Classical Mechanics

Morten Hjorth-Jensen

Dec 16, 2020

CONTENTS

Morten Hjorth-Jensen, Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA and Department of Physics, University of Oslo, Norway

Date: **Dec 16, 2020**

CHAPTER

ONE

INTRODUCTION

Classical mechanics is a topic which has been taught intensively over several centuries. It is, with its many variants and ways of presenting the educational material, normally the first **real** physics course many of us meet and it lays the foundation for further physics studies. Many of the equations and ways of reasoning about the underlying laws of motion and pertinent forces, shape our approaches and understanding of the scientific method and discourse, as well as the way we develop our insights and deeper understanding about physical systems.

There is a wealth of well-tested (from both a physics point of view and a pedagogical standpoint) exercises and problems which can be solved analytically. However, many of these problems represent idealized and less realistic situations. The large majority of these problems are solved by paper and pencil and are traditionally aimed at what we normally refer to as continuous models from which we may find an analytical solution. As a consequence, when teaching mechanics, it implies that we can seldomly venture beyond an idealized case in order to develop our understandings and insights about the underlying forces and laws of motion.

On the other hand, numerical algorithms call for approximate discrete models and much of the development of methods for continuous models are nowadays being replaced by methods for discrete models in science and industry, simply because **much larger classes of problems can be addressed** with discrete models, often by simpler and more generic methodologies.

As we will see below, when properly scaling the equations at hand, discrete models open up for more advanced abstractions and the possibility to study real life systems, with the added bonus that we can explore and deepen our basic understanding of various physical systems

Analytical solutions are as important as before. In addition, such solutions provide us with invaluable benchmarks and tests for our discrete models. Such benchmarks, as we will see below, allow us to discuss possible sources of errors and their behaviors. And finally, since most of our models are based on various algorithms from numerical mathematics, we have a unique oppotunity to gain a deeper understanding of the mathematical approaches we are using.

With computing and data science as important elements in essentially all aspects of a modern society, we could then try to define Computing as **solving scientific problems using all possible tools, including symbolic computing, computers and numerical algorithms, and analytical paper and pencil solutions**. Computing provides us with the tools to develope our own understanding of the scientific method by enhancing algorithmic thinking.

The way we will teach this course reflects this definition of computing. The course contains both classical paper and pencil exercises as well as computational projects and exercises. The hope is that this will allow you to explore the physics of systems governed by the degrees of freedom of classical mechanics at a deeper level, and that these insights about the scientific method will help you to develop a better understanding of how the underlying forces and equations of motion and how they impact a given system. Furthermore, by introducing various numerical methods via computational projects and exercises, we aim at developing your competences and skills about these topics.

These competences will enable you to

- understand how algorithms are used to solve mathematical problems,
- derive, verify, and implement algorithms,
- understand what can go wrong with algorithms,

- use these algorithms to construct reproducible scientific outcomes and to engage in science in ethical ways, and
- think algorithmically for the purposes of gaining deeper insights about scientific problems.

All these elements are central for maturing and gaining a better understanding of the modern scientific process *per se*.

The power of the scientific method lies in identifying a given problem as a special case of an abstract class of problems, identifying general solution methods for this class of problems, and applying a general method to the specific problem (applying means, in the case of computing, calculations by pen and paper, symbolic computing, or numerical computing by ready-made and/or self-written software). This generic view on problems and methods is particularly important for understanding how to apply available, generic software to solve a particular problem.

However, verification of algorithms and understanding their limitations requires much of the classical knowledge about continuous models.

1.1 A well-known examples to illustrate many of the above concepts

Before we venture into a reminder on Python and mechanics relevant applications, let us briefly outline some of the abovementioned topics using an example many of you may have seen before in for example CMSE201. A simple algorithm for integration is the Trapezoidal rule. Integration of a function $f(x)$ by the Trapezoidal Rule is given by following algorithm for an interval $x \in [a, b]$

$$\int_a^b (f(x)dx = \frac{1}{2}\left[f(a) + 2f(a+h) + \cdots + 2f(b-h) + f(b)\right] + O(h^2),$$

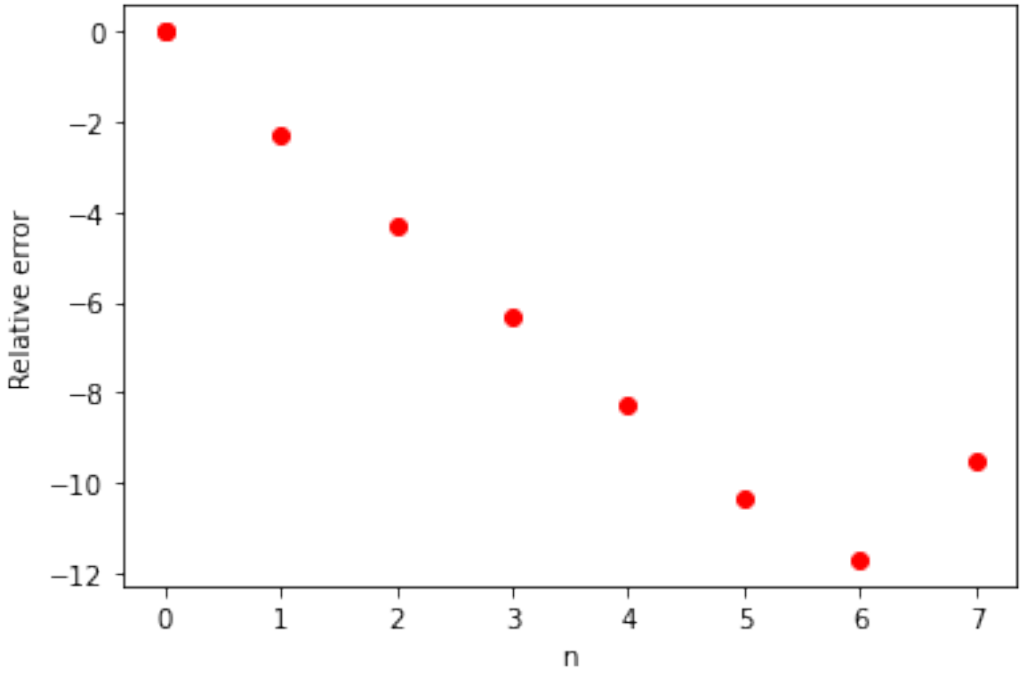
where h is the so-called stepsize defined by the number of integration points N as $h = (b-a)/(n)$. Python offers an extremely versatile programming environment, allowing for the inclusion of analytical studies in a numerical program. Here we show an example code with the **trapezoidal rule**. We use also **SymPy** to evaluate the exact value of the integral and compute the absolute error with respect to the numerically evaluated one of the integral $\int_0^1 dx x^2 = 1/3$. The following code for the trapezoidal rule allows you to plot the relative error by comparing with the exact result. By increasing to 10^8 points one arrives at a region where numerical errors start to accumulate.

```
%matplotlib inline

from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the trapezoidal rule
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s
#  function to compute pi
def function(x):
    return x*x
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
```

(continues on next page)

(continued from previous page)

```
exact = integrate(function(x), (x, a, b))
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Trapez(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```



This example shows the potential of combining numerical algorithms with symbolic calculations, allowing us to

- Validate and verify their algorithms.
- Including concepts like unit testing, one has the possibility to test and test several or all parts of the code.
- Validation and verification are then included *naturally* and one can develop a better attitude to what is meant with an ethically sound scientific approach.
- The above example allows the student to also test the mathematical error of the algorithm for the trapezoidal rule by changing the number of integration points. The students get **trained from day one to think error analysis**.
- With a Jupyter notebook you can keep exploring similar examples and turn them in as your own notebooks.

In this process we can easily bake in

1. How to structure a code in terms of functions
2. How to make a module
3. How to read input data flexibly from the command line

4. How to create graphical/web user interfaces
5. How to write unit tests (test functions or doctests)
6. How to refactor code in terms of classes (instead of functions only)
7. How to conduct and automate large-scale numerical experiments
8. How to write scientific reports in various formats (LaTeX, HTML)

The conventions and techniques outlined here will save you a lot of time when you incrementally extend software over time from simpler to more complicated problems. In particular, you will benefit from many good habits:

1. New code is added in a modular fashion to a library (modules)
2. Programs are run through convenient user interfaces
3. It takes one quick command to let all your code undergo heavy testing
4. Tedious manual work with running programs is automated,
5. Your scientific investigations are reproducible, scientific reports with top quality typesetting are produced both for paper and electronic devices.

1.1.1 PHY321: Review of Vectors, Math and first Numerical Examples for simple Motion Problems

Morten Hjorth-Jensen, Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA and Department of Physics, University of Oslo, Norway

Date: **Dec 16, 2020**

1.1.2 Space, Time, Motion, Reference Frames and Reminder on vectors and other mathematical quantities

Our studies will start with the motion of different types of objects such as a falling ball, a runner, a bicycle etc etc. It means that an object's position in space varies with time. In order to study such systems we need to define

- choice of origin
- choice of the direction of the axes
- choice of positive direction (left-handed or right-handed system of reference)
- choice of units and dimensions

These choices lead to some important questions such as

- is the physics of a system independent of the origin of the axes?
- is the physics independent of the directions of the axes, that is are there privileged axes?
- is the physics independent of the orientation of system?
- is the physics independent of the scale of the length?

Dimension, units and labels

Throughout this course we will use the standardized SI units. The standard unit for length is thus one meter 1m, for mass one kilogram 1kg, for time one second 1s, for force one Newton 1kgm/s^2 and for energy 1 Joule 1kgm^{2}s^{-2}.

We will use the following notations for various variables (vectors are always boldfaced in these lecture notes):

- position $\boldsymbol{r}$, in one dimention we will normally just use x,
- mass m,
- time t,
- velocity $\boldsymbol{v}$ or just v in one dimension,
- acceleration $\boldsymbol{a}$ or just a in one dimension,
- momentum $\boldsymbol{p}$ or just p in one dimension,
- kinetic energy K,
- potential energy V and
- frequency ω.

More variables will be defined as we need them.

It is also important to keep track of dimensionalities. Don't mix this up with a chosen unit for a given variable. We mark the dimensionality in these lectures as $[a]$, where a is the quantity we are interested in. Thus

- $[\boldsymbol{r}] =$ length
- $[m] =$ mass
- $[K] =$ energy
- $[t] =$ time
- $[\boldsymbol{v}] =$ length over time
- $[\boldsymbol{a}] =$ length over time squared
- $[\boldsymbol{p}] =$ mass times length over time
- $[\omega] =$ 1/time

Elements of Vector Algebra

Note: This section is under revision

In these lectures we will use boldfaced lower-case letters to label a vector. A vector $\boldsymbol{a}$ in three dimensions is thus defined as

$$\boldsymbol{a} = (a_x, a_y, a_z),$$

and using the unit vectors in a cartesian system we have

$$\boldsymbol{a} = a_x\boldsymbol{e}_x + a_y\boldsymbol{e}_y + a_z\boldsymbol{e}_z,$$

where the unit vectors have magnitude $|\boldsymbol{e}_i| = 1$ with $i = x, y, z$.

Using the fact that multiplication of reals is distributive we can show that

$$\boldsymbol{a}(\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a}\boldsymbol{b} + \boldsymbol{a}\boldsymbol{c},$$

Similarly we can also show that (using product rule for differentiating reals)

$$\frac{d}{dt}(\boldsymbol{ab}) = \boldsymbol{a}\frac{d\boldsymbol{b}}{dt} + \boldsymbol{b}\frac{d\boldsymbol{a}}{dt}.$$

We can repeat these operations for the cross products and show that they are distribuitive

$$\boldsymbol{a} \times (\boldsymbol{b} + \boldsymbol{c}) = \boldsymbol{a} \times \boldsymbol{b} + \boldsymbol{a} \times \boldsymbol{c}.$$

We have also that

$$\frac{d}{dt}(\boldsymbol{a} \times \boldsymbol{b}) = \boldsymbol{a} \times \frac{d\boldsymbol{b}}{dt} + \boldsymbol{b} \times \frac{d\boldsymbol{a}}{dt}.$$

The rotation of a three-dimensional vector $\boldsymbol{a} = (a_x, a_y, a_z)$ in the xy plane around an angle ϕ results in a new vector $\boldsymbol{b} = (b_x, b_y, b_z)$. This operation can be expressed in terms of linear algebra as a matrix (the rotation matrix) multiplied with a vector. We can write this as

$$\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}.$$

We can write this in a more compact form as $\boldsymbol{b} = \boldsymbol{Ra}$, where the rotation matrix is defined as

$$\boldsymbol{R} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Falling baseball in one dimension

We anticipate the mathematical model to come and assume that we have a model for the motion of a falling baseball without air resistance. Our system (the baseball) is at an initial height y_0 (which we will specify in the program below) at the initial time $t_0 = 0$. In our program example here we will plot the position in steps of Δt up to a final time t_f. The mathematical formula for the position $y(t)$ as function of time t is

$$y(t) = y_0 - \frac{1}{2}gt^2,$$

where $g = 9.80665 = 0.980655 \times 10^1 \text{m/s}^2$ is a constant representing the standard acceleration due to gravity. We have here adopted the conventional standard value. This does not take into account other effects, such as buoyancy or drag. Furthermore, we stop when the ball hits the ground, which takes place at

$$y(t) = 0 = y_0 - \frac{1}{2}gt^2,$$

which gives us a final time $t_f = \sqrt{2y_0/g}$.

As of now we simply assume that we know the formula for the falling object. Afterwards, we will derive it.

Our Python Encounter

We start with preparing folders for storing our calculations, figures and if needed, specific data files we use as input or output files.

```
%matplotlib inline

# Common imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

#in case we have an input file we wish to read in
#infile = open(data_path("MassEval2016.dat"),'r')
```

You could also define a function for making our plots. You can obviously avoid this and simply set up various **matplotlib** commands every time you need them. You may however find it convenient to collect all such commands in one function and simply call this function.

```
from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

def MakePlot(x,y, styles, labels, axlabels):
    plt.figure(figsize=(10,6))
    for i in range(len(x)):
        plt.plot(x[i], y[i], styles[i], label = labels[i])
        plt.xlabel(axlabels[0])
        plt.ylabel(axlabels[1])
    plt.legend(loc=0)
```

Thereafter we start setting up the code for the falling object.

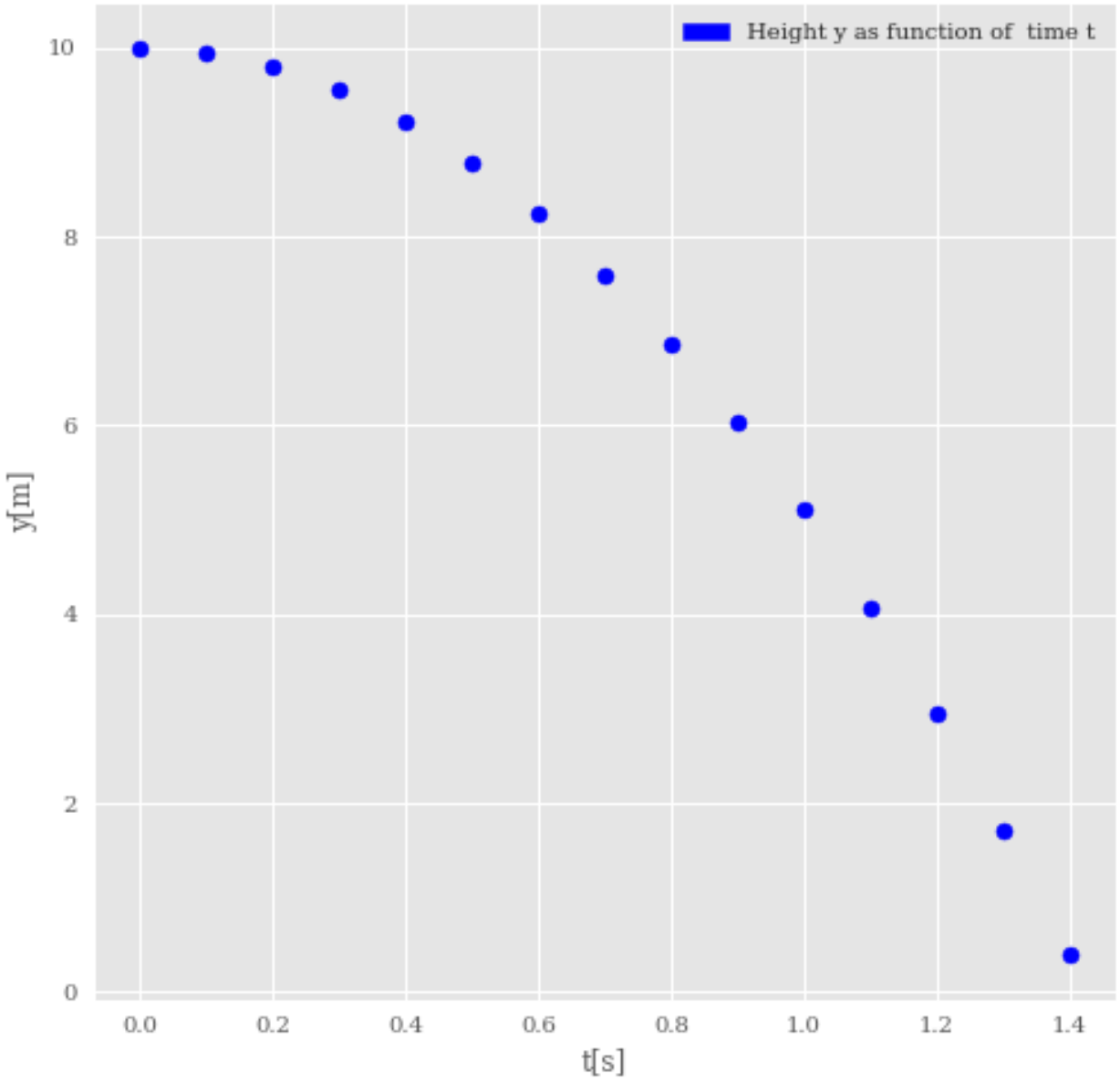
```
%matplotlib inline
import matplotlib.patches as mpatches

g = 9.80655 #m/s^2
y_0 = 10.0 # initial position in meters
```

(continues on next page)

(continued from previous page)

```
DeltaT = 0.1  # time step
# final time when y = 0, t = sqrt(2*10/g)
tfinal = np.sqrt(2.0*y_0/g)
#set up arrays
t = np.arange(0,tfinal,DeltaT)
y =y_0 -g*.5*t**2
# Then make a nice printout in table form using Pandas
import pandas as pd
from IPython.display import display
data = {'t[s]': t,
        'y[m]': y
        }
RawData = pd.DataFrame(data)
display(RawData)
plt.style.use('ggplot')
plt.figure(figsize=(8,8))
plt.scatter(t, y, color = 'b')
blue_patch = mpatches.Patch(color = 'b', label = 'Height y as function of  time t')
plt.legend(handles=[blue_patch])
plt.xlabel("t[s]")
plt.ylabel("y[m]")
save_fig("FallingBaseball")
plt.show()
```

```
    t[s]       y[m]
0    0.0  10.000000
1    0.1   9.950967
2    0.2   9.803869
3    0.3   9.558705
4    0.4   9.215476
5    0.5   8.774181
6    0.6   8.234821
7    0.7   7.597395
8    0.8   6.861904
9    0.9   6.028347
10   1.0   5.096725
11   1.1   4.067037
12   1.2   2.939284
13   1.3   1.713465
14   1.4   0.389581
```



Here we used **pandas** (see below) to systemize the output of the position as function of time.

Average quantities

We define now the average velocity as

$$\overline{v}(t) = \frac{y(t+\Delta t) - y(t)}{\Delta t}.$$

In the code we have set the time step Δt to a given value. We could define it in terms of the number of points n as

$$\Delta t = \frac{t_{\mathrm{final}-}t_{\mathrm{initial}}}{n+1}.$$

Since we have discretized the variables, we introduce the counter i and let $y(t) \rightarrow y(t_i) = y_i$ and $t \rightarrow t_i$ with $i = 0, 1, \ldots, n$. This gives us the following shorthand notations that we will use for the rest of this course. We define

$$y_i = y(t_i), \quad i = 0, 1, 2, \ldots, n.$$

This applies to other variables which depend on say time. Examples are the velocities, accelerations, momenta etc. Furthermore we use the shorthand

$$y_{i\pm1} = y(t_i \pm \Delta t),\ i = 0, 1, 2, \dots, n.$$

Compact equations

We can then rewrite in a more compact form the average velocity as

$$\overline{v}_i = \frac{y_{i+1} - y_i}{\Delta t}.$$

The velocity is defined as the change in position per unit time. In the limit $\Delta t \to 0$ this defines the instantaneous velocity, which is nothing but the slope of the position at a time t. We have thus

$$v(t) = \frac{dy}{dt} = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t}.$$

Similarly, we can define the average acceleration as the change in velocity per unit time as

$$\overline{a}_i = \frac{v_{i+1} - v_i}{\Delta t},$$

resulting in the instantaneous acceleration

$$a(t) = \frac{dv}{dt} = \lim_{\Delta t \to 0} \frac{v(t + \Delta t) - v(t)}{\Delta t}.$$

A note on notations: When writing for example the velocity as $v(t)$ we are then referring to the continuous and instantaneous value. A subscript like v_i refers always to the discretized values.

A differential equation

We can rewrite the instantaneous acceleration as

$$a(t) = \frac{dv}{dt} = \frac{d}{dt}\frac{dy}{dt} = \frac{d^2y}{dt^2}.$$

This forms the starting point for our definition of forces later. It is a famous second-order differential equation. If the acceleration is constant we can now recover the formula for the falling ball we started with. The acceleration can depend on the position and the velocity. To be more formal we should then write the above differential equation as

$$\frac{d^2y}{dt^2} = a(t, y(t), \frac{dy}{dt}).$$

With given initial conditions for $y(t_0)$ and $v(t_0)$ we can then integrate the above equation and find the velocities and positions at a given time t.

If we multiply with mass, we have one of the famous expressions for Newton's second law,

$$F(y, v, t) = m\frac{d^2y}{dt^2} = ma(t, y(t), \frac{dy}{dt}),$$

where F is the force acting on an object with mass m. We see that it also has the right dimension, mass times length divided by time squared. We will come back to this soon.

Integrating our equations

Formally we can then, starting with the acceleration (suppose we have measured it, how could we do that?) compute say the height of a building. To see this we perform the following integrations from an initial time t_0 to a given time t

$$\int_{t_0}^{t} dta(t) = \int_{t_0}^{t} dt\frac{dv}{dt} = v(t) - v(t_0),$$

or as

$$v(t) = v(t_0) + \int_{t_0}^{t} dta(t).$$

When we know the velocity as function of time, we can find the position as function of time starting from the defintion of velocity as the derivative with respect to time, that is we have

$$\int_{t_0}^{t} dtv(t) = \int_{t_0}^{t} dt\frac{dy}{dt} = y(t) - y(t_0),$$

or as

$$y(t) = y(t_0) + \int_{t_0}^{t} dtv(t).$$

These equations define what is called the integration method for finding the position and the velocity as functions of time. There is no loss of generality if we extend these equations to more than one spatial dimension.

Constant acceleration case, the velocity

Let us compute the velocity using the constant value for the acceleration given by $-g$. We have

$$v(t) = v(t_0) + \int_{t_0}^{t} dta(t) = v(t_0) + \int_{t_0}^{t} dt(-g).$$

Using our initial time as $t_0 = 0$s and setting the initial velocity $v(t_0) = v_0 = 0$m/s we get when integrating

$$v(t) = -gt.$$

The more general case is

$$v(t) = v_0 - g(t - t_0).$$

We can then integrate the velocity and obtain the final formula for the position as function of time through

$$y(t) = y(t_0) + \int_{t_0}^{t} dtv(t) = y_0 + \int_{t_0}^{t} dtv(t) = y_0 + \int_{t_0}^{t} dt(-gt),$$

With $y_0 = 10$m and $t_0 = 0$s, we obtain the equation we started with

$$y(t) = 10 - \frac{1}{2}gt^2.$$

Computing the averages

After this mathematical background we are now ready to compute the mean velocity using our data.

```
# Now we can compute the mean velocity using our data
# We define first an array Vaverage
n = np.size(t)
Vaverage = np.zeros(n)
for i in range(1,n-1):
    Vaverage[i] = (y[i+1]-y[i])/DeltaT
# Now we can compute the mean accelearatio using our data
# We define first an array Aaverage
n = np.size(t)
Aaverage = np.zeros(n)
Aaverage[0] = -g
for i in range(1,n-1):
    Aaverage[i] = (Vaverage[i+1]-Vaverage[i])/DeltaT
data = {'t[s]': t,
        'y[m]': y,
        'v[m/s]': Vaverage,
        'a[m/s^2]': Aaverage
        }
NewData = pd.DataFrame(data)
display(NewData[0:n-2])
```

	t[s]	y[m]	v[m/s]	a[m/s^2]
0	0.0	10.000000	0.000000	-9.80655
1	0.1	9.950967	-1.470982	-9.80655
2	0.2	9.803869	-2.451638	-9.80655
3	0.3	9.558705	-3.432292	-9.80655
4	0.4	9.215476	-4.412948	-9.80655
5	0.5	8.774181	-5.393602	-9.80655
6	0.6	8.234821	-6.374258	-9.80655
7	0.7	7.597395	-7.354913	-9.80655
8	0.8	6.861904	-8.335567	-9.80655
9	0.9	6.028347	-9.316222	-9.80655
10	1.0	5.096725	-10.296878	-9.80655
11	1.1	4.067037	-11.277533	-9.80655
12	1.2	2.939284	-12.258187	-9.80655

Note that we don't print the last values!

Including Air Resistance in our model

In our discussions till now of the falling baseball, we have ignored air resistance and simply assumed that our system is only influenced by the gravitational force. We will postpone the derivation of air resistance till later, after our discussion of Newton's laws and forces.

For our discussions here it suffices to state that the accelerations is now modified to

$$\boldsymbol{a}(t) = -g + D\boldsymbol{v}(t)|v(t)|,$$

where $|v(t)|$ is the absolute value of the velocity and D is a constant which pertains to the specific object we are studying. Since we are dealing with motion in one dimension, we can simplify the above to

$$a(t) = -g + Dv^2(t).$$

We can rewrite this as a differential equation

$$a(t) = \frac{dv}{dt} = \frac{d^2y}{dt^2} = -g + Dv^2(t).$$

Using the integral equations discussed above we can integrate twice and obtain first the velocity as function of time and thereafter the position as function of time.

For this particular case, we can actually obtain an analytical solution for the velocity and for the position. Here we will first compute the solutions analytically, thereafter we will derive Euler's method for solving these differential equations numerically.

Analytical solutions

For simplicity let us just write $v(t)$ as v. We have

$$\frac{dv}{dt} = -g + Dv^2(t).$$

We can solve this using the technique of separation of variables. We isolate on the left all terms that involve v and on the right all terms that involve time. We get then

$$\frac{dv}{g - Dv^2(t)} = -dt,$$

We scale now the equation to the left by introducing a constant $v_T = \sqrt{g/D}$. This constant has dimension length/time. Can you show this?

Next we integrate the left-hand side (lhs) from $v_0 = 0$ m/s to v and the right-hand side (rhs) from $t_0 = 0$ to t and obtain

$$\int_0^v \frac{dv}{g - Dv^2(t)} = \frac{v_T}{g}\text{arctanh}(\frac{v}{v_T}) = -\int_0^t dt = -t.$$

We can reorganize these equations as

$$v_T\text{arctanh}(\frac{v}{v_T}) = -gt,$$

which gives us v as function of time

$$v(t) = v_T \tanh -(\frac{gt}{v_T}).$$

Finding the final height

With the velocity we can then find the height $y(t)$ by integrating yet another time, that is

$$y(t) = y(t_0) + \int_{t_0}^t dtv(t) = \int_0^t dt[v_T \tanh -(\frac{gt}{v_T})].$$

This integral is a little bit trickier but we can look it up in a table over known integrals and we get

$$y(t) = y(t_0) - \frac{v_T^2}{g}\log[\cosh(\frac{gt}{v_T})].$$

Alternatively we could have used the symbolic Python package **Sympy** (example will be inserted later).

In most cases however, we need to revert to numerical solutions.

Our first attempt at solving differential equations

Here we will try the simplest possible approach to solving the second-order differential equation

$$a(t) = \frac{d^2y}{dt^2} = -g + Dv^2(t).$$

We rewrite it as two coupled first-order equations (this is a standard approach)

$$\frac{dy}{dt} = v(t),$$

with initial condition $y(t_0) = y_0$ and

$$a(t) = \frac{dv}{dt} = -g + Dv^2(t),$$

with initial condition $v(t_0) = v_0$.

Many of the algorithms for solving differential equations start with simple Taylor equations. If we now Taylor expand y and v around a value $t + \Delta t$ we have

$$y(t + \Delta t) = y(t) + \Delta t\frac{dy}{dt} + \frac{\Delta t^2}{2!}\frac{d^2y}{dt^2} + O(\Delta t^3),$$

and

$$v(t + \Delta t) = v(t) + \Delta t\frac{dv}{dt} + \frac{\Delta t^2}{2!}\frac{d^2v}{dt^2} + O(\Delta t^3).$$

Using the fact that $dy/dt = v$ and $dv/dt = a$ and keeping only terms up to Δt we have

$$y(t + \Delta t) = y(t) + \Delta tv(t) + O(\Delta t^2),$$

and

$$v(t + \Delta t) = v(t) + \Delta ta(t) + O(\Delta t^2).$$

Discretizing our equations

Using our discretized versions of the equations with for example $y_i = y(t_i)$ and $y_{i\pm1} = y(t_i + \Delta t)$, we can rewrite the above equations as (and truncating at Δt)

$$y_{i+1} = y_i + \Delta tv_i,$$

and

$$v_{i+1} = v_i + \Delta ta_i.$$

These are the famous Euler equations (forward Euler).

To solve these equations numerically we start at a time t_0 and simply integrate up these equations to a final time t_f, The step size Δt is an input parameter in our code. You can define it directly in the code below as

```
DeltaT = 0.1
```

With a given final time **tfinal** we can then find the number of integration points via the **ceil** function included in the **math** package of Python as

```
#define final time, assuming that initial time is zero
from math import ceil
tfinal = 0.5
n = ceil(tfinal/DeltaT)
print(n)
```

```
5
```

The **ceil** function returns the smallest integer not less than the input in say

```
x = 21.15
print(ceil(x))
```

```
22
```

which in the case here is 22.

```
x = 21.75
print(ceil(x))
```

```
22
```

which also yields 22. The **floor** function in the **math** package is used to return the closest integer value which is less than or equal to the specified expression or value. Compare the previous result to the usage of **floor**

```
from math import floor
x = 21.75
print(floor(x))
```

```
21
```

Alternatively, we can define ourselves the number of integration(mesh) points. In this case we could have

```
n = 10
tinitial = 0.0
tfinal = 0.5
DeltaT = (tfinal-tinitial)/(n)
print(DeltaT)
```

```
0.05
```

Since we will set up one-dimensional arrays that contain the values of various variables like time, position, velocity, acceleration etc, we need to know the value of n, the number of data points (or integration or mesh points). With n we can initialize a given array by setting all elelements to zero, as done here

```
# define array a
a = np.zeros(n)
print(a)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Code for implementing Euler's method

In the code here we implement this simple Eurler scheme choosing a value for $D = 0.0245$ m/s.

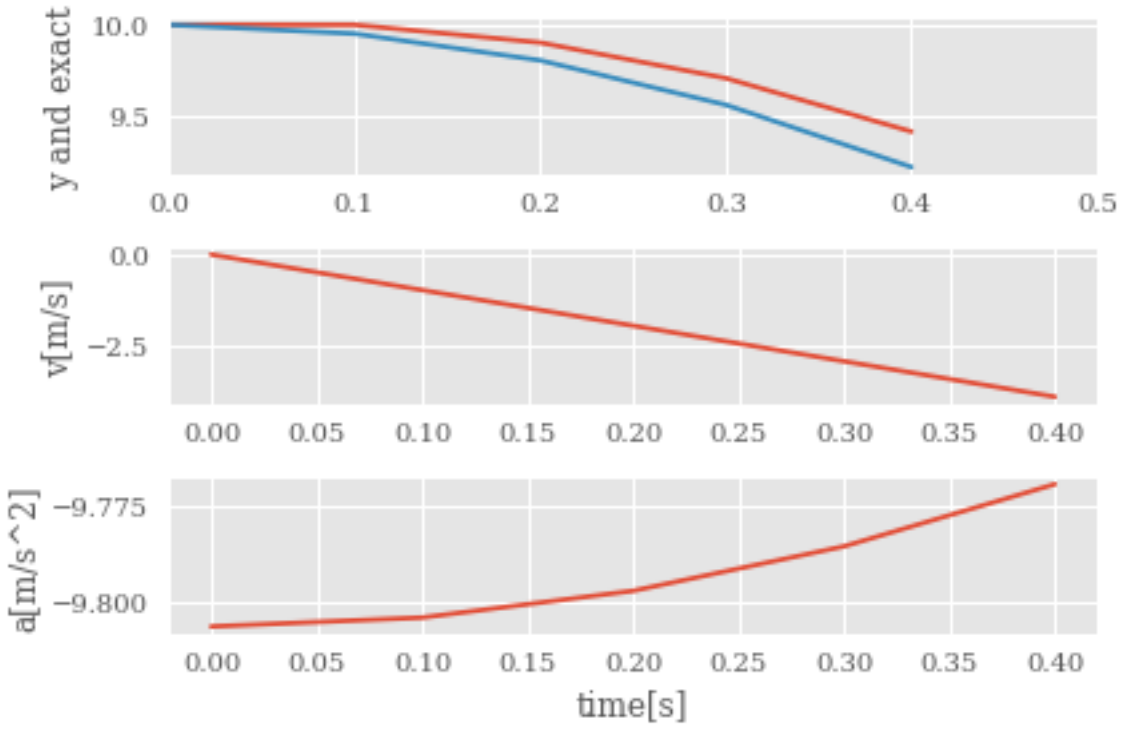
```
# Common imports
import numpy as np
import pandas as pd
from math import *
import matplotlib.pyplot as plt
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')


g = 9.80655 #m/s^2
D = 0.00245 #m/s
DeltaT = 0.1
#set up arrays
tfinal = 0.5
n = ceil(tfinal/DeltaT)
# define scaling constant vT
vT = sqrt(g/D)
# set up arrays for t, a, v, and y and we can compare our results with analytical ones
t = np.zeros(n)
a = np.zeros(n)
v = np.zeros(n)
y = np.zeros(n)
yanalytic = np.zeros(n)
# Initial conditions
v[0] = 0.0  #m/s
y[0] = 10.0 #m
yanalytic[0] = y[0]
# Start integrating using Euler's method
for i in range(n-1):
    # expression for acceleration
    a[i] = -g + D*v[i]*v[i]
    # update velocity and position
```

(continues on next page)

(continued from previous page)

```
    y[i+1] = y[i] + DeltaT*v[i]
    v[i+1] = v[i] + DeltaT*a[i]
    # update time to next time step and compute analytical answer
    t[i+1] = t[i] + DeltaT
    yanalytic[i+1] = y[0]-(vT*vT/g)*log(cosh(g*t[i+1]/vT))
    if ( y[i+1] < 0.0):
        break
a[n-1] = -g + D*v[n-1]*v[n-1]
data = {'t[s]': t,
        'y[m]': y-yanalytic,
        'v[m/s]': v,
        'a[m/s^2]': a
        }
NewData = pd.DataFrame(data)
display(NewData)
#finally we plot the data
fig, axs = plt.subplots(3, 1)
axs[0].plot(t, y, t, yanalytic)
axs[0].set_xlim(0, tfinal)
axs[0].set_ylabel('y and exact')
axs[1].plot(t, v)
axs[1].set_ylabel('v[m/s]')
axs[2].plot(t, a)
axs[2].set_xlabel('time[s]')
axs[2].set_ylabel('a[m/s^2]')
fig.tight_layout()
save_fig("EulerIntegration")
plt.show()
```

	t[s]	y[m]	v[m/s]	a[m/s^2]
0	0.0	0.000000	0.000000	-9.806550
1	0.1	0.049031	-0.980655	-9.804194
2	0.2	0.098034	-1.961074	-9.797128
3	0.3	0.146963	-2.940787	-9.785362
4	0.4	0.195770	-3.919323	-9.768915



Try different values for Δt and study the difference between the exact solution and the numerical solution.

Simple extension, the Euler-Cromer method

The Euler-Cromer method is a simple variant of the standard Euler method. We use the newly updated velocity v_{i+1} as an input to the new position, that is, instead of

$$y_{i+1} = y_i + \Delta t v_i,$$

and

$$v_{i+1} = v_i + \Delta t a_i,$$

we use now the newly calculate for v_{i+1} as input to y_{i+1}, that is we compute first

$$v_{i+1} = v_i + \Delta t a_i,$$

and then

$$y_{i+1} = y_i + \Delta t v_{i+1},$$

Implementing the Euler-Cromer method yields a simple change to the previous code. We only need to change the following line in the loop over time steps

```
for i in range(n-1):
    # more codes in between here
    v[i+1] = v[i] + DeltaT*a[i]
    y[i+1] = y[i] + DeltaT*v[i+1]
    # more code
```

Python practicalities, Software and needed installations

We will make extensive use of Python as programming language and its myriad of available libraries. You will find Jupyter notebooks invaluable in your work.

If you have Python installed (we strongly recommend Python3) and you feel pretty familiar with installing different packages, we recommend that you install the following Python packages via **pip** as

1. pip install numpy scipy matplotlib ipython scikit-learn mglearn sympy pandas pillow

For Python3, replace **pip** with **pip3**.

For OSX users we recommend, after having installed Xcode, to install **brew**. Brew allows for a seamless installation of additional software via for example

1. brew install python3

For Linux users, with its variety of distributions like for example the widely popular Ubuntu distribution, you can use **pip** as well and simply install Python as

1. sudo apt-get install python3 (or python for pyhton2.7)

etc etc.

Python installers

If you don't want to perform these operations separately and venture into the hassle of exploring how to set up dependencies and paths, we recommend two widely used distrubutions which set up all relevant dependencies for Python, namely

- Anaconda,

which is an open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system **conda**.

- Enthought canopy

is a Python distribution for scientific and analytic computing distribution and analysis environment, available for free and under a commercial license.

Furthermore, Google's Colab is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. Try it out!

Useful Python libraries

Here we list several useful Python libraries we strongly recommend (if you use anaconda many of these are already there)

- NumPy is a highly popular library for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays
- The pandas library provides high-performance, easy-to-use data structures and data analysis tools
- Xarray is a Python package that makes working with labelled multi-dimensional arrays simple, efficient, and fun!
- Scipy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering.

- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.
- Autograd can automatically differentiate native Python and Numpy code. It can handle a large subset of Python's features, including loops, ifs, recursion and closures, and it can even take derivatives of derivatives of derivatives
- SymPy is a Python library for symbolic mathematics.
- scikit-learn has simple and efficient tools for machine learning, data mining and data analysis
- TensorFlow is a Python library for fast numerical computing created and released by Google
- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano
- And many more such as pytorch, Theano etc

Your jupyter notebook can easily be converted into a nicely rendered **PDF** file or a Latex file for further processing. For example, convert to latex as

```
pycod jupyter nbconvert filename.ipynb --to latex
```

And to add more versatility, the Python package SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) and is entirely written in Python.

Numpy examples and Important Matrix and vector handling packages

There are several central software libraries for linear algebra and eigenvalue problems. Several of the more popular ones have been wrapped into ofter software packages like those from the widely used text **Numerical Recipes**. The original source codes in many of the available packages are often taken from the widely used software package LAPACK, which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. We describe them shortly here.

- LINPACK: package for linear equations and least square problems.
- LAPACK:package for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website http://www.netlib.org it is possible to download for free all source codes from this library. Both C/C++ and Fortran versions are available.
- BLAS (I, II and III): (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. Blas I is vector operations, II vector-matrix operations and III matrix-matrix operations. Highly parallelized and efficient codes, all available for download from http://www.netlib.org.

Basic Matrix Features

Matrix properties reminder.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The inverse of a matrix is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I$$

Some famous Matrices

- Diagonal if $a_{ij} = 0$ for $i \neq j$
- Upper triangular if $a_{ij} = 0$ for $i > j$
- Lower triangular if $a_{ij} = 0$ for $i < j$
- Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$
- Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$
- Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$
- Lower banded with bandwidth p: $a_{ij} = 0$ for $i > j + p$
- Upper banded with bandwidth p: $a_{ij} = 0$ for $i < j + p$
- Banded, block upper triangular, block lower triangular....

More Basic Matrix Features

Some Equivalent Statements.

For an $N \times N$ matrix $\mathbf{A}$ the following properties are all equivalent

- If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.
- The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.
- The rows of $\mathbf{A}$ form a basis of R^N.
- The columns of $\mathbf{A}$ form a basis of R^N.
- $\mathbf{A}$ is a product of elementary matrices.
- 0 is not eigenvalue of $\mathbf{A}$.

Numpy and arrays

Numpy provides an easy way to handle arrays in Python. The standard way to import this library is as

```
import numpy as np
```

Here follows a simple example where we set up an array of ten elements, all determined by random numbers drawn according to the normal distribution,

```
n = 10
x = np.random.normal(size=n)
print(x)
```

```
[-1.77950599  0.4112661  -0.02612504 -0.68556198 -0.35550204  2.39495247
  0.10453737 -0.34196605 -0.02755488  0.64322597]
```

We defined a vector x with $n = 10$ elements with its values given by the Normal distribution $N(0, 1)$. Another alternative is to declare a vector as follows

```
import numpy as np
x = np.array([1, 2, 3])
print(x)
```

```
[1 2 3]
```

Here we have defined a vector with three elements, with $x_0 = 1$, $x_1 = 2$ and $x_2 = 3$. Note that both Python and C++ start numbering array elements from 0 and on. This means that a vector with n elements has a sequence of entities $x_0, x_1, x_2, \ldots, x_{n-1}$. We could also let (recommended) Numpy to compute the logarithms of a specific array as

```
import numpy as np
x = np.log(np.array([4, 7, 8]))
print(x)
```

```
[1.38629436 1.94591015 2.07944154]
```

In the last example we used Numpy's unary function $np.log$. This function is highly tuned to compute array elements since the code is vectorized and does not require looping. We normaly recommend that you use the Numpy intrinsic functions instead of the corresponding **log** function from Python's **math** module. The looping is done explicitely by the **np.log** function. The alternative, and slower way to compute the logarithms of a vector would be to write

```
import numpy as np
from math import log
x = np.array([4, 7, 8])
for i in range(0, len(x)):
    x[i] = log(x[i])
print(x)
```

```
[1 1 2]
```

We note that our code is much longer already and we need to import the **log** function from the **math** module. The attentive reader will also notice that the output is $[1, 1, 2]$. Python interprets automagically our numbers as integers (like the **automatic** keyword in C++). To change this we could define our array elements to be double precision numbers as

```
import numpy as np
x = np.log(np.array([4, 7, 8], dtype = np.float64))
print(x)
```

```
[1.38629436 1.94591015 2.07944154]
```

or simply write them as double precision numbers (Python uses 64 bits as default for floating point type variables), that is

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0])
print(x)
```

```
  File "<ipython-input-20-f6d7a289d493>", line 3
    print(x)
    ^
SyntaxError: invalid syntax
```

To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can use simple use the **itemsize** functionality (the array x is actually an object which inherits the functionalities defined in Numpy) as

```
import numpy as np
x = np.log(np.array([4.0, 7.0, 8.0])
print(x.itemsize)
```

Matrices in Python

Having defined vectors, we are now ready to try out matrices. We can define a 3×3 real matrix $\hat{A}$ as (recall that we user lowercase letters for vectors and uppercase letters for matrices)

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
print(A)
```

If we use the **shape** function we would get $(3, 3)$ as output, that is verifying that our matrix is a 3×3 matrix. We can slice the matrix and print for example the first column (Python organized matrix elements in a row-major order, see below) as

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[:,0])
```

We can continue this was by printing out other columns or rows. The example here prints out the second column

```
import numpy as np
A = np.log(np.array([ [4.0, 7.0, 8.0], [3.0, 10.0, 11.0], [4.0, 5.0, 7.0] ]))
# print the first column, row-major order and elements start with 0
print(A[1,:])
```

Numpy contains many other functionalities that allow us to slice, subdivide etc etc arrays. We strongly recommend that you look up the Numpy website for more details. Useful functions when defining a matrix are the **np.zeros** function which declares a matrix of a given dimension and sets all elements to zero

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to zero
A = np.zeros( (n, n) )
print(A)
```

or initializing all elements to

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to one
A = np.ones( (n, n) )
print(A)
```

or as unitarily distributed random numbers (see the material on random number generators in the statistics part)

```
import numpy as np
n = 10
# define a matrix of dimension 10 x 10 and set all elements to random numbers with x \
 ↪in [0, 1]
A = np.random.rand(n, n)
print(A)
```

Meet the Pandas

Another useful Python package is pandas, which is an open source library providing high-performance, easy-to-use data structures and data analysis tools for Python. **pandas** stands for panel data, a term borrowed from econometrics and is an efficient library for data analysis with an emphasis on tabular data. **pandas** has two major classes, the **DataFrame** class with two-dimensional data objects and tabular data organized in columns and the class **Series** with a focus on one-dimensional data objects. Both classes allow you to index data easily as we will see in the examples below. **pandas** allows you also to perform mathematical operations on the data, spanning from simple reshapings of vectors and matrices to statistical operations.

The following simple example shows how we can, in an easy way make tables of our data. Here we define a data set which includes names, place of birth and date of birth, and displays the data in an easy to read way. We will see repeated use of **pandas**, in particular in connection with classification of data.

```
import pandas as pd
from IPython.display import display
data = {'First Name': ["Frodo", "Bilbo", "Aragorn II", "Samwise"],
        'Last Name': ["Baggins", "Baggins","Elessar","Gamgee"],
        'Place of birth': ["Shire", "Shire", "Eriador", "Shire"],
        'Date of Birth T.A.': [2968, 2890, 2931, 2980]
        }
data_pandas = pd.DataFrame(data)
display(data_pandas)
```

In the above we have imported **pandas** with the shorthand **pd**, the latter has become the standard way we import **pandas**. We make then a list of various variables and reorganize the above lists into a **DataFrame** and then print out a neat table with specific column labels as *Name*, *place of birth* and *date of birth*. Displaying these results, we see that the indices are given by the default numbers from zero to three. **pandas** is extremely flexible and we can easily change the above indices by defining a new type of indexing as

```
data_pandas = pd.DataFrame(data,index=['Frodo','Bilbo','Aragorn','Sam'])
display(data_pandas)
```

Thereafter we display the content of the row which begins with the index **Aragorn**

```
display(data_pandas.loc['Aragorn'])
```

We can easily append data to this, for example

```
new_hobbit = {'First Name': ["Peregrin"],
              'Last Name': ["Took"],
              'Place of birth': ["Shire"],
              'Date of Birth T.A.': [2990]
              }
data_pandas=data_pandas.append(pd.DataFrame(new_hobbit, index=['Pippin']))
display(data_pandas)
```

Here are other examples where we use the **DataFrame** functionality to handle arrays, now with more interesting features for us, namely numbers. We set up a matrix of dimensionality 10×5 and compute the mean value and standard deviation of each column. Similarly, we can perform mathematial operations like squaring the matrix elements and many other operations.

```
import numpy as np
import pandas as pd
from IPython.display import display
np.random.seed(100)
# setting up a 10 x 5 matrix
```

(continues on next page)

(continued from previous page)

```
rows = 10
cols = 5
a = np.random.randn(rows,cols)
df = pd.DataFrame(a)
display(df)
print(df.mean())
print(df.std())
display(df**2)
```

Thereafter we can select specific columns only and plot final results

```
df.columns = ['First', 'Second', 'Third', 'Fourth', 'Fifth']
df.index = np.arange(10)

display(df)
print(df['Second'].mean() )
print(df.info())
print(df.describe())

from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

df.cumsum().plot(lw=2.0, figsize=(10,6))
plt.show()


df.plot.bar(figsize=(10,6), rot=15)
plt.show()
```

We can produce a 4×4 matrix

```
b = np.arange(16).reshape((4,4))
print(b)
df1 = pd.DataFrame(b)
print(df1)
```

and many other operations.

The **Series** class is another important class included in **pandas**. You can view it as a specialization of **DataFrame** but where we have just a single column of data. It shares many of the same features as _DataFrame. As with **DataFrame**, most operations are vectorized, achieving thereby a high performance when dealing with computations of arrays, in particular labeled arrays. As we will see below it leads also to a very concice code close to the mathematical operations we may be interested in. For multidimensional arrays, we recommend strongly xarray. **xarray** has much of the same flexibility as **pandas**, but allows for the extension to higher dimensions than two.

1.1.3 PHY321: Forces, Newton's Laws and Motion Example

Morten Hjorth-Jensen, Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA and Department of Physics, University of Oslo, Norway

Date: **Dec 16, 2020**

1.1.4 Basic Steps of Scientific Investigations

An overarching aim in this course is to give you a deeper understanding of the scientific method. The problems we study will all involve cases where we can apply classical mechanics. In our previous material we already assumed that we had a model for the motion of an object. Alternatively we could have data from experiment (like Usain Bolt's 100m world record run in 2008). Or we could have performed ourselves an experiment and we want to understand which forces are at play and whether these forces can be understood in terms of fundamental forces.

Our first step consists in identifying the problem. What we sketch here may include a mix of experiment and theoretical simulations, or just experiment or only theory.

Identifying our System

Here we can ask questions like

1. What kind of object is moving
2. What kind of data do we have
3. How do we measure position, velocity, acceleration etc
4. Which initial conditions influence our system
5. Other aspects which allow us to identify the system

Defining a Model

With our eventual data and observations we would now like to develop a model for the system. In the end we want obviously to be able to understand which forces are at play and how they influence our specific system. That is, can we extract some deeper insights about a system?

We need then to

1. Find the forces that act on our system
2. Introduce models for the forces
3. Identify the equations which can govern the system (Newton's second law for example)
4. More elements we deem important for defining our model

Solving the Equations

With the model at hand, we can then solve the equations. In classical mechanics we normally end up with solving sets of coupled ordinary differential equations or partial differential equations.

1. Using Newton's second law we have equations of the type $\boldsymbol{F} = m\boldsymbol{a} = md\boldsymbol{v}/dt$
2. We need to define the initial conditions (typically the initial velocity and position as functions of time) and/or initial conditions and boundary conditions
3. The solution of the equations give us then the position, the velocity and other time-dependent quantities which may specify the motion of a given object.

We are not yet done. With our lovely solvers, we need to start thinking.

Now it is time to ask the big questions. What do our results mean? Can we give a simple interpretation in terms of fundamental laws? What do our results mean? Are they correct? Thus, typical questions we may ask are

1. Are our results for say $\boldsymbol{r}(t)$ valid? Do we trust what we did? Can you validate and verify the correctness of your results?
2. Evaluate the answers and their implications
3. Compare with experimental data if possible. Does our model make sense?
4. and obviously many other questions.

The analysis stage feeds back to the first stage. It may happen that the data we had were not good enough, there could be large statistical uncertainties. We may need to collect more data or perhaps we did a sloppy job in identifying the degrees of freedom.

All these steps are essential elements in a scientific enquiry. Hopefully, through a mix of numerical simulations, analytical calculations and experiments we may gain a deeper insight about the physics of a specific system.

Let us now remind ourselves of Newton's laws, since these are the laws of motion we will study in this course.

Newton's Laws

When analyzing a physical system we normally start with distinguishing between the object we are studying (we will label this in more general terms as our **system**) and how this system interacts with the environment (which often means everything else!)

In our investigations we will thus analyze a specific physics problem in terms of the system and the environment. In doing so we need to identify the forces that act on the system and assume that the forces acting on the system must have a source, an identifiable cause in the environment.

A force acting on for example a falling object must be related to an interaction with something in the environment. This also means that we do not consider internal forces. The latter are forces between one part of the object and another part. In this course we will mainly focus on external forces.

Forces are either contact forces or long-range forces.

Contact forces, as evident from the name, are forces that occur at the contact between the system and the environment. Well-known long-range forces are the gravitional force and the electromagnetic force.

Setting up a model for forces acting on an object

In order to set up the forces which act on an object, the following steps may be useful

1. Divide the problem into system and environment.
2. Draw a figure of the object and everything in contact with the object.
3. Draw a closed curve around the system.
4. Find contact points—these are the points where contact forces may act.
5. Give names and symbols to all the contact forces.
6. Identify the long-range forces.
7. Make a drawing of the object. Draw the forces as arrows, vectors, starting from where the force is acting. The direction of the vector(s) indicates the (positive) direction of the force. Try to make the length of the arrow indicate the relative magnitude of the forces.
8. Draw in the axes of the coordinate system. It is often convenient to make one axis parallel to the direction of motion. When you choose the direction of the axis you also choose the positive direction for the axis.

Newton's Laws, the Second one first

Newton's second law of motion: The force $\boldsymbol{F}$ on an object of inertial mass m is related to the acceleration a of the object through

$$\boldsymbol{F} = m\boldsymbol{a},$$

where $\boldsymbol{a}$ is the acceleration.

Newton's laws of motion are laws of nature that have been found by experimental investigations and have been shown to hold up to continued experimental investigations. Newton's laws are valid over a wide range of length- and time-scales. We use Newton's laws of motion to describe everything from the motion of atoms to the motion of galaxies.

The second law is a vector equation with the acceleration having the same direction as the force. The acceleration is proportional to the force via the mass m of the system under study.

Newton's second law introduces a new property of an object, the so-called inertial mass m. We determine the inertial mass of an object by measuring the acceleration for a given applied force.

Then the First Law

What happens if the net external force on a body is zero? Applying Newton's second law, we find:

$$\boldsymbol{F} = 0 = m\boldsymbol{a},$$

which gives using the definition of the acceleration

$$\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt} = 0.$$

The acceleration is zero, which means that the velocity of the object is constant. This is often referred to as Newton's first law. An object in a state of uniform motion tends to remain in that state unless an external force changes its state of motion. Why do we need a separate law for this? Is it not simply a special case of Newton's second law? Yes, Newton's first law can be deduced from the second law as we have illustrated. However, the first law is often used for a different purpose: Newton's First Law tells us about the limit of applicability of Newton's Second law. Newton's Second law can only be used in reference systems where the First law is obeyed. But is not the First law always valid?

No! The First law is only valid in reference systems that are not accelerated. If you observe the motion of a ball from an accelerating car, the ball will appear to accelerate even if there are no forces acting on it. We call systems that are not accelerating inertial systems, and Newton's first law is often called the law of inertia. Newton's first and second laws of motion are only valid in inertial systems.

A system is an inertial system if it is not accelerated. It means that the reference system must not be accelerating linearly or rotating. Unfortunately, this means that most systems we know are not really inertial systems. For example, the surface of the Earth is clearly not an inertial system, because the Earth is rotating. The Earth is also not an inertial system, because it ismoving in a curved path around the Sun. However, even if the surface of the Earth is not strictly an inertial system, it may be considered to be approximately an inertial system for many laboratory-size experiments.

And finally the Third Law

If there is a force from object A on object B, there is also a force from object B on object A. This fundamental principle of interactions is called Newton's third law. We do not know of any force that do not obey this law: All forces appear in pairs. Newton's third law is usually formulated as: For every action there is an equal and opposite reaction.

Motion of a Single Object

Here we consider the motion of a single particle moving under the influence of some set of forces. We will consider some problems where the force does not depend on the position. In that case Newton's law $m\dot{\boldsymbol{v}} = \boldsymbol{F}(\boldsymbol{v})$ is a first-order differential equation and one solves for $\boldsymbol{v}(t)$, then moves on to integrate $\boldsymbol{v}$ to get the position. In essentially all of these cases we cna find an analytical solution.

Air Resistance in One Dimension

Air resistance tends to scale as the square of the velocity. This is in contrast to many problems chosen for textbooks, where it is linear in the velocity. The choice of a linear dependence is motivated by mathematical simplicity (it keeps the differential equation linear) rather than by physics. One can see that the force should be quadratic in velocity by considering the momentum imparted on the air molecules. If an object sweeps through a volume dV of air in time dt, the momentum imparted on the air is

$$dP = \rho_m dV v, \quad (1.1)$$

where v is the velocity of the object and ρ_m is the mass density of the air. If the molecules bounce back as opposed to stop you would double the size of the term. The opposite value of the momentum is imparted onto the object itself. Geometrically, the differential volume is

$$dV = Avdt, \quad (1.2)$$

where A is the cross-sectional area and vdt is the distance the object moved in time dt.

Resulting Acceleration

Plugging this into the expression above,

$$\frac{dP}{dt} = -\rho_m A v^2. \quad (1.3)$$

This is the force felt by the particle, and is opposite to its direction of motion. Now, because air doesn't stop when it hits an object, but flows around the best it can, the actual force is reduced by a dimensionless factor c_W, called the drag coefficient.

$$F_{\text{drag}} = -c_W \rho_m A v^2, \quad (1.4)$$

and the acceleration is

$$to$$

$$\frac{dv}{dt} = -\frac{c_W \rho_m A}{m} v^2. \quad (1.5)$$

For a particle with initial velocity v_0, one can separate the dt to one side of the equation, and move everything with vs to the other side. We did this in our discussion of simple motion and will not repeat it here.

On more general terms, for many systems, e.g. an automobile, there are multiple sources of resistance. In addition to wind resistance, where the force is proportional to v^2, there are dissipative effects of the tires on the pavement, and in the axel and drive train. These other forces can have components that scale proportional to v, and components that are independent of v. Those independent of v, e.g. the usual $f = \mu_K N$ frictional force you consider in your first Physics courses, only set in once the object is actually moving. As speeds become higher, the v^2 components begin to dominate relative to the others. For automobiles at freeway speeds, the v^2 terms are largely responsible for the loss of efficiency. To travel a distance L at fixed speed v, the energy/work required to overcome the dissipative forces are fL, which for a force of the form $f = \alpha v^n$ becomes

$$to$$

$$W = \int dx\, f = \alpha v^n L. \quad (1.6)$$

For $n = 0$ the work is independent of speed, but for the wind resistance, where $n = 2$, slowing down is essential if one wishes to reduce fuel consumption. It is also important to consider that engines are designed to be most efficient at a chosen range of power output. Thus, some cars will get better mileage at higher speeds (They perform better at 50 mph than at 5 mph) despite the considerations mentioned above.

Going Ballistic, Projectile Motion or a Softer Approach, Falling Raindrops

As an example of Newton's Laws we consider projectile motion (or a falling raindrop or a ball we throw up in the air) with a drag force. Even though air resistance is largely proportional to the square of the velocity, we will consider the drag force to be linear to the velocity, $\boldsymbol{F} = -m\gamma\boldsymbol{v}$, for the purposes of this exercise. The acceleration for a projectile moving upwards, $\boldsymbol{a} = \boldsymbol{F}/m$, becomes

$$to$$

$$\frac{dv_x}{dt} = -\gamma v_x,$$
$$\frac{dv_y}{dt} = -\gamma v_y - g,$$

and γ has dimensions of inverse time.

If you on the other hand have a falling raindrop, how do these equations change? See for example Figure 2.1 in Taylor. Let us stay with a ball which is thrown up in the air at $t = 0$.

Ways of solving these equations

We will go over two different ways to solve this equation. The first by direct integration, and the second as a differential equation. To do this by direct integration, one simply multiplies both sides of the equations above by dt, then divide by the appropriate factors so that the vs are all on one side of the equation and the dt is on the other. For the x motion

one finds an easily integrable equation,

$$to$$

$$\frac{dv_x}{v_x} = -\gamma dt,$$

$$\int_{v_{0x}}^{v_x} \frac{dv_x}{v_x} = -\gamma \int_0^t dt,$$

$$\ln\left(\frac{v_x}{v_{0x}}\right) = -\gamma t,$$

$$v_x(t) = v_{0x} e^{-\gamma t}.$$

$$= -\gamma dt, = \int_{v_{0x}}^{v_x} \frac{dv_x}{v_x} -\gamma \int_0^t = dt, \ln\left(\frac{v_x}{v_{0x}}\right) -\gamma t, v_{\overline{x}}(t) v_{0x} e^{-\gamma t}.$$

This is very much the result you would have written down by inspection. For the y-component of the velocity,

$$to$$

$$\frac{dv_y}{v_y + g/\gamma} =$$
$$-\gamma dt$$
$$\ln\left(\frac{v_y + g/\gamma}{v_{0y} - g/\gamma}\right) =$$
$$-\gamma t_f,$$
$$v_{fy} =$$
$$-\frac{g}{\gamma} + \left(v_{0y} + \frac{g}{\gamma}\right) e^{-\gamma t}.$$

$$=$$
$$-\gamma dt \ln\left(\frac{v_y + g/\gamma}{v_{0y} - g/\gamma}\right)$$
$$-\gamma t_f, v_{fy} =$$
$$-\frac{g}{\gamma} + \left(v_{0y} + \frac{g}{\gamma}\right) e^{-\gamma t}.$$

Whereas v_x starts at some value and decays exponentially to zero, v_y decays exponentially to the terminal velocity, $v_t = -g/\gamma$.

Solving as differential equations

Although this direct integration is simpler than the method we invoke below, the method below will come in useful for some slightly more difficult differential equations in the future. The differential equation for v_x is straight-forward to solve. Because it is first order there is one arbitrary constant, A, and by inspection the solution is

$$v_x = Ae^{-\gamma t}.$$

The arbitrary constants for equations of motion are usually determined by the initial conditions, or more generally boundary conditions. By inspection $A = v_{0x}$, the initial x component of the velocity.

Differential Equations, contn

The differential equation for v_y is a bit more complicated due to the presence of g. Differential equations where all the terms are linearly proportional to a function, in this case v_y, or to derivatives of the function, e.g., v_y, dv_y/dt, $d^2v_y/dt^2 \cdots$, are called linear differential equations. If there are terms proportional to v^2, as would happen if the drag force were proportional to the square of the velocity, the differential equation is not longer linear. Because this expression has only one derivative in v it is a first-order linear differential equation. If a term were added proportional to d^2v/dt^2 it would be a second-order differential equation. In this case we have a term completely independent of v, the gravitational acceleration g, and the usual strategy is to first rewrite the equation with all the linear terms on one side of the equal sign,

$$\frac{dv_y}{dt} + \gamma v_y = -g.$$

Splitting into two parts

Now, the solution to the equation can be broken into two parts. Because this is a first-order differential equation we know that there will be one arbitrary constant. Physically, the arbitrary constant will be determined by setting the initial velocity, though it could be determined by setting the velocity at any given time. Like most differential equations, solutions are not "solved". Instead, one guesses at a form, then shows the guess is correct. For these types of equations, one first tries to find a single solution, i.e. one with no arbitrary constants. This is called the {\it particular} solution, $y_p(t)$, though it should really be called "a" particular solution because there are an infinite number of such solutions. One then finds a solution to the {\it homogenous} equation, which is the equation with zero on the right-hand side,

$$\frac{dv_{y,h}}{dt} + \gamma v_{y,h} = 0.$$

Homogenous solutions will have arbitrary constants.

The particular solution will solve the same equation as the original general equation

$$\frac{dv_{y,p}}{dt} + \gamma v_{y,p} = -g.$$

However, we don't need find one with arbitrary constants. Hence, it is called a **particular** solution.

The sum of the two,

$$v_y = v_{y,p} + v_{y,h},$$

is a solution of the total equation because of the linear nature of the differential equation. One has now found a *general* solution encompassing all solutions, because it both satisfies the general equation (like the particular solution), and has an arbitrary constant that can be adjusted to fit any initial condition (like the homogneous solution). If the equation were not linear, e.g if there were a term such as v_y^2 or $v_y \dot{v}_y$, this technique would not work.

More details

Returning to the example above, the homogenous solution is the same as that for v_x, because there was no gravitational acceleration in that case,

$$v_{y,h} = Be^{-\gamma t}.$$

In this case a particular solution is one with constant velocity,

$$v_{y,p} = -g/\gamma.$$

Note that this is the terminal velocity of a particle falling from a great height. The general solution is thus,

$$v_y = Be^{-\gamma t} - g/\gamma,$$

and one can find B from the initial velocity,

$$v_{0y} = B - g/\gamma, \quad B = v_{0y} + g/\gamma.$$

Plugging in the expression for B gives the y motion given the initial velocity,

$v_y = (v_{0y} + g/\gamma)e^{-\gamma t} - g/\gamma.$

It is easy to see that this solution has $v_y = v_{0y}$ when $t = 0$ and $v_y = -g/\gamma$ when $t \to \infty$.

One can also integrate the two equations to find the coordinates x and y as functions of t,

$$to$$

$$x = \int_0^t dt'\, v_{0x}(t') = \frac{v_{0x}}{\gamma}\left(1 - e^{-\gamma t}\right),$$

$$y = \int_0^t dt'\, v_{0y}(t') = -\frac{gt}{\gamma} + \frac{v_{0y} + g/\gamma}{\gamma}\left(1 - e^{-\gamma t}\right).$$

$$= \int_0^t dt'\, v_{0x}(t') = \frac{v_{0x}}{\gamma}\left(1 - e^{-\gamma t}\right) \text{,y}$$

$$\int_0^t dt'\, v_{0y}(t') = -\frac{gt}{\gamma} + \frac{v_{0y} + g/\gamma}{\gamma}\left(1 - e^{-\gamma t}\right).$$

If the question was to find the position at a time t, we would be finished. However, the more common goal in a projectile equation problem is to find the range, i.e. the distance x at which y returns to zero. For the case without a drag force this was much simpler. The solution for the y coordinate would have been $y = v_{0y}t - gt^2/2$. One would solve for t to make $y = 0$, which would be $t = 2v_{0y}/g$, then plug that value for t into $x = v_{0x}t$ to find $x = 2v_{0x}v_{0y}/g = v_0 \sin(2\theta_0)/g$. One follows the same steps here, except that the expression for $y(t)$ is more complicated. Searching for the time where $y = 0$, and we get

$$0 = -\frac{gt}{\gamma} + \frac{v_{0y} + g/\gamma}{\gamma}\left(1 - e^{-\gamma t}\right).$$

This cannot be inverted into a simple expression $t = \cdots$. Such expressions are known as "transcendental equations", and are not the rare instance, but are the norm. In the days before computers, one might plot the right-hand side of the above graphically as a function of time, then find the point where it crosses zero.

Now, the most common way to solve for an equation of the above type would be to apply Newton's method numerically. This involves the following algorithm for finding solutions of some equation $F(t) = 0$.

1. First guess a value for the time, $t_{\rm guess}$.
2. Calculate F and its derivative, $F(t_{\rm guess})$ and $F'(t_{\rm guess})$.
3. Unless you guessed perfectly, $F \neq 0$, and assuming that $\Delta F \approx F' \Delta t$, one would choose
4. $\Delta t = -F(t_{\rm guess})/F'(t_{\rm guess})$.
5. Now repeat step 1, but with $t_{\rm guess} \to t_{\rm guess} + \Delta t$.

If the $F(t)$ were perfectly linear in t, one would find t in one step. Instead, one typically finds a value of t that is closer to the final answer than $t_{\rm guess}$. One breaks the loop once one finds F within some acceptable tolerance of zero. A program to do this will be added shortly.

Motion in a Magnetic Field

Another example of a velocity-dependent force is magnetism,

$$to$$

$$\boldsymbol{F} = q\boldsymbol{v} \times \boldsymbol{B},$$

$$F_i = q \sum_{jk} \epsilon_{ijk} v_j B_k.$$

$$= q\boldsymbol{v} \times \boldsymbol{B}, F_i = q \sum_{jk} \epsilon_{ijk} v_j B_k.$$

For a uniform field in the z direction $\boldsymbol{B} = B\hat{z}$, the force can only have x and y components,

$$
\begin{aligned}
&to \\
&F_x = \\
&qBv_y \\
&F_y = \\
&-qBv_x.
\end{aligned}
$$

$$
\begin{aligned}
&= \\
&=qBv_yF_y \\
&-qBv_x.
\end{aligned}
$$

The differential equations are

$$
\begin{aligned}
&to \\
&\dot{v}_x = \\
&\omega_c v_y, \omega_c = qB/m \\
&\dot{v}_y = \\
&-\omega_c v_x.
\end{aligned}
$$

$$
\begin{aligned}
&= \\
&\omega_c v_y, \omega_c = qB/m\dot{v}_y \\
&-\omega_c v_x.
\end{aligned}
$$

One can solve the equations by taking time derivatives of either equation, then substituting into the other equation,

$$to$$

$$\ddot{v}_x = \omega_c \dot{v_y} = -\omega_c^2 v_x,$$

$$\ddot{v}_y =$$

$$-\omega_c \dot{v}_x = -\omega_c v_y.$$

$$=$$

$$-\omega_c \dot{v}_x = -\omega_c v_y.$$

The solution to these equations can be seen by inspection,

$$to$$

$$v_x =$$

$$A\sin(\omega_c t + \phi),$$

$$v_y =$$

$$A\cos(\omega_c t + \phi).$$

$$=$$

$$A\sin(\omega_c t + \phi), v_y$$

$$A\cos(\omega_c t + \phi).$$

One can integrate the equations to find the positions as a function of time,

$$to$$

$$x - x_0 = \int_{x_0}^{x} dx = \int_0^t dtv(t) = \frac{-A}{\omega_c}\cos(\omega_c t + \phi),$$

$$y - y_0 = \frac{A}{\omega_c}\sin(\omega_c t + \phi).$$

$$= \int_{x_0}^{x} dx = \not{=} \int_0^t dtv(t) \frac{-A}{\omega_c}\cos(\omega_c t + \phi), y - y_0 \frac{A}{\omega_c}\sin(\omega_c t + \phi).$$

The trajectory is a circle centered at x_0, y_0 with amplitude A rotating in the clockwise direction.

The equations of motion for the z motion are

$$\dot{v_z} = 0,$$

which leads to

$$z - z_0 = V_z t.$$

Added onto the circle, the motion is helical.

Note that the kinetic energy,

$$T = \frac{1}{2}m(v_x^2 + v_y^2 + v_z^2) = \frac{1}{2}m(\omega_c^2 A^2 + V_z^2),$$

is constant. This is because the force is perpendicular to the velocity, so that in any differential time element dt the work done on the particle $\boldsymbol{F} \cdot dr = dt\boldsymbol{F} \cdot v = 0$.

One should think about the implications of a velocity dependent force. Suppose one had a constant magnetic field in deep space. If a particle came through with velocity v_0, it would undergo cyclotron motion with radius $R = v_0/\omega_c$. However, if it were still its motion would remain fixed. Now, suppose an observer looked at the particle in one reference frame where the particle was moving, then changed their velocity so that the particle's velocity appeared to be zero. The motion would change from circular to fixed. Is this possible?

The solution to the puzzle above relies on understanding relativity. Imagine that the first observer believes $\boldsymbol{B} \neq 0$ and that the electric field $\boldsymbol{E} = 0$. If the observer then changes reference frames by accelerating to a velocity $\boldsymbol{v}$, in the new frame $\boldsymbol{B}$ and $\boldsymbol{E}$ both change. If the observer moved to the frame where the charge, originally moving with a small velocity v, is now at rest, the new electric field is indeed $\boldsymbol{v} \times \boldsymbol{B}$, which then leads to the same acceleration as one had before. If the velocity is not small compared to the speed of light, additional γ factors come into play, $\gamma = 1/\sqrt{1-(v/c)^2}$. Relativistic motion will not be considered in this course.

Sliding Block tied to a Wall

Another classical case is that of simple harmonic oscillations, here represented by a block sliding on a horizontal frictionless surface. The block is tied to a wall with a spring. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx.$$

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

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

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x,$$

with the angular frequency $\omega_0^2 = k/m$.

The above differential equation has the advantage that it can be solved analytically with solutions on the form

$$x(t) = Acos(\omega_0 t + \nu),$$

where A is the amplitude and ν the phase constant. This provides in turn an important test for the numerical solution and the development of a program for more complicated cases which cannot be solved analytically.

With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t),$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t).$$

We are now going to solve these equations using first the standard forward Euler method. Later we will try to improve upon this.

Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.

Since functions like cos are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t),$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}.$$

Observe that T depends only on k/m and not on the amplitude of the solution.

In addition to the periodicity test, the total energy has also to be conserved.

Suppose we choose the initial conditions

$$x(t=0) = 1 \text{ m} \qquad v(t=0) = 0 \text{ m/s},$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2.$$

We will derive this equation in our discussion on energy conservation.

An algorithm which implements these equations is included below.

- Choose the initial position and speed, with the most common choice $v(t=0) = 0$ and some fixed value for the position.
- Choose the method you wish to employ in solving the problem.
- Subdivide the time interval $[t_i, t_f]$ into a grid with step size

$$h = \frac{t_f - t_i}{N},$$

where N is the number of mesh points.

- Calculate now the total energy given by

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

- Choose ODE solver to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i.
- When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1} = t_i + h$.
- This iterative process continues till we reach the maximum time t_f.
- The results are checked against the exact solution. Furthermore, one has to check the stability of the numerical solution against the chosen number of mesh points N.

The following python program (code will be added shortly)

```
#
# This program solves Newtons equation for a block sliding on
# an horizontal frictionless surface.
# The block is tied to the wall with a spring, so N's eq takes the form:
#
#  m d^2x/dt^2 = - kx
#
# In order to make the solution dimless, we set k/m = 1.
# This results in two coupled diff. eq's that may be written as:
#
#  dx/dt = v
#  dv/dt = -x
#
# The user has to specify the initial velocity and position,
# and the number of steps. The time interval is fixed to
# t \in [0, 4\pi) (two periods)
#
```

The classical pendulum and scaling the equations

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0,$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt},$$

and

$$a = l\frac{d^2\theta}{dt^2}.$$

More on the Pendulum

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0,$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t),$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

More on the Pendulum

We define

$$\omega_0 = \sqrt{g/l},$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t,$$

with the dimensionless driving frequency

$$\hat{\omega} = \frac{\omega}{\omega_0},$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu},$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg}$$

We have

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}).$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}).$$

These are the equations to be solved. The factor Q represents the number of oscillations of the undriven system that must occur before its energy is significantly reduced due to the viscous drag. The amplitude $\hat{A}$ is measured in units of the maximum possible gravitational torque while $\hat{\omega}$ is the angular frequency of the external torque measured in units of the pendulum's natural frequency.