all history
line_ani = ani.FuncAnimation(fig1,drawPendulum,len(time),fargs=(result.y,line1[0],
    line2[0],line3[0],line4[0],length,trailLen),interval=5,
    blit=False, repeat=False)
plt.show()
```