MoMeDa_5

June 18, 2024

1 Exercise 5

1.1 Exercise 5.1: Parameterization of Data

1.1.1 Exercisse 5.1.1 (obligatory)

If the underlying probability distribution function (PDF) of a dataset is unknown, empirical fit functions have to be employed. The most common empirical fit functions are n-th order polynomials with constant coefficients p_k to be determined by the fit:

$$P_n(x) = \sum_{k=0}^{n} p_k \, x^k \ .$$

The fit results can usually be "stabilized" by using orthogonal polynomials

$$L_n(x) = \sum_{k=0}^{n} p_k \, l_k(x) \ ,$$

where $l_k(x)$ are Legendre polynomials, which can be defined recursively by

$$l_0(x) = 1; \quad l_1(x) = x; \quad (k+1)\, l_{k+1}(x) = (2k+1)\, x\, l_k(x) - k\, l_{k-1}(x) \ .$$

The Legendre polynomials fulfill the orthogonality relation

$$\int_{-1}^{1} \mathrm{d}x \, l_m(x) \; l_n(x) = \frac{2}{2n-1} \delta_{mn} \ ,$$

where δ_{mn} denotes the Kronecker delta.

Fit the data points given by the following pairs of x and y values assuming a constant uncertainty of $\sigma_y = 0.5$ for y and no uncertainty for x:

```
x = { -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1,
       0.0,  0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 }
y = { 5.0935, 2.1777, 0.2089, -2.3949, -2.4457, -3.0430, -2.2731,
     -2.0706, -1.6231, -2.5605, -0.7703, -0.3055, 1.6817, 1.8728,
      3.6586, 3.2353, 4.2520, 5.2550, 3.8766, 4.2890 }
```

1. Use $P_2(x)$, $P_3(x)$, ..., $P_7(x)$ as fit functions.
2. Use $L_2(x)$, $L_3(x)$, ..., $L_7(x)$ as fit functions.

Plot the data and the fitted curves for all fits. Compare the resulting values for p_k and their correlation matrices (to be obtained most conveniently via the `GetCorrelationMatrix()` method of Root class `TFitResult` or via the `scipy.optimize.curve_fit()` function). In which sense is the fit using orthogonal polynomials "more stable"? Discuss which order you would choose for the fit function.

Hint: A convenient framework for fitting and visualisation of problems like this one is included in the Root class `TGraphErrors` and its methods.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
```

[3]:
```
nPoints = 20
data_x = np.array([-0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0.0,  0.
 ↪1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0], dtype=float)
data_y = np.array([5.0935, 2.1777, 0.2089, -2.3949, -2.4457, -3.0430, -2.2731,␣
 ↪-2.0706, -1.6231, -2.5605, -0.7703, -0.3055, 1.6817, 1.8728, 3.6586, 3.2353,␣
 ↪4.2520, 5.2550, 3.8766, 4.2890], dtype=float)
sigma_x = np.array(nPoints*[0.], dtype=float)
sigma_y = np.array(nPoints*[0.5], dtype=float)
```

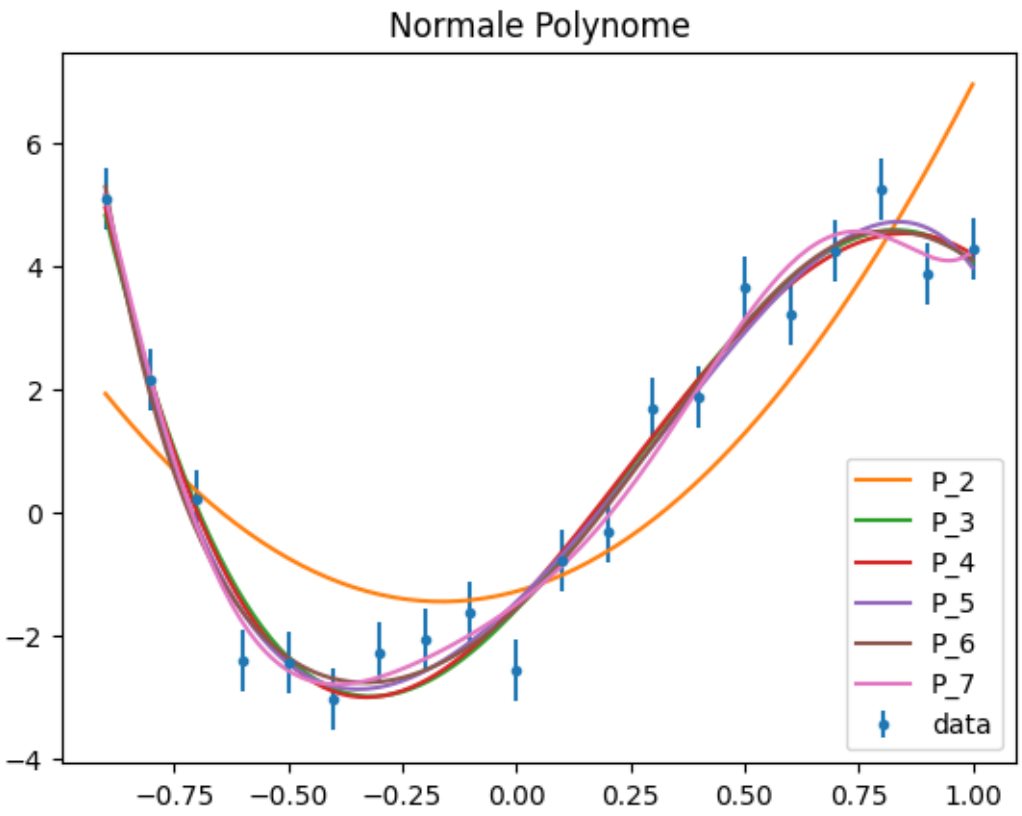
[4]:
```
# define polynomials
def P_2(x, a, b, c):
    return a + b * x + c * x**2

def P_3(x, a, b, c, d):
    return a + b * x + c * x**2 + d * x**3

def P_4(x, a, b, c, d, e):
    return a + b * x + c * x**2 + d * x**3 + e * x**4

def P_5(x, a, b, c, d, e, f):
    return a + b * x + c * x**2 + d * x**3 + e * x**4 + f * x**5

def P_6(x, a, b, c, d, e, f, g):
    return a + b * x + c * x**2 + d * x**3 + e * x**4 + f * x**5 + g * x**6

def P_7(x, a, b, c, d, e, f, g, h):
    return a + b * x + c * x**2 + d * x**3 + e * x**4 + f * x**5 + g * x**6 + h *␣
 ↪x**7

# define Legendre polynomials
def L_2(x, a, b, c):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.)

def L_3(x, a, b, c, d):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.) + d * 0.5 * (5. * x**3 - 3. * x)
```

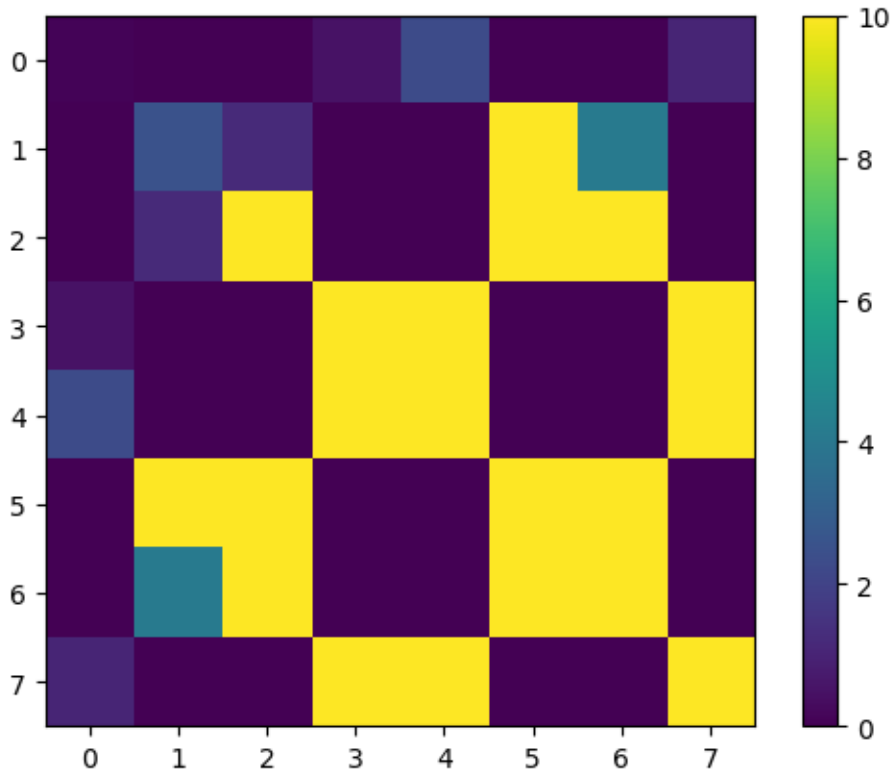
```
def L_4(x, a, b, c, d, e):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.) + d * 0.5 * (5. * x**3 - 3. *␣
 ↪x) + e * 0.125 * (35. * x**4 - 30. * x**2 + 3.)

def L_5(x, a, b, c, d, e, f):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.) + d * 0.5 * (5. * x**3 - 3. *␣
 ↪x) + e * 0.125 * (35. * x**4 - 30. * x**2 + 3.) + f * 0.125 * (63. * x**5 - 70.
 ↪ * x**3 + 15. * x)

def L_6(x, a, b, c, d, e, f, g):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.) + d * 0.5 * (5. * x**3 - 3. * x)␣
 ↪+ e * 0.125 * (35. * x**4 - 30. * x**2 + 3.) + f * 0.125 * (63. * x**5 - 70. *␣
 ↪x**3 + 15. * x) + g * 0.0625 * (231. * x**6 - 315. * x**4 + 105. * x**2 - 5.)

def L_7(x, a, b, c, d, e, f, g, h):
    return a + b * x + c * 0.5 * (3. * x**2 - 1.) + d * 0.5 * (5. * x**3 - 3. *␣
 ↪x) + e * 0.125 * (35. * x**4 - 30. * x**2 + 3.) + f * 0.125 * (63. * x**5 - 70.
 ↪ * x**3 + 15. * x) + g * 0.0625 * (231. * x**6 - 315. * x**4 + 105. * x**2 - 5.
 ↪) + h * 0.0625 * (429. * x**7 - 693. * x**5 + 315. * x**3 -35. * x)
```

[5]:
```
def PP(x, *coeffs):
    funcs = [P_2, P_3, P_4, P_5, P_6, P_7]
    return funcs[len(coeffs)-3](x, *coeffs)

def LL(x, *coeffs):
    funcs = [L_2, L_3, L_4, L_5, L_6, L_7]
    return funcs[len(coeffs)-3](x, *coeffs)

plt.errorbar(data_x, data_y, sigma_y, label="data", fmt=".")
overhang = 0
xlin = np.linspace(np.min(data_x)-overhang, np.max(data_x)+overhang, 200)
for i in range(2,8):
    coeffs = [1,1,1,1,1,1,1,1]
    res = scipy.optimize.curve_fit(PP, data_x, data_y, p0=coeffs[:i+1],␣
 ↪sigma=sigma_y)
    plt.plot(xlin, PP(xlin, *res[0]), label=f"P_{i}")
    print(np.round(res[0],2))
    np.set_printoptions(suppress=True)
    if i == 7: print(np.round(res[1],2))
    np.set_printoptions(suppress=False)
plt.legend()
plt.title("Normale Polynome")
plt.show()

plt.imshow(res[1], vmin=0, vmax=10)
plt.colorbar()
```

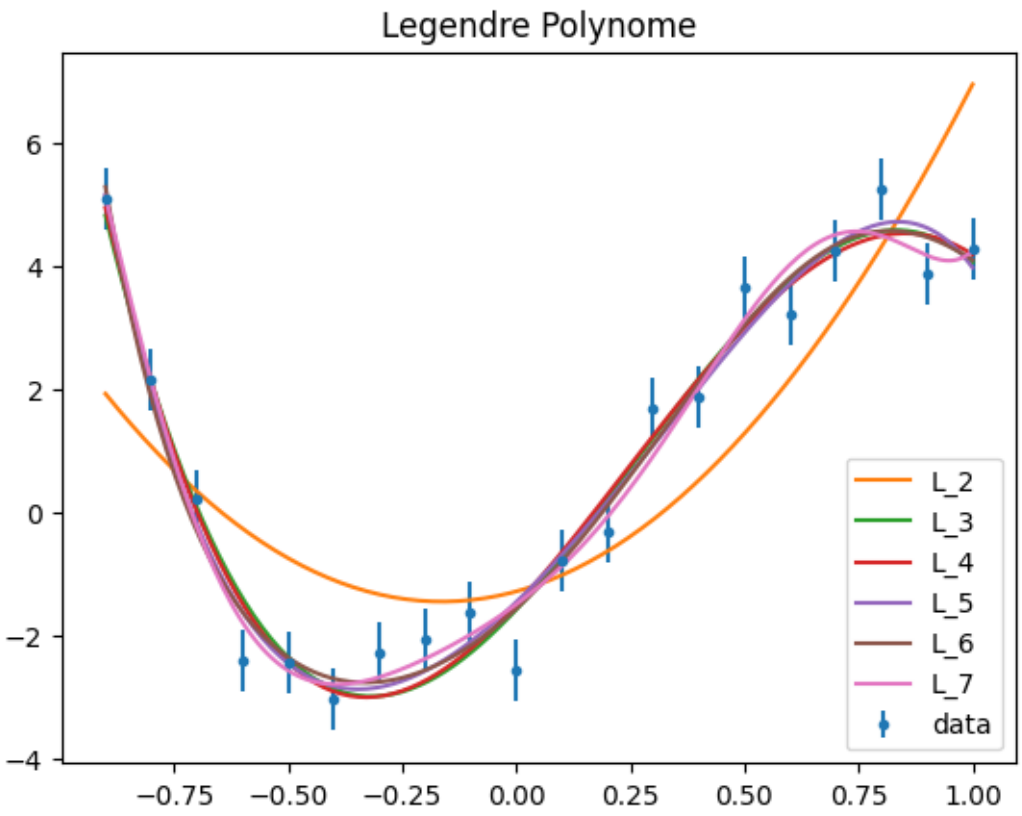
```
plt.show()

plt.errorbar(data_x, data_y, sigma_y, label="data", fmt=".")
for i in range(2,8):
    coeffs = [1,1,1,1,1,1,1,1]
    res = scipy.optimize.curve_fit(LL, data_x, data_y, p0=coeffs[:i+1],␣
 ↪sigma=sigma_y)
    plt.plot(xlin, LL(xlin, *res[0]), label=f"L_{i}")
    print(np.round(res[0],2))
    if i == 7: print(np.round(res[1],2))
plt.legend()
plt.title("Legendre Polynome")
plt.show()

plt.imshow(res[1], vmin=0, vmax=0.5)
plt.colorbar()
plt.show()
```

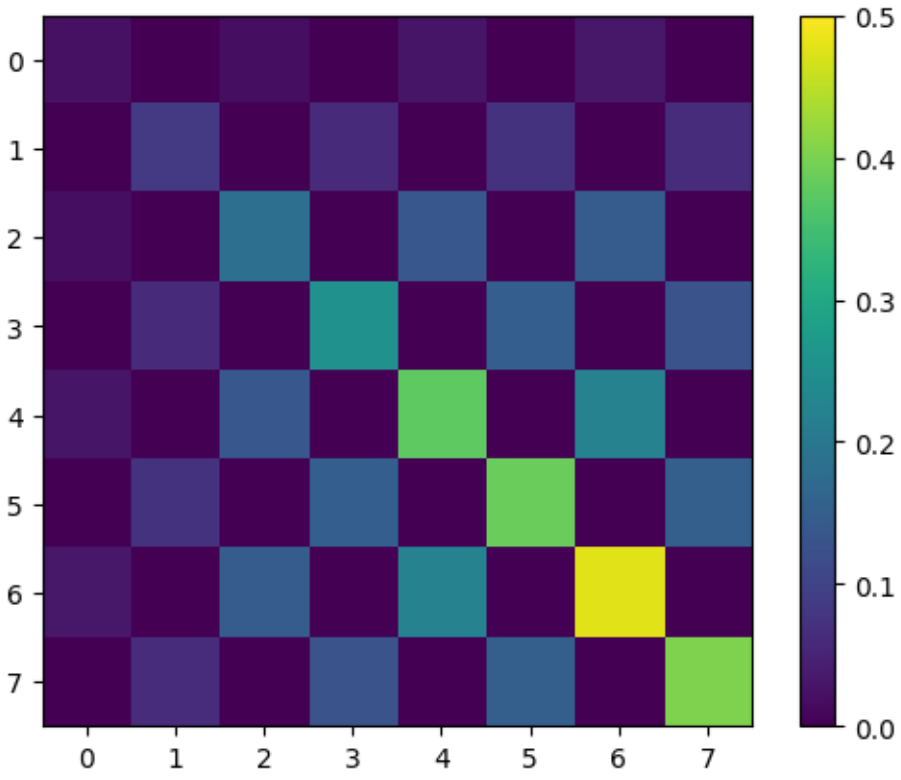
```
[-1.28  2.03  6.22]
[-1.57  7.89  7.71 -9.96]
[ -1.5    7.97   6.9  -10.15   0.97]
[-1.45  7.12  6.28 -6.03  1.93 -3.86]
[-1.56  6.87  8.69 -4.49 -5.66 -5.63  5.89]
[ -1.49   5.48   6.62   8.79   2.31 -36.8   -1.36  20.71]
[[   0.09   -0.04   -0.93    0.49    2.27   -1.35   -1.56    1.02]
 [  -0.04    2.52    1.18  -17.24   -4.54   33.44    4.12  -19.52]
 [  -0.93    1.18   16.7   -13.81  -48.58   38.25   36.43  -29.06]
 [   0.49  -17.24  -13.81  141.53   53.16 -301.04  -48.33  186.26]
 [   2.27   -4.54  -48.58   53.16  154.21 -147.2  -121.85  111.85]
 [  -1.35   33.44   38.25 -301.04 -147.2   678.5   133.82 -437.25]
 [  -1.56    4.12   36.43  -48.33 -121.85  133.82   99.65 -101.69]
 [   1.02  -19.52  -29.06  186.26  111.85 -437.25 -101.69  290.53]]
```


Normale Polynome
6
4
2
0
−2
−4
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
P_2
P_3
P_4
P_5
P_6
P_7
data




```
[0.79 2.03 4.14]
[ 1.    1.92  5.14 -3.98]
[ 1.    1.88  5.16 -4.06  0.22]
[ 1.03  1.84  5.29 -4.13  0.44 -0.49]
[ 1.04  1.77  5.36 -4.29  0.54 -0.71  0.41]
[ 0.98  1.89  5.08 -4.05  0.1  -0.42 -0.09  0.77]
[[ 0.02 -0.01  0.02 -0.02  0.03 -0.03  0.03 -0.03]
 [-0.01  0.08 -0.05  0.06 -0.09  0.07 -0.1   0.06]
 [ 0.02 -0.05  0.18 -0.11  0.14 -0.15  0.15 -0.14]
 [-0.02  0.06 -0.11  0.25 -0.18  0.15 -0.21  0.13]
 [ 0.03 -0.09  0.14 -0.18  0.38 -0.23  0.22 -0.23]
 [-0.03  0.07 -0.15  0.15 -0.23  0.39 -0.27  0.15]
 [ 0.03 -0.1   0.15 -0.21  0.22 -0.27  0.48 -0.26]
 [-0.03  0.06 -0.14  0.13 -0.23  0.15 -0.26  0.4 ]]
```


Legendre Polynome
6
4
2
0
−2
−4
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
L_2
L_3
L_4
L_5
L_6
L_7
data



It looks like Order 3 Polynomials are sufficient to fit the data given the uncertanties. Anything above order 4 would be overfitting. The 2D-color-Plots show the values of the covariance Matrix. It can be seen that the variance for Lagrange-Polomials is way smaller in general and the covariance between different coefficients is also way smaller. That makes them numerically more stable. Adding a new, higher polinomial does not change the other coefficients.

1.1.2 Exercise 5.1.2: (obligatory)

In an accelerator experiment, the following data are numbers of events measured in 60 energy intervals equally distributed between 0 and 3 GeV:

6	1	10	12	6	13	23	22	15	21	23	26	36	25	27	35	40	44	66	81
75	57	48	45	46	41	35	36	53	32	40	37	38	31	36	44	42	37	32	32
43	44	35	33	33	39	29	41	32	44	26	39	29	35	32	21	21	15	25	15

The data show a signal resonance visible on top of a background sample. For the uncertainties of all data points we assume the statistical uncertainty according to a Poisson distribution. The goal of this exercise is to extract information on the signal by parameterizing both signal and background. The information we are interested in are the width of the signal (which is related to the lifetime) and the number of signal events. Let us assume that the background can be parametrized as a polynomial of second order in the energy (i.e., a function with 3 parameters), and the signal as a Lorentz function (also 3 parameters) given by

$$L\left(x; A_{\text{norm}}, \mu, \Gamma\right) = \frac{A_{\text{norm}}}{\pi} \frac{\Gamma/2}{\left(x-\mu\right)^2 + \Gamma^2/4}$$

There are two possible methods to extract the signal:

1. Fit the data with a function with 6 parameters composed by the signal function plus the background function.
2. Define two intervals, left and right of the signal peak, to fit the background function. Then, fit the signal function to the data in the signal region after subtracting the background function.

   There are (at least) two ways how to exclude certain points for the fit. Either you can define new arrays for the fit which contain only a subset of the original data points, or you can define your own fit function which excludes certain intervals. For a Root example how to do this (not in Python, but in C) see here: https://root.cern/doc/master/fitExclude_8C.html.

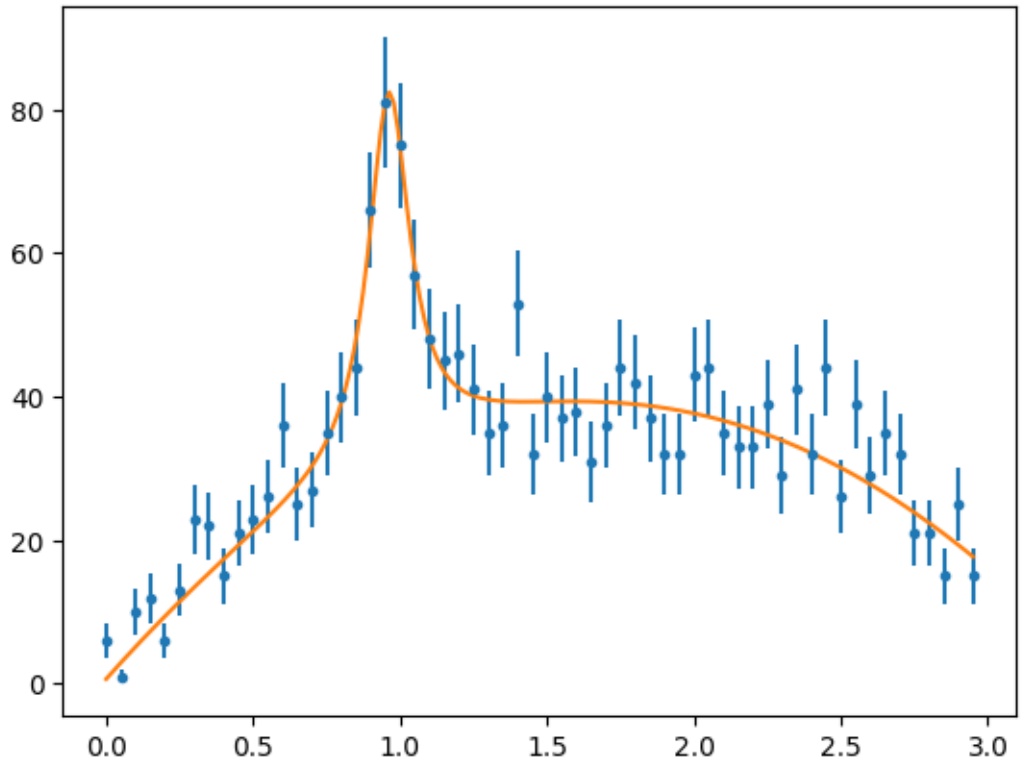
Plot the fitted functions on top of the data. Determine the width of the Lorentz peak and the number of signal events and their statistical uncertainties, and compare the results of both methods.

[6]:
```
nPoints = 60
data_x = np.array(np.arange(0, 3, 0.05), dtype=float) # 3 GeV / 60 bins = 0.05␣
 ↪GeV per bin
data_y = np.array([6, 1, 10, 12, 6, 13, 23, 22, 15, 21, 23, 26, 36, 25, 27, 35,␣
 ↪40, 44, 66, 81, 75, 57, 48, 45, 46, 41, 35, 36, 53, 32, 40, 37, 38, 31, 36,␣
 ↪44, 42, 37, 32, 32, 43, 44, 35, 33, 33, 39, 29, 41, 32, 44, 26, 39, 29, 35,␣
 ↪32, 21, 21, 15, 25, 15], dtype=float)
sigma_x = np.array(nPoints*[0], dtype=float)
sigma_y = np.array(np.sqrt(data_y), dtype=float)
```

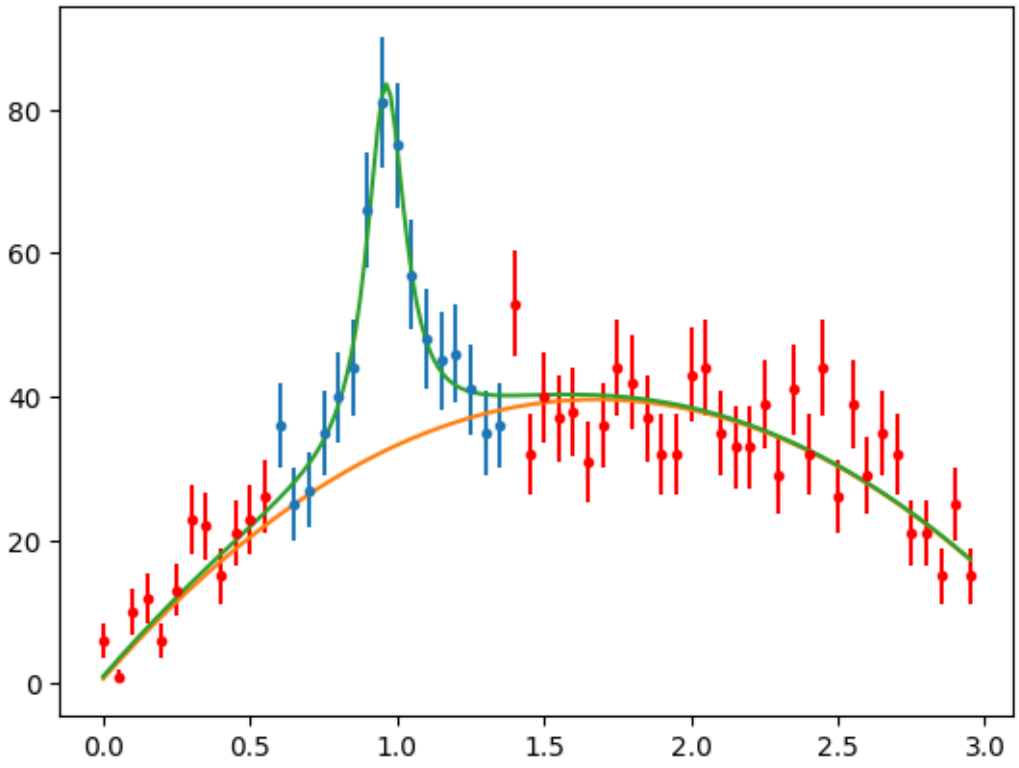
Using ROOT or pure Python, implement your solution following the steps below:

[26]:
```
# For the ROOT approach: Store the values in a TGraphErrors

# Part 1: Define the fit function with 6 parameters and fit signal and back␣
 ↪simultaneously.
def backgound(E,a,b,c):
    return a*E**2+b*E+c

def Signal(E,Anorm,mu,Gamma):
    return Anorm/np.pi*Gamma/2/((E-mu)**2+Gamma**2/4)

fullFunc = lambda E,a,b,c,Anorm,mu,Gamma:␣
 ↪backgound(E,a,b,c)+Signal(E,Anorm,mu,Gamma)

res = scipy.optimize.curve_fit(fullFunc, data_x, data_y, p0=(-10,40,0.1,10,1,0.
 ↪1), sigma=sigma_y)


plt.errorbar(data_x, data_y, sigma_y, label="data", fmt=".")
```

```
xlin = np.linspace(np.min(data_x), np.max(data_x), 200)
plt.plot(xlin, fullFunc(xlin, *res[0]))
plt.show()
display(res[0])

# Part 2: Split the data in signal and background region. First fit the␣
 ↪background distribution in the background region.
#         Afterwards, subtract the background expectation from data in the␣
 ↪signal region and fit the signal function.

backslice = np.logical_or(np.arange(0,data_x.shape[0]) < 12, np.arange(0,data_x.
 ↪shape[0]) > 27)
plt.errorbar(data_x[np.logical_not(backslice)], data_y[np.
 ↪logical_not(backslice)], sigma_y[np.logical_not(backslice)], label="data",␣
 ↪fmt=".")
plt.errorbar(data_x[backslice], data_y[backslice], sigma_y[backslice],␣
 ↪label="data", fmt=".r")

backres = scipy.optimize.curve_fit(backgound, data_x[backslice],␣
 ↪data_y[backslice], p0=(-10,40,0.1), sigma=sigma_y[backslice])
foreres = scipy.optimize.curve_fit(Signal, data_x[np.logical_not(backslice)],␣
 ↪data_y[np.logical_not(backslice)]-backgound(data_x[np.
 ↪logical_not(backslice)], *backres[0]), p0=(10,1,0.1), sigma=sigma_y[np.
 ↪logical_not(backslice)])

xlin = np.linspace(np.min(data_x)-overhang, np.max(data_x)+overhang, 200)
plt.plot(xlin, backgound(xlin, *backres[0]))
plt.plot(xlin, backgound(xlin, *backres[0]) + Signal(xlin, *foreres[0]))
plt.show()
display(backres[0], foreres[0])
```




```
array([-13.32141374,  45.17729489,   0.27304432,  13.80744981,
         0.96228106,   0.17230977])
```




```
array([-13.81242332,  46.36874167,   0.71021315])
```

```
array([12.71704027,  0.9623414 ,  0.15864221])
```

1.2 Exercise 5.2: Minimization via Simulated Annealing (obligatory)

Data analysis often requires to find the optimal solution, e.g., the minimum of a function in a multi-dimensional space.

As an example we use the following two-dimensional function which has several local minima, but just one global minimum:

$$f(x, y) = (x^2 + y - a)^2 + (x + y^2 - b)^2 + c \cdot (x + y)^2$$

with arbitrary parameters a , b and c.

For $ c= 0$ this function would be the Rosenbrock function which is often used to validate minimization algorithms.

For this exercise sheet, we make the arbitrary choice $a = 11$, $b = 7$, $c = 0.1$. Thus, $f(x, y)$ has four local minima of different depth.

In this exercise you will write your own minimization algorithm following the **Simulated Annealing** strategy and test with the above defined function. Use the code fragment given in the jupyter notebook as help.

1. Play with the parameters: initial and final temperature, cooling speed, and step size. Choose a starting point close to the global minimum, and check if the algorithm converges into the minimum.

2. Choose a starting point close to a local minimum which is not the global minimum, e.g., $(x, y) = (3, -2)$. Find a set of parameters for the algorithm such that it converges to the global minimum, but still keeping the number of iterations as low as possible, and motivate your choice.

   For tuning the parameters, you have two possibilities: Either you perform a scan over a meaningful range for each parameter, or you study how the algorithm reacts on changing certain parameters and then try to tune them by hand. In any case, first think what is the role of each parameter in the algorithm. E.g., both, the difference between initial and final temperature, and the cooling speed directly affect the number of iterations, but the temperature scale in addition affects the probability for jumps.

3. Repeat the analysis for different random seeds. If the minimum found depends on the random seed, re-tune the parameters of the algorithm until the result is independent of the random seed.

Python Approach:

[8]:
```
import matplotlib.animation
from IPython.display import Image
```

[9]:
```
# modified rosenbrock function: f(x,y) = (x^2+y-a)^2 + (x+y^2-b)^2 + c*(x+y)^2
def modified_rosenbrock_function(x, par):
    xx = x[0]
    yy = x[1]
    a = par[0]
    b = par[1]
    c = par[2]

    return (xx**2+yy-a)**2 + (xx+yy**2-b)**2 + c*(xx+yy)**2
```

[10]:
```
def plotFunction(function, listOfPoints):
    """Draw or return plot objects of scanned values of x and y and the surface␣
 ↪of the function.

    Helper function for visualization of the minimization procedure.
    """
    return None
```

[11]:
```
# Code fragment for exercise 6.2 of the Computerpraktikum Datenanlyse 2014
# Authors: Ralf Ulrich, Frank Schroeder (Karlsruhe Institute of Technology)
# Modified: 2020-05-26 Maximilian Burkart (Karlsruhe Institute of Technology)
# This code fragment probably is not the best and fastest implementation
# for "simulated annealing", but it is a simple implementation which does its␣
 ↪job.
```

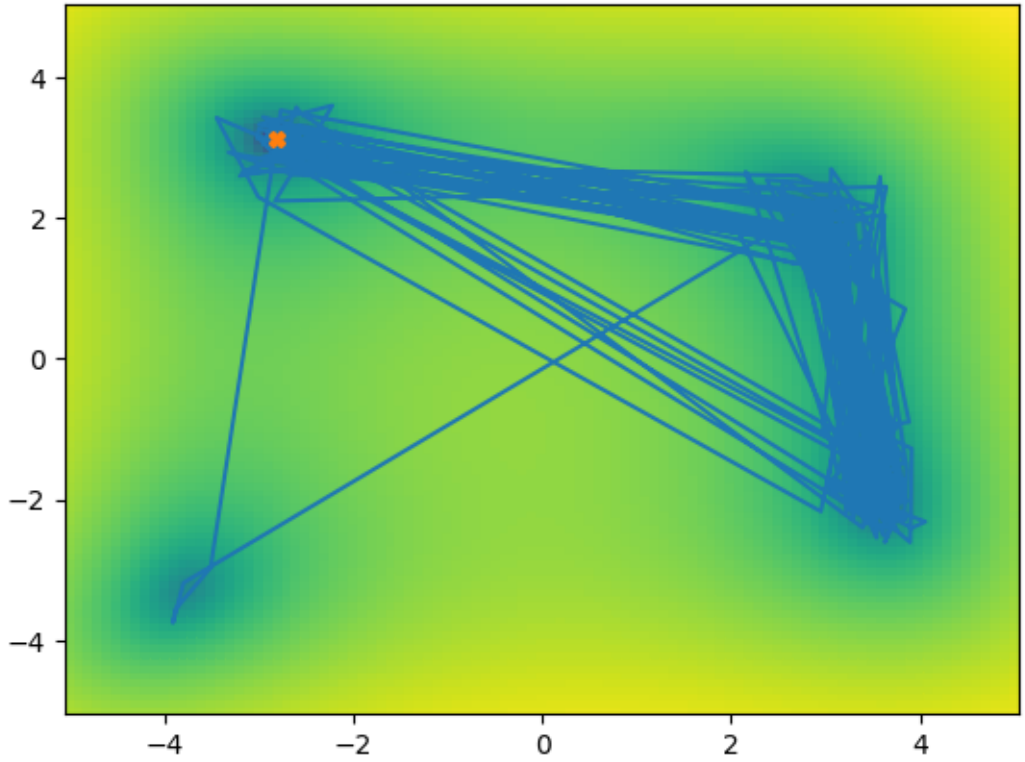
```
def simulated_annealing(init_vals=[0,0], rosenbrock_pars=[0, 0, 0],
                        init_temp=100, final_temp=1, cool_speed=1, step_size=1,
                        seed=None):
    """Minimize the modified Rosenbrock function using simulated annealing.

    params:
        init_vals:       Initial x and y values.
        rosenbrock_pars: Parameters of the modified Rosenbrock function.
        init_temp:       Initial temperature the cooling starts from.
        final_temp:      Final temperature of the cooling.
        cool_speed:      Cooling speed in percent of the current temperature.
        step_size:       Step size used in the cooling procedure.

    returns:
        min_pars: List of floats.
                    List of the x and y values at the found minimum.
        listOfPoints: List of floats.
                    List of the visited points during the minimization process.
    """
    nParameter = 2 # 2 parameters: x and y
    if len(init_vals) != nParameter:
        raise Exception("Number of function parameters does not correspond to␣
 ↪given number of initial values."
                        "Aborting...")

    # TODO: Implement setting of seed if a seed is given.
    if seed is not None:
        np.random.seed(seed)

    # Starting point: test the dependence of the algorithm on the initial values
    initialXvalue, initialYvalue = init_vals

    # Parameters of the  algorithm:
    # Find a useful set of parameters which allows to determine the global
    # minimum of the given function:
    # The temperature scale must be in adequate relation to the scale of the␣
 ↪function values,
    # the step size must be in adequate relation to the scale of the distance␣
 ↪between the
    # different local minima
    initialTemperature = init_temp
    finalTemperature = final_temp
    coolingSpeed = cool_speed # in percent of current temperature --> defines␣
 ↪number of iterations
    stepSize = step_size
```

```
    # Current parameters and cost
    currentParameters  = [initialXvalue, initialYvalue] # x and y in our case
    currentFunctionValue =  modified_rosenbrock_function(currentParameters,␣
↪[a,b,c]) # you have to implement the function first!

    # keep reference of best parameters
    bestParameters = currentParameters
    bestFunctionValue = currentFunctionValue

    listOfPoints= []
    # Heat the system
    temperature = initialTemperature

    iteration = 0

    # Start to slowly cool the system
    while (temperature > finalTemperature):

        # Change parameters
        newParameters = [0]*nParameter

        #for ipar in range(nParameter):
        #    newParameters[ipar] = gRandom.Gaus(currentParameters[ipar], stepSize)
        newParameters = np.array(
                [np.random.normal(loc=currentParameters[0], scale=step_size),
                 np.random.normal(loc=currentParameters[1], scale=step_size)])

        # Get the new value of the function
        newFunctionValue = modified_rosenbrock_function(newParameters, [a,b,c])

        # Compute Boltzman probability
        deltaFunctionValue = newFunctionValue - currentFunctionValue
        saProbability = np.exp(-deltaFunctionValue / temperature)

        # Acceptance rules :
        # if newFunctionValue < currentFunctionValue then saProbability > 1
        # else accept the new state with a probability = saProbability
        if ( saProbability > np.random.random() ):
            currentParameters = newParameters
            currentFunctionValue = newFunctionValue
            listOfPoints.append(currentParameters)  # log keeping: keep track of␣
↪path

        if (currentFunctionValue < bestFunctionValue):
            bestFunctionValue = currentFunctionValue
            bestParameters = currentParameters
```

```
        #print "T = ", temperature, "(x,y) = ",currentParameters, " Current value: ", currentFunctionValue, " delta = ", deltaFunctionValue # debug output

        # Cool the system
        temperature *= 1 - coolingSpeed / 100.

        # Count iterations
        iteration += 1

    # end of cooling loop

    return bestParameters, listOfPoints
```

[12]:
```
a = 11
b = 7
c = 0.1

initial_values = [3, -2]

initial_temp, final_temp, cool_speed, step_size = 5, 0.001, 0.01, 3.15

bestParameters, listOfPoints = simulated_annealing(init_vals=initial_values,
                                                   rosenbrock_pars=[a, b, c],
                                                   init_temp=initial_temp,
                                                   final_temp=final_temp,
                                                   cool_speed=cool_speed,
                                                   step_size=step_size
                                                   )

minValue = modified_rosenbrock_function(bestParameters, [a, b, c])

print(f"Resultat für das Minimum: f(x,y)={minValue} bei x={bestParameters[0]}, y={bestParameters[1]}")

# check if minimum is the global one (then the mimimum value should be around 0.01):
if minValue < 0.1:
    print("Ja, das ist das globale Minimum!")
else:
    print("Oh nein, das ist nicht das globale Minimum, sondern nur ein lokales!")

grid = np.mgrid[-5:5.1:0.1,-5:5.1:0.1]
res = modified_rosenbrock_function(grid, [a, b, c])
plt.pcolor(*grid, np.log(res))
plt.plot(*zip(*listOfPoints))
plt.plot(*bestParameters, "X")
plt.show()
```

```
Resultat für das Minimum: f(x,y)=0.014867851399921354 bei x=-2.8024689917217933,
y=3.140221483133727
Ja, das ist das globale Minimum!
```

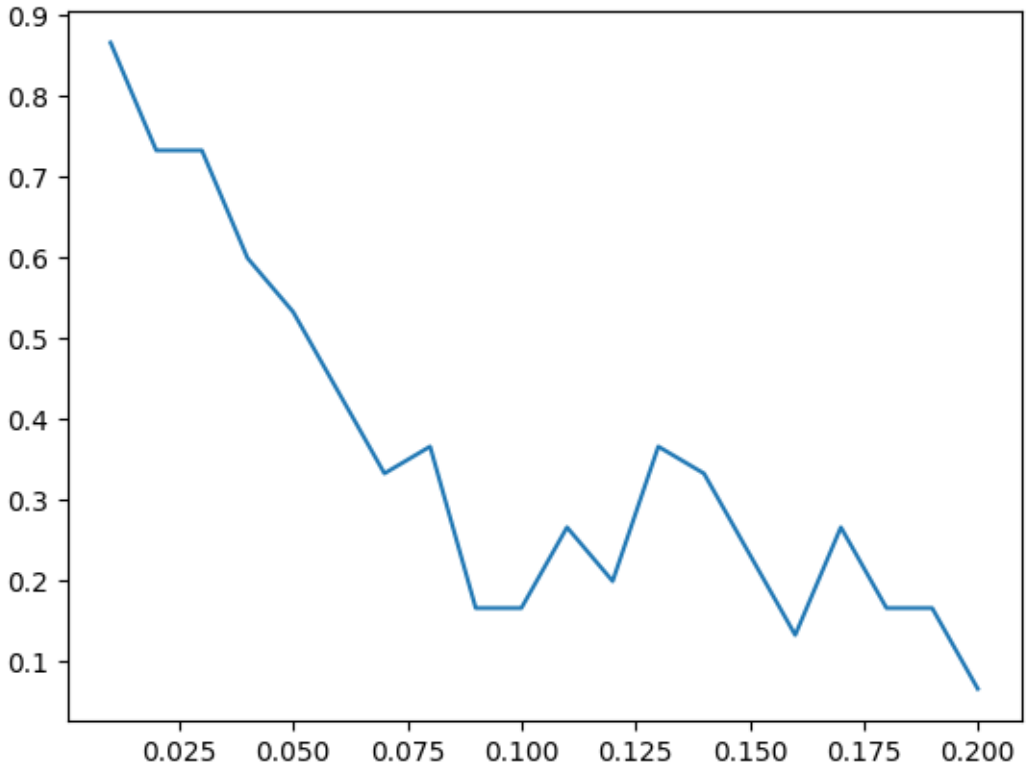


[13]:
```
from concurrent.futures import ProcessPoolExecutor

def score(params):
    initial_temp, final_temp, cool_speed, step_size = params
    s = 0
    N = 30
    for i in range(N):
        bestParameters, listOfPoints =␣
 ↪simulated_annealing(init_vals=initial_values,
                                                    rosenbrock_pars=[a, b, c],
                                                       init_temp=initial_temp,
                                                        final_temp=final_temp,
                                                        cool_speed=cool_speed,
                                                         step_size=step_size
                                                         )
        s += (modified_rosenbrock_function(bestParameters, [a, b, c]) < 0.1)
    print(-s/N,":",params)
    return s/N
```

```python
def parallel_score(params_list):
    with ProcessPoolExecutor() as executor:
        scores = list(executor.map(score, params_list))
    return scores

def generate_params(params, i, values):
    params_list = []
    for v in values:
        new_params = list(params)
        new_params[i] = v
        params_list.append(tuple(new_params))
    return params_list

initial_temp, final_temp, cool_speed, step_size = 5, 0.001, 0.1, 3.15
params = [initial_temp, final_temp, cool_speed, step_size]

i = 2
values = np.linspace(0.01, 0.2, 20)
params_list = generate_params(params, i, values)

results = parallel_score(params_list)
plt.plot(values, results)
#scipy.optimize.minimize(score, x0=[10, 0.001, 0.1, 3], method='Nelder-Mead',␣
 ↪tol=1e-6)
```

```
-0.36666666666666664 : (5, 0.001, 0.08, 3.15)
-0.3333333333333333 : (5, 0.001, 0.06999999999999999, 3.15)
-0.43333333333333335 : (5, 0.001, 0.060000000000000005, 3.15)
-0.5333333333333333 : (5, 0.001, 0.05, 3.15)
-0.16666666666666666 -0.16666666666666666 :: (5, 0.001, 0.09999999999999999,
3.15)
 (5, 0.001, 0.09, 3.15)
-0.6 : (5, 0.001, 0.04, 3.15)
-0.26666666666666666 : (5, 0.001, 0.11, 3.15)
-0.2 : (5, 0.001, 0.12, 3.15)
-0.3333333333333333 : (5, 0.001, 0.14, 3.15)
-0.36666666666666664 : (5, 0.001, 0.13, 3.15)
-0.13333333333333333 : (5, 0.001, 0.16, 3.15)
-0.23333333333333334 : (5, 0.001, 0.15000000000000002, 3.15)
-0.7333333333333333 : (5, 0.001, 0.03, 3.15)
-0.26666666666666666 : (5, 0.001, 0.17, 3.15)
-0.16666666666666666 : (5, 0.001, 0.19, 3.15)
-0.16666666666666666 : (5, 0.001, 0.18000000000000002, 3.15)
-0.06666666666666667 : (5, 0.001, 0.2, 3.15)
-0.7333333333333333 : (5, 0.001, 0.02, 3.15)
-0.8666666666666667 : (5, 0.001, 0.01, 3.15)
```

[13]: [<matplotlib.lines.Line2D at 0x7fbac1c64b50>]



[ ]: