Karlsruher Institut für Technologie

Vorlesung 05b: Rechnernutzung in der Physik

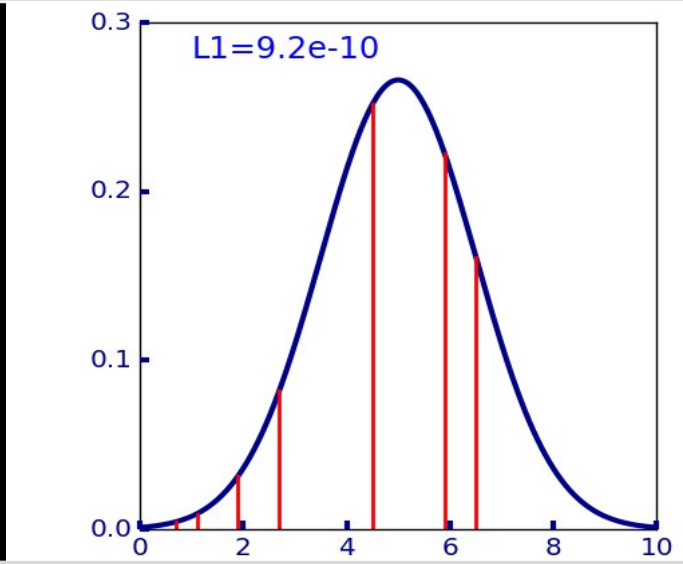
Modellanpassung

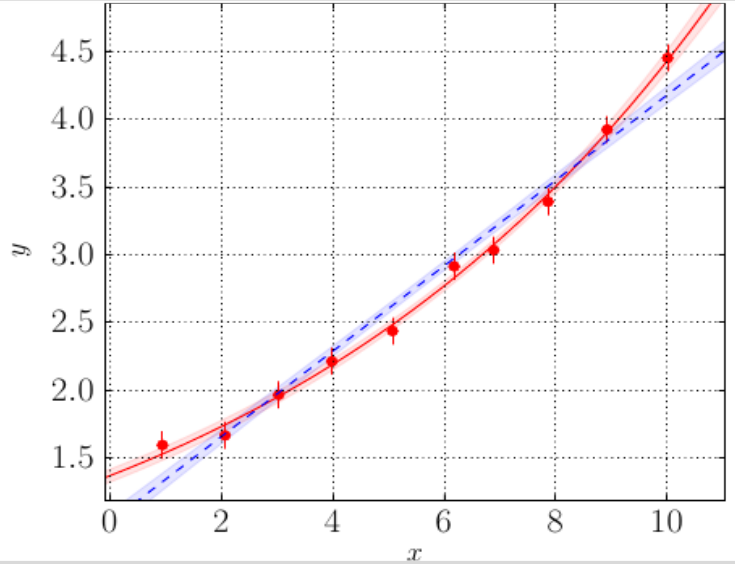
Fragen, Antworten und Beispiele

Günter Quast

Fakultät für Physik
Institut für Experimentelle Kernphysik

WS 2023/24





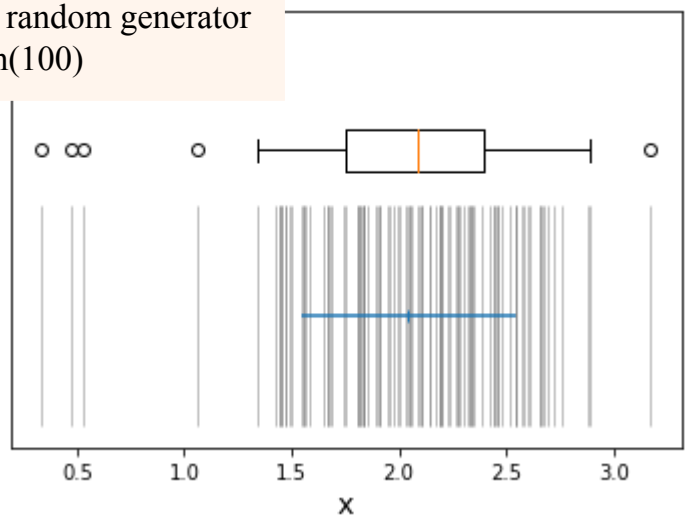
Jupyter Tutorial

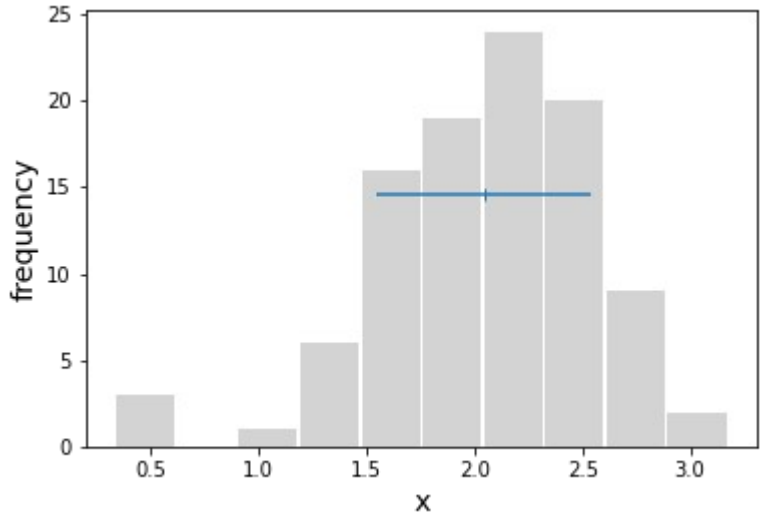
negLogLFits

Jupyter-Tutorial negLogLFits

```
# generate Gaussian-distributed data
mu0=2.
sig0=0.5
np.random.seed(314159)  # initialize random generator
data = mu0 + sig0 * np.random.randn(100)
```

Fit an gaußverteilte Daten





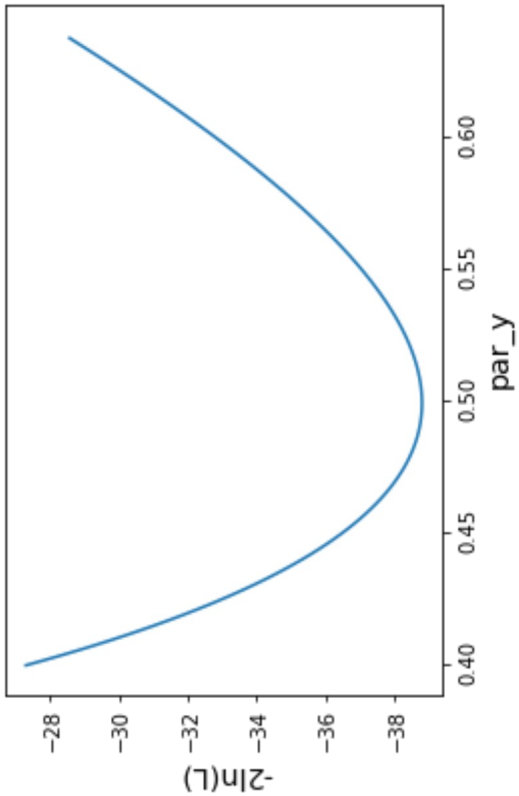

```
# define cost function: 2 * negative log likelihood of Gauß distribution
def myCost(mu=1., sigma=1.):
  # simple -2*log-likelihood of a 1-d Gauss distribution
  r = (data-mu)/sigma
  return np.sum( r*r) + 2.*len(data)*np.log(sigma)
```

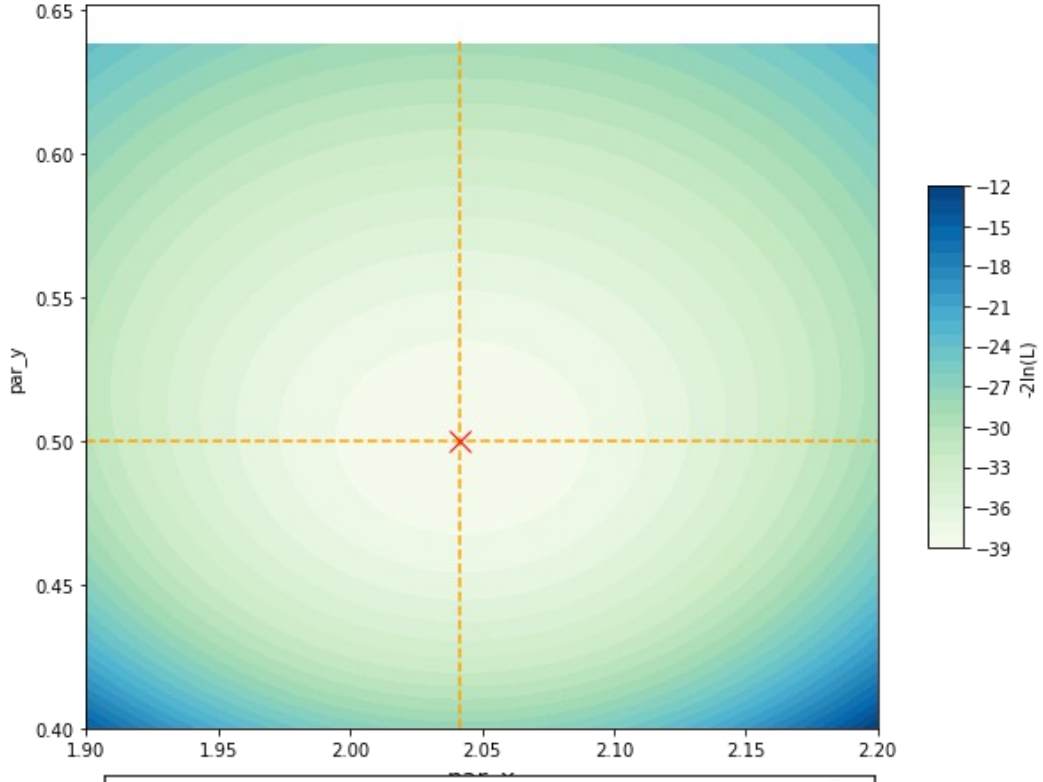
mit iminuit

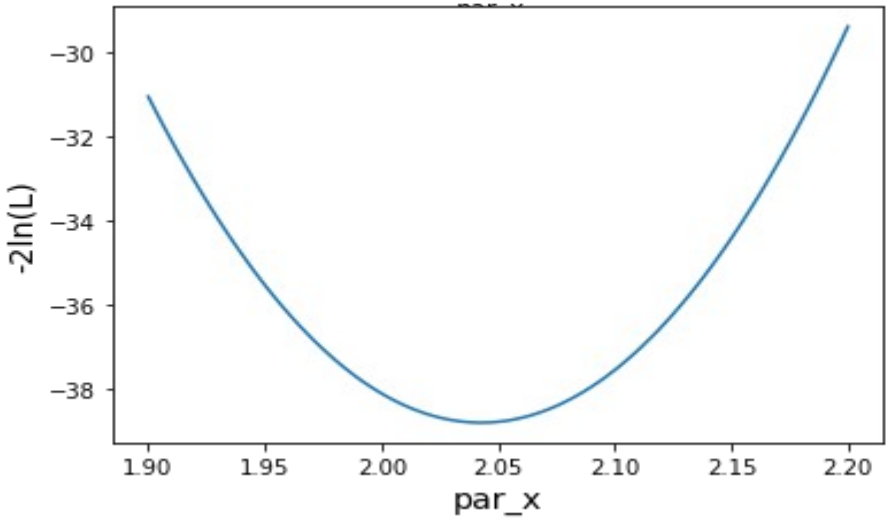
```
parameter names:  ('mu', 'sigma')
best-fit values:  (2.042703589998085, 0.49920606604299755)
```

```
# code for iminuit vers. >2.0
from iminuit import Minuit

# initialize Minuit object
m = Minuit(myCost, mu=1., sigma=1.)
m.errordef = 1.        # internal parameter,
      #  needed to control uncertainty analysis

# perform optimization
m.migrad()

# print results
print("parameter names: ", m.parameters)
print("best-fit values: ", tuple(m.values))
```

Grafische Darstellunng von -nlL





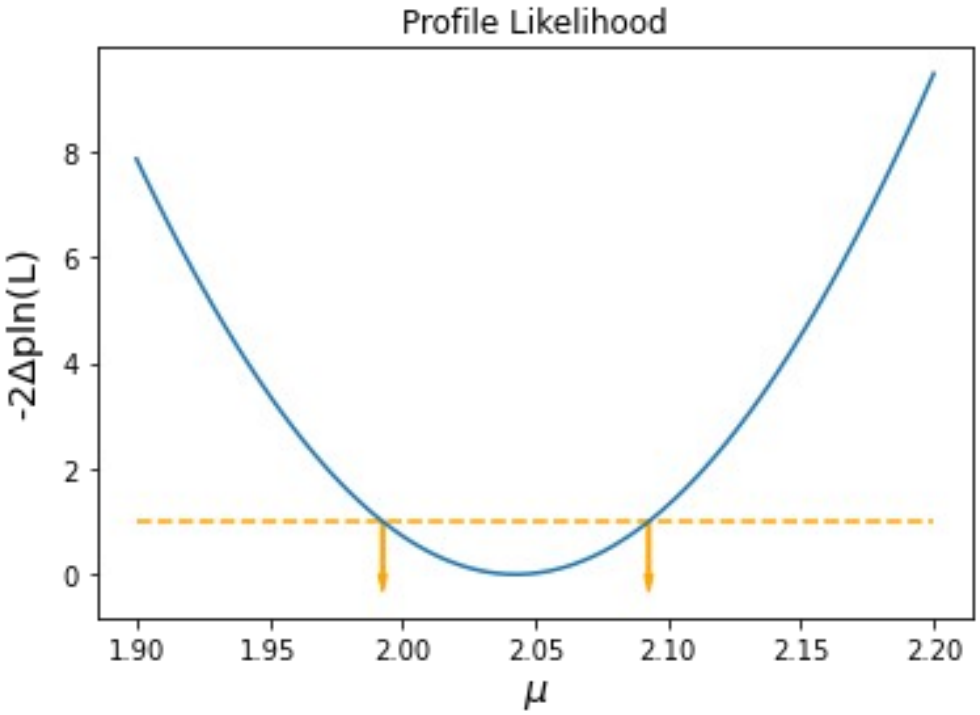


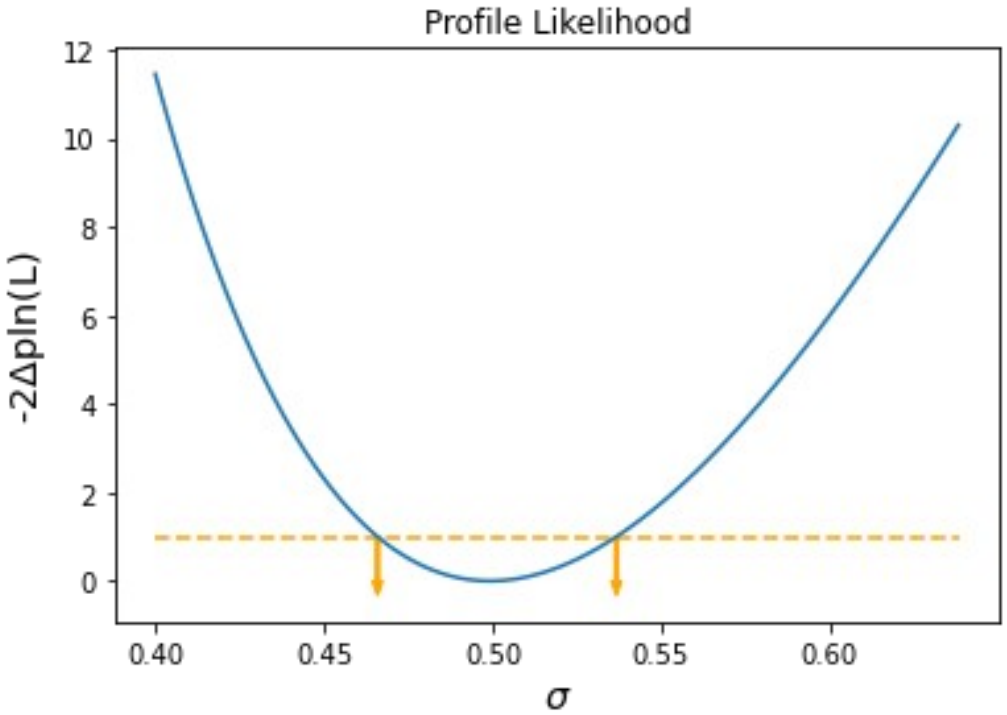
numerisch sehr aufwändig,
aber für viele Problemstellungen
schon „die Lösung“.

Jupyter-Tutorial negLogLFits

Unsicherheiten aus Scan der Profile Likelihood

„Intervallschätzung“



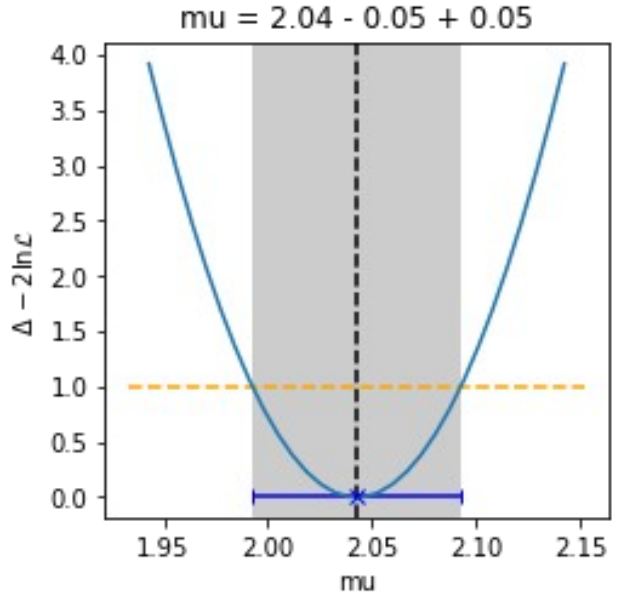


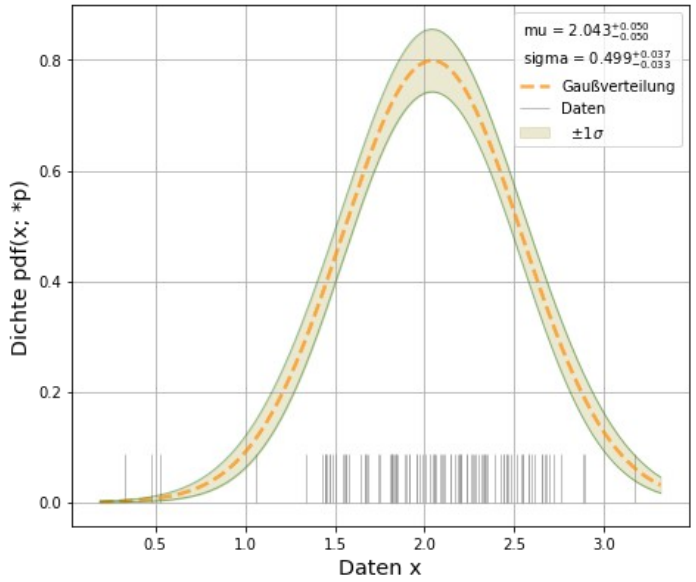
Kurven sehen in diesem Fall genau so aus wie die einfache Projektion. Das liegt daran, dass die Korrelation zwischen den Parametern klein ist und deshalb eine Änderung des x-Parameters keinen Einfluss auf den y-Parameter hat.

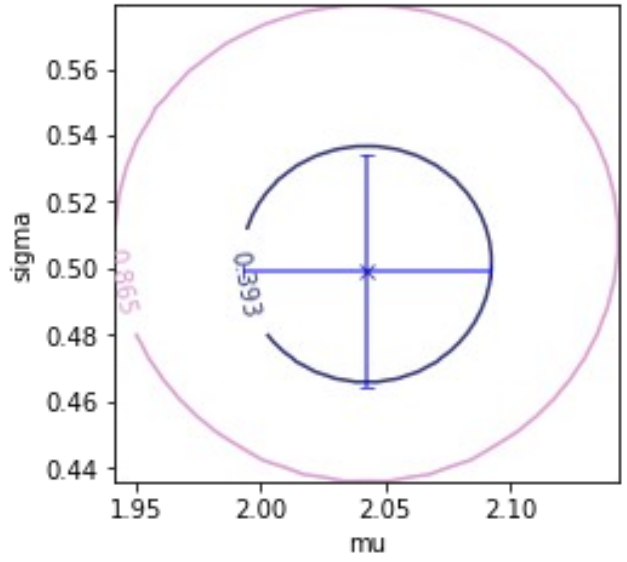
Jupyter-Tutorial negLogLFits

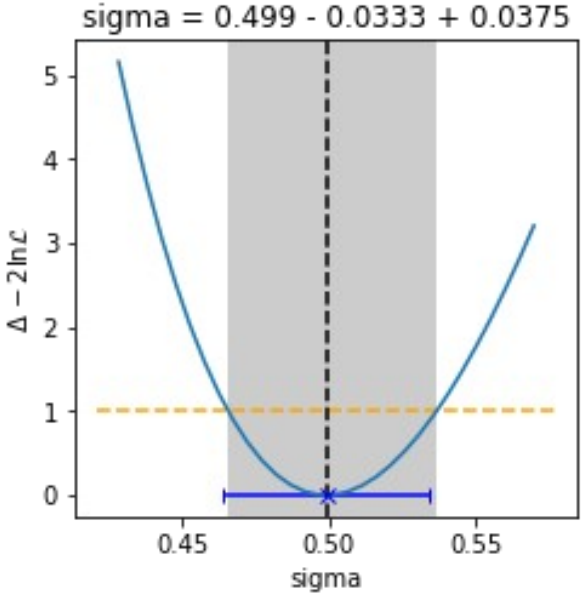
Fit an gaußverteilte Daten mit **PhyPraKit.mFit**

```
fit_results = mFit(myCost, plot_cor=True)
```



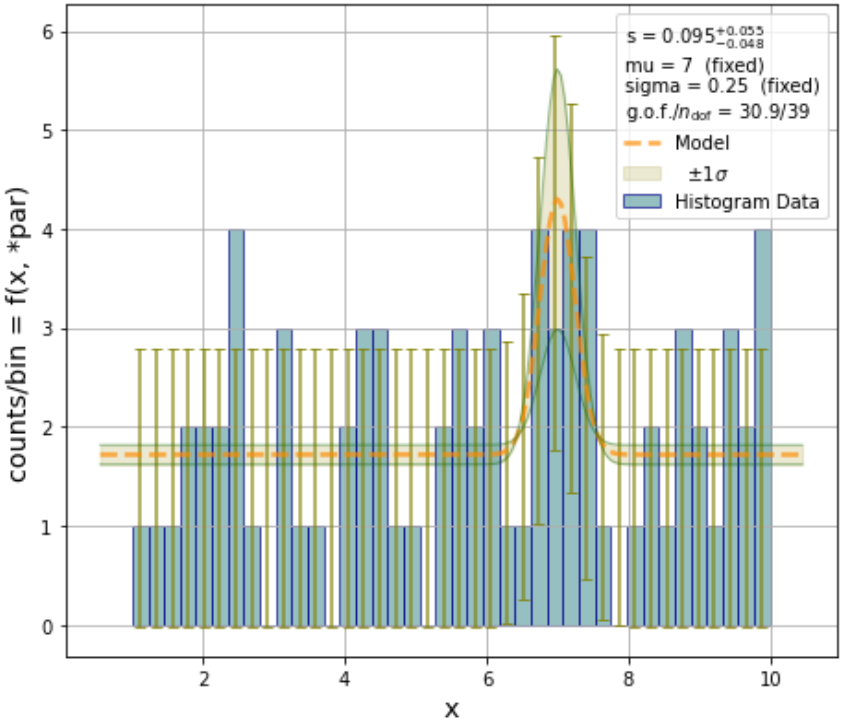






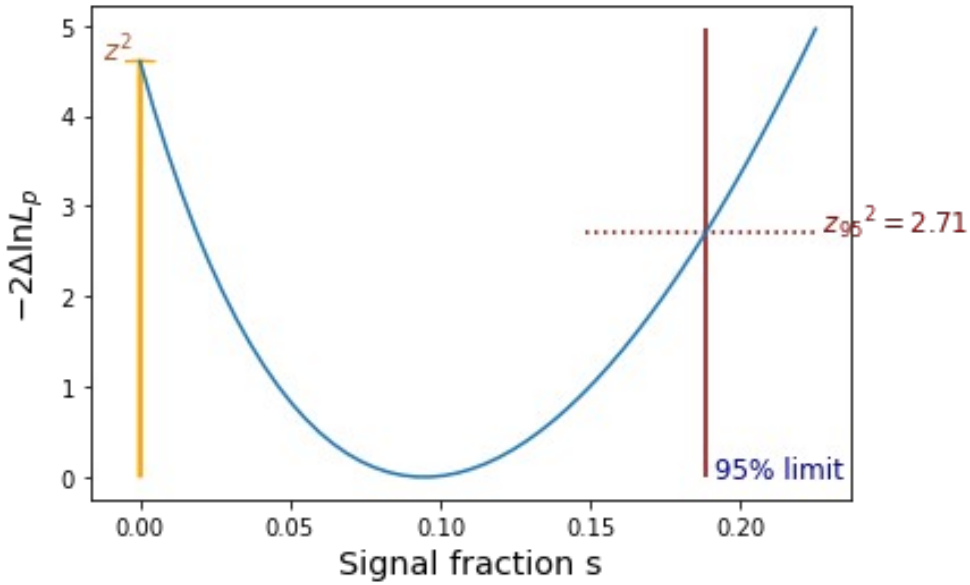
Jupyter-Tutorial negLogLFits

Histogramm-Fit mit **PhyPraKit.hFit** und **Analyse der Profile-Likelihood**



Gaußförmiges Signal auf flachem Untergrund
mit Signalanteil s als Parameter

Ist das ein signifikantes Signal ?



== Analysis of profile likelihood
Significance, z-value: 2.14
p-value of Null-hypothesis: 0.016
95% limit: s < 0.188

Werkzeuge zur numerischen Anpassung von Modellen

Software-Werkzeuge zur Parameteranpassung

In der alltäglichen Praxis verwendet man heute Programme,
(bzw. besser **Softwarepakete**) zur Anpassung von Modellen an Parameter:

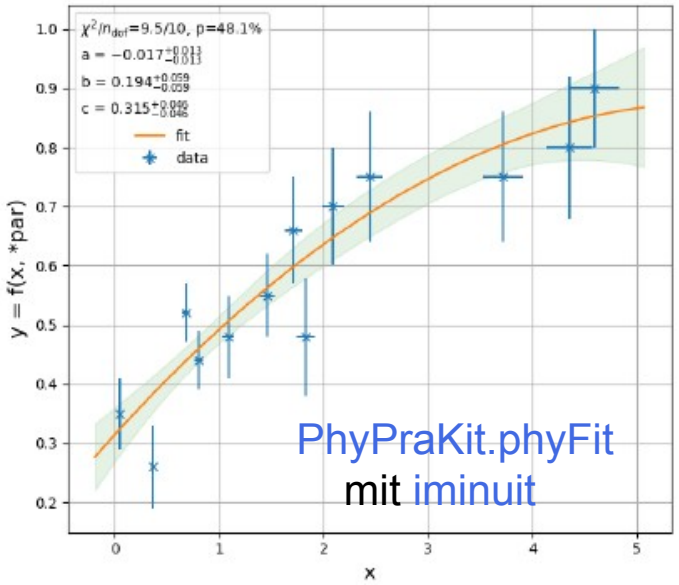
- Einlesen der Eingabedaten, evtl. Unterstützung verschiedener Formate
- Verwaltung der Unsicherheiten, evtl. für Abszisse und Ordinate,
  evtl. Kovarianzmatrizen bzw. Korrelationen
- Berücksichtigung von Einschränkungen an Modellparameter
- Bereitstellung verschiedener Fit-Funktionen, evtl. frei programmierbar
- Durchführung der Anpassung mittels numerischer Optimierung, evtl. anpassbar
- Ausgabe des Ergebnisses, als Textdatei und Grafik
- Bewertung der Qualität der Anpassung (außer χ^2 auch andere gebräuchlich)
- Ausgabe der Korrelationen bzw. Kovarianz-Matrix der Parameter
- graphische Darstellung der Modellunsicherheit
- asymmetrische Fehler mittels Scan der (Profile-) Likelihood - Kurve am Minimum
- Ausgabe von Kovarianz-Ellipsen (d. h. Wahrscheinlichkeitskonturen)

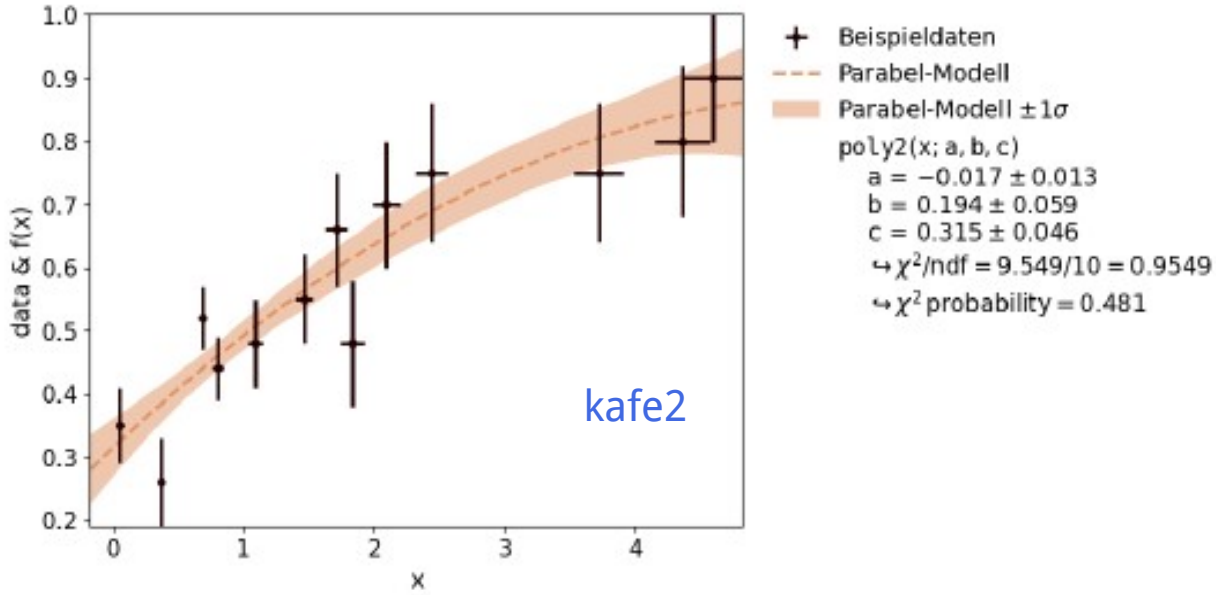
kafe2 und PhyPraKit.phyFit unterstützen alle der genannten Punkte;
fast alle großen Projekt in der Experimentalphysik haben eigene, spezialisierte Pakete.

Beispiele

Beispiele für Werkzeuge zur Anpassung

```
import numpy as np
from scipy.optimize import curve_fit
# fit function definition
def poly2(x, a=1.0, b=0.0, c=0.0):
return a * x**2 + b * x + c
#1. load data
x, y, sy = np.loadtxt("fitexample.dat", unpack=True)
# linear least squares with scipy.optimize.curve_fit
par, cov = curve_fit( poly2, x, y, sigma=sy,
                      absolute_sigma=True )
print("Fit parameters:\n", par)
print("Covariance matrix:\n", cov)
```





Werkzeuge: scipy.optimize

```
#! /usr/bin/env python
'''test_odFit
   test fitting an arbitrary fucntion with scipy odr,
   with uncertainties in x and y
'''
from PhyPraKit import generateXYdata, odFit
import numpy as np, matplotlib.pyplot as plt

# -- the model function
def model(x, a=0.3, b=1., c=1.):
   return a*x**2 + b*x + c
```

Anpassung mit scipy.optimize.curve_fit und scipy.odr

Skript PhyPraKit/examples/test_odFit.py

```
if __name__ == "__main__": # -------------------------------------

# parameters for the generation of test data
  sigx_abs = 0.2 # absolute error on x
  sigy_abs = 0.1 # relative error on y
  xmin =  1.
  xmax =  10.
  xdata=np.arange(xmin, xmax+1. ,1.)
  nd=len(xdata)
  mpars=[0.3, -1.5, 0.5]

# generate the data
  xt, yt, ydata = generateXYdata(xdata, model, sigx_abs, sigy_abs, mpar=mpars)

# fit with odFit (uses scipy.curve_fit and scipy.odr)
  par, pare, cor, chi2 = odFit(model,
    xdata, ydata, sigx_abs, sigy_abs, # data and uncertaintites
    p0=None)
```

Ergebnis:

```
*==* fit result:
-> chi2:         2.45
 -> parameters:    [ 0.309 -1.57   0.568]
 -> uncertainties: [0.02 0.15 0.28]
 -> correlation matrix:
[[ 1.   -0.95  0.84]
[-0.95  1.   -0.95]
[ 0.84 -0.95  1.  ]]
```

Werkzeuge: scipy.optimize (2)

```
def odFit(fitf, x, y, sx=None, sy=None, p0=None):
  """fit an arbitrary function with errors on x and y
  uses numerical "orthogonal distance regression" from package scipy.odr

  Args:
  * fitf: function to fit, arguments (array:P, float:x)
  * x:  np-array, independent data
  * y:  np-array, dependent data
  * sx: scalar or np-array, uncertainty(ies) on x
  * sy: scalar or np-array, uncertainty(ies) on y
  * p0: array-like, initial guess of parameters

  Returns:
  * np-array of float: parameter values
  * np-array of float: parameter errors
  * np-array: cor   correlation matrix
  * float: chi2  \chi-square
  """
  from scipy.optimize import curve_fit
  from scipy import odr

     ...

  par0, cov0 = curve_fit( fitf, x, y, sigma=sy, absolute_sigma=True, p0=p0 )

     ...

  mod = odr.Model(fitf_ODR)
  dat = odr.RealData(x, y, sx, sy)
  odrfit = odr.ODR(dat, mod, beta0 = par0)
```

Achtung:
curve_fit benötigt
speziellen Parameter

curve_fit default für Parametrisierung, nicht für Modellanpassung, d.h. $\chi^2/ndf = 1$ wird erzwungen und keine Überprüfung der Modell-hypothese möglich !

Parameterfehler ohne Datenfehler ?

Manche Programme zur Anpassung (QTIplot, Origin, curve_fit, ...) geben Parameterunsicherheiten aus, ohne dass Unsicherheiten auf die Datenpunkte angegeben wurden. **Wie geht das ?!**

Annahmen:

Modell beschreibt die Daten perfekt: $\chi^2 := n_f$ statt $< \chi^2 >= n_f$

Alle Datenpunkte haben den gleichen Fehler: $\sigma_i := \sigma$ (nur selten so !)

$$S'(\hat{p}) = \sum_{i=1}^{N} \left(y_i - f(x_i;\hat{p})\right)^2 =: S'_{\min} \stackrel{!}{=} n_f \cdot \sigma^2 \quad \text{d.h. alle} \quad \sigma_i = \sqrt{\frac{S'_{\min}}{n_f}}$$

dann $S(p) =: \sum_{i=1}^{N} \frac{\left(y_i - f(x_i;p)\right)^2}{S'_{\min}/n_f}$ setzen und Parameterfehler bestimmen

$$\Rightarrow \sigma_p{}^2 = 2\left(\frac{\partial^2 S(\hat{p})}{\partial p^2}\right)^{-1} = 2\left(\frac{\partial^2 S'(\hat{p})}{\partial p^2}\right)^{-1} \cdot S'_{\min}/n_f$$

- **Verlust der χ^2-Wahrscheinlichkeit**
- **angenommenes Fehlermodell oft falsch !**

Anpassungen mit PhyPraKit.phyFit

Code, Dokumentation und Beispiele: https://github.com/GuenterQuast/PhyPraKit/

Schlanke Wrapper-Funktionen um CERN iminuit

Interfaces:

`PhyPraKit.phyFit.mFit` (*ufcn, data=None, p0=None, constraints=None, limits=None, fixPars=None, neg2logL=True, plot=False, plot_band=True, plot_cor=False, showplots=True, quiet=True, axis_labels=['x', 'Density = f(x, *par)'], data_legend='data', model_legend='model', return_fitObject=False*)

Wrapper function to directly fit a user-defined cost funtion

`PhyPraKit.hFit` (*fitf, bin_contents, bin_edges, DeltaMu=None, p0=None, constraints=None, fixPars=None, limits=None, use_GaussApprox=False, fit_density=True, plot=True, plot_cor=True, showplots=True, plot_band=True, quiet=False, axis_labels=['x', 'counts/bin = f(x, *par)'], data_legend='Histogram Data', model_legend='Model'*)

Wrapper function to fit a density distribution f(x, *par) to binned data (histogram) with class mnFit

`PhyPraKit.xyFit` (*fitf, x, y, sx=None, sy=None, srelx=None, srely=None, xabscor=None, xrelcor=None, yabscor=None, yrelcor=None, ref_to_model=True, p0=None, constraints=None, fixPars=None, limits=None, use_negLogL=True, plot=True, plot_cor=False, plot_band=True, showplots=True, quiet=False, axis_labels=['x', 'y = f(x, *par)'], data_legend='data', model_legend='model'*)

Fit an arbitrary function fitf(x, *par) to data points (x, y) with independent and correlated absolute and/or relative errors on x- and y- values with class mnFit from package phyFit (uses iminuit).

Beispiel: Anpassung mit PhyPraKit.phyFit

```
...

# Komponenten der Messunsicherheit
#  - Genauigkeit U-Messung: 4000 Counts, +/-(0.5% + 3 digits)
#     - Messbereich 2V
crel_U = 0.005
Udigits = 3
Urange = 2
Ucounts = 4000
#  - Genauigkeit I-Messung: 2000 Counts, +/-(1.0% + 3 digits)
#  - Messbereiche 200µA, 20mA und 200mA
crel_I = 0.010
Idigits = 3
Icounts = 2000
Irange1 = 0.2
Irange2 = 20
Irange3 = 200
# - Rauschanteil (aus Fluktuationen der letzten Stelle)
#   - delta U = 0.005 V
deltaU = 0.005
#   - delta I = 0.025 mA
deltaI = 0.025
# - Anzeigegenauigkeit der Spannung (V)
sx = Udigits * Urange / Ucounts
sabsx = np.sqrt(deltaU**2 + sx**2) # Rauschanteil addieren
# - korrelierte Kalibrationsunsicherheit
crelx = crel_U
  # - Anzeigegenauigkeit des Stroms (mA), 3 Messbereiche
sy = np.asarray(   2 * [Idigits * Irange1 / Icounts] + \
                  12 * [Idigits * Irange2 / Icounts] + \
                   6 * [Idigits * Irange3 / Icounts])
sabsy = np.sqrt(deltaI**2 + sy**2) # Rauschanteil addieren
# - korrelierte Kalibrationsunsicherheit
crely = crel_I
```

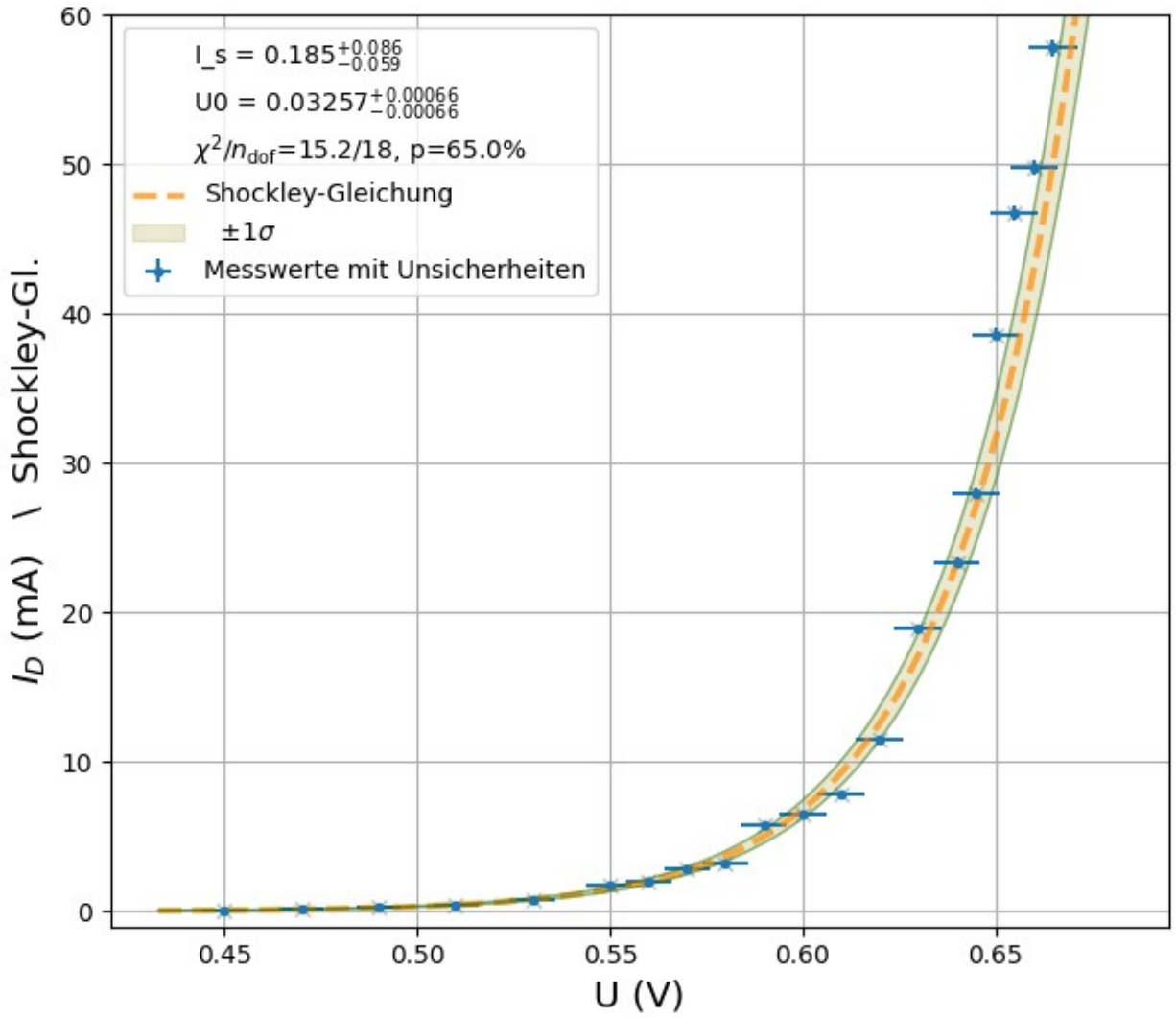
Skript

PhyPraKit/examples/Beispiel_Diodenkennlinie.py

```
 fitResult = xyFit( model,
# - data and uncertainties
    data_x, data_y,      # data x and y coordinates
    sx=sabsx,           # indep x
    sy=sabsy,           # indel y
    xrelcor=crelx,       # correlated rel. x
    yrelcor=crely,       # correlated rel. y
    ref_to_model=True,   # reference of rel. uncert. to model
# - fit control
    p0=(0.2, 0.05),   # initial guess for parameter values
    limits=('U0', 0.005, None), # parameter limits
# - output options
    plot=True,          # plot data and model
    plot_cor=False,      # plot profiles likelihood and contours
    showplots = False,   # plt.show() in user code
    quiet=False,         # suppress informative printout
    axis_labels=['U (V)', '$I_D$ (mA)   \  Shockley-Gl.'],
    data_legend = 'Messwerte mit Unsicherheiten',
    model_legend = 'Shockley-Gleichung' )
```

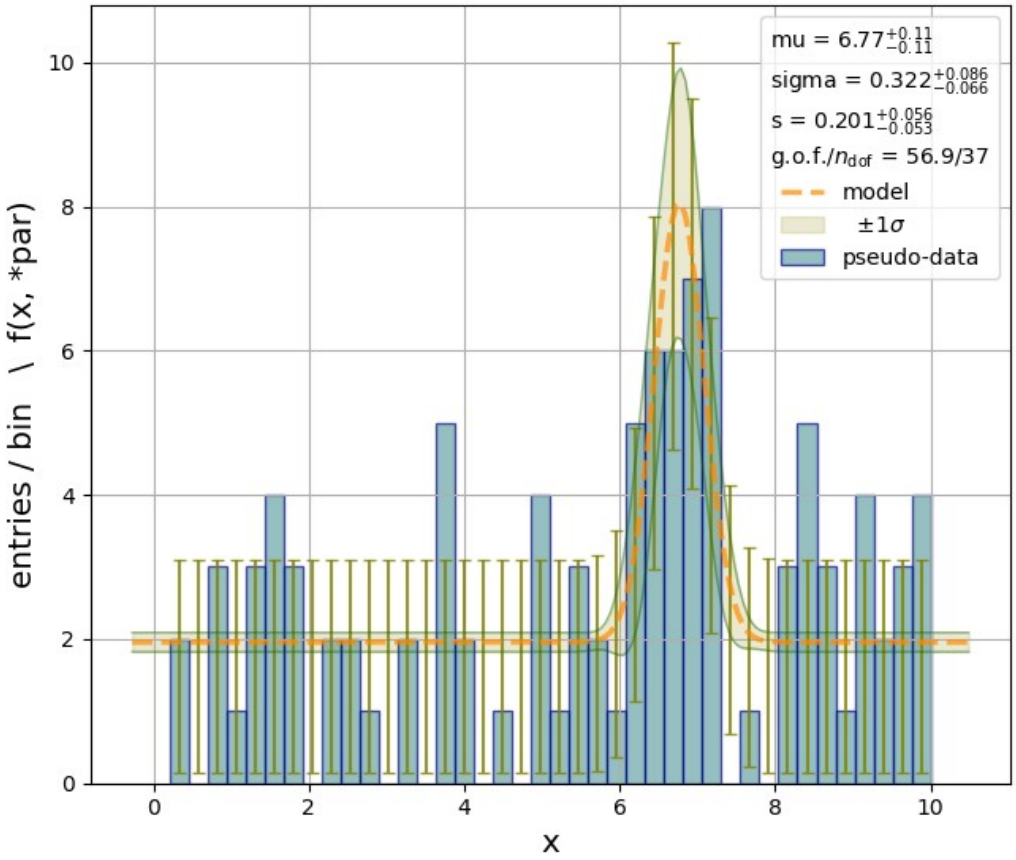
Beispiel: Anpassung mit PhyPraKit.phyFit

Skript PhyPraKit/examples/Beispiel_Diodenkennlinie.py



Beispiel: Anpassung mit PhyPraKit.phyFit.hFit()

PhyPraKit//examples/test_hFit.py

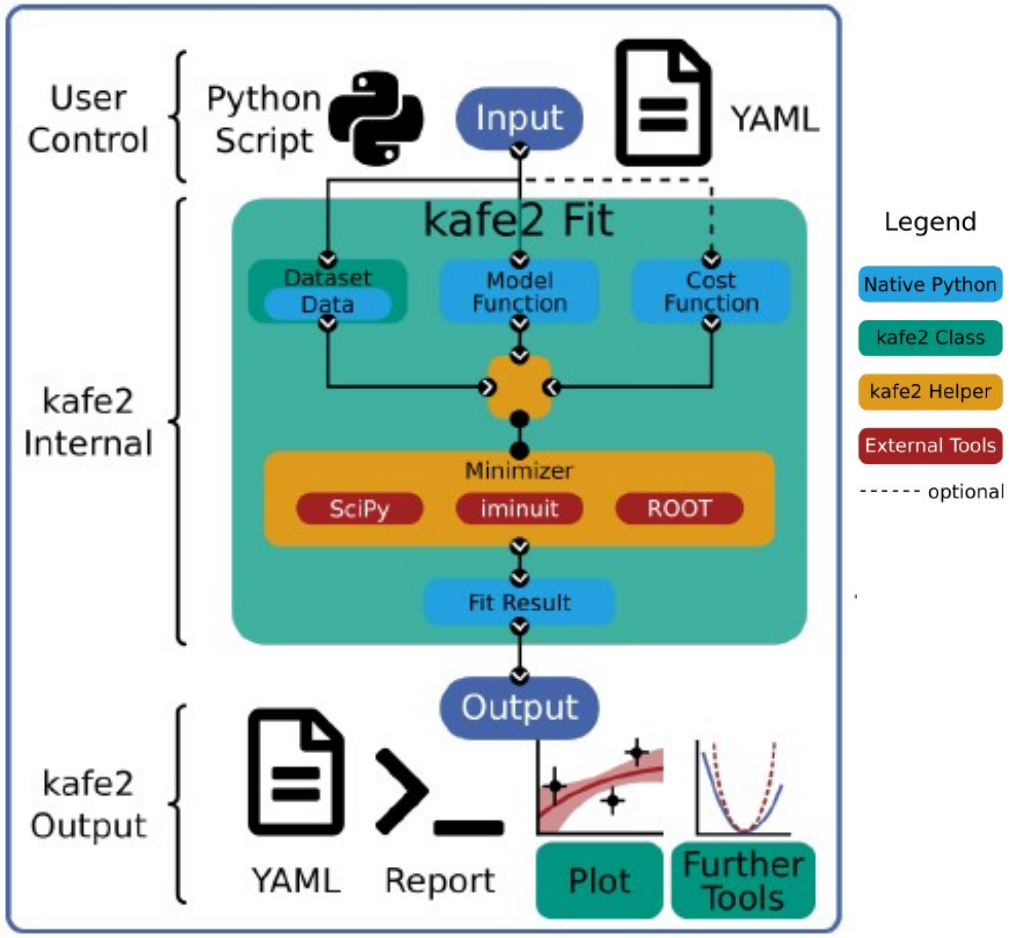


Anpassungen mit kafe2

Aus der Dokumentation zu kafe2:

kafe2 is an open-source Python package designed to provide a flexible Python interface for the estimation of model parameters from measured data. It is the spiritual successor to the original kafe package.

kafe2 offers support for several types of data formats (including series of indexed measurements, xy value pairs, and histo-grams) and data uncertainty models, as well as arbitrarily complex parametric models. The numeric aspects are handled using the scientific Python stack (NumPy, SciPy, ...). Visualization of the data and the estimated model are provided by matplotlib.

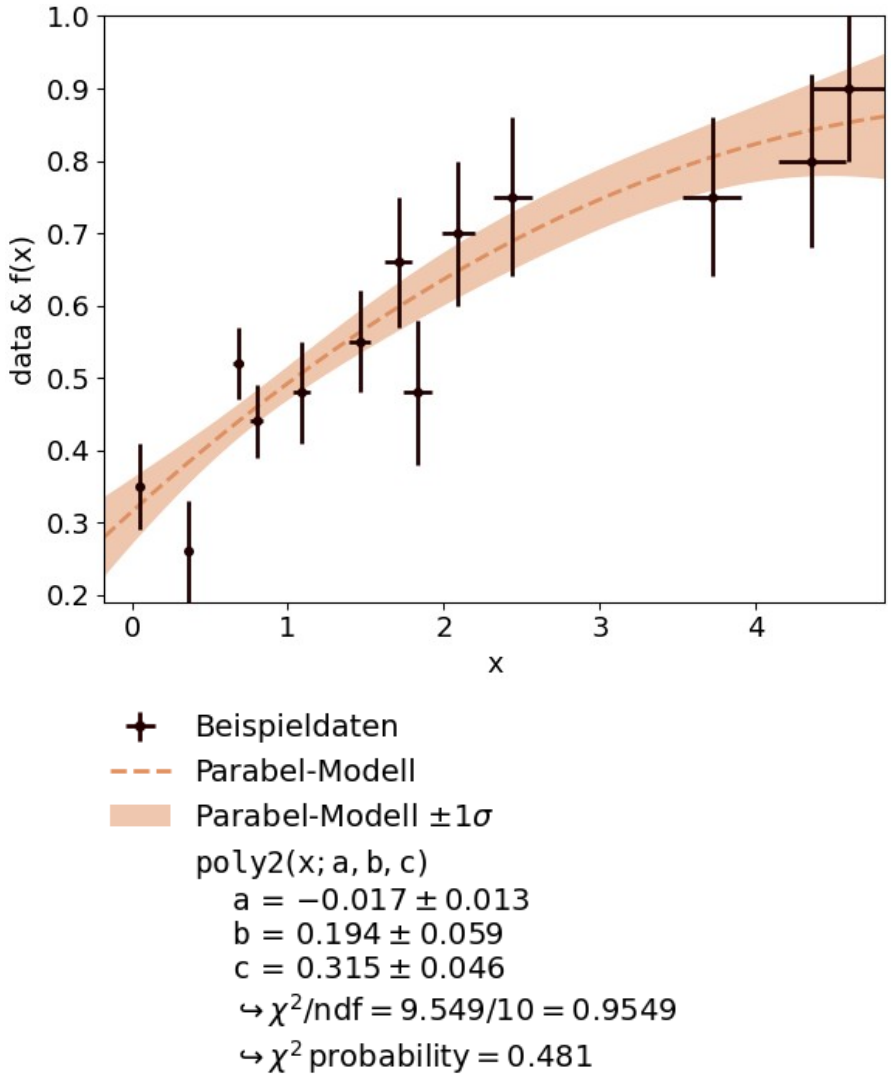


https://github.com/dsavoiu/kafe2

kafe2 Anwendung

Skript fitexample_kafe2.py

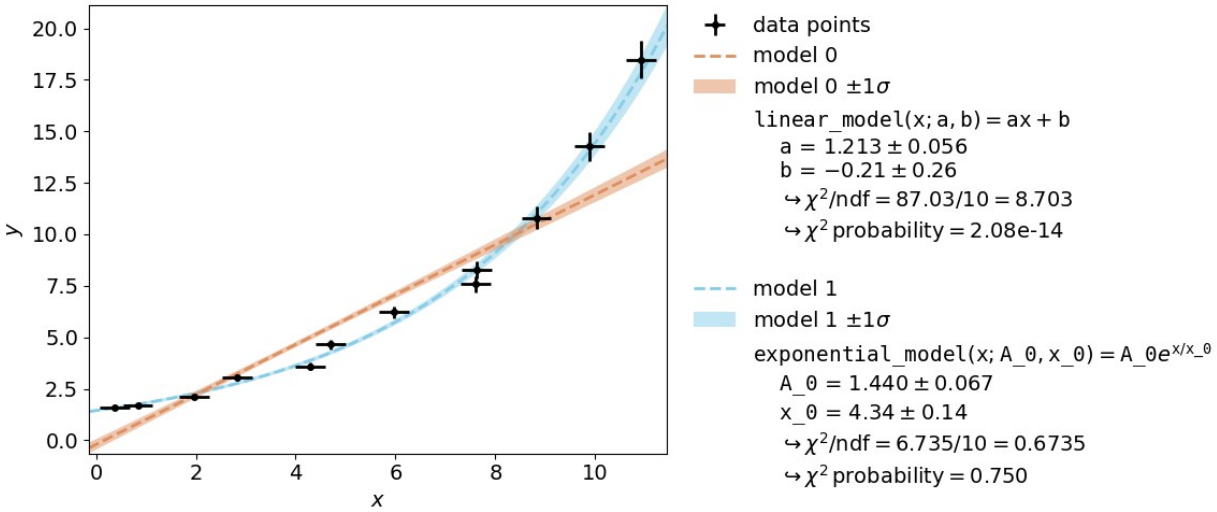
```
# Imports  #
from kafe2 import XYContainer, Fit, Plot
import numpy as np, matplotlib.pyplot as plt

### define the model function
def poly2(x, a=1.0, b=0.0, c=0.0):
    return a * x**2 + b * x + c

# Workflow #
# 1. load the experimental data from a file
x, y, e = np.loadtxt('fitexample.dat', unpack=True)
# 2. convert to kafe2 data structure and add uncertainties
xy_data = XYContainer(x, y)
xy_data.add_error('y', e)                  # independent erros y
xy_data.add_error('x', 0.05, relative=True)  # independent relative errors x
# set meaningful names
xy_data.label = 'Beispieldaten'
xy_data.axis_labels = ['x', 'data & f(x)']
# 3. create the Fit object
my_fit = Fit(xy_data, poly2)
# set meaningful names for model
my_fit.model_label = 'Parabel-Modell'
# 4. perform the fit
my_fit.do_fit()
# 5. report fit results
my_fit.report()
# 6. create and draw plots
my_plot = Plot(my_fit)
my_plot.plot()
# 7. show or save plots
## plt.savefig('kafe_fitexample.pdf')
plt.show()
```



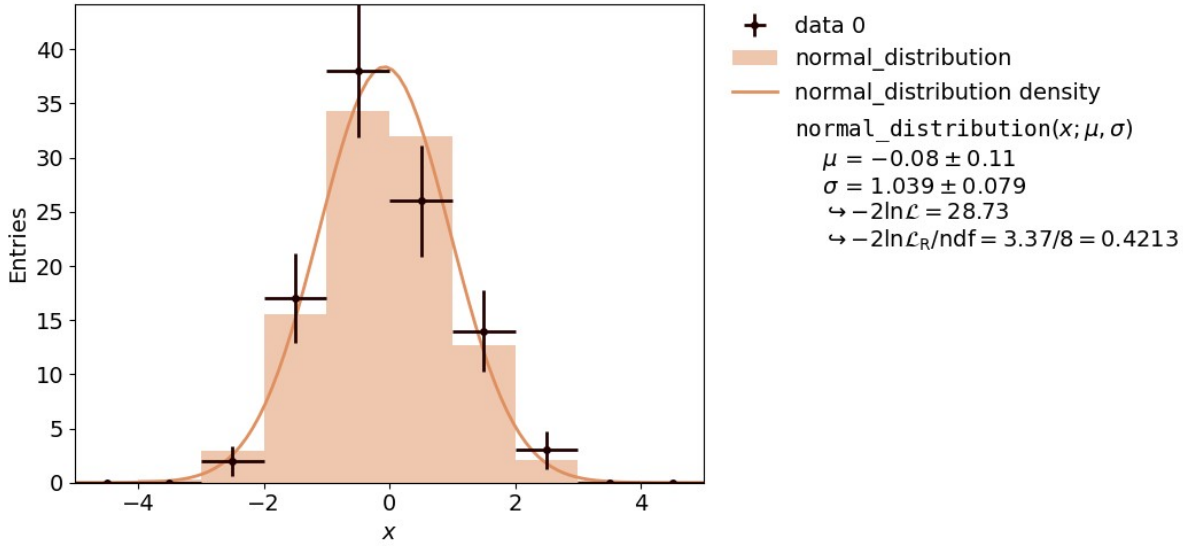
Jupyter Tutorial zu kafe2

Zu kafe2 gibt es umfangreiche Dokumentation und Jupyter-Tutorials in deutscher englischer Sprache sowie eine ausführliche Sammlung an Beispielen.

kafe2/examples/002_model_functions/model_functions.py



kafe2/examples/009_histogram_fit/histogram_fit.py



Anpassungen im kafe-Framework sowie die Text- und Grafik-basierte Ausgabe sind sehr flexiblel konfigurierbar.

→

Beispiele und Dokumentation zu Rate ziehen !

kafe2 Methode .add_error()

Aufbau der Kovarianzmatrix aus einzelnen Komponenten der Unsicherheit:

- unabhängige und korrelierte,
- absolute oder relative Unsicherheiten
- für Ordinate und Abszisse

add_error (*err_val*, *name=None*, *correlation=0*, *relative=False*)

aus kafe2 Dokumentation

Add an uncertainty source to the data container. Returns an error id which uniquely identifies the created error source.

Parameters

- **err_val** (*float or iterable of float*) – pointwise uncertainty/uncertainties for all data points
- **name** (str or `None`) – unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – correlation coefficient between any two distinct data points
- **relative** (*bool*) – if `True`, **err_val** will be interpreted as a *relative* uncertainty

Returns error name

Return type str

An- und Abschalten einzelner Komponenten mit `enable_error()` bzw. `disable_error()`

Fit auf der Kommandozeile: „kafe2go“

- kein spezieller Code notwendig,
- Daten und Modell in einer Datei (im yml-Format)

```
> kafe2go fit.yml
```

Datei fit.yml

```
x_data: [ 0.38,  0.84,  1.97,  2.83, 4.28,  4.70,  5.97, 7.61, 7.63, 8.82, 9.87,  10.91]
x_errors:
- correlation_coefficient: 0.0
  error_value: 0.3                  # use absolute errors, in this case always 0.3
  relative: false
  type: simple
y_data: [1.60, 1.67,  2.13, 3.05, 3.58, 4.65, 6.21, 7.58,  8.27,  10.80, 14.27, 18.49]
y_errors:
- correlation_coefficient: 0.0
  error_value: 0.05
  relative: true                    # use relative errors, in this case 5% of each value
  type: simple

model_function: |
  def exponential_model(x, A_0=1., x_0=5.):
      # simple exponential function, the kwargs in the function header are parameter defaults.
      return A_0 * np.exp(x/x_0)
```

Wrapper-Funktion für kafe2 in PhyPraKit

Einfache Anwendung von kafe2 mit PhyPraKit.k2Fit()

```
PhyPraKit.k2Fit(func, x, y, sx=None, sy=None, srelx=None, srely=None, xabscor=None,
                yabscor=None, xrelcor=None, yrelcor=None, ref_to_model=True, con-
                straints=None, p0=None, limits=None, plot=True, axis_labels=['x-data', 'y-
                data'], data_legend='data', model_expression=None, model_name=None,
                model_legend='model', model_band='$\\pm 1 \\sigma$', fit_info=True,
                plot_band=True, asym_parerrs=True, plot_cor=False, showplots=True, quiet=True)
```

Fit an arbitrary function func(x, *par) to data points (x, y) with independent and correlated absolute and/or relative errors on x- and y- values with package iminuit.

Beispiel PhyPraKit/examples/test_k2Fit.py

Übersicht Anpassungswerkzeuge

	kafe2	phyFit	scipy curve_fit	scipy odr	root TGraph
Kovarianzmatrix	✓	✓	✓	x	x
Verwalten von Unsicherheiten	✓	(✓)	x	x	x
Unsicherheiten in x	✓	✓	x	✓	✓
relative Unsicherheiten wrt. Model	✓	✓	x	x	x
volle Gauß-Likelihood	✓	✓	x	x	x
Constraints	✓	✓	x	x	(✓)
Fixieren von Parametern	✓	✓	x	x	✓
Histogramm - Fits	✓	✓	x	x	root.TH1
Unbinned ML	✓	✓	x	x	x
Verschiedene Minimizer	✓	x	x	x	x
Grafische Ausgabe	✓	✓	x	x	✓
Asymmetrische Unsicherheiten	✓	✓	x	x	(✓)

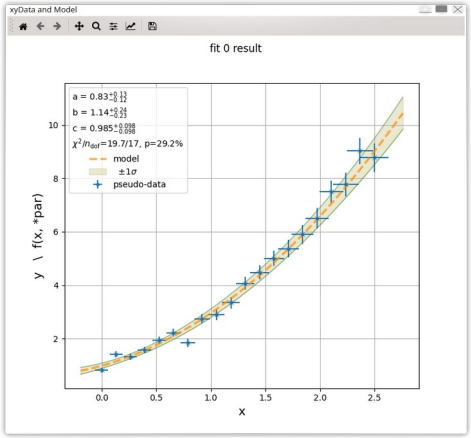
Ensemble-Tests

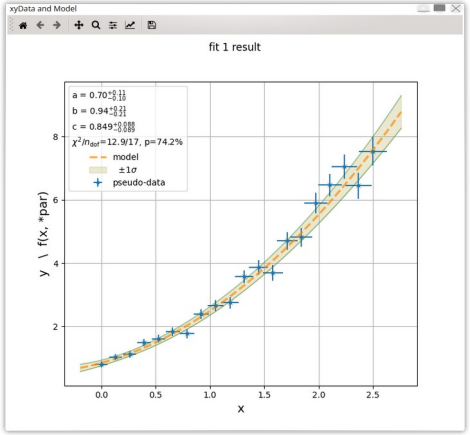
Ensemble-Tests

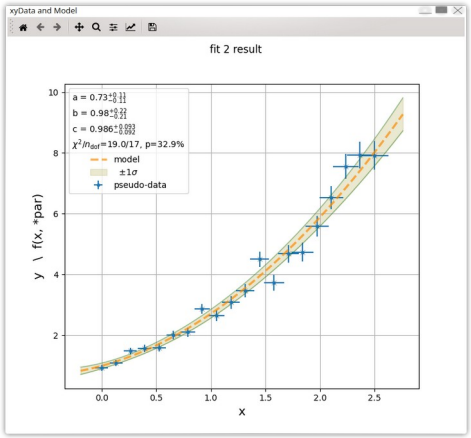
Zur Überprüfung der Eigenschaften von Schätzverfahren:

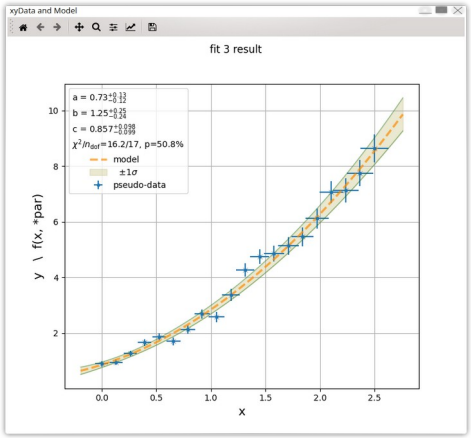
- Wiederholte Erzeugung simulierter Daten mit mit MC-Methode
- Jeweils Durchführung der Anpassung
- Auswertung der Ergebnisse:
  - Verteilung der Schätzwerte der Parameter
  - Korrelationen der Parameter
  - Verteilung der „Goodness-of-Fit“
  - Test der Coverage:
    Überprüfung, wie häufig Parameterwerte im Unsicherheitsintervall liegen

Script toyMC_Fit.py enthalten in PhyPraKit/examples oder im Tutorial negLogLFits.ipynb









Beispiel: Ensemble-Test

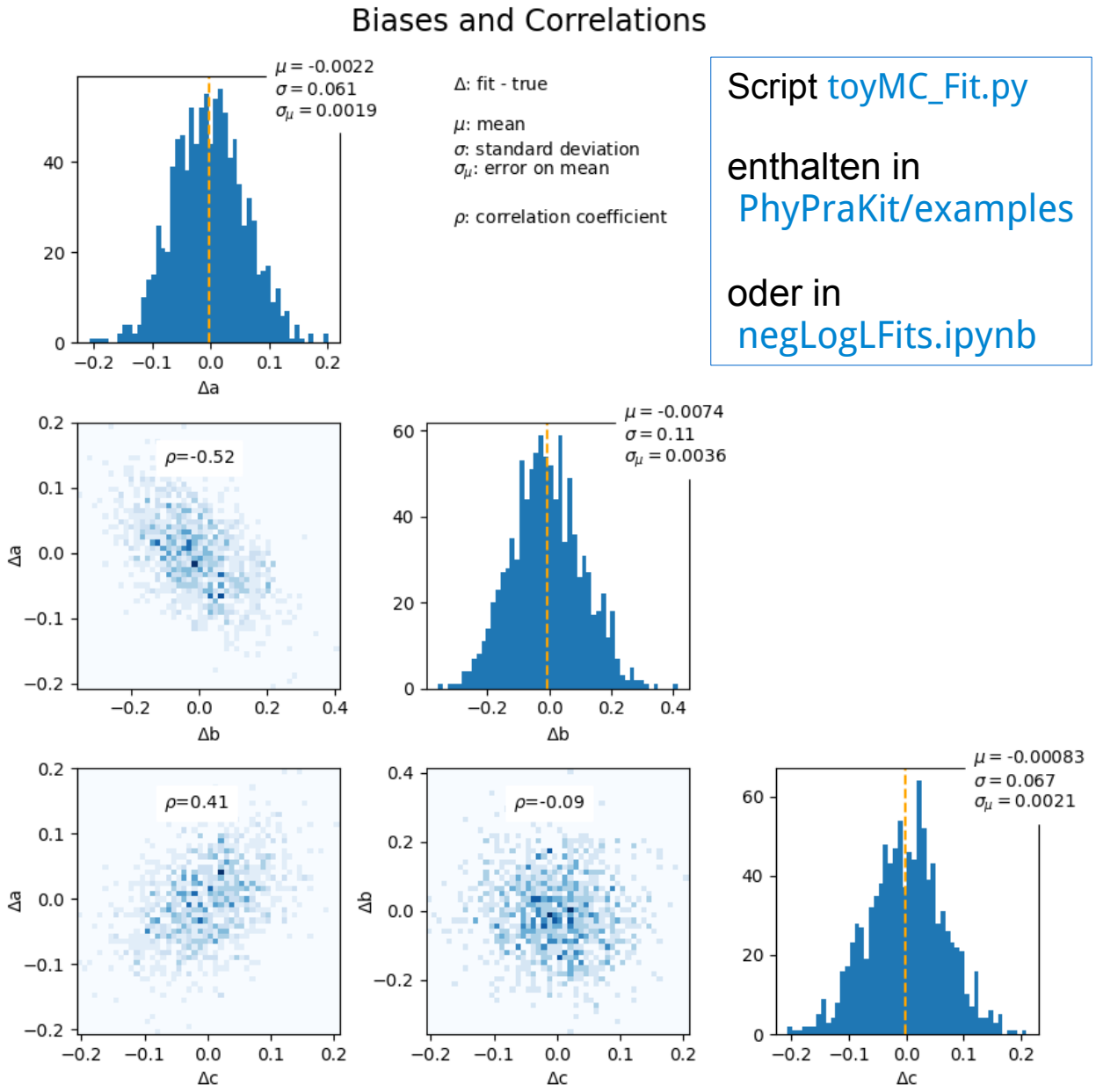
Beispiel:

Anpassung einer
Parabel an Daten
mit Unsicherheiten
in x- und y-Richtung

(1000 Pseudoexeperimente)

Ausgabe:

* biases:
  0: -0.00235 +\- 0.0019, std 0.0605
  1: -0.00754 +\- 0.0036, std 0.114
  2: -0.000882 +\- 0.0021, std 0.0667
* coverage:
  0: 97.4%
  1: 96.8%
  2: 98%



Script toyMC_Fit.py

enthalten in
PhyPraKit/examples

oder in
negLogLFits.ipynb

Beispiel: Ensemble-Test

Ausgabe von toyMC_Fit für verschiedene Szenarien:

```
*==*  1000  successful fits done:
* parameter names:
  0: a
  1: b
  2: c
```

-2 ln $\mathcal{L}$, rel. wrt. model

* biases:
  0: -0.0053 +\- 0.0038, std 0.119
  1: -0.0103 +\- 0.0071, std 0.225
  2: -0.0066 +\- 0.0029, std 0.091

-2 ln $\mathcal{L}$, rel. wrt. data

* biases: größere Verzerrung !
  0: -0.0169 +\- 0.0037, std 0.118
  1: -0.0328 +\- 0.0070, std 0.223
  2: -0.0215 +\- 0.0028, std 0.089

χ^2 , rel. wrt. model

* biases: größere Verzerrung !
  0: 0.0452 +\- 0.0040, std 0.127
  1: 0.0189 +\- 0.0074, std 0.234
  2: 0.0198 +\- 0.0029, std 0.093

χ^2 , rel. wrt. data

* biases: größere Verzerrung !
  0: 0.0238 +\- 0.0039, std 0.124
  1: -0.0236 +\- 0.0073, std 0.230
  2: -0.0118 +\- 0.0029, std 0.091

Numerische Optimierung

Beispiel **Simplex-Verfahren**

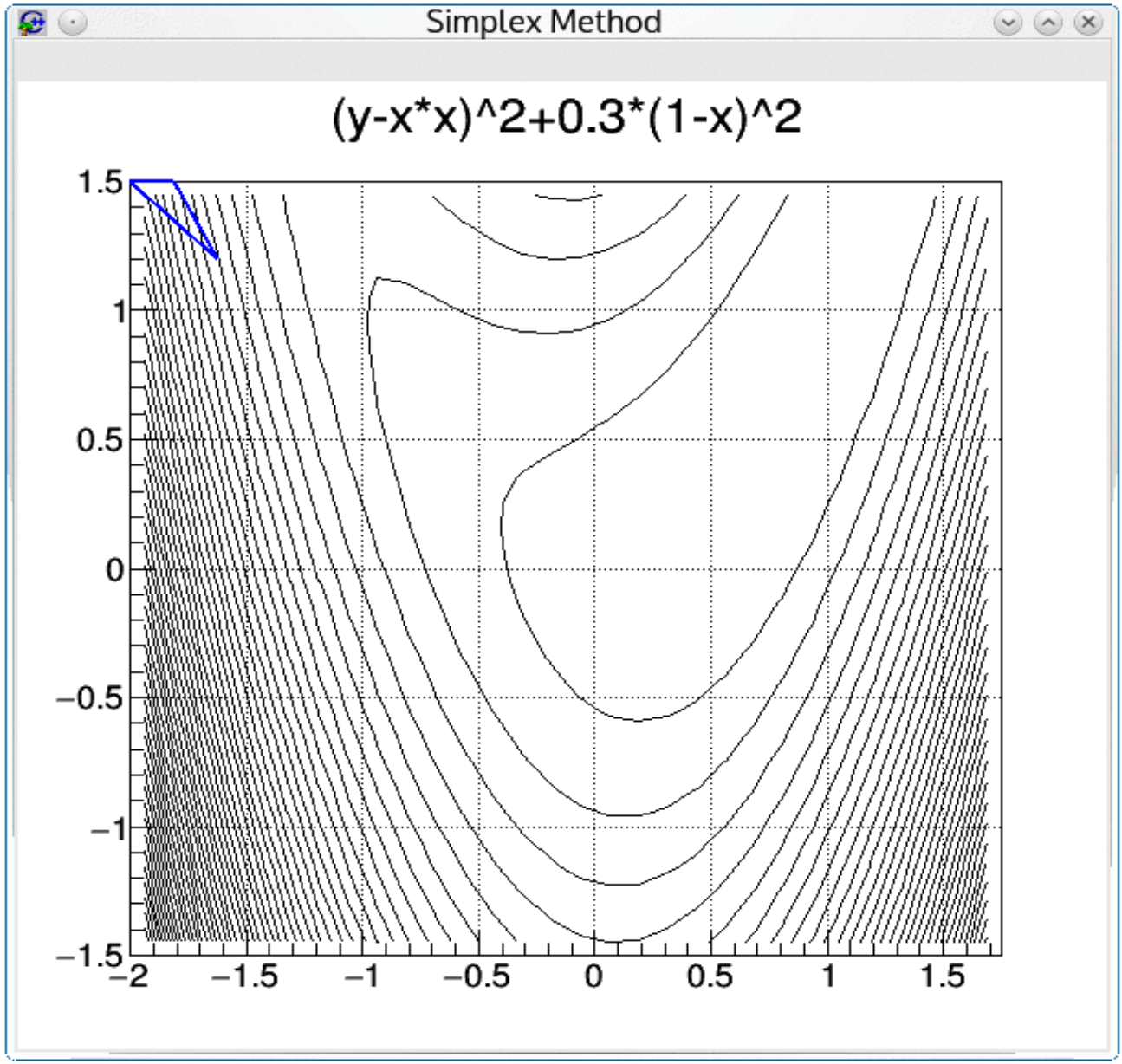
Illustration der **Simplex-Methode**

am Beispiel der
Rosenbrock'schen
„Bananenfunktion“

Simplex:

n-dimensionaler Polyeder aus *n+1* Punkten im R^n

Bei jedem Schritt wird der schlechteste Punkt $\mathbf{x}_r$ mit größtem Funktionswert $F(\mathbf{x}_r)$ durch einen neuen, besseren, ersetzt, oder der Simplex ggf. verkleinert oder vergrößert.



„Simuiertes Ausglühen“ – Simulated Annealing

Monte Carlo-Methoden sind sehr effizient bei der Suche über große Bereiche in hochdimensionalen Problemen –

sehr gut geeignet zur Bestimmung von Startwerten

Beispiel: Simulated Annealing *(Übungsaufgabe)*

Idee aus der Physik entlehnt:

Abkühlung eines thermodynamischen Systems, das der Boltzmann-Statistik genügt

Wahrscheinlichkeit, Mikrozustand E_j anzutreffen: $p(E_j) \propto \exp(-\frac{E_j}{k_B T})$

energetisch günstigster Zustand E_0 wird durch Abkühlen, T→0, erreicht.

Anschaulich: „Herausschütteln“ aus lokalen Minima wie Reis in einem Sieb.

Die Energie aus dem physikalischen Beispiel entspricht allgemein der „Kostenfunktion“ *F*

also –ln $\mathcal{L}$ oder χ^2 im Falle der Parameteranpassung

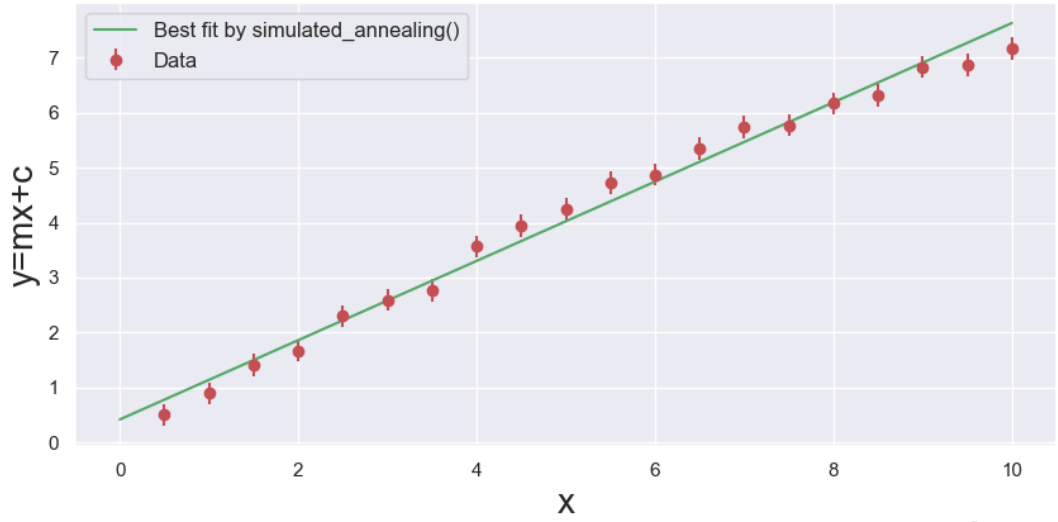
transformiere $F(x) \rightarrow \tilde{F}(x) = \exp\left(-\frac{F(x)}{T}\right)$;

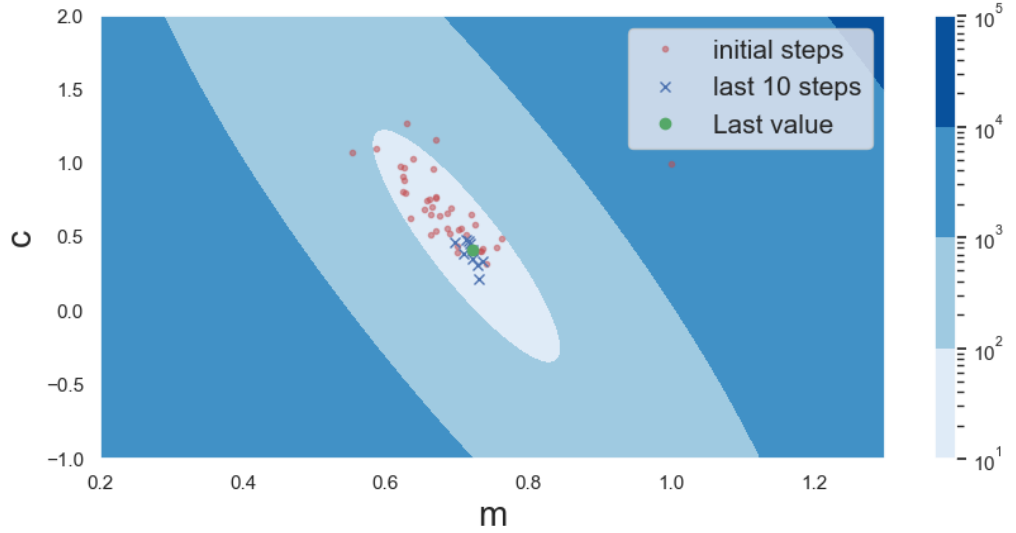
für zwei Zustände 1, 2 gilt $\frac{\tilde{F}(x_1)}{\tilde{F}(x_2)} = \frac{\tilde{F}_1}{\tilde{F}_2} = \exp\left(-\frac{F_2 - F_1}{T}\right)$

Beispiel: Simulated Annealing

```
def simulated_annealing(func, x_data, y_data, y_unc, m_start,
c_start):
    """ Simple algorithm for Simulated Annealing
    """
    # Define temperature setting
    initial_temp = 10
    final_temp = 1
    alpha = 0.01
    # Initialize solution as starting state
    current_temp = initial_temp
    solution = [[m_start], [c_start]]
    # annealing loop
    while current_temp > final_temp:
        # Compute normally distributed steps
        step = np.random.randn(2)*0.2
        # Build next state from old state and steps
        next_state = [solution[0][-1]-step[0],
                      solution[1][-1]-step[1]]
        # Compute costfunction difference
        cost_diff = (func(x_data, y_data, y_unc, solution[0][-1],
                    solution[1][-1]) - func(x_data, y_data, y_unc,
                    next_state[0], next_state[1]))
        # Accept (append) new state if difference is positive
        if cost_diff > 0:
            solution[0].append(next_state[0])
            solution[1].append(next_state[1])
        else:  # ... with tempeature-dependent probability
            if np.random.rand() < np.exp(cost_diff/current_temp):
                solution[0].append(next_state[0])
                solution[1].append(next_state[1])
        # Reduce temperature
        current_temp  -= alpha
    return solution
```

Geradenanpassung mit simulated annealing





Script SimulatedAnnealing.py

Simulated Annealing – der Algorithmus

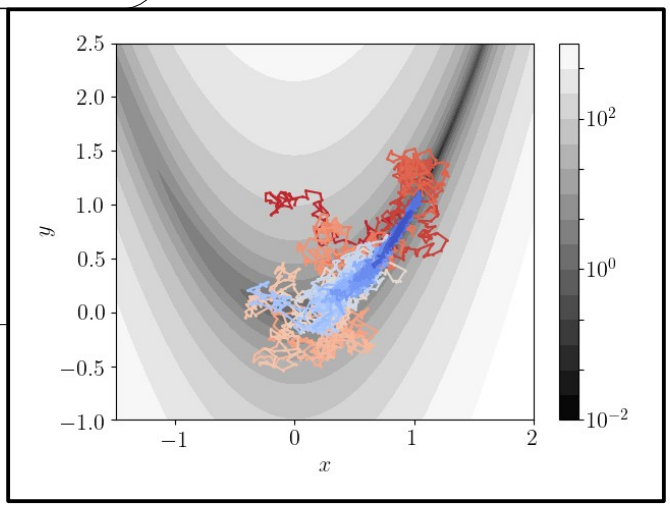
Der Algorithmus:

1. Start mit einem zufälligen Zustand x_1 bei der Temperatur T
2. erzeuge zufälligen neuen Zustand x_{n+1}
3. akzeptiere x_{n+1}, wenn $F_{n+1} < F_n$

   oder wenn $r \leqslant \exp\left(-\frac{F_{n+1}-F_n}{T}\right)$ mit Zufallszahl $r \in]0,1]$
4. wiederhole Schritt 2 bis 3 k-mal
5. kontrolliere Konvergenz
6. reduziere Temperatur um ΔT und gehe zu Schritt 2, falls Endtemperatur noch nicht erreicht ist

Zahlreiche freie Parameter:

- Anfangstemperatur ***T***
- Zahl ***k*** der Iterationen bei gleicher Temperatur
- das „Kühl-Schema“, also die Bestimmung von ***ΔT*** (linear, exponentiell, logarithmisch, oder „adaptiv“)



Erfolgreiches „annealing“ benötigt Fein-Abstimmung und Kontrolle