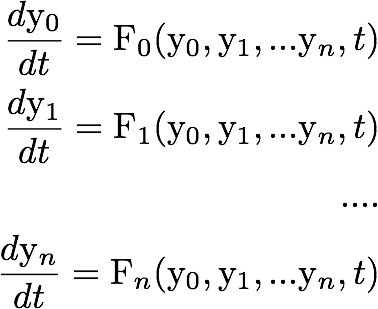
Solution to differential equations

The time dependence of any phenomena in physics is described by one or several **differential equations** which involves time derivatives. Many problems do not have an analytical solution because the system of differential equations is too complex to solve (e.g. N-body problems with gravity or fluid equations that govern local weather or the global climate).

Computers provide us with the computing power necessary to solve these complex problems numerically. In this project you will use one way of doing this which essentially consists in calculating time-derivatives of the important physical quantities of the problem and iterate over a succession of small time steps to calculate how these quantities change over time.

There are typically simple rules for how time-derivatives depend on other quantities. If we call yi (for i=0,…,n) the physical quantities (e.g. 3 components of velocity and 3 components of position is already 6 variables), and if we can write individual equations describing how each of these quantities vary over time:

where Fi are some function which may also explicitly depend on time. Other variables yi could be temperature, pressure, electric and magnetic fields, …really any physical quantity. The *Euler* and *Runge-Kutta* methods are the traditional techniques to resolve this kind of problem using small iterations in time Δt. The basic idea is that, starting with a set of initial conditions yi(t0)=yi0

we can calculate the quantity yi(tk+1) at the next time step tk+1 from the value of all other variables yi(tk) at time tk:

It is not the goal of this course to look at the details of these methods, however, it is important that you spend some time exploring how they work because the **SciPy** differential equation solver you will use is built on that. I would particularly recommend reading about the Euler method first (<https://en.wikipedia.org/wiki/Euler_method>) and Runge-Kutta next (<https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods>).

The Python SciPy differential equation solver is called **odeint** (<https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html>). It can solve the system of equations dyi/dt shown above by estimating a solution at time steps provided by the programmer. For second order differential equations, you will have to rewrite them manually as a combination of first order equation first. I give an example below.

For **odeint** to work, **you must first write a Python function that returns a list or a numpy array of all the dy/dt values at a given time**. If there are many dy/dt values, your function must build a list/array of them. If there is only a single equation, the single dy/dt value must still be wrapped in square brackets to make it a "list" or numpy array of size one. The name of the function doesn't matter. I will call it dydtLF, to make it clear that the function is a list, or a numpy array, of floats.

The arguments to dydtLF are a lists/arrays of the yi function values at the current time, and the single current time t. To make it clear, I'm using the convention that a list or array of floats ends in LF, and a single float variable ends in F. The code you write for the derivatives should look like

**def dydtLF(yLF, tF) :**

**code to calculate all the dy/dt values from the y-list and t**

**return [dydt0, dydt1, dydt2 ... ]**

**y0LF = [y00, y10, …, yn0] #** *This is the list of initial values for variables y0, y1, y2, ….*

**tLF=[t0, t1, t2, …, tk]** *# This is the list of time steps where you want to evaluate the solution*

**yM = odeint(dydtLF, y0LF, tLF)**

The routine odeint returns the solution as a list **yM=[y0, y1, …, yn]**, where each **yi** is a list of values for the time steps **tk**. Therefore the whole **yM** is a 2-dimensional list of size **n x k**.

First example: solving **dv/dt=1-v2**, the free fall equation with a drag term. For this problem we will only look at one variable, the velocity v (we could have done v and position x, but not for now, see next example).

**from scipy.integrate import odeint**

**import matplotlib.pyplot as plt**

**def dydtLF(yLF, tF):**

**v = yLF[0]** *# [this is velocity, first (and only) element in yLF*

**return [1.0 - v\*\*2]** *# this is dv/dt, returned as a list (even when there is only one variable like here)*

**v0 = 0.0** *# initial condition v(t=0)*

**y0LF = [v0]** *# proper variable (list) for dydtLF initial condition*

**t0 = 0.0** *# initial time*

**tmax = 3.0** *# maximum time*

**steps = 10** *# number of time steps*

**dt = (tmax - t0)/steps** *# time step*

**tLF = []** *# this initiates the 1-D array for time values*

**for i in range(steps+1):** *# this sets the time steps values*

**tLF.append(t0+i\*dt)**

**yM = odeint(dydtLF, y0LF, tLF)** *# solve the equation*

**print("yM=", yM)** *# print the results*

The following statements show you the data types used in the code:

**In [8]: type(tLF)**

**Out[8]: list**

**In [9]: type(yM)**

**Out[9]: numpy.ndarray**

**In [10]: yM.shape**

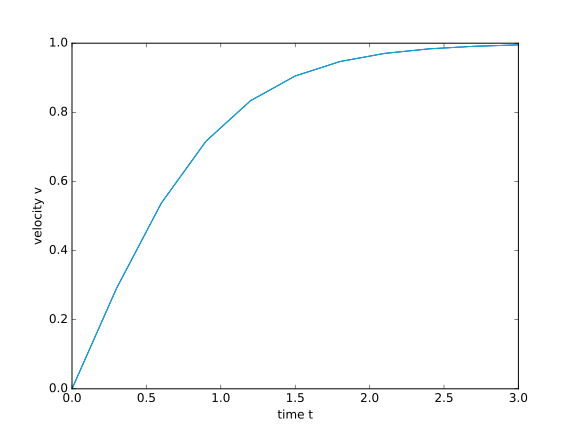
**Out[10]: (11, 1)**

Note that odeint works as well than the **y0LF** and **tLF** lists are numpy arrays. In fact, as you can see above, odeint returns a numpy array. The following statement plots **yM[:,0]** as a function of **tLF**, that is velocity v as a function of time t:

**plt.plot(tLF,yM[:,0])**

**plt.xlabel('time t')**

**plt.ylabel('velocity v')**



Second example: solving **mdv/dt=-k x - c v**, the damped harmonic oscillator.

For this problem we are interested in two variables, velocity v and position x, so we have two physical quantities, following the notation above, y0 and y1. We can assign either one to x and v.

m is the mass, k the spring constant and c the damping constant. Set k=m=1 for simplicity and I choose c=0.1. The script is:

**from scipy.integrate import odeint**

**import matplotlib.pyplot as plt**

**import numpy as np**

**import math**

**c = 0.1**

**def dydtLF(yLF, tF):**

**x = yLF[0]** *# this is position (first element of yLF)*

**v = yLF[1]** *# this is velocity (second element of yLF)*

**return [v, -x - c\*v]** *# this is [dx/dt, dv/dt]*

**t0 = 0.0** *# this is my origin of time*

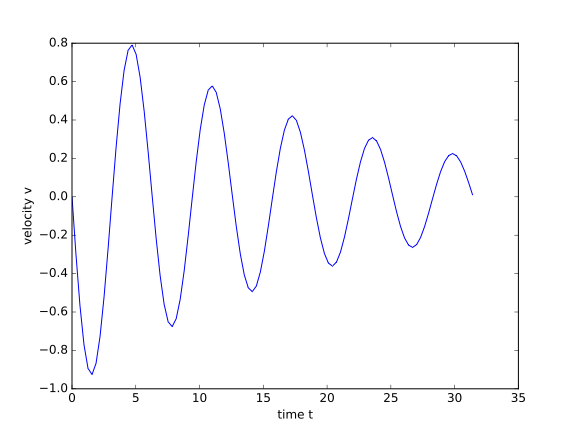
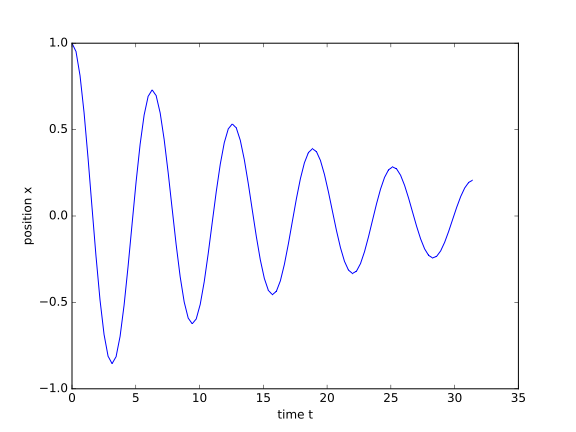
**tmax=10.0\*math.pi** *# this is my final time*

**steps=100** *# this is my number of time step*

**tLF = np.linspace(t0, tmax, steps+1)** *# this is my time 1-D array*

**y0LF = [1.0, 0.0]** *# this is initial condition for position, velocity*

**yM = odeint(dydtLF, y0LF, tLF)** *# solves differential equation*

Experiment with it (e.g. try different values of c). Running the above script, I get the following plots for position **x(t)** and velocity **v(t)**:

**Comments**: experiment with these examples before you get started with project 2. Make sure you understand how **odeint** works.

As a general comment, there are other ways of solving differential equations too that don’t involve iterations. As the physics problem gets more complicated, e.g. with non-linear terms, coupled systems and source terms, other techniques might be better, e.g. Green function approach or Fourier transforms. These are too advanced for this course, but you will surely meet them one day!

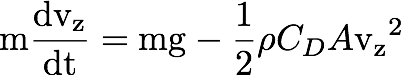
Some differential equations are still unsolved today, e.g. the full Navier-Stokes equation which governs the physics of fluids in the most possible general way. Any significant progress on this problem can make you win a million dollars prize :-) check the Clay institute millenium problems (<http://www.claymath.org/millennium-problems/navier%E2%80%93stokes-equation>).

**Project 2 (Due date Thursday November 3rd 12 a.m.)**



On October 2012, Felix Baumgartner broke the world record of skydiving at that time, by jumping from an altitude of 39 km and reaching a maximum speed of 1357 km/h.

In this problem you will calculate Baumgartner’s trajectory by solving numerically the free fall equation, taking into account the varying air density and the varying drag coefficient. We will have to make some simplifying assumptions which will prevent us from reproducing the exact characteristics of his jump, but we will get a good idea of what happened.

For this project we will look at a free fall case with a drag term that is proportional to velocity square, which is more realistic for high speed situations than the drag term we used in project one. The equation of motion becomes:

where Vz is the vertical speed (oriented positively downwards), ρ is the air density, CD is the drag coefficient, m is the mass of the jumper and A the jumper’s cross-section. Note that our altitude axis, "z" is also oriented downwards, so ground is located "below" with a positive z.

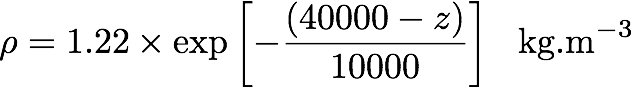
The jump has two phases:

**phase 1** is the free fall without parachute which **starts at an altitude of 40 km** and ends at an altitude of 2 km.

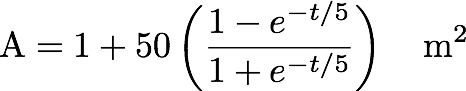
**phase 2** starts at altitude of 2 km when the parachute opens and ends when reaching ground.

Felix’s mass with equipment is m=118 kg. We will assume that the Earth’s gravity remains constant g=9.8 m/s2.

**Phase 1**: Take CD=0.2, A=0.85 m2 and the air density is equal to:



Use z as the altitude, expressed in meters, the origin of coordinates (z=0) being the beginning of the jump, with ground 40000 meters below. Initial velocity is Vz=0 m/s.

**Phase 2**: Parachute opens at an altitude of 2 km. Then suddenly the drag coefficient becomes CD=1.5, the air density can assumed to be constant with ρ=1.22 kg.m-3, and the cross-section A has to take into account the fact that the parachute takes a few seconds to unfold completely (a brutal parachute opening would certainly kill the jumper or destroy the parachute!). You will assume that the parachute cross-section A can be described as:

where time t here is measured starting from the beginning of phase 2.

**Objectives**: Plot the free fall velocity and the jumper altitude as function of time, from the beginning of the jump until reaching ground. Your Python script should be contained in a file called **skydive.py**, your plots should be in the file **skydive.pdf**. The script should generate the plots and also returns the answers to the questions below (so we can see how you have calculated them).

**Answer these questions**: What is the jumper maximum velocity in km/h? When is maximum velocity reached in seconds? What is the terminal velocity on the ground in m/s? what is the total duration of the jump in seconds? Why is velocity increasing a lot at the beginning and starts to decrease even before the parachute is opened? Write your answers in a text file called **skydive.txt**.

Make sure you copy all your files under a directory called /home2/phys210/yourusername/yourusername\_project\_2

**Tips:** You have to use **odeint** to solve this problem. The difficulty lies in the fact that you have to make sure you properly match the two phases of the jump: the conditions at the end of phase 1 are the initial conditions of the phase 2, and the physical constants of the problem go through a sudden change, i.e. the equation is also changed.