

Exercise_3

October 12, 2018

1 Exercise 3

In these week's exercises you will practice Poissonian statistics, delve deeper into the meaning of the error matrix and see an example of a case in which the standard error propagation formula is not directly applicable.

For the first exercise you should install the package *tqdm* from the Anaconda navigator. It provides a progress bar, which is nice to have when a long computation is running.

1.1 1. Poisson statistics

This exercise is about two variants of a counting experiment: in the first, simpler case, we will see that the observations are well described by a Poisson distribution. In the second case we will have events which are not independent from each other and we will see that the results deviate from a Poisson distribution.

Consider a beam of particles impinging on a thin target. Most particles will go through the target without interacting, while a few will be absorbed. The target is connected to a detector, which fires a signal when a particle is absorbed by the target. In the first part of the exercise we assume to have a perfect detector: it is able to detect each and every particle hitting the target.

The experiment consists in counting how many particles are absorbed by the target in a fixed time interval, e.g. 1s. We will repeat the counting n times and see how the results are distributed.

To simulate the setup, assume the following numbers: - Number of particles arriving at the target per second - Probability that a particle is absorbed by the target

```
In [ ]: '''
        Let's start by importing some useful modules and functions...
        '''
        from numpy.random import rand
        import matplotlib.pyplot as plt
        from scipy.stats import poisson
        from tqdm import tqdm_notebook as tqdm # Nice progress bar for long computations. Instal
        %matplotlib inline

        ...
        ... and by defining the relevant parameters of the experiment
        (feel free to change the values and see how the result changes)
        ...

        particle_rate = 1e6 # Number of particles arriving at the target per second
```

```

delta_t = 1 # duration of one experiment in seconds
absorption_probability = 2e-6 # Probability that a particle is absorbed by the target
n_trials = 200 # How many times you repeat the experiment

```

Write a function which decides if a single particle is absorbed by the target (return True) or not (return False).

Hint: generate a uniformly distributed random number between 0 and 1 with the `rand()` function. Use it to decide if the particle is detected or not, based on the known `absorption_probability`.

```
In [ ]: def particle_is_detected(...):
```

Now write a function to simulate the experiment running for a time `delta_t`. It should do the following: - compute how many particles reach the target during `delta_t` with the known `particle_rate`, - for each of those check if they get absorbed or not (cf. `particle_is_detected()`), - return the number of particles which are absorbed by the target during `delta_t`.

```
In [ ]: def run_experiment(...):
```

You are now ready to run the experiment, i.e. run the function. Do this a few times to get a feeling for the results: is the number of counted particles the same every time or does it change? What kind of result do you expect from the chosen `particle_rate`, `delta_t` and `absorption_probability`?

```
In [ ]: for i in range(10):
        print run_experiment()
```

Let's now analyse the results more systematically: run the experiment `n_trials` times and save the results in a list or array. Depending on your computer, this might take some time.

```
In [ ]: progress_bar = tqdm(total=n_trials, unit=' trials')
        results = []
        for i in range(n_trials):
            results.append(run_experiment(...))
            progress_bar.update()
```

Before plotting the results in a histogram, let's define the expected Poisson distribution in order to make a comparison. If you are not sure how to do this, have a look at last week's exercise. What is the expected `mu` parameter for the given `particle_rate`, `delta_t` and `absorption_probability`?

```
In [ ]: mu = ...
```

Now plot the results of the experiment together with the parent distribution. Again, have a look at last week's exercise if you need help. When plotting the histogram remember to set `density=True` in order to have it normalized to unity for a meaningful comparison with the Poisson distribution (if this option does not work, which might be the case for older versions, try replacing it with `normed=True`).

```
In [ ]: plt.hist(...) # histogram for the measurements
        plt.plot(...) # plot of the Poisson distribution
        plt.xlabel('Counted particles')
        plt.legend()
```

What do you observe? Is the data from the experiment well described by the Poisson distribution?

Let's now make a different assumption about the detector: it has no longer perfect efficiency, but whenever it detects a particle it needs some time to process the signal. During this time the detector is blind to any particle which might be absorbed by the target. In this way the recorded particles are not independent from each other anymore, and as you will see this will cause the result of the experiment to deviate from a Poisson distribution.

Modify your implementation of the function `run_experiment()` in order to account for the dead time of the detector. Assume that whenever the detector records a particle it is blind to the next 500000 particles reaching the target.

```
In [ ]: def run_experiment_dead_time(...):  
  
        # Run the function n_trials times and save the results as before
```

Plot the new results with detector dead time together with the same Poisson distribution from before. What do you observe?

```
In [ ]: plt.hist(...) # histogram for the measurements  
        plt.plot(...) # plot of the Poisson distribution  
        plt.xlabel('Counted particles')  
        plt.legend()
```

1.2 2. Correlated variables and error matrix

In this exercise you will work on a pair of correlated variables, compute the error matrix and visualize the error ellipse. Let's start from the case of two uncorrelated variables, saved in the file `data_uncorrelated.txt`. Have a look at the file: each line represents one measurement, the first number being the value of the x variable and the second number the value of the y variable.

```
In [ ]: import numpy as np  
        from matplotlib import pyplot as plt  
  
        data = np.genfromtxt('data_uncorrelated.txt')
```

Plot the data. How can you recognise that x and y are not correlated? Compare the 2D distribution in the xy plane and the histograms of the x and y values. What do you notice about e.g. the range of the axes and the position of the means?

```
In [ ]: f, ax = plt.subplots(1,3, figsize=(20, 6))  
        ax = ax.flatten()  
  
        ax[0].set_xlabel(r'$x$')  
        ax[0].set_ylabel(r'$y$')  
        ax[0].axis('equal')  
        ax[0].plot(data[:,0],data[:,1],'.')  
        ax[1].set_xlabel(r'$x$')  
        ax[1].hist(data[:,0],bins=range(-7,7),rwidth=.9)
```

```
ax[2].set_xlabel(r'$y$')
ax[2].hist(data[:,1],bins=range(-7,7),rwidth=.9)
```

Compute the covariance matrix. You can either write a function to do this yourself, or use the numpy implementation. Have a look at last week's exercise if you feel lost.

Compare your result with the error matrix you expect for two uncorrelated variables, cf. slides from Lecture 3: $\text{diag}(\sigma_x^2, \sigma_y^2)$. Is your result compatible with this expression? We will now compute the eigenvectors and eigenvalues of the covariance matrix and interpret them in terms of the properties of the distributions we just saw. As before, you can compute the values yourself or use the numpy implementation `np.linalg.eig(matrix)`

```
In [ ]: #Use this block to compute eigenvalues and eigenvectors of the covariance matrix and pri
```

For this easy case of uncorrelated variables you should recognize the following: the eigenvectors are aligned with the x and y axes and the eigenvalues are the variances of the data along the same axes; this means that the standard deviation in the x and y directions are the square root of the respective eigenvalue. Keep this in mind, as we will later see what changes if the variables are correlated.

For a visual interpretation do the following: plot again the 2D distribution of the data together with the eigenvectors multiplied by the square root of the corresponding eigenvalue (in this way the length of the vector will be the corresponding standard deviation).

```
In [ ]: plt.plot(data[:,0],data[:,1],'.',zorder=0)
plt.axis('equal')
```

```
#Compute the x and y components of the two vectors:
```

```
vector1_x = ...
vector1_y = ...
vector2_x = ...
vector2_y = ...
```

```
#Use the following function to draw the vectors. The options are needed to draw them in
```

```
plt.quiver(vector1_x, vector1_y ,angles='xy', scale_units='xy', scale=1,zorder=5)
plt.quiver(vector2_x, vector2_y ,angles='xy', scale_units='xy', scale=1,zorder=5)
```

Let's now draw an ellipse with half-axes σ_x and σ_y : this is the equivalent of the 1σ interval for a 1D Gaussian distribution. Fill in the values for σ_x, σ_y .

```
In [ ]: sigma_x = ...
sigma_y = ...
```

```
phi = np.linspace(0,2*np.pi)
```

```
ellipse_x = sigma_x*np.cos(phi) # x-coordinates of the points on the ellipse
```

```
ellipse_y = sigma_y*np.sin(phi) # y-coordinates of the points on the ellipse
```

```
plt.plot(ellipse_x,ellipse_y,'r')
```

```
plt.axis('equal')
```

```
plt.plot(data[:,0],data[:,1],'.',zorder=0)

#Redraw also the vectors (copy-paste from previous block)
plt.quiver(vector1_x, vector1_y ,angles='xy', scale_units='xy', scale=1,zorder=5)
plt.quiver(vector2_x, vector2_y ,angles='xy', scale_units='xy', scale=1,zorder=5)
```

Let us now look at the case of two correlated measurements. Load the data from *data_uncorrelated.txt* and repeat the steps from above up to the drawing of the vectors; do not draw the ellipse yet, we will do that in the next step. You should be able to copy-paste most of the code from the uncorrelated case.

```
In [ ]: data_correlated = np.genfromtxt('data_correlated.txt')
```

You should now see that the eigenvectors are not aligned with the coordinate axes anymore. However, as before, the eigenvectors represent the direction of the largest spread of the data, and the eigenvalues, i.e. the variances, define how large this spread is.

Let's now draw the ellipse. There are different ways in which this can be done. Let's start by determining the angle of rotation *theta*. *Hint*: take the *x* and *y* components of one of the eigenvectors and use the *np.arctan2()* function.

```
In [ ]: theta = ...
```

To draw the rotated ellipse, define the *x* and *y* coordinates as before, and then rotate them by multiplying them with a rotation matrix by the angle *theta*.

```
In [ ]: phi = np.linspace(0,2*np.pi)
        ellipse_x = sigma[0]*np.cos(phi) # x-coordinates of the points on the ellipse
        ellipse_y = sigma[1]*np.sin(phi) # y-coordinates of the points on the ellipse

        rotation = np.array([[np.cos(theta),np.sin(theta)],[-np.sin(theta),np.cos(theta)]])
        ellipse = np.dot(rotation,[ellipse_x,ellipse_y]) # dot product
```

Now we are ready to plot everything together.

```
In [ ]: plt.plot(data_correlated[:,0],data_correlated[:,1],'.',zorder=0)
        for i in range(2):
            plt.quiver(sigma[i]*eigvec[0,i],sigma[i]*eigvec[1,i],angles='xy', scale_units='xy',
            plt.plot(ellipse[0,:],ellipse[1,:], 'r')
            plt.axis('equal')
```

1.2.1 Bonus

The data for this exercise has been generated with the following code. Feel free to change the parameters and rerun the exercise.

```
In [ ]: import numpy as np

        n_samples = 1000
        mu = np.array([0.,0.])
```

```

var_x = 4.
var_y = 1.
cov_xy = 1.
r = np.array([
    [ var_x, cov_xy,],
    [ cov_xy, var_y,]
])

y = np.random.multivariate_normal(mu, r, size=n_samples)

with open('output.txt', 'w') as outfile:
    np.savetxt(outfile, y, fmt='%3.2f')

```

1.3 3. Computing uncertainties on inefficiencies

Consider an imperfect particle detector: out of all the particles hitting the detector, a fraction passes through unnoticed. The efficiency of the detector, i.e. the fraction of particles which are detected, is a very important parameter for any experimental setup. Suppose you want to measure the efficiency of a new detector. A possible approach is the following: you shoot n particles on the detector, and count the number of signals k which are recorded. The efficiency is then given by $\varepsilon = k / n$. What is the uncertainty on this quantity? As a first approach, let's assume that k and n are Poisson distributed (and thus $\delta k = \sqrt{k}$ and $\delta n = \sqrt{n}$) and that we can apply the standard error propagation formula you saw in lecture 1:

$$\delta f = \sqrt{\sum_{i=1}^N \left(\left. \frac{\partial f}{\partial x_i} \right|_{x_i=x_i^0} \delta x_i \right)^2}.$$

- Show that this formula yields the following result:

$$\delta \varepsilon = \sqrt{\frac{k}{n^2} + \frac{k^2}{n^3}}.$$

What happens to the uncertainty for *extreme* values of k , i.e. $k = 0$ and $k = n$? Do these results make sense? Remember that the efficiency is by definition a number between 0 and 1.

The source of the problem is that k and n are not independent (the particles which are recorded are a subset of all particles which hit the detector). A way to handle this is noting that the efficiency measurement is in fact a binomial process with total events n and success probability ε (see slides of Lecture 3). - Using the known variance of the binomial distribution show that in this case the uncertainty is given by

$$\delta \varepsilon = \sqrt{\frac{\varepsilon(1 - \varepsilon)}{n}}.$$

An equivalent approach is to consider, instead of the total number of particles n , the number n_f of particles which fail to be detected. In this approach, $n = k + n_f$ is not fixed anymore and k and n_f are uncorrelated; thus, the standard error formula is valid. - Show that applying the standard error formula to $\varepsilon = k / (k + n_f)$ yields again

$$\delta \varepsilon = \sqrt{\frac{\varepsilon(1 - \varepsilon)}{n}}.$$

Note that when evaluating this formula one has to use the measured (estimated) value of ε , since the true value is unknown. This is a good approximation for *intermediate* values of k , i.e. ε not too close to 0 or 1. The exact meaning of *too close* is a matter of judgement - the extreme values $\varepsilon = 0$ and $\varepsilon = 1$ are clearly too close, as they yield 0 uncertainty which is nonsense.