Problem Sheet 7
Some simple numerical methods to solve ODEs

Physical laws are expressed as mathematical relations which can then be solved under desired boundary conditions to obtain the results. Consider for example the relationship between electric and magnetic fields as represented by Maxwell's equations:

$$\nabla \times E = -\frac{\partial B}{\partial t}; \nabla \times B = \mu_0 J + \mu_0 \epsilon_0 \frac{\partial E}{\partial t}; \nabla \cdot B = 0$$

In order to find out, for example, the field distribution inside a resonator cavity, we need to solve these equations by applying suitable boundary conditions dictated by the geometry of the cavity and electromagnetic properties of the material used for making it. For simple geometries, analytic solutions are possible but in most cases numerical methods are essential and that is one reason why computers are important for physics. In this weeks exercise we will be looking at ways in which to visualise the fields encountered in electromagnetism and have a first look at some of the numerical tools available to solve these equations.

Note: *Scipy* contains modules for solving first order ordinary differential equations scipy.integrate.ode (a Nth order equation can also be solved using SciPy by transforming it into a system of first order equations), scipy.integrate.odeint provides a module for solving a system of first order ODEs. Partial differential equation solvers are not included in the Enthought distribution and are beyond the scope of this introductory course but modules such as FiPy (Finite volume approach) and SfePy (Simple Finite Element approach) are capable of solving this class of equation.

You will encounter differential equations that cannot be solved analytically. However, there are numerical methods that can be used to solve them with a computer. For many second-order, ordinary differential equations, a fairly simple approach called the Euler-Cromer Method is sufficient.

Notation We will often use shorthand notation for time derivatives where $\dot{x} = \frac{dx}{dt}$ and $\ddot{x} = \frac{d^2x}{dt^2}$. In other words, the number of dots indicates the number of derivatives. Since we'll be finding the value of a function $x(t)$ at discrete values of t, we can label those times with integer subscripts as $t_1 = t_0 + \Delta t, t_2 = t_0 + 2 \cdot \Delta t, \ldots, t_i = t_0 + i \cdot \Delta t$, where t_0 is the initial time and Δt is the time interval between the subsequent solutions. We will label the values of the function as , $x_0 = x(t_0), x_1 = x(t_0 + \Delta t), \ldots, x_i = x(t_0 + i \cdot \Delta t)$. Similarly, the notation for derivatives is $\dot{x}_i = \dot{x}(t_i)$ and $\ddot{x}_i = \ddot{x}(t_i)$.

The form of the second-order differential equations is:

$$\frac{d^2x}{dt^2} = f(x, \frac{dx}{dt}, t)$$

or

$$\ddot{x} = f(x, \dot{x}, t)$$

where the initial values of the function x_0 and first derivative $\dot{x}_0$ are known at the initial time t_0. We want to find the function at later times.

Euler Method

The simplest numerical method to solve differential equations is the Euler Method. If time is divided into small steps Δt, then

$$\dot{x}(t) \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{1}$$

and

$$\ddot{x}(t) \approx \frac{\dot{x}(t+\Delta t) - \dot{x}(t)}{\Delta t} \tag{2}$$

Suppose that the solution is known at one time. Rearranging equation (1), the approximate solution after a step forward in time is $x(t+\Delta t) \approx x(t) + \dot{x}(t)\Delta t$ To find the solution at another step forward in time will require knowing the first derivative at that later time. According to equation (2), that derivative is approximately $\dot{x}(t+\Delta t) \approx \dot{x}(t) + \ddot{x}(t)\Delta t$, where we use $\ddot{x}_i = f(x_i, \dot{x}_i, t_i)$. These final two equations can be used iteratively to find the approximate solution at later times. Unfortunately, this method is not very accurate after a while.

Euler-Cromer Method

A simple modification to the method described above greatly improves the accuracy of the numerical solution. First, find the approximate first derivative after a step forward in time as before,

$$\dot{x}(t+\Delta t) \approx \dot{x}(t) + \ddot{x}(t)\Delta t \tag{3}$$

Next, the value of the function at a later time is approximated by

$$x(t+\Delta t) \approx x(t) + \dot{x}(t+\Delta t)\Delta t \tag{4}$$

where the approximate first derivative at this later time is used. Note the slight difference between this approach and the Euler Method. The *previous* value of $\ddot{x}$ (which may depend on the previous values of x and $\dot{x}$ at the previous time) is used to calculate a new value of $\dot{x}$. However, the *new* value of $\dot{x}$ is used to calculate the new value of x.

1 PS7Ex1:

The example code *PS7Ex1.py* finds the solution to the differential equation

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

for $k = 5$, which should be a sinusoidal function. In the program, the first and second derivatives of x are called v and a, respectively.

```
# PS7Ex1.py an Example of a Euler-Cromer method to solve an ODE
#  Example for a=-kx, where 'a' is the second, and 'v' is the first
    differntial of 'x'
import matplotlib.pyplot as plt
# Make empty lists
tlist = []
xlist = []
vlist = []
#The initial values and constant are set next.
# Set inital values and constants
t = 0.0 # initial time=0
dt = 0.001 # time step
x = 10.0 # initial 'x'
v = 0.0 # initial 'v'
tf = 10.0 # end time
k = 5.0

#The initial values are appended to the lists.
# Append inital values to the lists
tlist.append(t)
xlist.append(x)
vlist.append(v)
#Inside of a loop, equations (3) and (4) are used to find values of the
    derivative and the function (in that order) at the next time.
#The type of loop used depends on how you want to determine when the program
    will stop.
#The new values are appended to the lists.
while t<tf:
    # Calculate new values
    a = -k*x # Calculate 'a' using the current 'x' and or 'v'
    v = v + a*dt  # Use the value 'a' to update 'v' ~ v(t+dt) eqn. [3]
    x = x + v*dt # calculate new 'x' using new 'v' eqn. [4]
    t = t + dt    # Advance the time by a step dt

    # Append new values to lists
    tlist.append(t)
    xlist.append(x)
    vlist.append(v)

# Plot the position vs. time
plt.figure()
plt.plot(tlist, xlist, ls='-', c='r')
plt.xlabel('time')
plt.ylabel('position')
plt.show()
```

PS7Ex1.py

a) Modify the example program *PS7Ex1.py* to find, and plot, the solution of the following differential equation:

$$\ddot{x} = -4x - 0.2\dot{x}$$

for t from 0 to 20 with the initial conditions $x(0) = 0$ and $\dot{x}(0) = 5$.

b) Further modify the example to plot both the Euler-Cromer and the Euler solution to

$$\ddot{x} = -4x - 0.2\dot{x}$$

Euler and Runge-Kutta applied to path of charged particle

The following code will calculate and display graphically the path of a charged particle (with initial position and velocity) under the influence of external magnetic and electric fields, the *Lorentz force* $f = q(\mathbf{E} + \mathbf{v} \times \mathbf{B})$. Download the example code from moodle and observe the effects of changing the initial parameters. In this example code the position x, y, z at $t + \delta t$ is obtained using the simplest of algorithms, Euler's method. This is a method for solving ordinary differential equations using the formula:

$$R_{n+1} = R_n + \delta t f(t_n, R_n)$$

, where $f(t_n, R_n) = \dot{R}(t)$ and in this case R is a vector $R[x, y, z]$. This advances a solution from R_n to R_{n+1} which approximates $R(t+\delta t)$. Note that the method increments a solution through an interval δt while using derivative information from only the beginning of the interval t. As a result, the step's error is $O(h^2)$.

A better solution can be obtained by using a method that calculates the average derivative over the interval such as the **Runge-Kutta** described in Appendix A of the mathematical methods notes. A fourth order *Runge-Kutta* is a method that evaluates the derivative four times; once at the initial point, twice at two trial midpoints and once at trial a endpoint, see Fig 1. The final value is evaluated from these derivatives using the equations:

$$k_1 = \delta t f(t_n, R_n)$$

$$k_2 = \delta t f(t_n + \frac{\delta t}{2}, R_n + \frac{k_1}{2})$$

$$k_3 = \delta t f(t_n + \frac{\delta t}{2}, R_n + \frac{k_2}{2})$$

$$k_4 = \delta t f(t_n + \delta t, R_n + k_3)$$

$$R_{n+1} = R_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

θ

Figure 1: RK4 Approximation is the weighted average of the gradient evaluated at 4 points

2 PS7Ex2:

a) Run the *field_simulation.py* code, explore how changing the initial conditions and time step, *dt*, effect the outcome.

b) The example code is written using lists and scalars, modify/re-write the code so that initial conditions are stored in arrays and that the force is calculated using a function such as:

```
import numpy as np
q=1
E=np.array([0.,1.,2.])
B=np.array([0,1,1.])

def Force(velocity):
    """Returns force vector"""
    return q * (E + np.cross(velocity, B) )
```

force.py

c) Modify the *field_simulation.py* code to use a 4th order *Runge-Kutta* method, plot and compare with the *Euler's* method result. Try increasing the total time interval to see how the errors vary with time. This can be done with either the initial code, or by modifying your answer to part b.

The *Runge-Kutta* method is discussed in most books on differential equations, a complete description can be found in *Numerical Recipes, The Art of Scientific Computing, Cambridge University Press, 3rd. Edition (2007), W.H.Press, S.A.Teukolsky, W.T. Vetterling, B.P. Flannery.* Although this book is not written for Python users it contains many good descriptions of the underlying methods that are common regardless of language.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
"""
The following example traces the movement of a charged particle under the
    influence of electric and
magnetic fields using Euler's Method. You can edit the program to change the
    components of the fields to see their effect on
the trajectory.
"""
Ex = 0.0 #  X Component of applied Electric Field
Ey = 2.0 #  Y Component of applied Electric Field
Ez = 0.0 #  Z Component of applied Electric Field
Bx = 0.0 # X Component of applied Magnetic field
By = 0.0
Bz = 4.0
m = 2.0 # Mass of the particle
q = 5.0 # Charge
x = 0.0 # initial position x
y = 0.0 # initial position y
z = 0.0 # initial position z
```

```
vx = 20.0 # initial velocity vx
vy = 10.0 # initial velocity vy
vz = 2.0 # initial velocity vz
a = []
b = []
c = []
t = 0.0
dt = 0.001 # time step (what happens if you increase this to 0.01?)
while t < 10: # trace path until time reaches value e.g. 10

    Fx = q * (Ex + (vy * Bz) - (vz * By) ) #Evaluates the 'x' componet of the
        force at 't' using Lorentz definition
    Fy = q * (Ey - (vx * Bz) + (vz * Bx) ) #Evaluates the 'y' componet of the
        force at 't'
    Fz = q * (Ez + (vx * By) - (vy * Bx) ) #Evaluates the 'z' componet of the
        force at 't'

    x = x + vx * dt # updates 'x' at (t+dt)
    y = y + vy * dt # updates 'y' at (t+dt)
    z = z + vz * dt # updates 'z' at (t+dt)

    vx = vx + Fx/m * dt #  Calculates v(t+dt) using v(t) and a(t),
        Acceleration = F/m; dv = a.dt
    vy = vy + Fy/m * dt
    vz = vz + Fz/m * dt

    a.append(x) # appending a list of x for plot
    b.append(y)
    c.append(z)
    t += dt # increment time

fig=plt.figure()
ax = Axes3D(fig)
ax.set_title("Path of charged particle under influence of electric and
    magnetic fields")
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.plot3D(a,b,c, color='red', label='path') #  creates 3-axis plot
ax.legend(loc='lower left')

plt.show()
```

field_simulation.py

Vector Fields

The code *vectorfield.py* is a basic script for generating a simple vector field around a point charge using Mayavi. Download the code from moodle and run it. Explore the functionality

of the Mayavi toolbar, the first button opens a dialog for the Mayavi pipeline that allows you to control many aspects of the view.

Figure 2: The Mayavi toolbar, the first button launches the Mayavi pipeline

3 PS7Ex3:

Modify the code such that the vector field is only calculated and plotted in the x,y plane. Add axis labels and a title to the plot and save a **.png** output of the plot, either via the script or by using the *mayavi* pipeline controls.

```
from numpy import mgrid, array, sqrt
from mayavi.mlab import quiver3d,axes
# this creates a 3D mesh grid which ranges from -100 to 100 in all
# coordinates, with intersections at every 20th value
# (-100, -80, -60, ...). See 'help mgrid' for details.
x,y,z = mgrid[-100.:101.:20., -100.:101.:20., -0.5:0.51:20.]
# We will now store the coordinates of our charge as a
# vector. All vectors in Python should be stored as arrays.
# Note that we are placing this charge in between the grid points.
# This is to get the best possible symmetry in this specific field
# If you place the charge _on_ the grid, you get a division of zero
# when calculating the E-field.
qpos = array([10.,10.,0.])
# the magnitude of our charge. This could be any number
# at the moment.
qcharge = 1.0
# Create a grid for the electric field. This has the size of
# x,y and z, but all the values are now set to zero.
Ex, Ey, Ez = x*0, y*0, z*0

# Calculate the x, y and z distance to the charge at every point in the grid:
rx = x - qpos[0]
ry = y - qpos[1]
rz = z - qpos[2]

# Calculate the distance at every point in the grid:
r = sqrt(rx**2 + ry**2 + rz**2)

# Calculate the field for each component at every point in the grid:
Ex = (qcharge / r**2) * (rx / r)
Ey = (qcharge / r**2) * (ry / r)
Ez = (qcharge / r**2) * (rz / r)

# Draw the vector field in 3D
```

```
#title
quiver3d(x,y,z,Ex,Ey,Ez)
axes(z_axis_visibility=False)
```

vectorfield.py

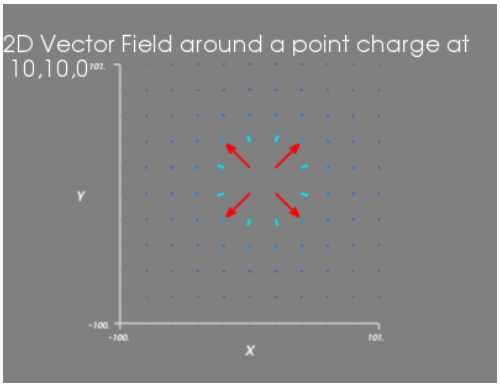

Figure 3: 2D vector plot with axis labels and title