MoMeDa_6

June 24, 2024

1 Moderne Methoden der Datenanalyse SS2024

2 Practical Exercise 6

2.1 Exercise 6.1: Hypothesis Testing

"Is this a new discovery or just a statistical fluctuation?" Statistics offers some methods to give a quantitative answer. But these methods should not be used blindly. In particular, one should know exactly what the obtained numbers mean and what they don't mean.

2.1.1 Exercise 6.1.1 (obligatory to solve either 6.1.1 or 6.1.2)

The following table shows the number of winners in a horse race for different track numbers:

track ||| 1 || 2 || 3 || 4 || 5 || 6 || 7 || 8 |

|———————|||———||———||———||———||———||———||———||———| | # winners ||| 29 || 19 || 18 || 25 || 17 || 10 || 15 || 11 |

Use a χ^2 test to check the hypothesis that the track number has *no* influence on the chance to win. Define a significance level, e.g., $\alpha = 5\,\%$ or $\alpha = 1\,\%$, *before* you do the test.

[14]:
```
# pick your poison
#from ROOT import TMath
from scipy.stats import chi2

def exercise6_1_1(confidenceLevel):

    # numbers given in the exercise
    nTracks = 8
    nWin = [29, 19, 18, 25, 17, 10, 15, 11]

    expected = np.average(nWin)

    return
```

[15]:
```
np.sum([29, 19, 18, 25, 17, 10, 15, 11])
```

[15]: 144

2.1.2 Exercise 6.1.2 (obligatory to solve either 6.1.1 or 6.1.2)

In a counting experiment, 5 events are observed while $\mu_B = 1.8$ background events are expected. Is this a significant ($= 3\sigma$) excess? Calculate the probability of observing 5 or more events when the expectation value is 1.8 using Poisson statistics.

[2]:
```
import numpy as np

# pick your poison
#from ROOT import gRandom, TMath, Math
from scipy.stats import poisson, norm
```

[9]:
```
def exercise6_1_2():

    # numbers given in the exercise
    nBackground = 1.8
    nObserved = 5

    return 1-poisson.cdf(nObserved-1,nBackground)

display(exercise6_1_2())
exercise6_1_2() < 0.0027
```

```
0.036406661001083473
```

[9]:
```
False
```

2.2 Exercise 6.2: Parameter Estimation

2.2.1 Exercise 6.2.1 (voluntary)

Consider the following set of values approximately following a Gaussian distribution (see also exercise_6_2_1.csv):

xi

yi

σi

xi

yi

σi

xi

yi

σi

xi

yi	σi
0.46	
0.19	
0.05	
0.69	
0.27	
0.06	
0.71	
0.28	
0.05	
1.04	
0.62	
0.01	
1.11	
0.68	
0.05	
1.14	
0.70	
0.07	
1.17	
0.74	
0.08	
1.20	
0.81	
0.09	
1.31	
0.93	
0.10	
2.03	
2.49	
0.03	

2.14

2.73

0.04

2.52

3.57

0.01

3.24

3.90

0.07

3.46

3.55

0.03

3.81

2.87

0.03

4.06

2.24

0.01

4.93

0.65

0.10

5.11

0.39

0.07

5.26

0.33

0.05

5.38

0.26

0.08

$$y(x) = a_1 \cdot \exp\{-\frac{1}{2}(\frac{x - a_2}{a_3})^2\}$$

with σ_i being the uncertainty on y_i.

- Determine the values of the three parameters a_1, a_2, and a_3 as well as their uncertainties.
- Afterwards, use the transformation $z = \ln(y)$ to obtain the linear function $z(x) = b_1 + b_2 x + b_3 x^2$. Determine the new parameters b_1, b_2, and b_3 and uncertainties in two ways and compare the results: first, by fitting this new function; second, by a calculation using the results for a_j and the transformation $z = \ln(y)$.

ROOT approach

[ ]:
```
from ROOT import TF1, TGraphErrors, TCanvas
import numpy as np

def exercise6_2_1(verbose = 0):
    # read exercise_6_2_1.csv or type are values explicitly

    # fit a Gaussian function

    #print(" --------  Fit Parameters ---------- ")
    #print(" a1 =     {0:.3f} +/- {1:.3f} ".format(a1, err_a1))
    #print(" a2 =     {0:.3f} +/- {1:.3f} ".format(a2, err_a2))
    #print(" a3 =     {0:.3f} +/- {1:.3f} ".format(a3, err_a3))
    #print(" chi2 =   {0:.3f} ".format(chi))


    # now with log

    # calculation


    #print(" --------  Fit Parameters ---------- ")
    #print(" b1 =     {0:.3f} +/- {1:.3f} ".format(b1, err_b1))
    #print(" b2 =     {0:.3f} +/- {1:.3f} ".format(b2, err_b2))
    #print(" b3 =     {0:.3f} +/- {1:.3f} ".format(b3, err_b3))
    #print(" chi2 =   {0:.3f} ".format(chi2))

    #print("\n --------  Estimated Parameters ---------- ")
    #print(" b1 =     {0:.3f} +/- {1:.3f} ".format(b1est, err_b1est))
    #print(" b2 =     {0:.3f} +/- {1:.3f} ".format(b2est, err_b2est))
    #print(" b3 =     {0:.3f} +/- {1:.3f} ".format(b3est, err_b3est))


    # draw graphs

    return
```

Python approach

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     import scipy.optimize
```

```
[ ]: def exercise_6_2_1():
         # read exercise_6_2_1.csv or type are values explicitly

         # fit a Gaussian function

         #print("Fit Result:")
         #print("a1 = {0:.3f} +- {3:.3f}\na2 = {1:.3f} +- {4:.3f}\na3 = {2:.3f} +- {5:
      ↪.3f}".
         #      format(a1, a2, a3, a1err, a2err, a3err))


         # now with log

         # calculation


         #print("Fit Result:")
         #print("b1 = {0:.3f} +- {3:.3f}\nb2 = {1:.3f} +- {4:.3f}\nb3 = {2:.3f} +- {5:
      ↪.3f}".
         #      format(*bopt, np.sqrt(bcov[0, 0]), np.sqrt(bcov[1, 1]), np.
      ↪sqrt(bcov[2, 2])))
         #print("Transformed Result:")
         #print("b1 = {0:.3f} +- {3:.3f}\nb2 = {1:.3f} +- {4:.3f}\nb3 = {2:.3f} +- {5:
      ↪.3f}".
         #      format(b1, b2, b3, b1err, b2err, b3err))

         # plot

         return
```

2.2.2 Exercise 6.2.2 (obligatory)

This exercise aims at constructing the error band around a function, $f(x)$, fitted to data points (x, y) - i.e., the errors on the fitted parameters are transformed into errors on the value of the function at each value of x. (If you do not want to use `ROOT` to optimize the curve fit in this exercise, you can instead use the fitting algorithm provided by scipy, e.g., scipy.optimize.curve_fit - see Python approach).

Let us attack the problem in steps:

- First, define a function for our problem: a straight line as $f(x) = a + bx$ with parameters a and b.
- Next, consider `npoints=11` data points in the interval `[xof, xof+npoints-1]` with `xof=10`

and y(x) = x. To simulate measurement errors, shift the data points in y-direction by a random shift, drawn from a Gaussian distribution with $\sigma = 0.5$ and $\mu = 0.0$.

- Now fit the straight line defined above to your data points.

  Draw the data points and the fit result. Print the correlation coefficient of the errors on the parameters a and b.

  Try to give an intuitive argument why the two parameters a (axis intercept) and b (slope) are strongly correlated.

- Next, construct the error band around the fit function: write a function `make_band(f, cov, withCorr=0)`, which takes the function f and the covariance matrix, as input arguments and calculates for each value of x the error on y, $\Delta_y(x)$. As a first approach, use the simple formula for error propagation, which, in this case, results in $\Delta_y(x)^2 = \Delta_a^2 + (x \cdot \Delta_b)^2$ if correlations are neglected.

  Draw the curves $y(x) \pm \Delta_y(x)$, i.e., the "error band" on top of your data and fit result. Does this look correct?

- Derive the appropriate formula for error propagation taking into account the correlation of the errors. Re-calculate $\Delta_y(x)$ and plot the corresponding error band. Compare with the above result obtained without correlations.

To get a better idea of what happens here and what the effect of the correlations is, you might want to repeat the whole exercise setting `xof=-5.`, meaning now that the mean of the x-values of the data points is approximately at 0. Alternatively, you may choose the parametrisation of the function as `y(x) = a' + b(x-(xof+(npoints-1)/2))`.

Python approach

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize
from copy import deepcopy
```

[4]:
```
# First: Define a function (e.g. a straight line)
def linear(x, *p):
    return x*p[0]+p[1]
```

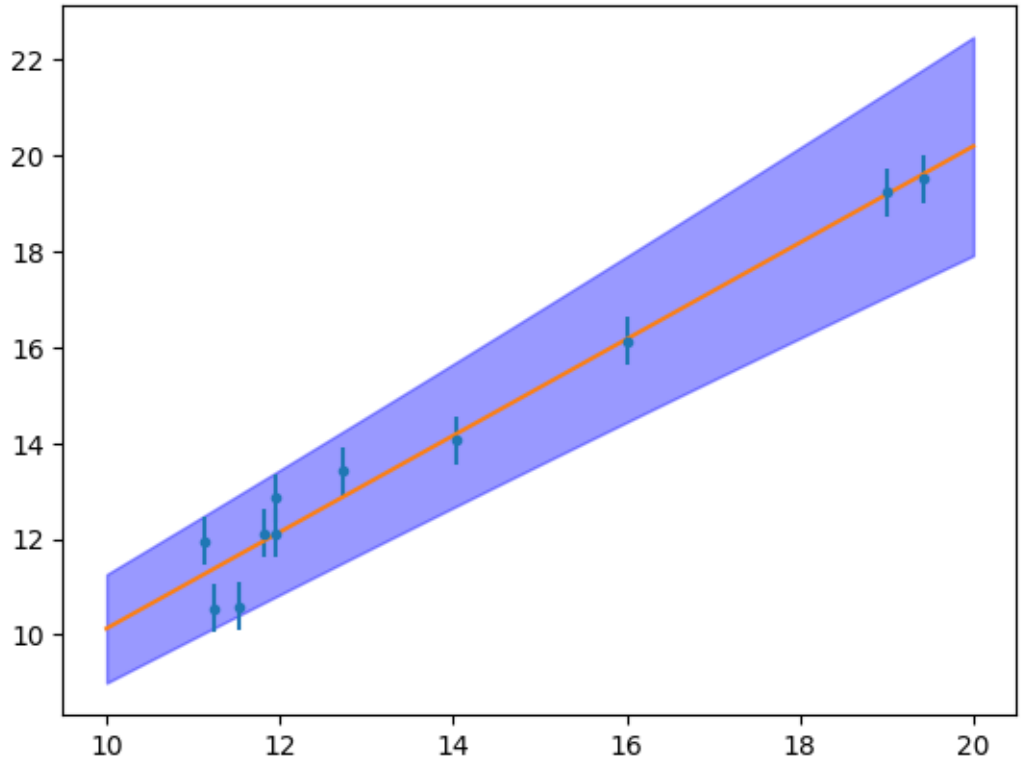
[44]:
```
def make_band_p(pcov, x_range, use_correlation=False):
    # As first approach, use the simple formula for error propagation
    pcov2 = deepcopy(pcov)
    if not use_correlation:
        pcov2[0,1] = pcov2[1,0] = 0

    grad = np.stack([x_range, np.ones(x_range.shape)])

    res = np.sum(np.dot(pcov2,grad)* grad, axis=0)
    return res
```

```
    # Derive the appropriate formula taking into account the correlation of the␣
 ↪errors
```

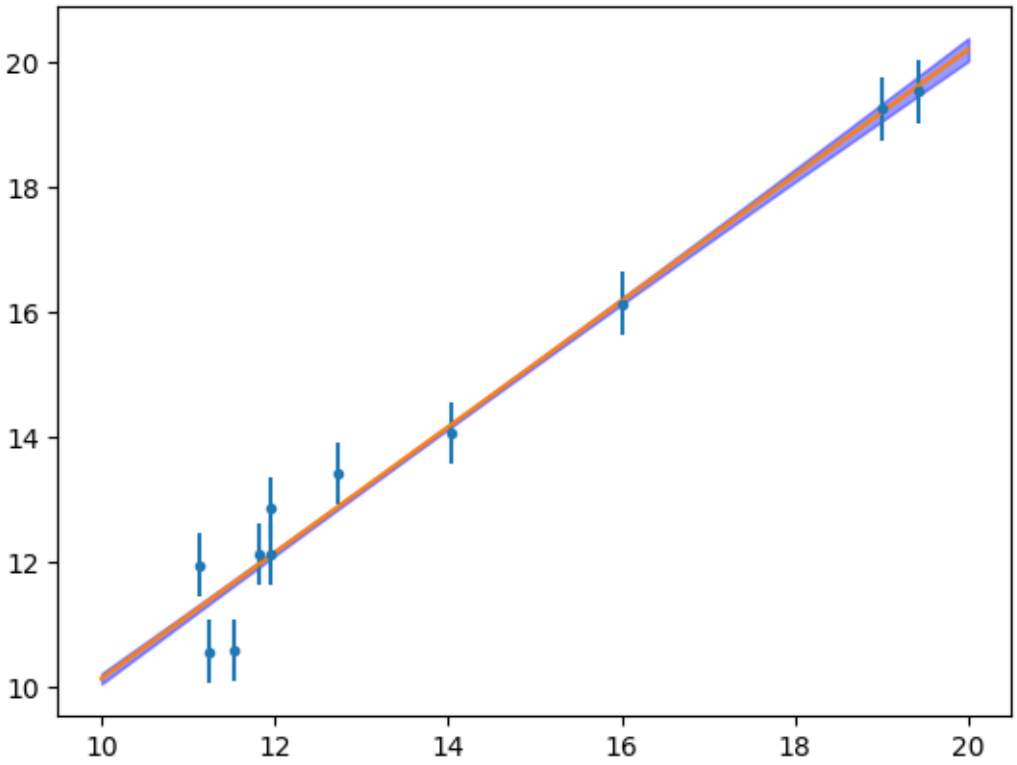
[53]:
```
def exercise_6_2_2(xof=10):
    # Next: `npoints` data points in the interval [xof, xof+npoints-1] with y(x)␣
 ↪= x.
    # Shift the data points in y-direction
    npoints = 11
    x = np.random.uniform(low=xof, high=xof+npoints-1, size=npoints)
    y = x + np.random.normal(0,0.5,size=npoints)

    # Now fit the straight line defined above to your data points
    fit_res = scipy.optimize.curve_fit(linear, x, y, p0=(1,1))

    print(fit_res)

    # Define above the function `make_band` and draw the "error band" on top of␣
 ↪your data and fit result
    # w/o correlation
    xx = np.linspace(xof, xof+npoints-1)
    plt.errorbar(x,y,yerr=0.5,fmt=".")
    plt.plot(xx, linear(xx, *fit_res[0]))
    plt.fill_between(xx,
                     linear(xx, *fit_res[0])-make_band_p(fit_res[1], xx,␣
 ↪use_correlation=False),
                     linear(xx, *fit_res[0])+make_band_p(fit_res[1], xx,␣
 ↪use_correlation=False),
                     alpha=0.4, color="blue")
    plt.show()

    # w/ correlation
    xx = np.linspace(xof, xof+npoints-1)
    plt.errorbar(x,y,yerr=0.5,fmt=".")
    plt.plot(xx, linear(xx, *fit_res[0]))
    plt.fill_between(xx,
                     linear(xx, *fit_res[0])-make_band_p(fit_res[1], xx,␣
 ↪use_correlation=True),
                     linear(xx, *fit_res[0])+make_band_p(fit_res[1], xx,␣
 ↪use_correlation=True),
                     alpha=0.4, color="blue")
    plt.show()

exercise_6_2_2(xof=10)
```

```
(array([1.00642896, 0.06331675]), array([[ 0.00382117, -0.0524028 ],
       [-0.0524028 ,  0.75138922]]))
```

```
(50,) (2, 50)
(50,)

(50,) (2, 50)
(50,)
```

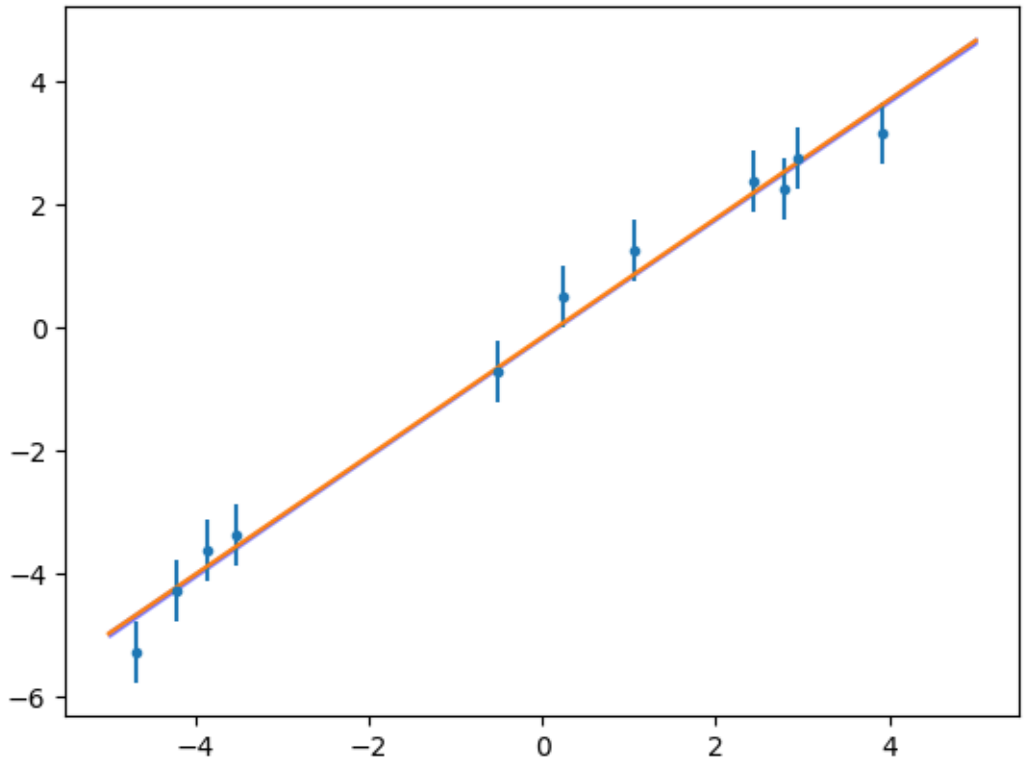



```
(50,) (2, 50)
(50,)

(50,) (2, 50)
(50,)
```

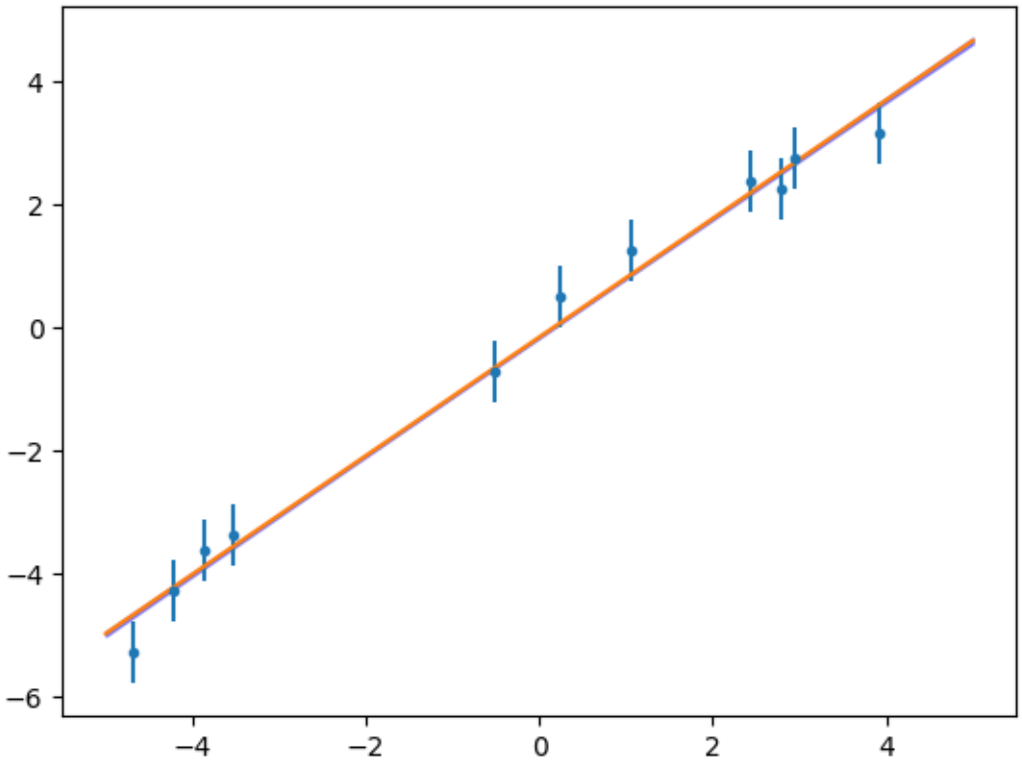


[54]:
```
exercise_6_2_2(xof=-5)
```

```
(array([ 0.96363562, -0.15369415]), array([[0.00122651, 0.00038062],
       [0.00038062, 0.01186361]]))

(50,) (2, 50)
(50,)

(50,) (2, 50)
(50,)
```




```
(50,) (2, 50)
(50,)

(50,) (2, 50)
(50,)
```


4
2
0
−2
−4
−6
−4
−2
0
2
4