

# PHY321: Introduction to Classical Mechanics and plans for Spring 2021

**Morten Hjorth-Jensen**<sup>1,2</sup>

<sup>1</sup>Department of Physics and Astronomy and Facility for Rare Ion Beams (FRIB), Michigan State University, USA

<sup>2</sup>Department of Physics, University of Oslo, Norway

Jan 6, 2021

## Aims and Overview

The first week starts on Monday January 11. This week is dedicated to a review of learning material and reminder on programming aspects, useful tools, where to find information and much more.

- Monday: Introduction to the course and reminder on vectors, space, time and motion.
- Wednesday: Python programming reminder, elements from CS201 and how they are used in this course. Installing software (anaconda). See slides at <https://mhjensen.github.io/Physics321/doc/pub/week2/html/week2.html>).
- Friday: Introduction to Git and GitHub and getting started with numerical exercises.

Recommended reading: John R. Taylor, Classical Mechanics (Univ. Sci. Books 2005), <https://www.uscibooks.com/taylor2.htm>, see also <https://github.com/mhjensen/Physics321/tree/master/doc/Literature>. Chapters 1.2 and 1.3

## Monday

Overview video

## 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.

## 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  $1\text{kgm/s}^2$  and for energy 1 Joule  $1\text{kgm}^2\text{s}^{-2}$ .

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

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

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

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

## 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  $\mathbf{a}$  in three dimensions is thus defined as

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

and using the unit vectors in a cartesian system we have

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

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

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

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

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

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

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

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

We have also that

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

The rotation of a three-dimensional vector  $\mathbf{a} = (a_x, a_y, a_z)$  in the  $xy$  plane around an angle  $\phi$  results in a new vector  $\mathbf{b} = (b_x, b_y, b_z)$ . 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  $\mathbf{b} = \mathbf{R}\mathbf{a}$ , where the rotation matrix is defined as

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

**Wednesday: Python programming reminder, elements from CS201 and how they are used in this course**

Overview video

## Numerical Elements

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 opportunity 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 develop our own understanding of the scientific method by enhancing algorithmic thinking.

## Computations and the Scientific Method

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.*

## 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  $x \in [a, b]$

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

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.

```
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')
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()

```

## Analyzing the above example

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 (L<sup>A</sup>T<sub>E</sub>X, 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.

## 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  $\mathbf{a}$  in three dimensions is thus defined as

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

and using the unit vectors in a cartesian system we have

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

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

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

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

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

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

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

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

We have also that

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



The rotation of a three-dimensional vector  $\mathbf{a} = (a_x, a_y, a_z)$  in the  $xy$  plane around an angle  $\phi$  results in a new vector  $\mathbf{b} = (b_x, b_y, b_z)$ . 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  $\mathbf{b} = \mathbf{R}\mathbf{a}$ , where the rotation matrix is defined as

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

## 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 python2.7)

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 distributions 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 other 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.**

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

The inverse of a matrix is defined by

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

Relations	Name	matrix elements
$A = A^T$	symmetric	$a_{ij} = a_{ji}$
$A = (A^T)^{-1}$	real orthogonal	$\sum_k a_{ik} a_{jk} = \sum_k a_{ki} a_{kj} = \delta_{ij}$
$A = A^*$	real matrix	$a_{ij} = a_{ij}^*$
$A = A^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$A = (A^\dagger)^{-1}$	unitary	$\sum_k a_{ik} a_{jk}^* = \sum_k a_{ki}^* a_{kj} = \delta_{ij}$

### 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)
```

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)
```

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  $x_0, x_1, x_2, \dots, x_{n-1}$ . 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)
```

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 normally recommend that you use the Numpy intrinsic functions instead of the corresponding **log** function from Python's **math** module. The looping is done explicitly 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)
```

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 automatically 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)
```

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)
```

To check the number of bytes (remember that one byte contains eight bits for double precision variables), you can simply use the **itemsizes** 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  $3 \times 3$  real matrix  $\hat{A}$  as (recall that we use 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  $3 \times 3$  matrix. We can slice the matrix and print for example the first column (Python organizes 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 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 reshaping 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  $10 \times 5$  and compute the mean value and standard deviation of each column. Similarly, we can perform mathematical 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
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  $4 \times 4$  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, and*

## Friday: Introduction to Git and GitHub and getting started with numerical exercises