

# (Advanced) Physics Lab 3 (HS18) - Exercise 1

Dr. Mauro Donegà, Dr. Severian Gvasaliya

September 24, 2018

Use the provided **Exercise\_1.ipynb** Jupyter notebook to solve the exercises, some code is already present which helps you in getting started. The **TODO:** indicates where you should fill in the missing code. (You are also free to start a new notebook from scratch.) We recommend to use Python version 2.7, but Python 3 should also work.

## 1 1. Basic plotting with matplotlib

In our first example, we plot a simple curve with matplotlib.

### 1.1 a) Generating set of data

First we need to create an array of our x values for the curve to plot.

Import basic libraries:

```
In [36]: from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
# the commands above is needed to have the plots displayed inside this
# notebook instead of in an external window

In [ ]: def equallySpacedNumbers(start, end, number):
    return # TODO: use numpy to return a 1-d array of equidistant
           # floating point numbers between start and end

# look at the function output by printing:
print(equallySpacedNumbers(2.0,3.0,4))

# test your function, it should print 'True' twice
print(all(equallySpacedNumbers(2.0,10.0,9)
        == [2.,3.,4.,5.,6.,7.,8.,9.,10.]))
print(all(abs(equallySpacedNumbers(-1.2,0.2,6)
        - [-1.2,-0.92,-0.64,-0.36,-0.08,0.2]) < 1e-6))
```

Note that in the second case, we can not make an exact comparison due to rounding errors. Having such test functions is useful in case we want to replace our generator of equally spaces numbers with a different (e.g. faster) version later.

### 1.1.1 b) Simple plots

As example, we now want to make a plot of the fall time vs. the height of which an apple is dropped. For both x and y we need one-dimensional numpy arrays of the same length.

You find some help on basic plot functionalities here:

[https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html)

For more special plots, first have a look in the gallery:

<https://matplotlib.org/gallery/index.html>

which already includes many common types of plots.

```
In [ ]: def Falltime(x, g):
        return np.sqrt(2*x/g)

        # create a dataset
        true_g = 9.8
        data_x = # TODO: create array of 50 equally spaced points
                  #           between 0 and 2.0 (height in m)
        data_y = # TODO: compute fall time for all height values

        # the simplest way to plot
        # TODO: create plot out of x/y data

        # always label the axes (the '$...$' for latex style)
        # hint: use raw strings, e.g. r'height [m]'
        # TODO: set labels for x and y-axis

        # make the plot appear
        # TODO: draw plot
```

### 1.1.2 c) Load measurements from text file

Above plot now shows the prediction of the fall time. To make a comparison with our measurements, we first have to load them from the text file **measurement.txt**.

Numpy provides a very convenient function for this purpose! look at the Numpy reference:

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html>

```
In [ ]: # load data from textfile
        #           format: height[m] time[s] height_error[m] time_error[s]
        measurements = # TODO: load measurements from measurement.txt

        # look at it
        print("shape:", measurements.shape, "\n")
        print("data:\n", measurements, "\n")
        print("first column:", measurements[:, 0], "\n")
        print("last row, first two columns:", measurements[-1,0:2])
```

### 1.1.3 d) Plot with error bars

Now we want to plot the measurement data (from the text file) with error bars together with the prediction from theory. In many cases there is a non-negligible uncertainty also on the theoretical

prediction. One way of visualizing this is to plot an error band, which in practice can be done by shading the area between two curves.

In this example, use  $\sigma_g = 0.4 \frac{\text{m}}{\text{s}^2}$  as the uncertainty of  $g$ .

There are examples of plots with error bars in the gallery linked above. For more detailed options look at the reference here:

[https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.errorbar.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html)

```
In [ ]: # create additional dataset for the uncertainty band
        # use 0.4 m/s^2 as symmetric uncertainty on g
        data_y_m = # TODO: g varied down by the uncertainty
        data_y_p = # TODO: g varied up by the uncertainty

        # plot uncertainty band of theory prediction
        # TODO: plot filled area between the two curves created
        #         above, hint: use plt.fill_between

        # plot mean value on top
        # TODO: add curve for nominal value of g in a different
        #         color than the uncertainty band

        # TODO: label the axes

        # plot measurement with errors
        # TODO: plot measurements loaded from text file on top of
        #         the theory curve with circular markers, no line between
        #         them and also include errorbars! hint: plt.errorbar

        # legend
        # TODO: create a legend, hint(1): you can give a label to the
        #         individual plots with e.g. label='theory' in the creation
        #         of the plots above
        #         hint(2): use numpoints=1 as argument for plt.legend to
        #         have only 1 point of your measurements in the legend

        # optional: set axis limits
        # TODO: set axes limits to [0, 2.0] for x and [0, 0.8] for y-axis

        # optional: grid lines
        # TODO: display grid lines

        # save the figure to a pdf file
        # TODO: save the plot as "exercise-1-plot.pdf"

        # make the plot appear
        # TODO: show the plot in the notebook
```

### 1.1.4 e) Histograms

A qualitative way to check compatibility of the measurement points with theory is to make a histogram of the pulls (pulls are defined below in the code). Create the histogram of pulls and overlay the expected pull distribution, which is Gaussian.

Instead of putting the formula for the Gaussian yourself, you can use `scipy.stats.norm.pdf`, see here: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html>

```
In [ ]: import scipy.stats

# as approximation we ignore the errors on the measured
# height since they are relatively small!

heights = measurements[:, 0]
times = measurements[:, 1]
time_errors = measurements[:, 3]
predictions = Falltime(heights, true_g)

# compute pulls
pulls = (times - predictions)/time_errors

# histogram of pulls
# TODO: create normalized histogram (meaning sum of all bin
#       contents equals 1) with 10 bins
#       hint: use histtype='stepfilled'

# unit gaussian
x = # TODO: 50 points between -3.0 and 3.0
y = # TODO: unit gaussian, hint: scipy.stats.norm.pdf
plt.plot(x, y, '--', color='r', linewidth=3.0)

# always label the axes, also for histograms
# TODO: labels

# annotation
# TODO: create a annotation with text 'unit gaussian' pointing
#       to the curve plotted above. hint: plt.annotate

# save the figure to a pdf file
# TODO: save as 'exercise-1-histogram.pdf'

# TODO: show in notebook
```

### 1.1.5 f) (optional) Creating a text file of toy measurements

```
In [9]: # create toy experiments instead of real measurements here

# TODO: create a text file in the same format as the
```

```
# 'measurement.txt' with 1000 random toy experiments,
# assuming the same uncertainties as above
```

## 2. Error propagation with Python

We consider a LC circuit with resonance frequency  $\omega_0 = \frac{1}{\sqrt{LC}}$ .

$C = 150 \pm 8$  pF

$L = 1 \pm 0.1$  mH

What is the resonance frequency and its uncertainty?

### 2.1 a) Calculation by hand

The mean value is computed to:

$\omega_0 =$

Since the uncertainties for both quantities come from independent electronic components, they can safely be assumed as uncorrelated and one can compute the uncertainty of  $\omega_0$  to

$\sigma_{\omega_0} =$

### 2.2 b) Installation of 'uncertainties' package

There are packages, which make handling of uncertainties very easy, e.g. the package simply called "uncertainties". It is not included in standard packages of Anaconda and therefore has to be installed with:

```
conda install -c conda-forge uncertainties
```

This can take several minutes, since anaconda has to resolve a lot of dependencies.

### 2.3 c) Use of 'uncertainties' package

Look at the example on the official website on how to use the library:

<https://pythonhosted.org/uncertainties/>

Define  $L$  and  $C$  as ufloats and compute the resonance frequency and print the result.

How can one obtain the central value and the uncertainty separately from the ufloat object?

```
In [1]: from uncertainties import ufloat
        from uncertainties.umath import *

        # TODO: ...
```

### 2.4 d) (optional) write your own uncertainty package

We can also try to write our own class for propagating uncertainties. Look at the myufloat class below and add the missing pieces marked with **TODO:**. Then test your **myufloat** class with the LC circuit example from above. It should lead to the same result (up to floating point rounding errors).

Addition and the square root already have been implemented, complete the minimal example by adding subtraction, multiplication and division.

```

In [35]: class myufloat:
    def __init__(self, n, s=0.0):
        self.n = float(n)
        self.s = float(s)

    def __add__(self, operand):
        n = self.n + operand.n
        s = np.sqrt(self.s * self.s + operand.s * operand.s)
        return myufloat(n, s)

    def __sub__(self, operand):
        return # TODO: implement subtraction

    def __mul__(self, operand):
        return # TODO: implement multiplication

    def __div__(self, operand):
        return # TODO: implement division

    # for Python3
    def __truediv__(self, operand):
        return self.__div__(operand)

    # used in np.sqrt
    def sqrt(self):
        return myufloat(np.sqrt(self.n), np.abs(0.5/np.sqrt(self.n)*self.s))

    def __str__(self):
        return "%1.2e ± %1.2e"%(self.n, self.s)

    # used for print function
    def __repr__(self):
        return "%1.2e ± %1.2e"%(self.n, self.s)

In [ ]: C = myufloat(150e-12, 8e-12)
        L = myufloat(1e-3, 0.1e-3)

        print(myufloat(1.0)/np.sqrt(C*L))

```

So the results agree for this case! (If not check your implementation)

Lets check some other cases:

create two values with uncertainties:

$$a = 1.0 \pm 0.1$$

$$b = 2.0 \pm 0.05$$

and compute the result including uncertainty both with the uncertainties package (ufloat) and your own implementation (using myufloat) of:

$$c = \frac{a+b}{a-b}$$

are they the same? If not, why?

```
In [ ]: a1 = ufloat(1.0, 0.1)
        b1 = ufloat(2.0, 0.05)

        a2 = myufloat(1.0, 0.1)
        b2 = myufloat(2.0, 0.05)

        c1 = (a1+b1)/(a1-b1)
        c2 = (a2+b2)/(a2-b2)

        print(c1)
        print(c2)
```

What is happening here?