Karlsruhe Institute of Technology

`kafe` – Ein *Python*-Paket für elementare Datenanalyse im Physikpraktikum

`kafe` – A *Python* Package for Basic Data Analysis in Physics Laboratory Courses

Bachelorarbeit
von

Daniel Săvoiu

An der Fakultät für Physik
Institut für Experimentelle
Kernphysik (IEKP)

Erstgutachter: Prof. Dr. Günter Quast
Zweitgutachter: Dr. Fred-Markus Stober

2. Dezember 2013

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 2. Dezember 2013

. . . . . .

(**Daniel Săvoiu**)

Inhaltsverzeichnis

1. Einleitung

Die erste Begegnung mit der Datenanalyse im Physikstudium erfolgt in der Regel im Rahmen der physikalischen Anfängerpraktika. Dort werden üblicherweise vorgegebene Modelle an selbst aufgenommene Messreihen angepasst, um die Parameter des jeweiligen Modells zu ermitteln.

Dieser Anpassungsvorgang ist komplex und wird von vielen Faktoren beeinflusst, welche selbst unterschiedlich behandelt werden können. Aus diesem Grund ist die Frage der verwendeten Analysesoftware von besonderem Interesse, denn diese muss sowohl didaktisch als auch praktisch mit den Lernzielen im Physikpraktikum verträglich sein.

Hinsichtlich der Funktionalität erscheint hier die Notwendigkeit besonders wichtig, für das Praktikum grundlegende Aspekte wie Fehlerkorrelationen oder die Messunsicherheit jener Messgrößen, die als unabhängige Variablen aufgefasst werden, (sog. „x-Fehler“), richtig zu behandeln. Didaktisch sollte die Auswertesoftware den im weiteren Studium und im Beruf vorhandenen Standards entsprechen, um zugleich für eine Einführung in die rechnergestützte Physik eingesetzt werden zu können.

Aus einem kurzen Überblick über die typischen Softwarepakete, die im Praktikum für die Datenauswertung eingesetzt werden wird klar, dass keines von diesen allen Voraussetzungen entspricht. Ziel dieser Bachelorarbeit ist der Entwurf einer Analysesoftware, welche für den Einsatz im Rahmen der physikalischen Praktika geeignet ist und das Erlangen von Grundkenntnissen im Bereich der Datenanalyse ermöglicht.

Im Folgenden wird zunächst ein Beispiel einer typischen Problemstellung im Physikpraktikum präsentiert. Dieses soll einen Überblick über die Anforderungen an eine Datenanalysesoftware für das Physikpraktikum verschaffen.

Diese Einleitung soll weiterhin durch die Präsentation einiger bereits bestehender Ansätze für Datenanalysesoftware im Physikpraktikum die Entwicklung einer neuen Softwarelösung motivieren und zu weiterführenden Gedanken in dieser Hinsicht verhelfen.

1.1. Ein elementares Beispiel – Spannungsabfall am Widerstand

Die Aufnahme einer Strom-Spannungskurve an einem elementaren elektrischen Bauteil, z.B. an einem ohmschen Widerstand, ist für das physikalische Anfängerpraktikum ein

typisches Beispiel für ein Messverfahren, das oft auftritt. In so einem Versuch wird mithilfe einer regelbaren Spannungsquelle bei verschiedenen Spannungswerten der Strom durch den Widerstand gemessen. Die aufgenommenen Messpunkte (I_i, U_i) liegen bei einem ohmschen Widerstand nach aller Erwartung auf einer Ursprungsgeraden, also wird bei der Auswertung von einem linearen Modell für die Daten ausgegangen:

$$U(I) = R\,I\,.$$

Der einzige zu bestimmende Modellparameter ist R, der Widerstand des Bauteils.

Die reine Messung und Protokollierung der Messwerte reicht jedoch für die Auswertung nicht aus, sondern es muss generell bei jedem Experiment mit Abweichungen der einzelnen Messpunkte von den tatsächlichen Erwartungen gerechnet werden. Jede Messung ist mit einer gewissen Unsicherheit der Messwerte U_i und I_i verbunden, welche von verschiedenen Quellen herrührt.

Es kann sich dabei zum Beispiel um Ablesefehler oder durch Rauschen hervorgerufene Unsicherheiten handeln. Diese sind Messfehler zufälliger Natur, auch *statistische Fehler* genannt. Darüber hinaus sind manche Messfehler jedoch *systematisch* bedingt, etwa aufgrund einer falsch kalibrierten Messapparatur. Diese Fehlerart zeichnet sich durch eine Korrelation der Messfehler aus, deren Ursache ist, dass typischerweise dieselben Messgeräte für die Aufnahme einer Messreihe verwendet werden. Kalibrierungsfehler dieser Geräte wirken sich auf alle Messpunkte in gleicher Weise aus, was eine Korrelation des systematischen Fehleranteils bewirkt.

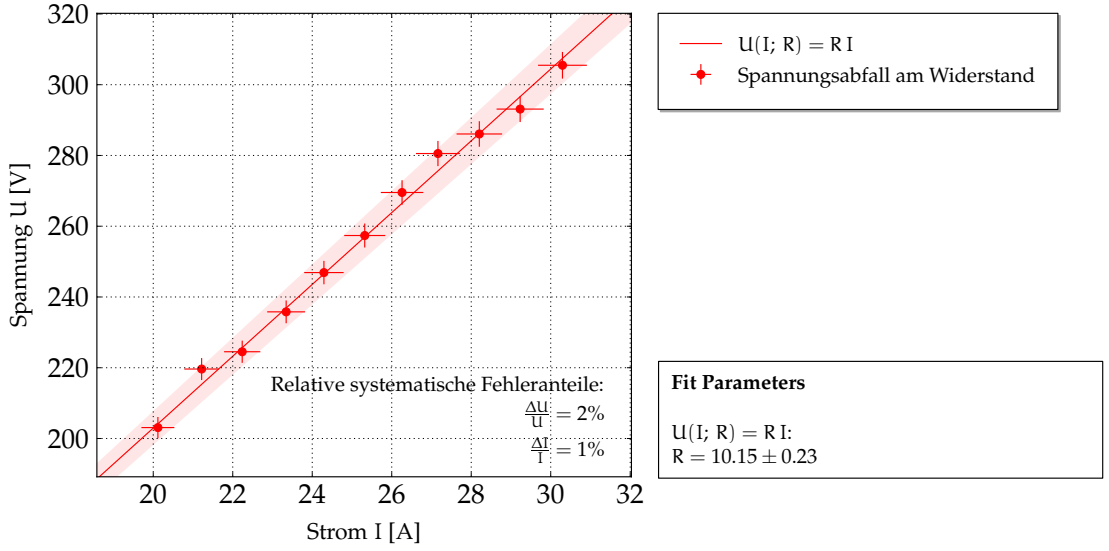


Abbildung 1.1.: Anpassung einer Ursprungsgeraden an Strom-/Spannungsmesspunkten, um den Modellparameter (den ohmschen Widerstand R) zu ermitteln. Die eingezeichneten Fehlerbalken repräsentieren die Gesamtfehler der Messpunkte. Der systematische, vollständig korrelierte Anteil des Gesamtfehlers wird gesondert durch einen Kommentar ausgewiesen. Die Daten wurden gemäß ihrer erwarteten Statistik generiert.

Schon bei diesem einfachen Beispiel einer Messung werden minimale Anforderungen an die Auswertesoftware klar. Zum einen ist der korrekte Umgang mit Fehlern verschiedener Art eine Voraussetzung. Dies läuft insbesondere auf die Eingabemöglichkeit

von statistischen und systematischen Fehlern hinaus, wobei letzteres auch eine korrekte Verarbeitung von Fehlerkorrelationen impliziert. Nicht weniger wichtig ist die Berücksichtigung von Fehlern in beide Achsenrichtungen, denn typischerweise sind beide Messgrößen, deren Zusammenhang man untersucht, fehlerbehaftet.

Bereits an diesen Forderungen, die selbst aus der elementarsten Praktikumsaufgabe hervorgehen, scheitert ein Großteil der gebräuchlichen Programme für grundlegende Datenanalyse. Dies motiviert die Entwicklung eines Datenanalysewerkzeugs, das sowohl über die oben genannte Funktionalität verfügt, als auch auf das Physikpraktikum ausgerichtet ist.

1.2. Bisherige Ansätze

Für die Datenanalyse im Praktikum wird gebräuchlicher Weise eine Reihe von Analysewerkzeugen verwendet, welche sich nicht nur bezüglich ihrer angebotenen Funktionalität, sondern auch in ihrer Flexibilität und Zugänglichkeit, unterscheiden.

Der Entwurf der meisten Softwarepakete ist darauf ausgerichtet, einen möglichst breiten Anwendungsspektrum abzudecken. Dabei wird von einer Personalisierung der Fit-Routine durch den Anwender in den meisten Fällen abgesehen, oder dies wird nur beschränkt angeboten. Einen alternativen Ansatz machen Pakete, bei denen die Steuerung ausschließlich über das Erstellen von eigenen Programmen erfolgt. Dies gewährt dem Anwender zwar mehr Flexibilität, fordert aber zugleich fortgeschrittene Kenntnisse. Erfahrungsgemäß liegen diese in den meisten Fällen im Physikpraktikum nicht vor.

Eine kurze Präsentation der gängigen Werkzeuge, die im Physikpraktikum für die Datenanalyse eingesetzt werden, wird im Kapitel 3/S. 13 diskutiert. Werkzeuge wie `gnuplot` (s. Abschn. 3.1.1/S. 14) oder *Origin* (s. Abschn. 3.1.2/S. 14) zeichnen sich durch verschiedene Funktionen aus, sind aber wegen entscheidenden Funktionalitätslücken keine geeigneten Auswertesoftware im Physikpraktikum.

Festzustellen ist auch, dass sonstige, fortgeschrittene Softwarepakete (z.B. *ROOT*, s. Abschn. 3.1.3/S. 16), obwohl sie über solche Funktionalität verfügen, für das Physikpraktikum aufgrund des langsamen Lernzuwachses und der Vorbedingung, eine Programmiersprache wie C++ zu beherrschen, ebenso ungeeignet sind.

Um den Umgang mit diesem Programmpaket Studierenden im Physikpraktikum dennoch zugänglich zu machen, wurde mit *RooFiLab* (s. Abschn. 3.1.4/S. 17) in einer früheren Arbeit ein erster Ansatz gemacht. Hierbei handelt es sich um eine graphische Schnittstelle zu *ROOT*, welche auf die Anpassung von Modellen an Datensätze ausgerichtet ist.

Die von *ROOT* angebotene Anpassungs- und Darstellungsfunktionalität wird damit durch eine vereinfachte, interaktive Steuerung den Studierenden zur Verfügung gestellt. Dies erlaubt eine schnelle und akkurate Bewältigung vieler im Physikpraktikum auftretenden Probleme.

1.3. Weiterführung: Datenanalyse in *Python* mit `kafe`

Die *RooFiLab*-Schnittstelle bleibt jedoch von einem in der Datenanalyse wichtigen Aspekt fern. Aufgrund der Vielfalt von Auswertungsmethoden, die diesen Bereich der Physik

auszeichnet, ist die genaue Dokumentation der angewandten Verfahren von ausschlaggebender Bedeutung. Eng damit verbunden ist auch die Reproduzierbarkeit, welche bei mangelnder Dokumentation des Analysevorgangs beeinträchtigt wird.

Aus diesem Grund ist Datenanalyse von einer selbst erbrachten Programmierleistung nicht zu trennen. Selbst bei einer Einführung in die Datenanalyse kann somit nicht davon abgesehen werden, wenigstens Grundkenntnisse des Programmierens zu vermitteln. Dadurch erfolgt eine nähere Auseinandersetzung der Studierenden mit den relevanten Verfahren und ihr Verständnis wird durch selbstständige Anwendung vertieft.

Diesem Sachverhalt entsprechend ist es für den Entwurf eines im Physikpraktikum eingesetzten Analysepakets angemessen, eine Ablösung von der graphischen Benutzeroberfläche anzustreben und dafür das Erlernen von skriptorientierter Auswertung von Messdaten zu fördern.

Durch die Erstellung von Fit-Verfahren in Form eines ausführbaren Programms besteht die Möglichkeit der genauen Anpassung der Analyseroutine an das jeweils vorliegende Problem. Zugleich wird das Verfahren durch den Code selber dokumentiert und kann deswegen auch bequem durch ein erneutes Ausführen des Skripts reproduziert werden.

Durch die Wahl von *Python* als Skriptsprache wird der Anwender von den komplizierten programmiertechnischen Aspekten, die etwa für die Programmerstellung in C++ erforderlich sind, ferngehalten. Die Einarbeitungszeit wird durch die einfache, intuitive Syntax von *Python* im Vergleich zu *ROOT*/C++ erheblich verkürzt.

Für die tatsächliche Umsetzung der Analyse ist in *Python* im Rahmen dieser Arbeit mit dem *Karslruhe Fit Environment* (kurz `kafe`) ein weiterer Ansatz für die Bereitstellung von Datenanalysesoftware im Physikpraktikum gemacht worden.

Das *Python*-Paket `kafe` wurde als einheitliche Softwarelösung für die Auswertung von im Praktikum typischen Analyseaufgaben konzipiert. Es bietet Funktionen für Modellanpassung an Messdaten und Hypothesentests, sowie für die graphische Darstellung der Ergebnisse an.

1.4. Überblick

In den folgenden Kapiteln wird eine Auffrischung der Anpassungsverfahren zugrundeliegenden Theorie vorgenommen (Kap. 2/S. 5). Dabei wird enger auf die von `kafe` eingesetzten theoretischen Grundlagen eingegangen.

Um einen Überblick über die Software, die typischerweise von Studenten im Praktikum eingesetzt wird, zu geben, werden zudem gängige Datenanalysewerkzeuge erwähnt und ihre Funktionalität kurz beschrieben (Kap. 3/S. 13). Im Anschluss wird näher auf Aufbau und Bedienung von `kafe`, sowie auf die Implementierung des Fit-Vorgangs eingegangen (Kap. 4/S. 19). Die technische Ausführung wird im Kapitel 5/S. 29 diskutiert.

Ein Ausblick hinsichtlich der Weiterentwicklung und des Einsatzes von `kafe` im Anfängerpraktikum wird im Kapitel 6/S. 33 gegeben. Des Weiteren steht eine Dokumentation der *API* im Anhang zur Verfügung.

2. Theoretische Betrachtungen

Die Auswertung experimenteller Daten erfolgt anhand eines Modells, welches durch einen oder mehr freie Parameter bestimmt ist. Für eine sinnvolle Analyse muss die Übereinstimmung dieses Modells mit den Beobachtungen aufgrund seiner Parameter quantitativ abgeschätzt und die optimalen Werte der Parameter bestimmt werden. Das Modell wird also dazu den Daten angepasst (engl. *fitted*).

Konkret geht es im Physikpraktikum um die Aufnahme von Datenpunkten. Dies beschränkt sich in der Regel auf den Fall einer einzigen unabhängigen Variablen, folglich wird hier nur dieser Fall betrachtet. Ein Messereignis entspricht somit der Aufnahme eines Datenpunkts (x_i, y_i). Ferner wird noch das Modell, welches zur Auswertung der Daten betrachtet werden soll, üblicherweise in Form einer Fit-Funktion $f(x)$ vorgegeben.

Um anzudeuten, dass diese Fit-Funktion noch von einem Parametersatz $\mathbf{p}$ abhängt, wird diese mit $f(x|\mathbf{p})$ notiert. Die Größe des Parametersatzes wird im Folgenden mit K, die des Datensatzes mit N bezeichnet.[1]

Die Anpassung der Fit-Funktion an den Messdaten impliziert vorerst die Definition eines mathematischen Abstandsbegriffes zwischen Beobachtung und Theorie, also zwischen den Messdaten (x_i, y_i) und dem Modell $f(x|\mathbf{p})$, in Abhängigkeit von seinen Parametern $\mathbf{p}$. Um eine möglichst gute Anpassung an den Daten zu erhalten muss dieser Abstand folglich minimiert werden.

Eine geeignete Wahl eines Abstandsbegriffs ist im allgemeinen von der Verteilungsart der Messgrößen abhängig. Im Folgenden wird ein Fit-Verfahren, welches einen natürlich aus der Verteilung der Messwerte hervorgehenden Abstandsbegriff einsetzt, beschrieben.

2.1. Methode der kleinsten Quadrate

Da Messgrößen statistischen Schwankungen unterliegen, werden diese als Zufallsvariablen aufgefasst, deren Wahrscheinlichkeitsverteilung zunächst unbekannt ist. Eine Messung entspricht dann einer einelementigen Stichprobe der Zufallsvariable.

[1] Unter „statistischen Fehlern“ versteht man vollständig unkorrelierte Unsicherheiten der Messdaten. Diese werden mit σ_x und σ_y bezeichnet. Für eine Liste der in dieser Bachelorarbeit verwendete Notation, siehe Anhang.

Oft wird im Praktikum die Annahme gemacht, dass die Messdaten einer (normierten) Gaußverteilung unterliegen (Abb. 2.1/S. 6):

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{2.1}$$

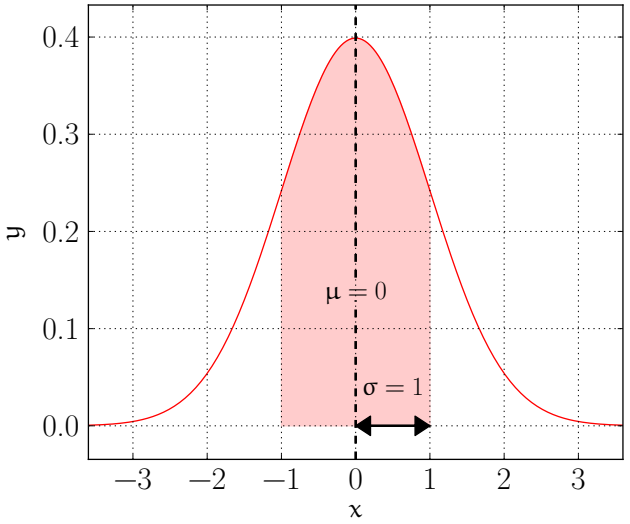


Abbildung 2.1.: Wahrscheinlichkeitsverteilungsfunktion der Standard-Normalverteilung. Die Standard-Normalverteilung ist eine Gauß-Verteilung mit $\mu = 0$ und $\sigma = 1$. Gezeigt wird auch der Bereich $[-\sigma, \sigma]$. Die Wahrscheinlichkeit, dass bei einer Stichprobe einer gaußverteilten Größe, sich ein Wert innerhalb dieses Bereichs ergibt, ist konstant und liegt bei $\approx 68.27\%$. Man spricht von einem 1σ-Bereich.

Dabei wird der Mittelwert μ der Verteilung mit dem sogenannten „wahren Wert“ der gemessenen Größe identifiziert. Für den i-ten Datenpunkt einer Messreihe entspricht dieser Wert der Modellvorhersage $f(x_i \,|\, \mathbf{p})$. Ferner wird die Größe σ, die ein Maß für die Breite der Verteilung ist, als Unsicherheit des Messwerts aufgefasst und kurz „Messfehler“ genannt.

Die Annahme von gaußverteilten Messwerten ist eine besonders im Praktikum oftangetroffene Vereinfachung. Diese basiert auf der Tatsache, dass diese von vielen Rauschquellen beeinflusst werden und somit laut dem zentralen Grenzwertsatz annähernd gaußverteilt sind.[Bar89, S. 49]

2.1.1. Bestimmung der Parameter der Bestanpassung

Man betrachte für alle Datenpunkte die Differenz $y_i - f(x_i \,|\, \mathbf{p})$ zwischen den Beobachtungen und den Modellvorhersagen. Diese Größen, genannt *Residuen*, unterliegen einer Gauß-Verteilung um $\mu = 0$. Normiert man zusätzlich diese Differenz auf den Messfehler σ_i, erhält man Standard-Normalverteilte Größen $[y_i - f(x_i \,|\, \mathbf{p})]/\sigma_i$.

Nach dieser Überlegung kann nun die Wahrscheinlichkeit, bei einem korrekten Modell, das i-te Messereignis zu beobachten, aufgestellt werden:

$$\mathcal{P}(x_i, y_i) = \frac{1}{\sqrt{2\pi{\sigma_i}^2}} \exp\left(-\frac{1}{2}\left(\frac{y_i - f(x_i \,|\, \mathbf{p})}{\sigma_i}\right)^2\right)$$

Dann ist die Wahrscheinlichkeit, alle N Messereignisse einer Messreihe zu beobachten gegeben durch:

$$\mathcal{P}(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} \mathcal{P}(x_i, y_i) = \left(\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi{\sigma_i}^2}}\right) \exp\left(-\frac{1}{2}\sum_{i=1}^{N}\left(\frac{y_i - f(x_i \,|\, \mathbf{p})}{\sigma_i}\right)^2\right)$$

Der Parametersatz $\mathbf{p}$, für den dieser Ausdruck maximal wird würde also der besten Anpassung entsprechen. Dies ist äquivalent zur Minimierung der Summe innerhalb der Exponentialfunktion, also der mit den inversen Varianzen der Messpunkte gewichteten Summe der Residuenquadrate (engl. *weighted sum of squared residuals*). In Kurzfassung wird dieses Verfahren als *Methode der kleinsten Quadrate* bezeichnet.

Der zu minimierende Ausdruck $F(\mathbf{p})$ ist also eine Summe von Quadraten standardnormalverteilter Variablen und unterliegt somit einer χ^2-Verteilung. Aus diesem Grund wird diese Größe auch mit χ^2 bezeichnet:

$$\chi^2 = F(\mathbf{p}) = \sum_{i=1}^{N}\left(\frac{y_i - f(x_i \,|\, \mathbf{p})}{\sigma_i}\right)^2 \tag{2.2}$$

Die Anzahl der Freiheitsgrade NDF (engl. *number of degrees of freedom*) dieser χ^2-Verteilung wird durch $\text{NDF} = N - K$ gegeben. Dadurch, dass die Modellvorhersage $f(x_i \,|\, \mathbf{p})$ für jeden Punkt nicht eine feste Zahl ist, sondern selber von p_1—p_K abhängt, müssen die K Modellfreiheitsgrade also von der Größe N des Datensatzes abgezogen werden.

Der Erwartungswert $\langle\chi^2\rangle = \langle F(\mathbf{p})\rangle$ liegt auch bei $\text{NDF} = N - K$. Das Verhalten von $\langle F(\mathbf{p})\rangle$ bei einer 1σ-Abweichung der y-Werte ($y_i \to y_i \pm \sigma_i$) lässt sich leicht prüfen:

$$\langle\chi^2(y_i \pm \sigma_i)\rangle = \sum_{i=1}^{N}\left\langle\left(\frac{y_i - f(x_i \,|\, \mathbf{p})}{\sigma_i} \pm 1\right)^2\right\rangle = \langle\chi^2(y_i)\rangle + 1 \tag{2.3}$$

Erwartet wird also, dass eine 1σ-Abweichung der y_i eine Änderung $\Delta\Big(F(\mathbf{p})\Big) = 1$ hervorbringt.

2.1.2. Matrixformulierung

Eine kompaktere Schreibweise dieser Größe ergibt sich unter Einführung der sogenannten *Varianz-Kovarianzmatrix* (oft nur *Kovarianzmatrix*). Diese Matrix ist für diesen noch einfachen Fall unkorrelierter Fehler eine Diagonalmatrix (für die allgemeine Form mit Fehlerkorrelationen, s. 2.1.4/S. 8). Die Einträge der Matrix entsprechen der Varianzen jedes Messereignisses:

$$\mathbf{V} = \begin{pmatrix} {\sigma_1}^2 & & & \\ & {\sigma_2}^2 & & \\ & & \ddots & \\ & & & {\sigma_N}^2 \end{pmatrix} \tag{2.4}$$

Mithilfe dieser Matrix und des Residuenvektors $\boldsymbol{\lambda}(\mathbf{p}) = \mathbf{y} - \mathbf{f}(\mathbf{x}\,|\,\mathbf{p})$ lässt sich (2.2/S. 7) umschreiben:

$$\chi^2 = F(\mathbf{p}) = \boldsymbol{\lambda}(\mathbf{p})^T \mathbf{V}^{-1} \boldsymbol{\lambda}(\mathbf{p}) \tag{2.5}$$

2.1.3. Bestimmung der Fehler der Parameter

Als direkte Konsequenz der Unsicherheit der Messdaten, sind auch die Fit-Parameter fehlerbehaftet. Nach einer erfolgreichen Ermittlung des Minimums $\mathbf{p}_{\min}$ von $F(\mathbf{p})$ ist es nun nötig, die Parameterfehler zu bestimmen, welche ein Maß für die Anfälligkeit des zu minimierenden Ausdrucks $F(\mathbf{p})$ auf eine Änderung der Parameter $\mathbf{p}$ sind. Um diese akkurat bestimmen zu können wird vorausgesetzt, dass $F(\mathbf{p})$ in der Umgebung von $\mathbf{p}_{\min}$ „oft genug“ differenzierbar ist. Die Existenz des Gradients $\boldsymbol{\nabla} F(\mathbf{p})$ und der *Hesse-Matrix*

$$\mathbf{H}(\mathbf{p}_{\min}) = \left(\begin{array}{cccc} \frac{\partial^2 F}{\partial {p_1}^2} & \frac{\partial^2 F}{\partial p_1 \partial p_2} & \cdots & \frac{\partial^2 F}{\partial p_1 \partial p_K} \\ \frac{\partial^2 F}{\partial p_2 \partial p_1} & \frac{\partial^2 F}{\partial {p_2}^2} & \cdots & \frac{\partial^2 F}{\partial p_2 \partial p_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 F}{\partial p_K \partial p_1} & \frac{\partial^2 F}{\partial p_K \partial p_2} & \cdots & \frac{\partial^2 F}{\partial {p_K}^2} \end{array} \right) \Bigg|_{\mathbf{p}_{\min}}$$

können also somit vorausgesetzt werden und $F(\mathbf{p})$ kann durch eine Taylor-Reihe zweiter Ordnung approximiert werden:

$$F(\mathbf{p}_{\min} + \boldsymbol{\Delta p}) \approx F(\mathbf{p}_{\min}) + \boldsymbol{\nabla} F(\mathbf{p}_{\min})^T \boldsymbol{\Delta p} + \frac{1}{2} \boldsymbol{\Delta p}^T \mathbf{H}(\mathbf{p}_{\min}) \boldsymbol{\Delta p}$$

Da im Funktionsminimum der Gradient $\boldsymbol{\nabla} F(\mathbf{p}_{\min})$ Null ist, gibt die quadratische Form $\boldsymbol{\Delta p}^T \mathbf{H}(\mathbf{p}_{\min}) \boldsymbol{\Delta p}$ allein Auskunft über das Verhalten von $F(\mathbf{p}_{\min})$ im Bereich der Minimums.

Diese quadratische Näherung erlaubt es, sogenannte *parabolische* Fehler der Parameter zu ermitteln. Diese werden oft in Form einer Kovarianzmatrix der Parameter angegeben, welche der inversen Hesse-Matrix bis auf einem Vorfaktor entspricht:[Qua13]

$$\mathbf{V}_p \propto \mathbf{H}^{-1}(\mathbf{p}_{\min})$$

Die Diagonaleinträge dieser Matrix entsprechen wiederum den Varianzen der Parameter, während die Nebendiagonaleinträge die Korrelation der Parameter angeben.

2.1.4. Behandlung korrelierter Fehler

Von einer Korrelation der Messungen einer Messreihe kann im allgemeinen nicht abgesehen werden. Solche Korrelationen kommen z.B. durch systematisch bedingte Unsicherheiten (Kalibrierungsfehler, *etc.*) der Messdaten zustande. Sind die systematischen Fehler gegenüber den statistischen vernachlässigbar klein, kann von Korrelationen abgesehen werden. Dies ist jedoch nicht der Regelfall und besonders für die im Physikpraktikum eingesetzte Apparatur sind hohe systematische Fehler vorauszusetzen.

Eine korrekte Behandlung solcher Fehler lässt sich durch die Ergänzung der diagonalen Kovarianzmatrix (2.4/S. 7) zu einer symmetrischen Matrix erreichen.[BZ10, S. 88–90] Das

Element V_{ij} der vollständigen Kovarianzmatrix entspricht somit der Kovarianz des i-ten und j-ten y-Messwerts. Die vollständige Matrix lautet:

$$\mathbf{V} = \begin{pmatrix} {\sigma_1}^2 & \mathrm{Cov}_{y,12} & \cdots & \mathrm{Cov}_{y,1N} \\ \mathrm{Cov}_{y,12} & {\sigma_2}^2 & \cdots & \mathrm{Cov}_{y,2N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}_{y,1N} & \mathrm{Cov}_{y,2N} & \cdots & {\sigma_N}^2 \end{pmatrix} \tag{2.6}$$

Korrelationskoeffizienten und -matrizen

Obwohl Kovarianzmatrizen für numerische Berechnungen geeignet sind, ist es für die Einschätzung des Korrelationsgrads der Messfehler oft angemessener, anstatt Kovarianzen Cov_{ij} Korrelationskoeffizienten Cor_{ij} anzugeben. Diese bringen das Verhältnis zwischen den zufälligen Fehlern und den korrelierten Fehlern zum Ausdruck:

$$\mathrm{Cor}_{ij} = \frac{\mathrm{Cov}_{ij}}{\sigma_i \sigma_j}$$

Die Unsicherheiten einer gesamten Messreihe sind somit durch Angabe eines Fehlervektors und einer Matrix, welche aus den Korrelationskoeffizienten besteht (die sogenannte Korrelationsmatrix), vollständig charakterisiert.

2.1.5. Behandlung von Fehlern in Richtung der Abszisse

Bisher wurden nur Unsicherheiten der y-Messwerte betrachtet. Da aber typischerweise auch die Messung der x-Werte fehlerbehaftet ist, muss das Verfahren für deren Behandlung erweitert werden.

Solche „x-Fehler“ lassen sich für differenzierbare Modellfunktionen $f(x\,|\,\mathbf{p})$ bequem durch Iteration behandeln.[Qua13] Bei einem solchen iterativen Verfahren wird zunächst eine Anpassung ausschließlich mit y-Fehlern durchgeführt. Anschließend werden die „x-Fehler“ auf die ursprünglichen y-Fehler „projiziert“.

Dafür wird zunächst für jedes x_i die Modellfunktion durch die Tangente in dem jeweiligen Punkt angenähert. Die Steigung der Tangente entspricht der Ableitung $\left.\frac{\partial f(x\,|\,\mathbf{p})}{\partial x}\right|_{x_i}$ der Modellfunktion. Die totalen Fehler werden dann mittels Fehlerfortpflanzung ermittelt:

$$\sigma_{i,\mathrm{tot}} = \sqrt{{\sigma_{i,y}}^2 + \left[\left.\frac{\partial f(x\,|\,\mathbf{p})}{\partial x}\right|_{x_i}\right]^2 {\sigma_{i,x}}^2}$$

In der Matrizenformulierung existieren in einem solchen Fall Kovarianzmatrizen $\mathbf{V}_x$ und $\mathbf{V}_y$ für beide Koordinatenachsen. Die zur Berechnung von $F(\mathbf{p})$ verwendete Matrix $\mathbf{V}$ wird für die erste Anpassung auf $\mathbf{V}_y$ gesetzt. Für jede anschließende Iteration wird $\mathbf{V}$ neu bestimmt:

$$\mathbf{V}_{ij} = \mathbf{V}_{y,ij} + \mathbf{V}_{x,ij} \left.\frac{\partial f(x\,|\,\mathbf{p})}{\partial x}\right|_{x_i} \left.\frac{\partial f(x\,|\,\mathbf{p})}{\partial x}\right|_{x_j}$$

2.2. Güte der Anpassung und Hypothesentest

Die Güte eines Fits gibt darüber Auskunft, wie gut das vorgegebene Modell zu den Messdaten passt. Es ist üblich hier eine ja/nein-Entscheidung zu treffen und die Hypothese, dass das Modell für die Beschreibung der Daten geeignet ist, entweder zu akzeptieren oder zu verwerfen.

Ferner existieren verschiedene Gütekriterien für einen Fit. Hier werden zwei gängige Indikatoren der Fit-Güte eingeführt und mögliche Schwellen für Hypothesentests diskutiert.

2.2.1. χ^2 pro Freiheitsgrad

Die zu minimierende Größe $F(\mathbf{p})$ unterliegt einer χ^2-Verteilung. Diese Verteilung hat einen einzigen Parameter, nämlich die Anzahl von Freiheitsgraden (NDF), welche sich aus der Differenz $N - K$ ergibt.

Der Mittelwert der Verteilung von χ^2/NDF liegt bei 1. Zu erwarten ist also, dass bei korrektem Modell und akkurat geschätzten Messfehlern, der Wert von χ^2/NDF für jede Anpassung sich in der Gegend von 1 befindet. Weicht dieser Wert erheblich von 1 ab, so kann man u. U. die Anpassung verwerfen.

Ein Wert $\chi^2/\text{NDF} \gg 1$ spricht für eine schlechte Übereinstimmung des Modells mit den Daten. Der entgegengesetzte Fall $\chi^2/\text{NDF} \ll 1$ signalisiert üblicherweise eine zu großzügige Schätzung der Messfehler.

2.2.2. Konfidenzniveau

Aussagekräftiger als χ^2 pro Freiheitsgrad ist ein Hypothesentest, der auf die Wahrscheinlichkeit, einen höheren χ^2-Wert (also eine schlechtere Anpassung) als der vorliegende zu erhalten, basiert. Diese erhält man durch Integration der χ^2-Funktion für die entsprechende Anzahl von Freiheitsgraden von dem Wert am Minimum bis ins Unendliche:

$$\chi^2{}_{\text{prob}} = \int_{\chi^2{}_{\text{min}}}^{\infty} \chi^2(t, \text{NDF})\, dt \tag{2.7}$$

Liegt diese Wahrscheinlichkeit unter einer gewissen Konfidenzgrenze (engl. *confidence level*, CL), so kann die Anpassung verworfen werden. Eine übliche Konfidenzgrenze liegt bei $CL = 5\%$.

Zu Bemerken ist noch, dass die Möglichkeit besteht, eine sehr hohe Wahrscheinlichkeit (2.7/S. 10) zu beobachten. Überschreitet diese $1 - CL$, so ist die Anpassung „verdächtig gut“. Nichtsdestoweniger spricht dies nicht für die Verwerfung der Anpassung,[BZ10, S.255] sondern es ist eher ein Indikator dafür, dass die Unsicherheiten der Datenpunkte zu hoch geschätzt wurden, wie es für einen zu niedrigen χ^2/NDF-Wert der Fall ist.

2.3. Konfidenzband

Die Ermittlung der Modellparameter legt die Fit-Funktion eindeutig fest. Jedoch entsteht durch die Unsicherheiten der Parameterwerte ein gewisser „Spielraum“ im Verlauf der Funktion. Eine 1σ-Abweichung der Fit-Funktion lässt sich durch die Fortpflanzung der Messfehler auf die Parameterfehler quantifizieren.[Str10]

Für jeden Wert der unabhängigen Variablen im Modell wird also dadurch ein Konfidenzintervall ermittelt. Besonders als visueller Hinweis auf die Parameterfehler ist für die graphische Darstellung dieser Parameterfehler die Zusammenführung dieser Konfidenzintervalle in ein sogenanntes *Konfidenzband* um die Funktion üblich.

Der 1σ-Bereich um die Fit-Funktion wird also durch zwei weitere Funktionen abgegrenzt, die symmetrisch um $f(x\,|\,\mathbf{p})$ angeordnet sind. Die Eingrenzungsfunktionen sind gegeben durch:

$$f_{\pm 1\sigma}(x) = f(x\,|\,\mathbf{p}) \pm \sum_{i,j=1}^{K} \frac{\partial f(x\,|\,\mathbf{p})}{\partial p_i}\, \mathbf{V}_{p,ij}\, \frac{\partial f(x\,|\,\mathbf{p})}{\partial p_j} \tag{2.8}$$

3. Datenanalyse im Physikpraktikum

Die zentralen Aufgaben in den physikalischen Praktika können in drei Etappen eingeteilt werden. Die *Datencharakterisierung* verlangt zunächst die Festlegung eines Modells, dessen Parameter bestimmt werden sollen. Ferner werden mögliche Fehlerquellen und Fehlerarten identifiziert und auf Korrelationsgrad oder Verteilungsart untersucht. Die theoretischen Betrachtungen, sowie das Modell, welches aus diesen hervorgeht, liegen im Praktikum typischerweise bereits vor. Die Aufgabe der Studierenden besteht nur darin, diese Überlegungen nachzuvollziehen.

Die Etappe der *Datenaufnahme* umfasst den Messprozess selber. Zusätzlich zur Durchführung von Messungen und der Erstellung von Datensätzen ist es notwendig, die Größe der Messfehler abzuschätzen. Die aufgenommenen Datensätze werden in der Etappe der *Datenanalyse* zur Modellparameterbestimmung genutzt.

Je nach Praktikumsversuch werden für diese Auswertung verschiedene Softwarepakete genutzt. Frei zugängliche Datenanalysepakete unterscheiden sich jedoch dramatisch in Funktionalität, was dazu führt, das im Rahmen einer einzigen Auswertung viele verschiedene Programme zur Anwendung kommen können. Dies soll durch den Entwurf eines Programmpakets, welches speziell auf das Praktikum zugeschnitten ist, vermieden werden.

In diesem Abschnitt werden einige gängige Datenanalysewerkzeuge vorgestellt und die von ihnen angebotene Funktionalität beschrieben.

3.1. Übersicht der typischen Datenanalysewerkzeuge

Die Softwareauswahl, welche in dieser Arbeit als Bezugssystem verwendet wurde, ergab sich zum Teil aus der eigenen Erfahrung des Autors im Physikpraktikum. Für die Organisation der Daten wurde hier zu gängigen Tabellenkalkulationsprogrammen gegriffen, und für die graphische Darstellung zu `gnuplot`.[WK12]

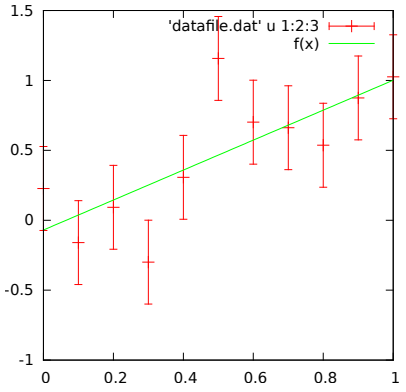
Eine eher auf die Analyse als auf die Darstellung gerichtete Lösung, die eine tabellenkalkulation-ähnliche Methode der Dateneingabe vorsieht, ist *Origin*.[ori] Die von jedem der beiden Softwarepaketen angebotenen Funktionen werden im Folgenden kurz dargelegt.

Obwohl kein standardmäßig im Praktikum eingesetztes Werkzeug, wird auch *ROOT*[BR97] und ein Bruchteil seiner Funktionspalette hier auch kurz vorgestellt.

3.1.1. gnuplot

Das Plotprogramm `gnuplot` ermöglicht es, relativ schnell Messdaten graphisch darzustellen. Darüber hinaus besteht die Möglichkeit, eine Funktion an die Daten anzupassen und die Parameter der besten Anpassung auszulesen.

Die Übergabe von Daten an `gnuplot` wird durch die Erstellung einer Skriptdatei, in welcher Angaben zur Fit- und Zeichenprozedur gemacht werden, erreicht. Das Programm liefert neben der graphischen Ausgabe auch eine textuelle, aus der z.B. die Fit-Parameter oder Daten zur Güte des Fits herausgelesen werden können. Ein Beispiel des Eingabeformats und der erzeugten Ausgabe wird in Abb. 3.1/S. 14 gezeigt.




```
#/usr/bin/gnuplot

set terminal pdf color enhanced
set output 'plot.pdf'
set size square

f(x) = a*(x-x0)
a=3;x0=0

fit f(x) 'datafile.dat' using 1:2:3 via a, x0
plot 'datafile.dat' u 1:2:3 w yerr, f(x)
```

Abbildung 3.1.: Ausgabe und Eingabeformat von `gnuplot` anhand eines einfachen Datensatzes. Das Beispiel umfasst die Anpassung einer Geraden an die Daten.

Wie es der Name schon verrät, handelt es sich jedoch bei `gnuplot` hauptsächlich um eine Software für graphische Darstellung, welche auf schnelle Graphikproduktion setzt. Wie in Abb. 3.1/S. 14 ersichtlich, lässt sich eine Graphik mit wenigen Eingabezeilen erstellen.

Die voreingestellten Optionen sind jedoch, etwa aufgrund der dünnen Linien oder des standardmäßig kleinen Texts der Beschriftung, für die Einbindung in Berichte oder Präsentationen ungeeignet. Die Stärken von `gnuplot` liegen vielmehr in der skript-basierten Herangehensweise (bei vorhandenen Skriptdatei gewährleistet dies die Reproduzierbarkeit des Fits) und in der relativ intuitiven Syntax. Für einfache Anpassungen bietet `gnuplot` beispielsweise den Befehl `fit`. Die Modellfunktion und die Startparameter der Anpassung werden direkt in der Eingabedatei spezifiziert.

Das von `gnuplot` bei einem solchen Fit implementierte Anpassungsverfahren[Cra98] basiert auf dem Levenberg-Marquardt-Algorithmus,[Lev44,Mar63] und erlaubt die Berücksichtigung von statistischen Messunsicherheiten der abhängigen Variablen (σ_y). Darauf beschränkt sich jedoch die Fit-Funktionalität von `gnuplot`: Messunsicherheiten in Richtung der Abszisse und korrelierte Messfehler lassen sich mit `gnuplot` nicht behandeln.

3.1.2. *Origin*

Das *GUI*-orientierte *Origin* (engl. *graphical user interface*) bietet eine viel breitere Funktionspalette als `gnuplot` und ist als integrierte Lösung für Datenorganisation, Analyse und Darstellung gedacht.

Im Gegensatz zu `gnuplot` erfolgt die Steuerung von *Origin* über eine graphische Benutzeroberfläche. Die Organisation der Daten in einer vertrauten Umgebung, die an gängige

Tabellenkalkulation-Programme erinnert, sowie die Bedienung per Mausklick machen die Software zugänglicher als andere. In *Origin* werden Messdaten und Messfehler in Tabellenform übergeben, und zwar ein Datenpunkt und die assoziierten Messfehler in x- und/oder y-Richtung pro Zeile.

Für die Modellanpassung stehen dem Benutzer mehrere Regressionsmöglichkeiten zur Auswahl. Für lineare, polynomielle oder sonstige nichtlineare Modelle bietet *Origin* ein Anpassungsverfahren, welches auf die Methode der kleinsten Quadrate (s. Abschn. 2.1/S. 5) basiert.

Zu bemerken ist jedoch, dass auch in *Origin* kein allgemeines Verfahren zur Behandlung von „x-Fehlern“ existiert: nur für den Sonderfall eines linearen Modells besteht die Möglichkeit der zusätzlichen Berücksichtigung dieser Unsicherheiten. Dies geschieht durch Erweiterung der üblichen χ^2-Funktion durch einen zweiten Term, welcher die x-Residuen enthält.[orh] Hierbei hängt die Gewichtung der Residuen in den Summen von den angegebenen Messfehlern ab. Es besteht jedoch keine Möglichkeit, Fehlerkorrelationen anzugeben.

Die Stärke von *Origin* liegt in der graphischen Benutzeroberfläche. Dadurch, dass jedoch die Anpassungen und Zeichnungen immer per Mausklick aufgerufen werden, besteht keine Möglichkeit, eine Anpassung vollständig zu einem späteren Zeitpunkt nachzuvollziehen und ggf. zu wiederholen.

Dies ist allgemein eine gravierende Limitation, die sich auf die Datenauswertung im Physikpraktikum nicht weniger stark auswirkt: kleine Korrekturen der Messdaten oder Messfehler, die eine Wiederholung des Fits voraussetzen, können also nur durch mühsame Wiederholung der Schritte auf der graphischen Oberfläche erbracht werden. Einen Eindruck der Komplexität dieses Vorgangs wird in Abb. 3.2/S. 15 angedeutet.

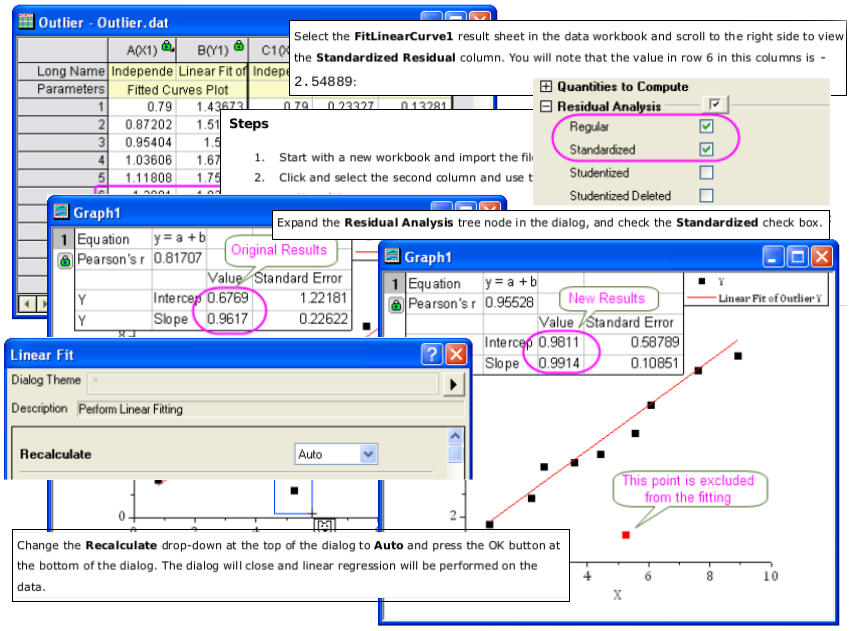


Abbildung 3.2.: Collage von Elementen der graphischen Benutzeroberfläche von *Origin*. Die Einzelgraphiken entstammen den *Origin*-Tutorials unter `http://www.originlab.com/pdfs/Tutorials.pdf`.

Insgesamt erweist sich *Origin* als beliebtes Werkzeug für Datenanalyse im Physikpraktikum, die jedoch aufgrund begrenzter Funktionalität und der Bedienung über die graphische Benutzeroberfläche kein vertieftes Verständnis von Datenanalyse verschafft. Ferner handelt es sich bei *Origin* um ein lizenzpflichtiges Programm. Da zu Lehrzwecken der Einsatz von freien Softwarepaketen zu empfehlen ist, stellt dies eine weitere Einschränkung von *Origin* dar.

3.1.3. *ROOT*

Bei dem am *CERN* entwickelten Programmpaket *ROOT* handelt es sich um ein generisches Datenanalyse-Framework, welches auch für die Analyse von experimentellen Daten aus teilchenphysikalischen Experimenten am *LHC*[1] verwendet wird.

Die von *ROOT* angebotenen Funktionen beschränken sich nicht auf die Modellanpassung an Daten, sondern sind für die Behandlung diverser wissenschaftlicher Problemstellungen einsetzbar. Die Software bietet auch die Erstellung von Graphiken und das Abspeichern von Ergebnissen in einem einheitlichen Datenformat.

Die Bedienung von *ROOT* erfolgt entweder über die Einbindung der *ROOT*-Bibliotheken in eigene C++-Programme, oder über die Erstellung von Makros in einer C++-ähnlichen, interpretierten Sprache.

Diese Makros enthalten alle notwendigen Informationen zum Programmablauf. Dies hat erstens den Vorteil, dass alle Anpassungen problemlos durch erneute Ausführung der Makros reproduzierbar sind. Zweitens erlaubt dies weitgehende Kontrolle über die Anpassungsroutine selber, so dass sämtliche Information über Datenpunkte, Fehler und Korrelationen an *ROOT* an einer zentralen Stelle weitergegeben werden können.

Ein Nachteil dieser Herangehensweise ist die Voraussetzung von guten C++-Kenntnissen, sowie die zusätzliche zeitintensive Einarbeitung in die *ROOT*-API (siehe dazu z.B. „Diving into *ROOT*“ [PQZ13]).

Minuit

Der ursprünglich unabhängig in FORTRAN geschriebene Funktionsminimierer *Minuit*,[JR75] ist in *ROOT* enthalten. *Minuit* bietet effiziente Methoden zur Auffindung von Minima einer Funktion mit mehreren Parametern und zur Berechnung der Parameterfehler.

Oft problematisch sind in der Modellanpassung die lokalen Minima der zu minimierenden Funktion, gegen welche Funktionsminimierer zu konvergieren neigen. Zum Teil lassen sich die globalen Minima durch geschickte Wahl der Startparameter einer Anpassung auffinden, im Fall einer komplexeren Abhängigkeit jedoch wird eine solche Schätzung nur schwer möglich. Die Suche nach globalen Minima wird in *Minuit* durch die Implementierung einer Monte-Carlo-Methode zur Funktionsminimierung und eines speziell auf die Auffindung globaler Minima ausgerichteten Algorithmus erleichtert.

Ein weiterer Vorteil von *Minuit* ist die Möglichkeit einer detaillierten Fehleranalyse mithilfe von *Minos*. Dieses Modul analysiert das Verhalten der zu minimierenden Funktion $F(\mathbf{p})$ in der Gegend des Minimums und Untersucht den Einfluss der Änderung einzelner Parameter auf den Funktionswert. Da oft die quadratische Näherung (s. Abschn. 2.1.3/S. 8), auf welche die Angabe symmetrischer (parabolischer) Parameterfehler basiert, generell keine gute Approximation von $F(\mathbf{p})$ ist, werden oft asymmetrische Fehler angegeben. Dafür stellt *Minos* die notwendige Funktionalität bereit.

[1] engl. *Large Hadron Collider* — Teilchenbeschleunigeranlage am *CERN*

3.1.4. *RooFiLab*

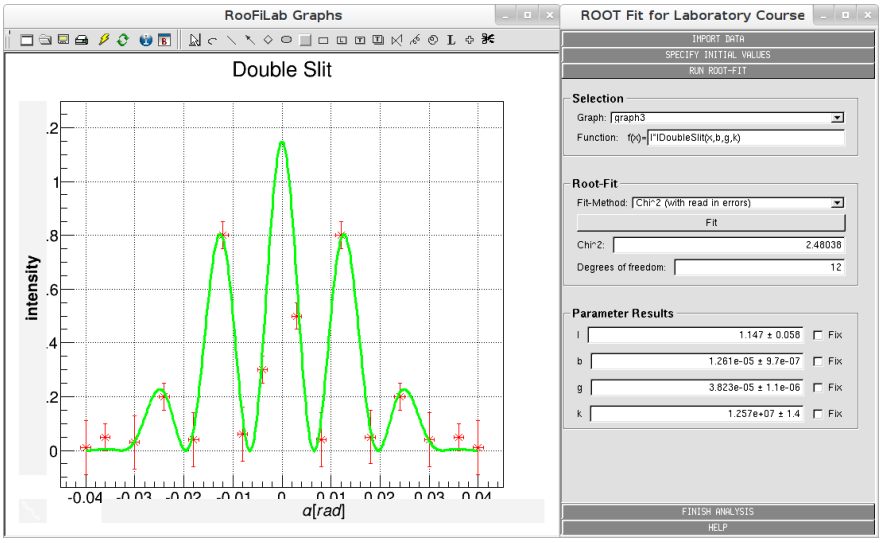


Abbildung 3.3.: Graphische Benutzeroberfläche von *RooFiLab*. Der Steuerbereich im rechten Fenster ist in sog. „Shutter“ gegliedert, die dem Anwender Funktionen zur Einlesung, Analyse und Darstellung der Daten bieten. Im linken Fenster werden die Daten und die angepasste Modellfunktion mithilfe von *ROOT*-Methoden dargestellt.

Für den Einsatz in Physikpraktika wurde am *Karlsruher Institut für Technologie* das Fit-Paket *RooFiLab*[Mü10] entwickelt. Dieses wurde ausgehend von den Bedürfnissen im Praktikum erstellt und ermöglicht eine korrekte Behandlung von Abszissenfehlern und Fehlerkorrelationen. Die Analyse der Daten und die graphische Darstellung der Ergebnisse werden mithilfe der dafür bestimmten *ROOT*-Klassen durchgeführt. Die Bedienung erfolgt über eine graphische Benutzeroberfläche, was den Umgang mit *ROOT* erleichtern soll, indem die dafür zu erbringende Programmierleistung durch eine graphische Schnittstelle überbrückt wird.

Anders als bei *Origin* wird bei *RooFiLab* jedoch die Reproduzierbarkeit gewährleistet, da die Messdaten, die Fehlerinformation und Anweisungen zum Fit-Algorithmus nicht im Rahmen des Programms selber eingegeben werden, sondern in einer oder mehreren Dateien organisiert werden, welchen dem Programm übergeben werden. Bedenklich ist jedoch der Einsatz von *RooFiLab* aus didaktischer Sicht erst dann, wenn das Aneignen von Programmierkenntnissen vermittelt werden soll. Hier wird durch die graphische Steuerung von dem selbstständigen Verfassen von Programmen abgesehen und abgewöhnt, was nicht der Standardpraxis in der Datenanalyse entspricht.

3.2. Zusammenfassung

Wie aus dem zuvor verschafften Überblick ersichtlich, kann keines der vorgestellten Softwarepakete die Anforderungen an eine Auswertesoftware für das Physikpraktikum erfüllen. Ein Überblick der interessanten Eigenschaften der Software wird in Tabelle 3.1/S. 18 gezeigt.

Das Hauptproblem besteht bei den einfacheren `gnuplot` und *Origin* in mangelnder Funktionalität, bei dem fortgeschrittenen *ROOT* eher in der Schwierigkeit der Bedienung und der mangelnden Zugänglichkeit. Diese Probleme werden zum Teil im Fit-

Paket für das Physikpraktikum *RooFiLab* aufgehoben. Die Bevorzugung einer graphischen Benutzeroberfläche steigert hier zwar die Zugänglichkeit, minimiert aber die Flexibilität der Software und zugleich ihren Nutzen als Basis für die Einführung in die rechnergestützte Analyse.

	Programm			
Eingenschaft	`gnuplot`	*Origin*	*ROOT*	*RooFiLab*
Zugänglichkeit	ja	ja	nein	ja
Transparenz des Fit-Algorithmus	nein	nein	zum Teil	zum Teil
Reproduzierbarkeit des Fit-Vorgangs	ja	nein	ja	ja
Flexibilität und Auswahlfreiheit	nein	zum Teil	ja	zum Teil
Skriptorientiertheit	ja	nein	ja	nein
Behandlung von korrelierten Fehlern	nein	nein	zum Teil	ja
Behandlung von „x-Fehlern“	nein	zum Teil	zum Teil	ja

Tabelle 3.1.: Vergleich von im Physikpraktikum häufig benutzter Auswertungssoftware.

Somit kann auf die Notwendigkeit geschlossen werden, ein Softwarepaket für die Einführung in die Datenanalyse bereitzustellen. Dieses soll die Bedürfnisse von Studierenden im Praktikum in ihrer Gesamtheit abdecken.

Um dieser Forderung entgegenzukommen wird im bereitgestellten Fit-Paket `kafe` auf alle in der Tabelle 3.1/S. 18 aufgetragenen Interessenbereiche eingegangen. Dabei wird die Kontrolle des Anwenders über den tatsächlichen Anpassungsvorgang erhöht und dadurch ein vertieftes Verständnis gefördert.

4. Das Analysewerkzeug `kafe`

Das *Python*-Paket `kafe` (Abk. für ***Ka**rlsruhe **F**it **E**nvironment*) soll der Einführung in die Datenanalyse für Studierende im Rahmen der physikalischen Praktika dienen. Das Grundprinzip beim Entwurf der Software, der Aufbau und die Funktionsweise werden in diesem Kapitel erläutert.

4.1. Voraussetzungen und Ansatz

Die Hauptfunktion von `kafe` ist das Anpassen von Modellfunktionen an Datenpunkte. Dabei ist die korrekte Behandlung von Unsicherheiten der abhängigen und unabhängigen Variablen möglich. Korrelationen der Messfehler werden mithilfe von Kovarianzmatrizen (s. Abschn. 2.1.4/S. 8) behandelt.

Darüber hinaus wird auch die Möglichkeit angeboten, Informationen über den Ablauf der Anpassung auszugeben und die Ergebnisse graphisch darzustellen. Damit erfüllt `kafe` den primären Zweck, als Datenanalysepaket im Physikpraktikum einsetzbar zu sein.

Ein weiteres Ziel von `kafe` ist es darüber hinaus auch einige didaktische Aspekte zu erfüllen. Zu den zentralen Aspekten, die bei dem Entwurf und Implementierung der Software berücksichtigt werden, gehören also die Zugänglichkeit für Studierende und das Ermöglichen eines schnellen Lernzuwachses. Weiterhin sollen im Praktikum zusätzlich zum praktischen Aspekt zugleich Programmierkenntnisse in einer höheren Programmiersprache vermittelt werden. Dies soll vor allem durch die vertiefte Auseinandersetzung mit Auswertungsverfahren ein tieferes Verständnis der grundlegenden Konzepte der Datenanalyse fördern.

Als höhere Programmiersprache ist die interpretierte Sprache *Python*[pyt] für diesen Zweck optimal. *Python* verfügt über eine einfache Syntax und ist auf Leserlichkeit und Modularität ausgerichtet. Bereits vorhandene *Python*-Bibliotheken für effiziente numerische Operationen und Datenvisualisierung machen *Python* zu einem vielfältigen und leistungsfähigen Werkzeug des Physikers.

Die Auffassung von Datenanalyse als Programmieraufgabe bietet die notwendige Flexibilität, um die im Kap. 3/S. 13 durchgenommenen Überlegungen zur Funktionalität

umsetzen zu können. Dies ermöglicht es weiterhin, kafe als Schnittstelle zwischen bereits vorhandenen Datenanalyse-Programmen und dem Studierenden im Praktikum, zu konzipieren. Durch diesen Ansatz wird zugleich die Notwendigkeit der erneuten Implementierung von Anpassungs- und Darstellungsmethoden vermieden und die bereits vielfältige Funktionspalette bestehender Software zum Vorteil gemacht.

4.1.1. Anwendungsbereich und Erweiterung

Bei der Entwicklung von kafe wurde stets unter Berücksichtigung des geplanten Anwendungsbereichs vorgegangen, um ein auf das Physikpraktikum maßgeschneidertes Werkzeug zu erhalten. Zum einen wird nur die Möglichkeit von Datenpunkten in der x–y-Ebene in Erwägung gezogen. Dies bedeutet, dass nur Modellfunktionen einer einzigen unabhängigen Variablen berücksichtigt werden. Der Anzahl freier Parameter oder der Datenpunkte werden *a priori* keine Grenzen gesetzt.

Ferner sind im angenommenen Fehlermodell keine Korrelationen zwischen den x- und den y-Fehlern angenommen. Dies ist wiederum auf auf die Tatsache zurückzuführen, dass dieser Fall bei Praktikumsversuchen selten auftritt.

Zu bemerken ist, dass obwohl die Standardeinstellungen von kafe auf eine möglichst reibungslose Anwendung ausgerichtet sind, die gesamte Funktionalität der zugrundeliegenden Pakete dem Benutzer offen gehalten wird. Somit ist eine genaue Anpassung an spezifische Bedürfnisse oder sogar die Erweiterung des Programms durch Zusatzmodule für den fortgeschrittenen Anwender möglich.

4.1.2. Zugrundeliegende Software

Im Rahmen eines *Python*-Pakets verbindet kafe Komponenten bereits vorhandener Software. Zum einen werden grundlegende Datenstrukturen wie *Arrays* oder Matrizen in der Implementierung des Numerik-Pakets *NumPy*[num] eingesetzt. Dies gewährleistet einen effizienten Umgang mit den Daten. Die Funktionalität wird durch das *Python*-Paket *SciPy* erweitert, etwa für numerische Berechnung von Funktionsableitungen oder speziellen Funktionen.

Die graphische Darstellung ist mithilfe der Bibliothek matplotlib[mat] realisiert. Diese ist vielseitig anwendbar, besonders jedoch zur Funktionsdarstellung durch das Untermodul pyplot. Der Einsatz von matplotlib erlaubt einen hohen Anpassungsgrad der graphischen Darstellungen, so dass sich mittels Festlegung sinnvoller Standard-Einstellungen für die Einbindung in geschriebene Berichte geeignete Graphiken ergeben.

Die Zentralkomponente von kafe ist allerdings der im generischen Analysepaket *ROOT* enthaltene Funktionsminimierer *Minuit*. Die Einbindung von *Minuit* in *Python* erfolgt über *PyROOT*[Lav], eine *Python–ROOT* Kommunikationsbrücke.

4.2. Implementierung

In der Implementierung wird nach dem Paradigma der Objektorientierung vorgegangen. Jede der Aufgaben, die sich mit kafe erledigen lassen, fällt einem dafür zuständigen Objekt zu. Ein kurzer Überblick über den Aufbau und der Funktionsweise von kafe wird hier erstellt.

4.2.1. Aufbau

Der Analyseprozess kann in drei Etappen eingeteilt werden. Zuerst werden die Messdaten und Messfehler eingelesen und ein Datensatz erstellt. Anschließend wird die tatsächliche Modellanpassung ausgeführt und es wird schließlich eine graphische Darstellung erzeugt.

Entsprechend werden in `kafe` die Klassen `Dataset`, `Fit` bzw. `Plot` implementiert, welche der Bewältigung dieser Aufgaben dienen. Eine Übersicht der inneren Struktur von `kafe` wird in Abb. 4.1/S. 21 gezeigt.

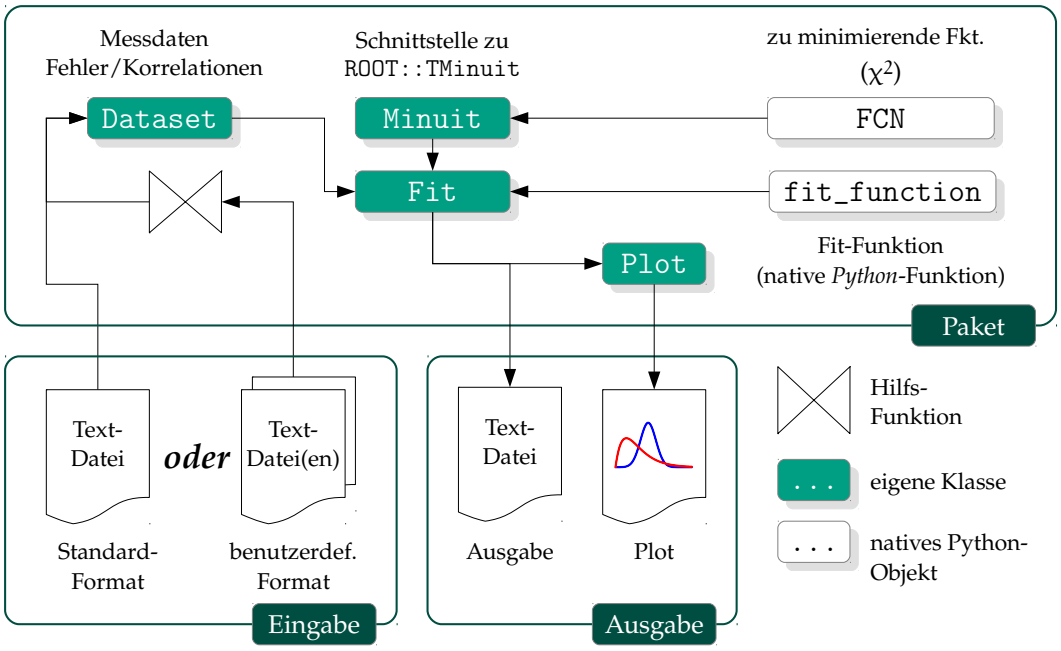


Abbildung 4.1.: Schematische Darstellung des Aufbaus von `kafe`. Die Messdaten und ggf. Messfehler und deren Korrelationen werden aus Dateien im Standard- oder in einem benutzerdefinierten Format eingelesen und ein `Dataset`-Objekt wird erstellt. Ein `Fit`-Objekt übernimmt die Anpassung der Modellfunktion `fit_function` an die Daten durch die Minimierung der Funktion `FCN` und gibt dabei Informationen über die Anpassung aus. Die graphische Darstellung erfolgt über ein `Plot`-Objekt.

`Dataset`

Die Messdaten selber werden im `Dataset`-Objekt in *NumPy Arrays* hinterlegt. Fehlerinformation für jede der zwei Achsen `x` und `y` speichert das `Dataset`-Objekt in Form von Kovarianzmatrizen. Auf diese Weise kann eine allgemeinere Fehlercharakterisierung und -behandlung erfolgen, die zugleich Korrelationen zwischen den Messwerten einer Achse berücksichtigt. Sonderfälle (wie etwa der Fall ausschließlich statistischer, unkorrelierter Messfehler) lassen sich dadurch ebenso behandeln.

Zur Initialisierung eines `Dataset`-Objekts werden *NumPy Arrays* und eventuell *NumPy* Matrix-Objekte dem Konstruktor übergeben. Im Regelfall sind jedoch experimentelle Daten in Dateien hinterlegt, die eingelesen werden müssen. Für die einheitliche

Speicherung und das spätere Einlesen von Datensätzen wird für `Dataset`-Objekte ein Standard-Format festgelegt (s. Abschn. 4.2.3/S. 24).

Ein Verweis auf eine Datei in Standardformat reicht bereits aus, um einen `Dataset`-Objekt erstellen zu können. Liegen die Messdaten in anderen Formaten vor, ist die Programmierung einer eigenen Routine für die Datensatzerstellung nötig. Eine vorprogrammierte Einlesemethode für Daten, welche in Tabellenformat gespeichert sind, wird als Hilfsfunktion bereits mitgeliefert. Das Einlesen von Kovarianzmatrizen aus separaten Dateien wird ebenfalls unterstützt.

`Fit`

Ein `Fit`-Objekt bietet die notwendige Funktionalität, um Anpassungen einer Modellfunktion an Daten durchführen zu können. Dazu benötigte Verweise auf den Datensatz und die Modellfunktion, die dem Datensatz angepasst werden soll, werden im Konstruktor des `Fit`-Objekts übergeben.

Modellfunktionen und die `FitFunction` Wrapper-Klasse

Als Modellfunktion ist jede aufrufbare *Python*-Struktur mit numerischem Rückgabewert zulässig. Im einfachsten Fall ist dies eine native *Python*-Funktion. Allerdings gilt hier die implizite Bedingung, dass das erste Argument der Funktion der unabhängigen Variablen `x` entspricht. Die restlichen Argumente entsprechen den Modellparametern. Weiterhin wird gefordert, dass für alle Modellparameter Standardwerte spezifiziert werden. Diese werden bei der Anpassung als Startwerte der Parameter übernommen.

Um eine Modellfunktion vom `Fit`-Objekt als solche erkennen zu lassen, wird diese mit der Klasse `FitFunction` dekoriert. Diese Wrapper-Klasse um die Modellfunktion ist wiederum ausführbar (und kann also als Modellfunktion selber verwendet werden), erlaubt es allerdings dem Entwickler, weitere Prozeduren und Eigenschaften als Instanzmethoden bzw. Attribute der `FitFunction` zu hinterlegen.

Dem Anwender bzw. den anderen Kontrollstrukturen im Rahmen von `kafe` wird somit ein bequemer Zugang zu diesen Attributen und Methoden ermöglicht. So kann zum Beispiel zur ersten Ableitung der Modellfunktion, welche an einigen Stellen im Programm gebraucht wird, wie zu einer üblichen Instanzmethode gegriffen werden.

Auch Metadaten rund um die Modellfunktion lassen sich unter Verwendung der Wrapper-Klasse behandeln. Beispielsweise wird im Attribut `latex_name` der Klasse `FitFunction` ein LaTeX-Ausdruck für den Funktionsnamen hinterlegt, welcher anschließend für die graphische Darstellung im Rahmen des `Plot`-Objekts verwendet wird.

Für das gezielte setzen dieser Attribute werden weitere Dekorator-Klassen bereitgestellt.

Zu minimierende Funktion $\mathrm{F}(\mathbf{p})$ (`FCN`)

Zusätzlich lässt sich durch die Verwendung einer eigenen zu minimierenden Funktion $\mathrm{F}(\mathbf{p})$ (in der Bezeichnung von *Minuit* auch `FCN` genannt) das Anpassungsverfahren weiter individuell einrichten. Diese kann ebenfalls als *Python*-Funktion geschrieben werden, allerdings muss dafür eine feste Argumentstruktur beachtet werden. Die Anpassung selber erfolgt durch Aufruf einer Instanzmethode. Dies setzt den Funktionsminimierer in Gang, um die Parameterwerte im Funktionsminimum zu ermitteln.

Fixierung von Parametern

Für komplexere Modellfunktionen ist die Wahrscheinlichkeit, durch den einfachen Minimierungsalgorithmus nur ein lokales Minimum zu erreichen, größer als bei einfachen Modellen. Daher ist es oft nötig, den Minimierer in die Richtung des globalen Minimums zu lenken.

Dies geschieht durch die von *Minuit* unterstützte Fixierung einzelner Parameter. Die zu minimierende Funktion wird dann nur hinsichtlich der freien Parameter minimiert. Nach dem Erreichen des gewünschten globalen Minimums werden alle fixierte Parameter freigesetzt und eine letzte Anpassung durchgeführt.

`Plot`

Um eine graphische Darstellung mit `matplotlib` zu ermöglichen wurde das `Plot`-Objekt implementiert. Durch die Versetzung der Darstellungsfunktionalität in ein spezielles Objekt ist auch die Darstellung mehrerer Anpassungen im selben Bild möglich.

Nachdem die notwendigen Datensätze deklariert und die Anpassungen durchgeführt worden sind, können die resultierende `Fit`-Objekte einem `Plot`-Objekt im Konstruktor übergeben werden. Das resultierende `Plot`-Objekt erstellt über `matplotlib` die tatsächliche graphische Darstellung der Datenpunkte, der angepassten Modellfunktion und ggf. der Fehlerbalken und des Konfidenzbandes. Die so erstellte Graphik kann dann angezeigt werden, um eventuell weitere Änderungen interaktiv vorzunehmen, oder direkt gespeichert werden.

4.2.2. Anpassungsalgorithmus

Die Funktionsanpassung an Daten ist ein Minimierungsproblem. Hierfür wird nach Angabe einer Modellfunktion und eines Datensatzes eine Funktion $F(\mathbf{p})$ definiert, die einem Abstandsmaß zwischen der Modellfunktion und den Daten entspricht. Diese wird anschließend minimiert, um die beste Anpassung zu erhalten.

Für einen Teil der angestrebten Funktionalität, nämlich die Berücksichtigung von Korrelationen, ist bereits durch eine geschickte Definition der zu minimierenden Funktion gesorgt. Für die voreingestellte χ^2-Methode ($F(\mathbf{p}) = \chi^2$) macht man sich hier Kovarianzmatrizen zunutze (s. Abschn. 2.1.2/S. 7–2.1.4/S. 8).

Es wird jedoch eine gewisse Steuerung des Minimierungsprozesses gefordert, um beispielsweise die erwünschte Berücksichtigung von „x-Fehlern“ zu erlangen. Dies wird folgendermaßen durch Iteration der Anpassung erzielt.

In einer ersten, vorläufigen Anpassung wird nur die y-Kovarianzmatrix berücksichtigt. Eine Minimierung wird mithilfe der *Minuit*-Routinen `MIGRAD` und `HESSE` ausgeführt und liefert dann eine Schätzung der Modellparameter und der Parameterunsicherheiten.

In jeder anschließenden Iteration wird mithilfe der ersten Ableitung der Modellfunktion die x-Kovarianzmatrix auf die y-Kovarianzmatrix projiziert, um eine Gesamtfehlermatrix zu erhalten (s. Abschn. 2.1.5/S. 9), und mit dieser Matrix dann eine erneute Anpassung mit `MIGRAD` und `HESSE` durchgeführt.

Der Iterationsprozess hört auf, sobald die durch Projektion gewonnene Gesamtkovarianzmatrix keine signifikanten Änderungen zur vorherigen Matrix mehr aufweist. Dem `Fit`-Objekt können am Ende des Fit-Vorgangs die optimalen Parameterwerte und die Kovarianzmatrix der Parameter entnommen werden. Eine Übersicht des Daten- und Steuerflusses beim Fit-Vorgang mit `kafe` kann in Abb. 4.2/S. 24 gesehen werden.

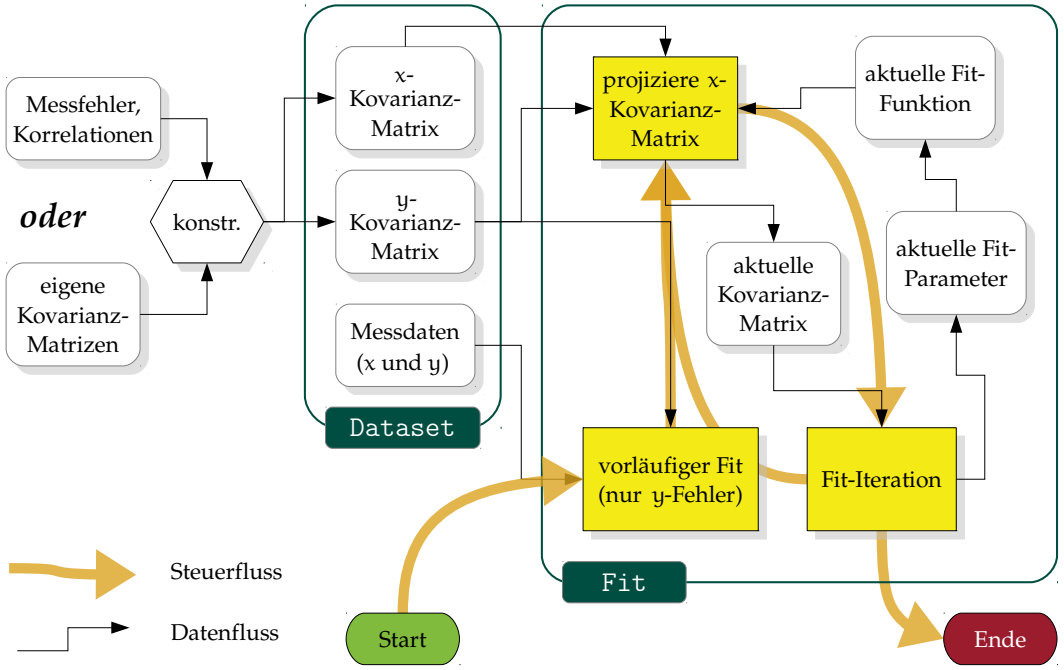


Abbildung 4.2.: Schematische Darstellung des Ablaufs einer Anpassung mit `kafe`. Ein `Dataset`-Objekt wird durch Eingabe der Messdaten erstellt. Messunsicherheiten werden in Form von Kovarianzmatrizen gespeichert, die entweder aus angegebenen Messfehlern und Korrelationen konstruiert werden, oder alternativ explizit angegeben werden. Zunächst wird eine vorläufige Anpassung unter Berücksichtigung der y-Fehler gemacht. Anschließend wird durch Projektion der x-Fehler die Kovarianzmatrix neu berechnet und eine Anpassung erneut durchgeführt. Dies wird solange iteriert, bis die Änderung der Kovarianzmatrix durch die Projektion vernachlässigbar ist.

4.2.3. Standard-Dateiformat für das Einlesen von Datensätzen

Die Organisation von Messdaten in Tabellenform mit einem Messpunkt pro Reihe reicht für Anpassungen ohne Korrelationen aus. Sobald die Messfehler jedoch Korrelationen aufweisen entsteht die Notwendigkeit, separat Kovarianzmatrizen (jede oft in eine eigene Datei gespeichert) einzulesen.

Dies verursacht bereits bei einer gemäßigten Anzahl von Datensätzen einen großen Organisationsaufwand. Um dies zu vermeiden wird ein Standard-Dateiformat für Datensätze eingeführt, welches Messdaten, Messfehler und Fehlerkorrelationen in einer einheitlichen Struktur zusammenfasst.

Der hierfür gemachte Ansatz ist die Einteilung eines Datensatzes in einzelne Blöcke, die je einer Achsenrichtung entsprechen. Die x- und y-Blöcke haben dabei dieselbe Struktur.

In jedem Block steht pro Reihe in der ersten Spalte der tatsächliche Messwert. Falls für die Achsenrichtung noch zusätzlich zufällige Messfehler vorliegen, werden diese in die zweite Spalte eingetragen. Schließlich werden im Fall korrelierter Messfehler weiter Korrelationskoeffizienten als untere Dreiecksmatrix in die Spalten 3 bis $N + 1$ eingetragen.

Die auf der Hauptdiagonalen liegenden Einsen sind implizit und werden aus Gründen der Übersichtlichkeit weggelassen. Um eine gute Lesbarkeit durch den Menschen zu bewahren werden die Datenblöcke durch beschreibende Kommentarzeilen eingeführt. Der x-Datenblock zum Beispiel ist dann nach folgendem Schema organisiert:

$$\begin{array}{cccccc} x_1 & \sigma_{x,1} & & & & \\ x_2 & \sigma_{x,2} & \mathrm{Cor}_{x,1\,2} & & & \\ x_3 & \sigma_{x,3} & \mathrm{Cor}_{x,1\,3} & \mathrm{Cor}_{x,2\,3} & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ x_N & \sigma_{x,N} & \mathrm{Cor}_{x,1\,N} & \mathrm{Cor}_{x,2\,N} & \dots & \mathrm{Cor}_{x,N-1\,N} \end{array} \tag{4.1}$$

Die in diesem Klartext-Format gespeicherten Datensätze können von `kafe` als `Dataset`-Objekt wieder eingelesen werden.

4.2.4. Hilfsfunktionen für das Einlesen anderer Formate

Oft liegen die Messdaten nicht in dem in Abschn. 4.2.3/S. 24 vorgestellten Format vor. Ein möglicher Grund hierfür ist die getrennte Speicherung der Fehlerinformationen für verschiedene Fehlerquellen in Form von Kovarianzmatrizen.

In diesem Fall werden die Messdaten in Tabellenformat mit einem Messpunkt pro Zeile gespeichert. Um Daten in diesem gängigen Format einlesen zu können, wird in einem Zusatzmodul eine Hilfsfunktion angeboten, welche aus dem Inhalt einer Datei mit diesem Format ein `Dataset`-Objekt erstellen soll.

Die Möglichkeit, Daten zu verwenden, welche in anderen Formaten gespeichert werden, bleibt durch das Programmieren weiterer solcher Hilfsfunktionen offen.

4.2.5. Ausgabeformat

Im Laufe einer Anpassung gibt `kafe` in einheitlicher Form Informationen über den Fit-Ablauf aus. Dazu gehören in erster Linie eine erneute Ausgabe des Datensatzes im Standardformat (s.o.) und die Angabe der Modellfunktion.

Die Modellparameter im gefundenen Minimum von $F(\mathbf{p})$ werden anschließend zusammen mit Korrelationskoeffizienten in die Ausgabedatei geschrieben. Das Format gleicht hier dem Datensatz-Standardformat: In der ersten Spalte steht wiederum der Parameterwert, in der zweiten der (parabolische) Parameterfehler und in den restlichen Spalten die Korrelationskoeffizienten zu den anderen Parametern. Im Gegensatz zum Datensatz-Standardformat werden hier außerdem die Parameternamen vor jeder Ausgabezeile in Form von Kommentaren eingefügt. Diese werden wie in *Python* mit einem Rautezeichen (#) gekennzeichnet und sorgen für eine verbesserte Lesbarkeit der Ausgabe.

Im eben genannten Abschnitt der Ausgabe werden zunächst die ermittelten Modellparameter und deren Korrelationskoeffizienten, so wie diese von *Minuit* ausgegeben werden, in die Datei geschrieben.

Um das Einbinden des Ergebnisses in andere Dokumenten zu erleichtern wird ebenfalls eine vereinfachte Ausgabe vorgenommen. Der Fehler jedes Parameters wird dafür auf zwei signifikante Stellen, und der Parameterwert anschließend auf die Größenordnung des so entstandenen Fehlers, gerundet. Jede Parameterangabe erfolgt dann in der Form:

$$\text{Parametername} = \text{Wert} \pm \text{Fehler (parabolisch)}$$

Als Letztes erfolgt die Ausgabe von Daten, welche die Güte der Anpassung betreffen. Der Funktionswert der zu minimierenden Funktion $F(\mathbf{p})$ im gefundenen Minimum, geteilt durch die Anzahl der Freiheitsgrade (s. Abschn. 2.2.1/S. 10) ist hierbei ein geeigneter Indikator der Fit-Güte und wird demzufolge ausgegeben.

Eine weitere Einschätzung der Güte ist über die Ermittlung der Wahrscheinlichkeit, einen schlechteren Fit für den Datensatz zu erhalten, möglich. Diese Einschätzung ist durch die Aufintegration der χ^2-Verteilung mit der entsprechenden Anzahl von Freiheitsgraden möglich. Es wird dabei über die Wahrscheinichkeitsdichte von dem ermittelten χ^2-Wert bis ins Unendliche aufintegriert und der resultierende Wert ausgegeben (s. Abschn. 2.2.2/S. 10).

Ein Beispiel der Textausgabe ist in Abb. 4.3/S. 27 zu sehen.

4.2.6. Graphische Ausgabe

Für die graphische Darstellung wird die *Python*-Bibliothek `matplotlib` verwendet. Diese bietet zahlreiche Funktionen für die Erstellung von Plots und enthält Methoden für eine weitere Personalisierung der graphischen Ausgabe.

Die Darstellungsschnittstelle zu `matplotlib` soll durch auf das Problem maßgeschneiderte Standard-Einstellungen Graphiken erstellen, die den Anforderungen im Praktikum entsprechen. Ein Beispiel der Ausgabe ist in Abb. 4.4/S. 28 zu sehen.

Ein wichtiger Aspekt hierbei ist die korrekte Beschriftung der Graphiken. Dafür muss die Möglichkeit bestehen, mathematische Notation in die Beschriftung einzufügen. Dies geschieht durch Aufrufen von LaTeX, einer Textsatzsoftware, die auf das verbreitete Textsatzsystem TeX beruht. [lat]

Durch geeignete LaTeX-Befehle, welche an `matplotlib` übergeben werden, können auch Schriftart und -größe eingestellt werden, so dass diese mit dem Dokument übereinstimmen, wo die Graphik später eingebunden werden soll. Die Schriftgröße wird standardmäßig auf einen vergleichsweise hohen Wert gesetzt, so dass die Graphik selbst bei einer Verkleinerung gut lesbar bleibt.

Bei Graphiken, die mehr als eine Messreihe enthalten, ist eine sinnvolle Farb- und Formkodierung standardmäßig eingestellt. Diese soll erlauben, die Messreihen leicht zu identifizieren. Dasselbe gilt für die schwarz-weiß-Version der Graphiken.

Einige weitere Maßnahmen für eine erleichterte Lesbarkeit wurden vorgenommen. Erstens wird ein Rand um die Gesamtheit der Datenpunkte gelassen. Dies soll gewährleisten, dass Datenpunkte, die am Rand des erfassten Bereichs liegen, sichtbar bleiben. Zweitens wird die Legende außerhalb des Plots gestellt. Dies verhindert eine mögliche Bedeckung wichtiger Stellen in der Darstellung. Letztens wird die Graphik mit einem Raster versehen, um die Lokalisierung einzelner Messpunkte zu vereinfachen.

Ein weiterer wichtiger Punkt ist die sinnvolle Darstellung von Informationen über die Anpassung. Dafür kann in dem unterhalb der Legende verbliebener Raum eine Info-Box angezeigt werden, wo Informationen über die Fit-Parameter angezeigt werden. Das Konzept der Konfidenzintervalle wird graphisch durch das Anlegen eines 1σ-Konfidenzbandes an die angepassten Funktionen wiedergegeben.

```
############
# Dataset #
############

# axis 0: x
# datapoints uncor. err.
9.438644e-01 1.000000e-01
2.067665e+00 1.000000e-01
3.030246e+00 1.000000e-01

# axis 1: y
# datapoints uncor. err.
2.211930e+00 1.000000e-01
2.435954e+00 1.000000e-01
2.912194e+00 1.000000e-01


#################
# Fit function #
#################

linear_2par(x; slope, y_intercept) = slope * x + y_intercept

###############
# Fit result #
###############

# slope
# value        uncor. err.
3.318614e-01   7.134603e-02

# y_intercept
# value        uncor. err.    correlations
1.851682e+00   1.560320e-01   -9.208725e-01

########################
# Final fit parameters #
########################

slope = 0.332 +- 0.071
y_intercept = 1.85 +- 0.16

################
# Fit details #
################

FCN      1.40598691372
FCN/ndf  1.40598691372
EdM      4.17362461965e-19
UP       1.0
STA      Error matrix accurate

chi2prob 0.23572373244
HYPTEST  accepted (CL 5%)
```

Abbildung 4.3.: Textausgabe für einen Beispieldatensatz

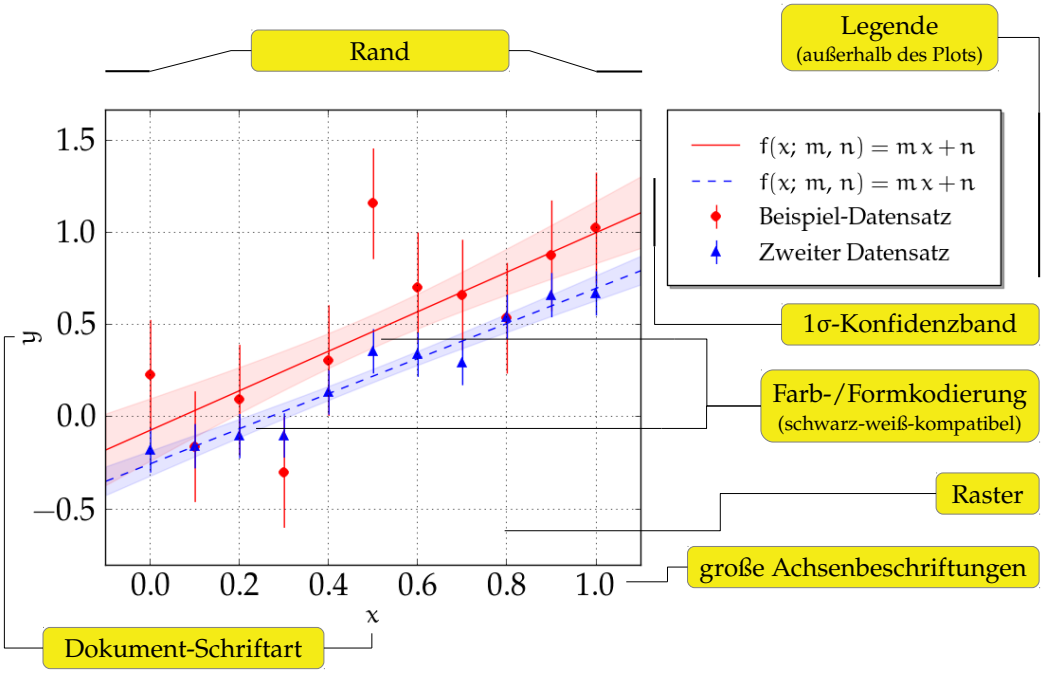


Abbildung 4.4.: Beispiel der graphischen Ausgabe mit `matplotlib`. Einige wichtige Elemente der Darstellung sind beschriftet.

5. Technische Ausführung

Das *Python*-Programmpaket `kafe` soll die Datenanalyse im Physikpraktikum erleichtern und zugleich den Studierenden das Programmieren nahebringen. Um dies zu erzielen wurde bei der Entwicklung besonders darauf Wert gelegt, gut lesbaren Code zu produzieren und die Bedienung besonders intuitiv zu gestalten.

Die gesamte im Rahmen dieser Bachelorarbeit entstandene Software wurde unter der *GNU General Public License* lizenziert, damit der Quellcode offen und zugänglich bleibt und um für fortgeschrittene Anwender das Studieren und Ändern des Codes zu ermöglichen.

Darüber hinaus wurde für `kafe` eine Dokumentation erstellt und öffentlich bereitgestellt, welche als Referenz der wesentlichen Kernfunktionen der Software dienen soll. In diesem Kapitel werden die technischen Maßnahmen beschrieben, welche für die Entwicklung und Verteilung der Software von besonderer Bedeutung sind.

5.1. Entwicklung

In einer ersten Phase der Entwicklung wurden der logische Aufbau und die Aufteilung des Pakets in Module festgelegt. Hierbei wurde jedem der drei Hauptobjekte `Dataset`, `Fit` und `Plot` (s. 4.2.1/S. 21) je ein Modul gewidmet. Ein gesondertes Modul existiert auch für die `FitFunction`-Klasse, welche als *Wrapper* für die von `kafe` verwendeten Modellfunktionen dient, also welche native *Python*-Funktionen an der Verwendung als Modellfunktionen anpasst. Andere Module wurden für Datei- und *Stream*-Manipulation, sowie für numerische Operationen erstellt.

Da bestimmte Modellfunktionen (z.B. lineare oder exponentielle Modelle mit verschiedenen Parametrisierungen) oft verwendet werden, wurde eine Funktionsbibliothek erstellt und als separates Modul bereitgestellt. Dies soll dem Anwender einerseits die Implementierung der Modellfunktion im Anpassungsprogramm selber ersparen. Andererseits dient diese Bibliothek als Beispiel dafür, wie Modellfunktionen in *Python* geschrieben werden, wobei insbesondere der Vorteil bei der Verwendung von Dekoratoren verdeutlicht werden soll.

Die vorgenommene Modularisierung des Quellcodes trägt somit beim Verständnis der Struktur und der Funktionsweise des Programms bei. In dieser Hinsicht sind auch die

Lesbarkeit und Verständlichkeit des Programmcodes im Entwicklungsprozess zentrale Punkte gewesen. Dies ist besonders deswegen wichtig, da es sich um ein didaktisches *open source*-Projekt handelt, und deswegen das Untersuchen des Quellcodes nicht nur erlaubt, sondern auch erwünscht ist. Aus diesem Grund erschien es sinnvoll, den Code nach den Richtlinien guten Programmierstils, etwa nach den Vorgaben des PEP 8,[vRWC01] zu gestalten.

Für die Schlussphase der Entwicklung wurde zur Versionskontrolle `git`[git] verwendet, um den Entwicklungsprozess genauer zu dokumentieren und für die Weiterentwicklung künftig auch eine Kollaboration zu ermöglichen. Ein vorläufiges, öffentliches *Repository* wurde unter `http://ekptrac.physik.uni-karlsruhe.de/git-public/kafe` angelegt.

5.2. Softwarepaket und Verteilung

Bei der Bereitstellung von `kafe` spielen die zahlreichen Softwareabhängigkeiten eine wichtige Rolle. Diese müssen auf dem System auf dem `kafe` zur Anwendung gebracht wird mit den richtigen Einstellungen installiert werden.

Dies erweist sich vor allem bei Softwarepaketen wie *ROOT*, die typischerweise nur als Quellcode vorliegen, zumeist als problematisch, da hier bei jeder Plattform mit unterschiedlichen Rahmenbedingungen gerechnet werden muss. Um dennoch den Installationsprozess zu erleichtern werden mehrere Installationsarten angeboten.

Zudem erfolgte die Entwicklung von `kafe` unter Linux[fed], weswegen auch hauptsächlich vergleichbare Unix-basierte Betriebssysteme als Zielplattformen für die Anwendung vorgesehen wurden.

5.2.1. Virtuelle Maschine

Eine Strategie, diese Hürden zu überwinden ist, die Software in eine virtuelle Maschine zu installieren, und diese dann zur Verfügung zu stellen. Dies hat weiterhin den Vorteil, dass die Software startbereit geliefert wird und dadurch der Nutzen für den Anwender maximal ist.

Zudem ist eine solche virtuelle Maschine bereits zum Einsatz in den praxisnahen Lehrveranstaltungen zur Teilchenphysik am Karlsruher Institut für Technologie bereits vorhanden.[Qua12] Als primäre Verteilungsmethode wurde `kafe` also in diese virtuelle Maschine installiert.

5.2.2. Eigenständige Installation

Für fortgeschrittene Anwender existiert die Möglichkeit der eigenständigen Installation. Dafür muss vorab gewährleistet werden, dass alle Softwareabhängigkeiten von `kafe` mit den richtigen Einstellungen auf das System installiert werden.

Als *Python*-Paket lässt sich `kafe` mittels standardmäßiger Werkzeuge wie `setuptools`[set], die Bestandteil vieler *Python*-Distributionen sind, oder durch weitere Paketverwaltungssoftware wie `pip`[pip] mit minimalem Aufwand installieren. Hierbei bietet `pip` den Vorteil, Pakete mit einem Befehl in der Kommandozeile aktualisieren oder entfernen zu können.

Eine genauere Anleitung zu der eigenständigen Installation wird als Textdatei mit der Software mitgeliefert.

5.3. Dokumentation und Anwendungsbeispiele

Ein Großteil der Dokumentation ist durch das ausführliche Kommentieren des Codes, sowie durch die von *Python* angebotenen *Docstrings* erfolgt. Mithilfe von *Sphinx* [sph] wurde anschließend aus ausgewählten Kommentarzeilen und den *Docstrings* eine externe Dokumentation erstellt. Diese steht in HTML-Format und als mithilfe von LaTeX generierte PDF-Datei zur Verfügung.

Die so entstandene Dokumentation gibt Auskunft über die für die Methoden von `kafe`, welche für den Aufruf durch den Anwender bestimmt sind. Diese verschafft jedoch nur einen Überblick über die gesamte Funktionalität und eignet sich somit lediglich als Nachschlagewerk für die bereits mit `kafe` vertrauten Anwender und für Entwickler.

Um hingegen Einsteigern zu ermöglichen, sich mit den Basisfunktionen von `kafe` vertraut zu machen, wurden einige kommentierte Beispielprogramme geschrieben und bereitgestellt.

6. Zusammenfassung und Ausblick

Analyse und Auswertung von Messdaten sind zentrale Aspekte jedes Physikstudiums. Daher muss die Suche nach einer passenden Gestaltung der rechnergestützten Analyse in den physikalischen Praktika stets praktischen und didaktischen Überlegungen unterliegen. Neben der Vermittlung von Grundwissen in Bereichen wie Wahrscheinlichkeitstheorie oder Stochastik ist die Frage der Softwarewahl von entscheidender Bedeutung.

Dabei ist diese Frage von einer scheinbar großen Wahlfreiheit in der Auswahl von Software geprägt. Wie in den vorherigen Kapiteln erläutert, decken gängige Softwarepakete den im Praktikum erforderlichen Funktionalitätsumfang allerdings nicht ab. Auch dem didaktischen Zweck der Vermittlung von Programmierkenntnissen, welche von der Datenanalyse nicht zu trennen sind, stehen die Programme mit oftmals vorgefertigten und undurchsichtigen Anpassungsverfahren im Wege.

Die in dieser Arbeit vorgeschlagene Lösung setzt für die Überwindung dieser Unzulänglichkeiten auf die Einführung der höheren Programmiersprache *Python*. Das in *Python* geschriebene Analysepaket `kafe` nutzt die Flexibilität der Programmiersprache, vorhandene Bibliotheken und die Einbindung von spezialisierten Drittprogrammen mit einer *Python*-Schnittstelle, um die oben erwähnten Mängel zu beheben.

Dieser Ansatz lässt nicht nur Entwicklern, sondern auch Anwendern selber die Möglichkeit einer Erweiterung der Software um zusätzliche Funktionalität völlig offen. Dies kann etwa durch die Implementierung einer von χ^2 verschiedenen zu minimierenden Funktion für die Anpassung oder durch eine Erweiterung der graphischen Darstellung realisiert werden. Dadurch lässt sich der Einsatzbereich des Programms auch auf nichtdidaktikbezogene Bereiche ausweiten.

Trotz Ausrichtung der Software auf Studierende im Physikpraktikum ist `kafe` auch für den Einsatz in den Übungen anderer einführenden Lehrveranstaltung zur rechnergestützten Physik oder zur Datenanalyse geeignet. Eine erstmalige Anwendung von `kafe` als Lehrmittel wird beispielsweise im Rahmen der Übungen zur Lehrveranstaltung *Rechnernutzung in der Physik* am Karlsruher Institut für Technologie geplant.

Der Entwurf sinnvoller Beispiele, die für eine Einführung in die Funktionsanpassung mit `kafe` geeignet sind, ist im Entwicklungsprozess priorisiert worden. Nicht weniger wichtig ist das Erstellen einer ausführlichen Dokumentation für den Anwender, welche die aktuelle Dokumentation erweitern soll.

Damit verbunden befindet sich ein `kafe`-*Tutorial* in Planung. Dieses soll durch eine geführte Bearbeitung von einfachen Beispielen die Datenanalyse mit `kafe` auf anschaulicher Weise vermitteln.

Angesichts der in dieser Arbeit erläuterten Aspekte erweist sich `kafe` als geeignet, um Studierenden im Rahmen ihres Studiums das Programmieren nahezulegen und dadurch grundlegende Begriffe der Datenanalyse effizient zu vermitteln. Das Bewältigen typischer Problemstellungen der Physik wird dadurch mithilfe eines maßgeschneiderten Werkzeugs gefördert.

7. Danksagung

Mein ganz herzlicher Dank geht an Herrn Prof. Dr. Günter Quast für die Gelegenheit, ein interessantes Thema in dieser Bachelorarbeit bearbeiten zu können, sowie für seine engagierte Betreuung und seine Anregungen während der gesamten Bearbeitungszeit.

Ganz besonderen Dank auch an Herrn Dr. Fred-Markus Stober für die freundliche Übernahme des Koreferats und für seine Unterstützung in vielen technischen Aspekten der Ausführung.

Für die angenehme und freundliche Ambiente und ihre fachliche und moralische Unterstützung bei der Arbeit danke ich der gesamten Arbeitsgruppe am IEKP. Ich bin darunter besonders für die aktive Beteiligung bei der Namensgebung der Software und der Korrekturlesung der vorliegenden Arbeit dankbar.

Schließlich möchte ich mich bei meinen Eltern bedanken, die mein gesamtes Studium überhaupt erst ermöglicht haben und mir bei jedem Schritt beiseite gestanden sind.

Literatur- und Softwareverzeichnis

[Bar89] Roger J Barlow: *Statistics: a guide to the use of statistical methods in the physical sciences*, volume 29. John Wiley & Sons, 1989.

[BR97] Rene Brun and Fons Rademakers: *ROOT – an object oriented data analysis framework*. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 389(1):81–86, 1997. `http://root.cern.ch`.

[BZ10] Gerhard Bohm und Günter Zech: *Introduction to statistics and data analysis for physicists*. DESY, 2010. `http://www-library.desy.de/preparch/books/vstatmp.pdf`.

[Cra98] Dick Crawford: *Gnuplot documentation*, 1998. `http://www.gnuplot.info/docs_4.0/gnuplot.html`.

[fed] *Fedora 18 (Spherical Cow)*. `http://fedoraproject.org/`.

[git] `git` *1.8.1.4*. `http://git-scm.com/`.

[JR75] F James und M Roos: *Minuit – a system for function minimization and analysis of the parameter errors and correlations*. Computer Physics Communications, 10(6):343–367, 1975.

[lat] *LaTeX – A document preparation system*. `http://www.latex-project.org/`.

[Lav] Wim Lavrijssen: *PyROOT – A Python–ROOT Bridge*. `http://wlav.web.cern.ch/wlav/pyroot/`.

[Lev44] Kenneth Levenberg: *A method for the solution of certain problems in least squares*. Quarterly of applied mathematics, 2:164–168, 1944.

[Mar63] Donald W Marquardt: *An algorithm for least-squares estimation of nonlinear parameters*. Journal of the Society for Industrial & Applied Mathematics, 11(2):431–441, 1963. `http://dx.doi.org/10.1137/0111030`.

[mat] `matplotlib` *1.2.0*. `http://www.matplotlib.org`.

[Mü10] Thomas Müller: *RooFiLab – Root Fit for Laboratory Courses (Dokumentation)*. Februar 2010. `http://www-ekp.physik.uni-karlsruhe.de/~quast/RooFiLab/RooFiLab.pdf`.

[num] *NumPy 1.7.1*. `http://www.numpy.org`.

[orh] *Origin user guide: Linear Fit with X Error*. `http://www.originlab.com/www/helponline/Origin/en/UserGuide/Linear_Fit_with_X_Error_Dialog.html`.

[ori] *Origin 9*. http://www.originlab.com.

[pip] pip *1.4.1*. https://pypi.python.org/pypi/pip.

[PQZ13] Danilo Piparo, Günter Quast, and Manuel Zeise: *A ROOT guide for students — „Diving into ROOT“*. 2013. http://www-ekp.physik.uni-karlsruhe.de/~quast/Skripte/diving_into_ROOT.pdf.

[pyt] *Python 2.7*. http://www.python.org.

[Qua12] Günter Quast: *Virtuelle Maschine zur Physik*. 2012. http://www-ekp.physik.uni-karlsruhe.de/~quast/VMroot/VMachine.pdf.

[Qua13] Günter Quast: *Skript zur Funktionsanpassung mit der χ^2-Methode*. 2013. http://www-ekp.physik.uni-karlsruhe.de/~quast/Skripte/Chi2Method.pdf.

[set] setuptols *1.3*. https://pypi.python.org/pypi/setuptools.

[sph] *Sphinx 1.1.3*. http://sphinx-doc.org/.

[Str10] Tilo Strutz: *Data Fitting and Uncertainty: A practical introduction to weighted least squares and beyond*. Vieweg+Teubner, 2010.

[vRWC01] Guido van Rossum, Barry Warsaw und Nick Coghlan, 2001. http://www.python.org/dev/peps/pep-0008/.

[WK12] Thomas Williams und Colin Kelley: *Gnuplot 4.6: An Interactive Plotting Program*, 2012. http://gnuplot.sourceforge.net/.

Anhang

A. Notation und Abkürzungen

Abkürzung/Notation	Bedeutung
$\mathrm{Cor}_{x,ij}$, $\mathrm{Cor}_{y,ij}$	Korrelationskoeffizient zwischen dem i-ten und j-ten x/y-Wert
$\mathrm{Cov}_{x,ij}$, $\mathrm{Cov}_{y,ij}$	Kovarianz des i-ten und j-ten x/y-Werts (Varianz falls $i = j$)
Δx, Δy	systematischer (vollständig korrelierter) Fehler der Datenpunkte
`FCN`	Bezeichnung der zu minimierenden Funktion in *Minuit*
$F(\mathbf{p})$	Die zu minimierende Funktion für den Parametervektor $\mathbf{p}$
$f(x \mid \mathbf{p})$	Modellfunktion für den Parametervektor $\mathbf{p}$
K	Größe des Parametersatzes (Anzahl der Modellparameter)
N	Größe des Datensatzes (Anzahl der Datenpunkte eines Datensatzes)
σ_x, σ_y	statistischer (vollständig unkorrelierter) Fehler der Datenpunkte
$\mathbf{V}_x$, $\mathbf{V}_y$	Kovarianzmatrix der x/y-Werte
$\mathbf{V}_p$	Kovarianzmatrix der Parameter

B. Dokumentation (in englischer Sprache)

(s. gesonderter Anhang)

kafe Documentation

Release

Daniel Savoiu

November 12, 2013

CONTENTS

kafe is a data fitting framework designed for use in undergraduate physics lab courses. It provides a basic *Python* toolkit for fitting and plotting using already available *Python* packages such as *NumPy* and *matplotlib*, as well as *CERN ROOT*'s version of the *Minuit* minimizer.

Contents

CHAPTER

ONE

SUMMARY

The package provides a simple approach to fitting using variance-covariance matrices, thus allowing for error correlations to be taken into account. This implementation's error model assumes the measurement data (dependent variable) is distributed according to a *Gaussian* distribution centered at its "true" value. The spread of the distribution is given as a (1σ)-error.

An "errors-in-variables" model is also implemented to take uncertainties in the independent variable (x errors) into account. This is done by specifying/constructing a separate variance-covariance matrix for the x axis and "projecting" it onto the y error matrix. If the fit function is approximated in each point by its tangent line, the *Gaussian* errors in the x direction are not warped by this projection.

...

For examples on how to use **kafe**, see the `examples` folder. Consulting the API can also be helpful.

CHAPTER

TWO

API DOCUMENTATION

2.1 kafe Package

2.1.1 `kafe` Package

kafe

A Python package for fitting and plotting for use in physics lab courses.

This Python package allows fitting of user-defined functions to data. A dataset is represented by a *Dataset* object which stores measurement data as *NumPy* arrays. The uncertainties of the data are also stored in the *Dataset* as an *error matrix*, allowing for both correlated and uncorrelated errors to be accurately represented.

The constructor of a *Dataset* object accepts several keyword arguments and can be used to construct a *Dataset* out of data which has been loaded into *Python* as *NumPy* arrays. Alternatively, a plain-text representation of a *Dataset* can be read from a file.

Also provided are helper functions which construct a *Dataset* object from a file containing column data (one measurement per row, column order can be specified).

2.1.2 `_version_info` Module

2.1.3 `constants` Module

`F_SIGNIFICANCE = 2`
: Set significance for returning results and errors N = rounding error to N significant digits and value to the same order of magnitude as the error.

`G_PADDING_FACTOR_X = 1.2`
: factor by which to expand x data range

`G_PADDING_FACTOR_Y = 1.2`
: factor by which to expand y data range

`G_PLOT_POINTS = 200`
: number of plot points for plotting the function

`M_CONFIDENCE_LEVEL = 0.05`
: Confidence level for hypothesis test

`M_MAX_ITERATIONS = 6000`
: Maximum *Minuit* iterations until abort

M_MAX_X_FIT_ITERATIONS = 2
Maximum additional iterations for *x* fit

M_TOLERANCE = 0.1
Minuit tolerance level

2.1.4 dataset Module

class Dataset(**kwargs*)
Bases: object

The *Dataset* object is a data structure for storing measurement and error data. In this implementation, the *Dataset* has the compulsory field *data*, which is used for storing the measurement data, and another field *cov_mats*, used for storing the covariance matrix for each axis.

There are several ways a *Dataset* can be constructed. The most straightforward way is to specify an input file containing a plain-text representation of the dataset:

```
>>> my_dataset = Dataset(input_file='/path/to/file')
```

or

```
>>> my_dataset = Dataset(input_file=my_file_object)
```

If an *input_file* keyword is provided, all other input is ignored. The *Dataset* plain-text representation format is as follows:

```
# x data
x_1  sigma_x_1
x_2  sigma_x_2  cor_x_12
...  ...        ...       ...
x_N  sigma_x_N  cor_x_1N  ...  cor_x_NN

# y data
y_1  sigma_y_1
y_2  sigma_y_2  cor_y_12
...  ...        ...       ...
y_N  sigma_y_N  cor_y_1N  ...  cor_y_NN
```

Here, the *sigma_...* represents the fully uncorrelated error of the data point and *cor_..._ij* is the correlation coefficient between the *i*-th and *j*-th data point.

Alternatively, field data can be set by passing iterables as keyword arguments. Available keywords for this purpose are:

data : tuple/list of tuples/lists/arrays of floats

a tuple/list of measurement data. Each element of the tuple/list must be iterable and be of the same length. The first element of the **data** tuple/list is assumed to be the *x* data, and the second to be the *y* data:

```
>>> my_dataset = Dataset(data=([0., 1., 2.], [1.23, 3.45, 5.62]))
```

Alternatively, x-y value pairs can also be passed as **data**. The following is equivalent to the above:

```
>>> my_dataset = Dataset(data=([0.0, 1.23], [1.0, 3.45], [2.0, 5.62]))
```

In case the *Dataset* contains two data points, the ordering is ambiguous. In this case, the first ordering (*x* data first, then *y* data) is assumed.

cov_mats : tuple/list of *numpy.matrix* (optional)

a tuple/list of two-dimensional iterables containing the covariance matrices for x and y, in that order. Covariance matrices can be any sort of two-dimensional NxN iterables, assuming N is the number of data points.

```
>>> my_dataset = Dataset(data=([0., 1., 2.], [1.23, 3.45, 5.62]), cov_mats=(my_cov_mat_x, my_cov_mat_
```

This keyword argument can be omitted, in which case covariance matrices of zero are assumed. To specify a covariance matrix for a single axis, replace the other with `None`.

```
>>> my_dataset = Dataset(data=([0., 1., 2.], [1.23, 3.45, 5.62]), cov_mats=(None, my_cov_mat_y))
```

title : string (optional)

the name of the *Dataset*. If omitted, the *Dataset* will be given the generic name 'Untitled Dataset'.

axis_labels : list of strings (optional)

labels for the x and y axes. If omitted, these will be set to `'x'` and `'y'`, respectively.

axis_units : list of strings (optional)

units for the x and y axes. If omitted, these will be assumed to be dimensionless, i.e. the unit will be an empty string.

`axis_labels` = None
axis labels

`cov_mat_is_regular`(*axis*)
Returns *True* if the covariance matrix for an axis is regular and `False` if it is singular.

axis [`'x'` or `'y'`] Axis for which to check for regularity of the covariance matrix.

`cov_mats` = None
list of covariance matrices

`get_axis`(*axis_alias*)
Get axis id from an alias.

axis_alias [string or int] Alias of the axis whose id should be returned. This is for example either `'0'` or `'x'` for the x-axis (id 0).

`get_cov_mat`(*axis*, *fallback_on_singular=None*)
Get the error matrix for an axis.

axis [`'x'` or `'y'`] Axis for which to load the error matrix.

fallback_on_singular [*numpy.matrix* or string (optional)] What to return if the matrix is singular. If this is `None` (default), the matrix is returned anyway. If this is a *numpy.matrix* object or similar, that is returned istead. Alternatively, the shortcuts `'identity'` or `1` and `'zero'` or `0` can be used to return the identity and zero matrix respectively.

`get_data`(*axis*)
Get the measurement data for an axis.

axis [string] Axis for which to get the measurement data. Can be `'x'` or `'y'`.

`get_data_span`(*axis*, *include_error_bars=False*)
Get the data span for an axis. The data span is a tuple (*min*, *max*) containing the smallest and highest coordinates for an axis.

axis [`'x'` or `'y'`] Axis for which to get the data span.

include_error_bars [boolean (optional)] `True` if the returned span should be enlarged to contain the error bars of the smallest and largest datapoints (default: `False`)

`get_formatted`(*format_string='.06e'*, *delimiter='t'*)
Returns the dataset in a plain-text format which is human-readable and can later be used as an input file for the creation of a new *Dataset*. The format is as follows:

```
# x data
x_1  sigma_x_1
x_2  sigma_x_2  cor_x_12
...  ...        ...       ...
x_N  sigma_x_N  cor_x_1N  ...  cor_x_NN

# y data
y_1  sigma_y_1
y_2  sigma_y_2  cor_y_12
...  ...        ...       ...
y_N  sigma_y_N  cor_y_1N  ...  cor_y_NN
```

Here, the `x_i` and `y_i` represent the measurement data, the `sigma_?_i` are the statistical uncertainties of each data point, and the `cor_?_ij` are the correlation coefficients between the *i*-th and *j*-th data point.

If the `x` or `y` errors are not correlated, then the entire correlation coefficient matrix can be omitted. If there are no statistical uncertainties for an axis, the second column can also be omitted. A blank line is required at the end of each data block!

format_string [string (optional)] A format string with which each entry will be rendered. Default is `'.06e'`, which means the numbers are represented in scientific notation with six significant digits.

delimiter [string (optional)] A delimiter used to separate columns in the output.

`get_size`()
Get the size of the *Dataset*. This is equivalent to the length of the *x*-axis data.

`has_correlations`(*axis*)
Returns *True* if the specified axis has correlation data, `False` if not.

axis [`'x'` or `'y'`] Axis for which to check for correlations.

`has_errors`(*axis*)
Returns *True* if the specified axis has statistical error data.

axis [`'x'` or `'y'`] Axis for which to check for error data.

`n_axes` = **None**
dimensionality of the *Dataset*. Currently, only 2D *Datasets* are supported

`n_datapoints` = **None**
number of data points in the *Dataset*

`read_from_file`(*input_file*)
Reads the *Dataset* object from a file.

returns [boolean] `True` if the read succeeded, `False` if not.

`set_cov_mat`(*axis*, *mat*)
Set the error matrix for an axis.

axis [`'x'` or `'y'`] Axis for which to load the error matrix.

mat [*numpy.matrix* or `None`] Error matrix for the axis. Passing `None` unsets the error matrix.

`set_data`(*axis*, *data*)
Set the measurement data for an axis.

axis [`'x'` or `'y'`] Axis for which to set the measurement data.

data [iterable] Measurement data for axis.

`write_formatted`(*file_path*, *format_string='.06e'*, *delimiter='t'*)

Writes the dataset to a plain-text file. For details on the format, see get_formatted.

file_path [string] Path of the file object to write. **WARNING**: *overwrites existing files*!

format_string [string (optional)] A format string with which each entry will be rendered. Default is `'.06e'`, which means the numbers are represented in scientific notation with six significant digits.

delimiter [string (optional)] A delimiter used to separate columns in the output.

`build_dataset`(*xdata*, *ydata*, *cov_mats=None*, ***kwargs*)

This helper function creates a *Dataset* from a series of keyword arguments.

Valid keyword arguments are:

xdata and ydata [list/tuple/*np.array* of floats] These keyword arguments are mandatory and should be iterables containing the measurement data.

cov_mats [`None` or 2-tuple (optional)] This argument defaults to `None`, which means no covariance matrices are used. If covariance matrices are needed, a tuple with two entries (the first for x covariance matrices, the second for y) must be passed.

Each element of this tuple may be either `None` or a NumPy matrix object containing a covariance matrix for the respective axis.

error specification keywords [iterable or numeric (see below)] In addition to covariance matrices, errors can be specified for each axis (x or y) according to a simplified error model.

In this respect, a valid keyword is composed of an axis, an error relativity specification (*abs* or *rel*) and error correlation type (*err* or *cor*). The errors are then set as follows:

1. **For totally uncorrelated errors (*err*):**
   - if keyword argument is iterable, the error list is set to that
   - if keyword argument is a number, an error list with identical entries is generated
2. **For fully correlated errors (*cor*):**
   - keyword argument *must* be a single number. The global correlated error for the axis is then set to that.

So, for example:

```
>>> myDataset = build_dataset(..., yabserr=0.3, yrelcor=0.1)
```

creates a Dataset with an uncorrelated error of 0.3 for each y coordinate and a fully correlated (systematic) error of y of 0.1.

title [string (optional)] The title of the *Dataset*.

basename [string or `None` (optional)] A basename for the *Dataset*. All output files related to this dataset will use this as a basename. If this is `None` (default), the basename will be inferred from the filename.

axis_labels [2-tuple of strings (optional)] a 2-tuple containing the axis labels for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.

axis_units [2-tuple of strings (optional)] a 2-tuple containing the axis units for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.

2.1.5 `file_tools` Module

`parse_column_data`(*file_to_parse, field_order='x,y', delimiter=' ', cov_mat_files=None, title='Untitled Dataset', basename=None, axis_labels=['x', 'y'], axis_units=['', '']*)

Parses a file which contains measurement data in a one-measurement-per-row format. The field (column) order can be specified. It defaults to *x,y*. *Valid field names are 'x*, *y*, *xabserr*, *yabserr*, *xrelerr*, *yrelerr*. Another valid field name is *ignore* which can be used to skip a field.

A certain type of field can appear several times. If this is the case, all specified errors are added in quadrature:

$$\sigma_{\mathrm{tot}} = \sqrt{{\sigma_1}^2 + {\sigma_2}^2 + \ldots}$$

Every valid measurement data file *must* have an *x* and a *y* field.

For more complex error models, errors and correlations may be specified as covariance matrices. If this is desired, then any number of covariance matrices (stored in separate files) may be specified for an axis by using the *cov_mat_files* argument.

Additionally, a delimiter can be specified. If this is a whitespace character or omitted, any sequence of whitespace characters is assumed to separate the data.

file_to_parse [file-like object or string containing a file path] The file to parse.

field_order [string (optional)] A string of comma-separated field names giving the order of the columns in the file. Defaults to `'x,y'`.

delimiter [string (optional)] The field delimiter used in the file. Defaults to any whitespace.

cov_mat_files [*several* (see below, optional)] This argument defaults to `None`, which means no covariance matrices are used. If covariance matrices are needed, a tuple with two entries (the first for *x* covariance matrices, the second for *y*) must be passed.

Each element of this tuple may be either `None`, a file or file-like object, or an iterable containing files and file-like objects. Each file should contain a covariance matrix for the respective axis.

When creating the *Dataset*, all given matrices are summed over.

title [string (optional)] The title of the *Dataset*.

basename [string or `None` (optional)] A basename for the *Dataset*. All output files related to this dataset will use this as a basename. If this is `None` (default), the basename will be inferred from the filename.

axis_labels [2-tuple of strings (optional)] a 2-tuple containing the axis labels for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.

axis_units [2-tuple of strings (optional)] a 2-tuple containing the axis units for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.

return [*Dataset*] A Dataset built from the parsed file.

`parse_matrix_file`(*file_like*, *delimiter=None*)

Read a matrix from a matrix file. The format of the matrix file should be:

```
# comment row
a_11  a_12  ...  a_1M
a_21  a_22  ...  a_2M
...   ...   ...  ...
a_N1  a_N2  ...  a_NM
```

file_like [string or file-like object] File path or file object to read matrix from.

delimiter [None or string (optional)] Column delimiter use in the matrix file. Defaults to None, meaning any whitespace.

2.1.6 fit Module

class Fit(*dataset*, *fit_function*, *external_fcn=<function chi2 at 0x27c76e0>*, *fit_label=None*)
Bases: object

Object representing a fit. This object references the fitted *Dataset*, the fit function and the resulting fit parameters.

Necessary arguments are a *Dataset* object and a fit function (which should be fitted to the *Dataset*). Optionally, an external function *FCN* (whose minima should be located to find the best fit) can be specified. If not given, the *FCN* function defaults to χ^2.

dataset [*Dataset*] A *Dataset* object containing all information about the data

fit_function [function] A user-defined Python function to be fitted to the data. This function's first argument must be the independent variable *x*. All other arguments *must* be named and have default values given. These defaults are used as a starting point for the actual minimization. For example, a simple linear function would be defined like:

```
>>> def linear_2par(x, slope=1, y_intercept=0):
...     return slope * x + y_intercept
```

Be aware that choosing sensible initial values for the parameters is often crucial for a succesful fit, particularly for functions of many parameters.

external_fcn [function (optional)] An external *FCN* (function to minimize). This function must have the following call signature:

```
>>> FCN(xdata, ydata, cov_mat, fit_function, parameter_values)
```

It should return a float. If not specified, the default χ^2 *FCN* is used. This should be sufficient for most fits.

fit_label [LaTeX-formatted string (optional)] A name/label/short description of the fit function. This appears in the legend describing the fitter curve. If omitted, this defaults to the fit function's LaTeX expression.

call_external_fcn(**parameter_names*)
Wrapper for the external *FCN*. Since the actual fit process depends on finding the right parameter values and keeping everything else constant we can use the *Dataset* object to pass known, fixed information to the external *FCN*, varying only the parameter values.

parameter_names [sequence of values] the parameter values at which *FCN* is to be evaluated

current_cov_mat = None
the current covariance matrix used for the *Fit*

dataset = None
this Fit instance's child *Dataset*

do_fit(*quiet=False*, *verbose=False*)
Runs the fit algorithm for this *Fit* object.

First, the *Dataset* is fitted considering only uncertainties in the *y* direction. If the *Dataset* has no uncertainties in the *y* direction, they are assumed to be equal to 1.0 for this preliminary fit, as there is no better information available.

Next, the fit errors in the *x* direction (if they exist) are taken into account by projecting the covariance matrix for the *x* errors onto the *y* covariance matrix. This is done by taking

the first derivative of the fit function in each point and "projecting" the x error onto the resulting tangent to the curve.

This last step is repeater until the change in the error matrix caused by the projection becomes negligible.

quiet [boolean (optional)] Set to `True` if no output should be printed.

verbose [boolean (optional)] Set to `True` if more output should be printed.

`external_fcn` = **None**
the (external) function to be minimized for this *Fit*

`fit_function` = **None**
the fit function used for this *Fit*

`fit_one_iteration`(*verbose=False*)
Instructs the minimizer to do a minimization.

`fix_parameters`(**parameters_to_fix*)
Fix the given parameters so that the minimizer works without them when *do_fit* is called next. Parameters can be given by their names or by their IDs.

`get_current_fit_function`()
This method returns a function object corresponding to the fit function for the current parameter values. The returned function is a function of a single variable.

returns [function] A function of a single variable corresponding to the fit function at the current parameter values.

`get_error_matrix`()
This method returns the covariance matrix of the fit parameters which is obtained by querying the minimizer object for this *Fit*

returns [*numpy.matrix*] The covariance matrix of the parameters.

`get_parameter_errors`(*rounding=False*)
Get the current parameter uncertainties from the minimizer.

rounding [boolean (optional)] Whether or not to round the returned values to significance.

returns [tuple] A tuple of the parameter uncertainties

`get_parameter_values`(*rounding=False*)
Get the current parameter values from the minimizer.

rounding [boolean (optional)] Whether or not to round the returned values to significance.

returns [tuple] A tuple of the parameter values

`latex_parameter_names` = **None**
LaTeX parameter names

`minimizer` = **None**
this *Fit*'s minimizer (*Minuit*)

`number_of_parameters` = **None**
the total number of parameters

`parameter_names` = **None**
the names of the parameters

`print_fit_details`()
prints some fit goodness details

`print_fit_results`()
prints fit results

`print_rounded_fit_parameters()`
prints the fit parameters

`project_x_covariance_matrix()`
Project the x errors from the x covariance matrix onto the total matrix.

This is done elementwise, according to the formula:

$$C_{tot,ij} = C_{y,ij} + C_{x,ij} \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j}$$

`release_parameters(*parameters_to_fix)`
Release the given parameters so that the minimizer begins to work with them when *do_fit* is called next. Parameters can be given by their names or by their IDs. If no arguments are provied, then release all parameters.

`set_parameters(*args, **kwargs)`
Sets the parameter values (and optionally errors) for this fit. This is usually called just before the fit is done, to establish the initial parameters. If a parameter error is omitted, it is set to 1/1000th of the parameter values themselves. If the default value of the parameter is 0, it is set, by exception, to 0.001.

This method accepts up to two positional arguments and several keyword arguments.

args[0] [tuple/list of floats (optional)] The first positional argument is expected to be a tuple/list containing the parameter values.

args[1] [tuple/list of floats (optional)] The second positional argument is expected to be a tuple/list of parameter errors, which can also be set as an approximate estimate of the problem's uncertainty.

no_warning [boolean (optional)] Whether to issue warnings (`False`) or not (`True`) when communicating with the minimizer fails. Defaults to `False`.

Valid keyword argument names are parameter names. The keyword arguments themselves may be floats (parameter values) or 2-tuples containing the parameter values and the parameter error in that order:

<parameter_name> [float or 2-tuple of floats (optional)] Set the parameter with the name <'parameter_name'> to the value given. If a 2-tuple is given, the first element is understood to be the value and the second to be the parameter error.

`xdata = None`
the x coordinates of the data points used for this *Fit*

`ydata = None`
the y coordinates of the data points used for this *Fit*

`chi2(xdata, ydata, cov_mat, fit_function, parameter_values)`
A simple χ^2 implementation. Calculates χ^2 according to the formula:

$$\chi^2 = \lambda^T C^{-1} \lambda$$

Here, λ is the residual vector $\lambda = \vec{y} - \vec{f}(\vec{x})$ and C is the covariance matrix.

xdata [iterable] The x measurement data

ydata [iterable] The y measurement data

cov_mat [*numpy.matrix*] The total covariance matrix

fit_function [function] The fit function $f(x)$

parameter_values [list/tuple] The values of the parameters at which $f(x)$ should be evaluated.

`round_to_significance`(*value*, *error*, *significance=2*)
Rounds the error to the established number of significant digits, then rounds the value to the same order of magnitude as the error.

value [float] value to round to significance

error [float] uncertainty of the value

significance [int (optional)] number of significant digits of the error to consider

2.1.7 `function_library` Module

2.1.8 `function_tools` Module

`ASCII`(***kwargs*)
Optional decorator for fit functions. This overrides a FitFunction's plain-text (ASCII) attributes. The new values for these attributes must be passed as keyword arguments to the decorator. Possible arguments:

name [string] Plain-text representation of the function name.

parameter_names [list of strings] List of plain-text representations of the function's arguments. The length of this list must be equal to the function's argument number. The argument names should be in the same order as in the function definition.

x_name [string] Plain-text representation of the independent variable's name.

expression [string] Plain-text-formatted expression representing the function's formula.

class `FitFunction`(*f*)
Decorator class for fit functions. If a function definition is decorated using this class, some information is collected about the function which is relevant to the fitting process, such as the number of parameters, their names and default values. Some details pertaining to display and representation are also set, such as LaTeX representations of the parameter names and the function name. Other decorators can be applied to a function object to specify things such as a LaTeX or plain-text expression for the fit function.

`derive_by_parameters`(*x_0*, *derivative_spacing*, *parameter_list*)
Returns the gradient of *func* with respect to its parameters, i.e. with respect to every variable of *func* except the first one.

`derive_by_x`(*x_0*, *derivative_spacing*, *parameter_list*)
If *x_0* is iterable, gives the array of derivatives of a function $\mathtt{f}(\mathtt{x}, \mathtt{par}_1, \mathtt{par}_2, \ldots)$ around $\mathtt{x} = \mathtt{x}_i$ at every $\mathtt{x}_i$ in $\vec{\mathtt{x}}$. If *x_0* is not iterable, gives the derivative of a function $\mathtt{f}(\mathtt{x}, \mathtt{par}_1, \mathtt{par}_2, \ldots)$ around $\mathtt{x} = \mathtt{x\_0}$.

`expression` = **None**
a math expression (string) representing the function's result

`get_function_equation`(*equation_format='latex'*, *equation_type='full'*, *ensuremath=True*)
Returns a string representing the function equation. Supported formats are LaTeX and ASCII inline math. Note that LaTeX math is wrapped by default in an `\ensuremath{}` expression. If this is not desired behaviour, the flag `ensuremath` can be set to `False`.

equation_format [string (optional)] Can be either "latex" (default) or "ascii".

equation_type [string (optional)] Can be either "full" (default), "short" or "name". A "name"-type equation returns a representation of the function name:

```
f
```

A "short"-type equation limits itself to the function name and variables:

```
f(x, par1, par2)
```

A "full"-type equation includes the expression which the function calculates:

```
f(x, par1, par2) = par1 * x + par2
```

ensuremath [boolean (optional)] If a LaTeX math equation is requested, `True` (default) will wrap the resulting expression in an `\ensuremath{}` tag. Otherwise, no wrapping is done.

`latex_expression` = None
a LaTeX math expression, the function's result

`latex_name` = None
The function's name in LaTeX

`latex_parameter_names` = None
A list of parameter names in LaTeX

`latex_x_name` = None
A LaTeX symbol for the independent variable.

`name` = None
The name of the function

`number_of_parameters` = None
The number of parameters

`parameter_defaults` = None
The default values of the parameters

`parameter_names` = None
The names of the parameters

`sourcelines` = None
string object holding the source code for the fit-function

`x_name` = None
The name given to the independent variable

`LaTeX`(***kwargs*)
Optional decorator for fit functions. This overrides a FitFunction's *latex_* attributes. The new values for the *latex_* attributes must be passed as keyword arguments to the decorator. Possible arguments:

name [string] LaTeX representation of the function name.

parameter_names [list of strings] List of LaTeX representations of the function's arguments. The length of this list must be equal to the function's argument number. The argument names should be in the same order as in the function definition.

x_name [string] LaTeX representation of the independent variable's name.

expression [string] LaTeX-formatted expression representing the function's formula.

`derivative`(*func*, *derive_by_index*, *variables_tuple*, *derivative_spacing*)
Gives $\frac{\partial f}{\partial x_k}$ for $f = f(x_0, x_1, \ldots)$. *func* is f, *variables_tuple* is $\{x_i\}$ and *derive_by_index* is k.

`outer_product`(*input_array*)
Takes a *NumPy* array and returns the outer (dyadic, Kronecker) product with itself. If *input_array* is a vector $\mathbf{x}$, this returns $\mathbf{x}\mathbf{x}^T$.

2.1.9 `latex_tools` Module

`ascii_to_latex_math`(*str_ascii*, *monospace=True*, *ensuremath=True*)

Escapes certain characters in an ASCII input string so that the result can be included in math mode without error.

str_ascii [string] A plain-text string containing characters to be escaped for LaTeX math mode.

monospace [boolean (optional)] Whether to render the whole expression as monospace. Defaults to `True`.

ensuremath [boolean (optional)] If this is `True`, the resulting formula is wrapped in an `\ensuremath{}` tag. Defaults to `True`.

2.1.10 `minuit` Module

`D_MATRIX_ERROR = {0: 'Error matrix not calculated', 1: 'Error matrix approximate!', 2: 'Error matrix forced positive d`

Error matrix status codes

class `Minuit`(*number_of_parameters*, *function_to_minimize*, *parameter_names*, *start_parameters*, *parameter_errors*, *quiet=True*, *verbose=False*)

A class for communicating with ROOT's function minimizer tool Minuit.

`FCN_wrapper`(*number_of_parameters*, *derivatives*, *f*, *parameters*, *internal_flag*)

This is actually a function called in *ROOT* and acting as a C wrapper for our *FCN*, which is implemented in Python.

This function is called by *Minuit* several times during a fit. It doesn't return anything but modifies one of its arguments (*f*). This is *ugly*, but it's how *ROOT*'s `TMinuit` works. Its argument structure is fixed and determined by *Minuit*:

number_of_parameters [int] The number of parameters of the current fit

derivatives [C array] If the user chooses to calculate the first derivative of the function inside the *FCN*, this value should be written here. This interface to *Minuit* ignores this derivative, however, so calculating this inside the *FCN* has no effect (yet).

f [C array] The desired function value is in f[0] after execution.

parameters [C array] A C array of parameters. Is cast to a Python list

internal_flag [int] A flag allowing for different behaviour of the function. Can be any integer from 1 (initial run) to 4(normal run). See *Minuit*'s specification.

`fix_parameter`(*parameter_number*)

Fix parameter number <*parameter_number*>.

parameter_number [int] Number of the parameter to fix.

`function_to_minimize` = **None**

the actual *FCN* called in `FCN_wrapper`

`get_chi2_probability`(*n_deg_of_freedom*)

Returns the probability that an observed χ^2 exceeds the calculated value of χ^2 for this fit by chance, even for a correct model. In other words, returns the probability that a worse fit of the model to the data exists. If this is a small value (typically <5%), this means the fit is pretty bad. For values below this threshold, the model very probably does not fit the data.

n_def_of_freedom [int] The number of degrees of freedom. This is typically $n_extdatapoints - n_extparameters$.

`get_contour`(*parameter1*, *parameter2*, *n_points=20*)

Returns a list of points (2-tuples) representing a sampling of the 1σ contour of the TMinuit fit. The `FCN` has to be minimized before calling this.

parameter1 [int] ID of the parameter to be displayed on the *x*-axis.

parameter2 [int] ID of the parameter to be displayed on the *y*-axis.

n_points [int (optional)] number of points used to draw the contour. Default is 20.

returns [2-tuple of tuples] a 2-tuple (x, y) containing `n_points+1` points sampled along the contour. The first point is repeated at the end of the list to generate a closed contour.

`get_error_matrix()`
Retrieves the parameter error matrix from TMinuit.

return : *numpy.matrix*

`get_fit_info(`*info*`)`
Retrieves other info from *Minuit*.

info [string] Information about the fit to retrieve. This can be any of the following:

- `'fcn'`: *FCN* value at minimum,
- `'edm'`: estimated distance to minimum
- `'err_def'`: *Minuit* error matrix status code
- `'status_code'`: *Minuit* general status code

`get_parameter_errors()`
Retrieves the parameter errors from TMinuit.

return [tuple] Current *Minuit* parameter errors

`get_parameter_info()`
Retrieves parameter information from TMinuit.

return [list of tuples] `(parameter_name, parameter_val, parameter_error)`

`get_parameter_name(`*parameter_nr*`)`
Gets the name of parameter number `parameter_nr`

parameter_nr [int] Number of the parameter whose name to get.

`get_parameter_values()`
Retrieves the parameter values from TMinuit.

return [tuple] Current *Minuit* parameter values

`max_iterations` = **None**
maximum number of iterations until `TMinuit` gives up

`minimize(`*log_print_level=3*`)`
Do the minimization. This calls *Minuit*'s algorithms `MIGRAD` for minimization and `HESSE` for computing/checking the parameter error matrix.

`number_of_parameters` = **None**
number of parameters to minimize for

`release_parameter(`*parameter_number*`)`
Release parameter number <*parameter_number*>.

parameter_number [int] Number of the parameter to release.

`reset()`
Execute TMinuit's *mnrset* method.

`set_err(`*up_value=1.0*`)`
Sets the `UP` value for Minuit.

up_value [float (optional, default: 1.0)] This is the value by which *FCN* is expected to change.

`set_parameter_errors`(*parameter_errors=None*)
Sets the fit parameter errors. If parameter_values='None', sets the error to 1% of the parameter value.

`set_parameter_names`(*parameter_names*)
Sets the fit parameters. If parameter_values='None', tries to infer defaults from the function_to_minimize.

`set_parameter_values`(*parameter_values*)
Sets the fit parameters. If parameter_values='None', tries to infer defaults from the function_to_minimize.

`set_print_level`(*print_level=1*)
Sets the print level for Minuit.

print_level [int (optional, default: 1 (frugal output))] Tells `TMinuit` how much output to generate. The higher this value, the more output it generates.

`set_strategy`(*strategy_id=1*)
Sets the strategy Minuit.

strategy_id [int (optional, default: 1 (optimized))] Tells `TMinuit` to use a certain strategy. Refer to `TMinuit`'s documentation for available strategies.

`tolerance` = **None**
`TMinuit` tolerance

`update_parameter_data`(*show_warnings=False*)
(Re-)Sets the parameter names, values and step size on the C++ side of Minuit.

`P_DETAIL_LEVEL` = **1**
default level of detail for TMinuit's output (typical range: -1 to 3, default: 1)

2.1.11 `numeric_tools` Module

`cor_to_cov`(*cor_mat*, *error_list*)
Converts a correlation matrix to a covariance matrix according to the formula

$$\mathrm{Cov}_{ij} = \mathrm{Cor}_{ij}\,\sigma_i\,\sigma_j$$

cor_mat [*numpy.matrix*] The correlation matrix to convert.

error_list [sequence of floats] A sequence of statistical errors. Must be of the same length as the diagonal of *cor_mat*.

`cov_to_cor`(*cov_mat*)
Converts a covariance matrix to a correlation matrix according to the formula

$$\mathrm{Cor}_{ij} = \frac{\mathrm{Cov}_{ij}}{\sqrt{\mathrm{Cov}_{ii}\,\mathrm{Cov}_{jj}}}$$

cov_mat [*numpy.matrix*] The covariance matrix to convert.

`extract_statistical_errors`(*cov_mat*)
Extracts the statistical errors from a covariance matrix. This means it returns the (elementwise) square root of the diagonal entries

cov_mat The covariance matrix to extract errors from. Type: *numpy.matrix*

`make_symmetric_lower`(*mat*)
Copies the matrix entries below the main diagonal to the upper triangle half of the matrix. Leaves the diagonal unchanged. Returns a *NumPy* matrix object.

mat [*numpy.matrix*] A lower diagonal matrix.

returns [*numpy.matrix*] The lower triangle matrix.

`zero_pad_lower_triangle`(*triangle_list*)
Converts a list of lists into a lower triangle matrix. The list members should be lists of increasing length from 1 to N, N being the dimension of the resulting lower triangle matrix. Returns a *NumPy* matrix object.

For example:

```
>>> zero_pad_lower_triangle([ [1.0], [0.2, 1.0], [0.01, 0.4, 3.0] ])
matrix([[ 1.  ,  0.  ,  0.  ],
        [ 0.2 ,  1.  ,  0.  ],
        [ 0.01,  0.4 ,  3.  ]])
```

triangle_list [list] A list containing lists of increasing length.

returns [*numpy.matrix*] The lower triangle matrix.

2.1.12 `plot` Module

class `Plot`(**fits*, ***kwargs*)
Bases: `object`

`axis_labels` = **None**
axis labels

`compute_plot_range`(*include_error_bars=True*)
Compute the span of all child datasets and sets the plot range to that

`draw_fit_parameters_box`(*plot_spec=0*)
Draw the parameter box to the canvas

plot_spec [int, list of ints, string or None (optional, default: 0)] Specify the plot id of the plot for which to draw the parameters. Passing 0 will only draw the parameter box for the first plot, and so on. Passing a list of ints will only draw the parameters for plot ids inside the list. Passing `'all'` will print parameters for all plots. Passing `None` will return immediately doing nothing.

`draw_legend`()
Draw the plot legend to the canvas

`extend_span`(*axis*, *new_span*)
Expand the span of the current plot.

This method extends the current plot span to include *new_span*

`fits` = **None**
list of 'Fit's to plot

`init_plots`()
Initialize the plots for each fit.

`plot`(*p_id*, *show_data=True*, *show_function=True*)
Plot the *Fit* object with the number *p_id* to its figure.

`plot_all`(*show_info_for='all'*, *show_data_for='all'*, *show_function_for='all'*)
Plot every *Fit* object to its figure.

`plot_range` = **None**
plot range

`plot_style` = **None**
plot style

`save`(*output_file*)
Save the *Plot* to a file.

`show`()
Show the *Plot* in a matplotlib interactive window.

`show_legend` = **None**
whether to show the plot legend (`True`) or not (`False`)

class `PlotStyle`
Class for specifying a style for a specific plot. This object stores a progression of marker and line types and colors, as well as preferences relating to point size and label size. These can be overriden by overwriting the instance variables directly. A series of *get_...* methods are provided which go through these lists cyclically.

`get_line`(*idm*)
Get a specific line type. This runs cyclically through the defined defaults.

`get_linecolor`(*idm*)
Get a specific line color. This runs cyclically through the defined defaults.

`get_marker`(*idm*)
Get a specific marker type. This runs cyclically through the defined defaults.

`get_markercolor`(*idm*)
Get a specific marker color. This runs cyclically through the defined defaults.

`get_pointsize`(*idm*)
Get a specific point size. This runs cyclically through the defined defaults.

`label_to_latex`(*label*)
Generates a simple LaTeX-formatted label from a plain-text label. This treats isolated characters and words beginning with a backslash as mathematical expressions and surround them with $ signs accordingly.

label [string] Plain-text string to convert to LaTeX.

`pad_span`(*span*, *pad_coeff=1*, *additional_pad=None*)
Enlarges the interval *span* (list of two floats) symmetrically around its center to length *pad_coeff*. Optionally, an *additional_pad* argument can be specified. The returned span is then additionally enlarged by that amount.

additional_pad can also be a list of two floats which specifies an asymmetric amount by which to enlarge the span. Note that in this case, positive entries in *additional_pad* will enlarge the span (move the interval end away from the interval's center) and negative amounts will shorten it (move the interval end towards the interval's center).

2.1.13 `stream` Module

class `StreamDup`(*out_file*, *suppress_stdout=False*)
Bases: `object`

Object for simultaneous logging to stdout and files. This object provides a file/like object for the outout to be written to. Writing to this object will write to stdout (usually the console) and to a file.

out_file [file path or file-like object or list of file paths ...] File(s) to which to log the output, along with stdout. If a file exists on disk, it will be appended to.

suppress_stdout [boolean] Whether to log to stdout simultaneously (`False`) or suppress output to stdout (`True`). Default to `False`.

`fileno()`
Returns the file handler id of the main (first) output file.

`flush()`

`write(`*message*`)`

`write_timestamp(`*prefix*`)`

`write_to_file(`*message*`)`

`write_to_stdout(`*message*, *check_if_suppressed=False*`)`
Explicitly write to stdout. This method will not check by default whether `suppress_stdout` is set for this *StreamDup*. If `check_if_suppressed` is explicitly set to `True`, then this check occurs.

2.1.14 Indices and tables

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

H

I

K

L

M

N

O

P

R

S

T

U

W

X

Y

Z