# Blatt04

November 27, 2023

---

## 1 Rechnernutzung in der Physik

**Institut für Experimentelle Teilchenphysik**
**Institut für Theoretische Teilchenphysik**
Prof. G. Quast, Prof. M. Steinhauser
Dr. A. Mildenberger, Dr. Th. Chwalek
Ilias Seite zum Kurs
WS 2023/24 – Blatt 04
Abgabe: Montag 4.12.2023 bzw. Dienstag 5.12.2023

---

Auf dem vierten Übungsblatt beschäftigen Sie sich vertieft mit der Parameterschätzung und wenden einen Hypothesentest an, um die Gemeinsamkeit zweier Datensätze zu überprüfen.

Auf dem vergangenen Übungsblatt haben Sie die Parameterschätzung selbst implementiert, indem Sie eine Kostenfunktion (*Likelihood*) geschrieben, diese an diskreten Stellen eines vorgegebenen Parameterraumes ausgewertet und das Minimum an einer der Stützstellen berechnet haben. Wie Sie aus der Vorlesung oder Ihrer eigenen Erfahrung kennen, weist das Vorgehen an vielen Stellen Optimierungsbedarf auf. Sie könnten viel mehr Stützstellen verwenden, um einen besseren Scan der Likelihood zu erhalten, dann müssen Sie aber viel mehr rechnen, zudem müssen Sie Unsicherheiten berücksichtigen, das Verfahren für alle zukünftigen Herausforderungen verallgemeinern und alles so niederschreiben, dass der Code lesbar und einfach verständlich ist. An dieser Stelle können wir Sie beruhigen, denn all diese Aufgaben haben bereits eine Vielzahl an Menschen für Sie erledigt.

Im Folgenden beschäftigen Sie sich mit einem Minimierungsverfahren, welches Ihnen erlaubt deutlich effizienter ein Minimum in einer Parameterlandschaft zu finden, anschließend sind Sie eingeladen mithilfe von *kafe2* die Parameterschätzung vollständig durchzuführen. Zum Abschluss lernen Sie den *Student'schen-t-Test* kennen.

Wie Sie in Ihrem Studium möglicherweise bereits erkannt haben, ist das Finden eines Optimums einer hochdimensionalen Kostenfunktion $F(x)$ eine häufige Fragestellung. Damit Sie eine Vorstellung der Lösungsansätze erhalten, sollen Sie in dieser Aufgabe selbst einen Algorithmus schreiben, der Ihnen dabei hilft, jenes Optimum zu bestimmen. Der Algorithmus ist bei Weitem nicht der beste oder schnellste, verdeutlich Ihnen aber den Einsatz von Monte-Carlo-Methoden in der Modernen Physik. Als Beispiel betrachten Sie die zweidimensionale, modifizierte Rosenbrock Funktion

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

mit drei Parametern $a, b$ und $c$. Die Funktion weist einige lokale Minima, aber immer nur ein globales Minimum auf. Die Bearbeitung der Aufgabe erfolgt für die zufällige Wahl an Startparametern: $a = 11$, $b = 7$, $c = 0.1$.

## 1.1 a) Implementierung des Algorithmus

In der ersten Teilaufgabe erstellen Sie das Grundgerüst der Minimierung, indem Sie zunächst die modifizierte Rosenbrock-Funktion und anschließend das simulierte Abkühlen als Funktionen definieren. Zur Regulierung der Temperatur wird eine konstante Kühlrate verwendet, die als Parameter des Algorithmus betrachtet wird. Zusätzlich wird eine Schrittweite als Parameter eingeführt, welche die Distanz eines neuen Zustandes zum alten beeinflusst.

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation
from IPython.display import Image
# initialize random number generator
rng = np.random.default_rng( 42 )
```

[ ]:
```python
# ToDo: implement modified rosenbrock function
def modified_rosenbrock_function(x, par):
    xValue = x[0]
    yValue = x[1]
    a = par[0]
    b = par[1]
    c = par[2]

    return xValue**2+yValue**2
```

[ ]:
```python
def simulated_annealing(
    init_vals=[0,0],
    rosenbrock_pars=[0, 0, 0],
    init_temp=100,
    final_temp=1,
    cool_speed=1,
    step_size=1
):
    """
    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."
                        f"Expected number of initial values: {nParameter}")

    #␣
↪============================================================================================
    # Presetting and book keeping
    #␣
↪============================================================================================

    # parameters of the algorithm
    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  = init_vals
    currentFunctionValue =  modified_rosenbrock_function(currentParameters,␣
↪rosenbrock_pars)

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

    # log keeping: keep track of path
    listOfPoints= []

    #␣
↪============================================================================================
    #␣
↪============================================================================================

    # Heat the system
    temperature = initialTemperature

    # Start to slowly cool the system
```

```
    while (temperature > finalTemperature):

        # ToDo: Change parameters
        newParameters = rng.standard_normal(nParameter)

        # Get the new value of the function
        newFunctionValue = modified_rosenbrock_function(newParameters,␣
↪rosenbrock_pars)

        # ToDo: Compute Boltzman probability

        # ToDo: Acceptance rules


        # Delete this as soon as acceptance rules are implemented
        bestParameters = currentParameters
        listOfPoints.append(currentParameters)

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


    # end of cooling loop
    return bestParameters, listOfPoints
```

## 1.2 b) Finden des globalen Minimums

Testen Sie zunächst Ihren Algorithmus aus, indem Sie Startwerte für $x$ und $y$ in der Nähe des globalen Minimums wählen $(x, y) = (-2, 3)$. Machen Sie sich dabei mit den Eigenschaften der Anfangs- und Endtemperaturen, der Abkühlrate und der Schrittgröße vertraut.

Sobald Sie mit Ihrem Algorithmus zufrieden sind, dieser also zuverlässig gegen das globale Minimum konvergiert, wählen Sie als Startpunkt einen Punkt in einem der Nebenminima (z.B. $(x, y) = (3, -2)$) aus und passen Sie die Parameter des Algorithmus so an, dass er wieder zuverlässig gegen das globale Minimum konvergiert. Zur korrekten Einstellung der Parameter bietet es sich an, entweder einen Bereich an Parametern systematisch zu untersuchen oder den Einfluss eines jeden Parameters graphisch zu untersuchen. Sie finden einen Vorschlag zur graphischen Untersuchung in den folgenden Zellen. In beiden Fällen bietet es sich jedoch an, sich die Eigenschaften der Algorithmusparameter vorher zu überlegen.

**Hinweise und Überlegungen:**

- Die Anfangs- und Endtemperatur sollen die Kostenfunktionswerte der Anfangsbedingung und des gesuchten Minimums widerspiegeln.
- Die Differenz zwischen Anfangs- und Endtemperatur und die Kühlrate beeinflussen die Anzahl an Iterationen.
- Die Temperaturskala beeinflusst die Wahrscheinlichkeit für Sprünge.
- Die Schrittgröße sollte adäquat im Bezug zum Abstand der Minima gewählt werden.

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


initial_values = [-2, 3]
initial_temp, final_temp, cool_speed, step_size = 100, 1, 1, 1

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!
 ↪")
```

[ ]:
```
%matplotlib notebook
```

[ ]:
```
Points = np.array(listOfPoints).T
frames = 30
total = len(Points[0])


def init():
    #p1.set_zscale('log')
    p1.set_xlim(-5, 5)
    p1.set_ylim(-5, 5)
    p1.set_zlim(0.0001, 1000)

    p1.set_xlabel('X')
    p1.set_ylabel('Y')
    p1.set_zlabel('Z')

    # plot function
```

```
    X = np.linspace(-5, 5, 1000)
    Y = np.linspace(-5, 5, 1000)
    X, Y = np.meshgrid(X, Y)

    Z = modified_rosenbrock_function([X, Y], [a, b, c])
    p1.plot_surface(X, Y, Z, cmap='coolwarm', alpha=0.4)
    p1.scatter(bestParameters[0], bestParameters[1],␣
 ↪modified_rosenbrock_function(bestParameters, [a, b, c]), c="k", label="foo")


def update(num, data, plotcom):
    low = int(num/frames * len(Points[0]))
    up = int((num+1)/frames * len(Points[0]))

    plotcom.set_data(data[0:2, low:up])
    plotcom.set_3d_properties(data[2, low:up])



fig=plt.figure(figsize=(10, 10))
p1=fig.add_subplot(1, 1, 1, projection='3d')

data = np.array([Points[0], Points[1], modified_rosenbrock_function(Points,␣
 ↪[a,b,c])])
plotcommand = p1.plot(data[0], data[1], data[2], color='blue', marker=".",␣
 ↪linestyle="")[0]

# animate
animation = matplotlib.animation.FuncAnimation(fig, update, frames=frames,␣
 ↪init_func=init, fargs=(data, plotcommand), interval=250, blit=False)

animation.save('annealing.gif', writer=matplotlib.animation.PillowWriter(fps=5))

from IPython.display import HTML
HTML(animation.to_jshtml())
```

Die zweite Aufgabe auf diesem Übungsblatt wird Ihnen sehr bekannt vorkommen, da Sie im Anfängerpraktikum garantiert eine Messung dieser Art durchgeführt haben. In der Datei *StromSpannungsKennlinie.csv* finden Sie 40 Strom- und Spannungsmesswerte. Diese Daten könnten aus einer beliebigen Kalibrationsmessung stammen. Ihre Aufgabe ist, aus diesen Strom- und Spannungswerten den elektrischen Widerstand des untersuchten Bauteils zu bestimmen.

Der Kernaspekt der Aufgabe liegt darin, die Anpassung korrekt durchzuführen und Ihnen zu verdeutlich, wie einfach eine Anpassung, unter der Berücksichtigung aller typischen Unsicherheiten, ist. Es wird Ihnen nahegelegt, hier *kafe2* als Werkzeug zu verwenden, da dieses *Python*-Paket viele wichtige Funktionen beinhaltet, die sonst durch mühseelige Handarbeit in die Anpassung eingebaut werden müssten.

Im Folgenden betrachten Sie einen Datensatz aus je 20 gemessenen Strom- und Spannungswerten aus zwei Messbereichen: Der erste Messbereich umfasst die Spannung $U \in [0, 2]$V und den Strom $I \in [0, 20]$mA, der zweite die Bereiche $U \in [2, 20]$V und $I \in [20, 200]$mA. Für die Unsicherheiten der Messungen können Sie sich auf die Anzeige des Messinstruments und die Herstellerangaben beziehen: * Jeder Messbereich umfasst eine Angabe von fünf Stellen mit einer **absoluten Unsicherheit** von **zwei Stellen**. Messen Sie also im ersten Messbereich die Spannung, wird Ihnen der Bereich *x,xxxx*V angezeigt, die Unsicherheit bezieht sich dann auf die letzten zwei Stellen. * Alle Strom- und Spannungsmessungen unterliegen einem **konstanten Untergrundrauschen**, bei der Spannung gibt der Hersteller des verwendeten Messgerätes **20mV**, beim Strom **1mA** an. * Zusätzlich wird eine **relative, vollständig korrelierte Unsicherheit** von **5%** auf alle Messwerte angegeben.

## 1.3 a) Vorbereitung der Auswertung

Bereiten Sie im ersten Aufgabenteil die Auswertung der Messdaten vor, indem Sie * eine lineare Funktion definieren, die Ihnen als Modellfunktion bei der Anpassung dienen wird, * die Messdaten korrekt einlesen * und alle beschriebenen Unsicherheiten korrekt aufschreiben und gegebenenfalls verrechnen.

Beachten Sie dabei die Aufteilung der Unsicherheiten auf die Messbereiche. Es bietet sich beispielsweise an, Unsicherheiten mithilfe eines Arrays zu definieren. In der folgenden Teilaufgabe haben Sie die Möglichkeit, entweder Fleißkommazahlen (*float*) oder Arrays der Länge der Messdaten anzugeben.

[ ]:
```
%matplotlib inline
```

[ ]:
```
import PhyPraKit as ppk
from kafe2 import XYContainer, Fit, Plot, ContoursProfiler
```

[ ]:
```
# linear model


# uncertainty components


# data read in
```

## 1.4 b) Durchführung der Anpassung

Führen Sie in dieser Teilaufgabe die Anpassung der Geraden durch. Beispiele, wie die Anpassung mithilfe von *kafe2* funktioniert, finden Sie in der Dokumentation zu *kafe2*.

Der Einfluss der relativen, korrelierten Unsicherheit wird zu einer Verzerrung der Anpassung führen. Führen Sie dafür die Anpassung einmal mit dem Bezug der relativen Unsicherheit auf die Daten und einmal mit Bezug auf das Modell durch und vergleichen Sie beide Ergebnisse.

**Hinweise:**

- Die Daten übergeben Sie am einfachsten an *kafe2* mithilfe der Klasse *XYContainer(x, y)*.

- Der Fit lässt sich mithilfe der Klasse *Fit(container)* erstellen.
- Die Unsicherheiten können Sie mithilfe der Methode *add_error(err_val)* der Klasse *Fit* definieren.
- Die Argumente *relative*, *correlation* und *reference* (*'data'* oder *'model'*) sind notwendig zur korrekten Fehlerbehandlung in der Methode *add_error()*.
- Verbessern Sie die Lesbarkeit des Plots mithilfe von geschickt ausgewählten Labeln und Farben im Plot.
- Beim Plotten mithilfe der Klasse *Plot()* bietet es sich an, *asymmetric_parameter_errors=True* zu setzen, um asymmetrische Unsicherheiten zu erhalten. Dies ist äquivalent zu dem Vorgehen, mit dem Sie auf dem vorigen Blatt Konfidenzintervalle berechnet haben.

**Zusatzaufgabe (ohne Wertung)**

- Betrachten Sie die Likelihood-Konturen der Parameter in beiden Fällen der Anpassung. Wo sehen Sie den Einfluss der unterschiedlichen Behandlung der relativen Unsicherheiten?

[ ]:
```
# prepare data for fit


# perform first fit: rel. uncertainty to data


# add uncertainties to first fit


# perform first fit


# perform second fit: rel. uncertainty to model


# add uncertainties to second fit


# perform second fit


# plot fits
```

In der letzten Aufgaben machen Sie sich an einem einfach Beispiel klar, wie ein sehr simpler Hypothesentest Ihnen dabei helfen kann, eine von Ihnen geforderte Entscheidung zu treffen. Als Beispiel betrachten Sie zwei Datensätze, *sample1.dat* und *sample2.dat*. Sie sollen die Aussage treffen, ob es möglich ist, dass die Datensätze aus einem gemeinsamen Datensatz stammen. Als Mittel der Überprüfung können Sie einen sogenannten verallgemeinerten *Student'schen-t-Test* durchführen.

Sie gehen von der Nullhypothese $H_0$ **Die Stichproben haben gleiche Erwartungswerte** $\mu_1 = \mu_2$ aus. Zur Quantifizierung des Unterschieds wird eine Prüfgröße (engl.: test statistics) $d$ berechnet,

die auf einen möglichen Unterschied der beiden Proben empfindlich ist. Für einen solchen Test wird der Betrag der Differenz der Erwartungswerte verwendet, die auf die erwartete Unsicherheit normiert werden:

$$d = \frac{|\mu_1 - \mu_2|}{\sigma},$$
$$\sigma = \sqrt{\frac{\sigma_1^2}{N_1} + \frac{\sigma_2^2}{N_2}}.$$

Dabei geben $\sigma_{1,2}$ bzw. $N_{1,2}$ die Standardabweichung bzw. Größe der Stichproben an.

### 1.5 a) Vorbereitung des Hypothesentests

Implementieren Sie eine Funktion zur Berechnung der Größe $d$ und der Anzahl der Freiheitsgrade $n_{\text{dof}} = N_1 + N_2 - 2$. Lesen Sie die Stichproben ein und histogrammieren Sie diese.

[ ]:
```
# Implement test statistic
```

[ ]:
```
# data read in


# plot both distributions
```

### 1.6 b) *Studentscher-t-Test*

Berechnen Sie nun die beobachtete Größe $d_{\text{obs}}$. Stellen Sie die $t$-Verteilung graphisch dar und zeichnen Sie den beobachteten Wert ein. Stellen Sie ebenfalls fest, ob Sie die Nullhypothese mit einer Irrtumswahrscheinlichkeit von $\alpha = 0.045(2\sigma)$ verwerfen können.

**Hinweise:**

- Die *SciPy*-Methode *scipy.stats.t.pdf()* ist eine große Hilfe bei der Darstellung der $t$-Verteilung.
- Die Überprüfung der Nullhypothese kann entweder über die Bestimmung des $p$-Wertes der beobachteten Größe $d_{\text{obs}}$ oder graphisch durch Einzeichnung von $d_0$ geschehen. In beiden Fällen bietet *scipy.stats.t* Methoden (*cdf*, *sf*) an, die bei der Berechnung hilfreich sind.

[ ]:
```
import scipy.stats as stat
import scipy.integrate as integrate
```

[ ]:
```
# calculate observed quantity


# accept or reject 0 hypothesis
alpha = 0.045


# plot t distribution
```