Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

HZB Helmholtz
Zentrum Berlin

Bachelorthesis

Entwicklung einer interaktiven Analyse-Software für zeitaufgelöste Spektroskopiemessdaten

vorgelegt am 5. August 2022

•

Fachbereich Duales Studium Wirtschaft / Technik
Hochschule für Wirtschaft und Recht Berlin

Name:	Alexander Matthias Schmidt
Ausbildungsbetrieb:	Helmholtz-Zentrum Berlin für Materialien und Energie GmbH
Studienbereich:	Technik
Fachrichtung:	Informatik
Studiengang:	Informatik
Studienjahrgang:	2019
Erstgutachter:	Dr. Christoph Merschjann
Zweitgutachter:	Prof. Dr. Rainer Höhne

Abstract

In der vorliegenden Bachelorarbeit wird die Entwicklung einer interaktiven Applikation zur Analyse und Darstellung von zeitaufgelösten Spektroskopiemessdaten aus materialwissenschaftlichen Experimenten dargestellt und diskutiert. Ausgangspunkt war dabei ein Projekt, in dem Teilkomponenten zur grafischen Visualisierung von spektroskopischen Datensätzen und zum Anlegen und Abspeichern mathematischer Modelle bereits implementiert waren. Gegenstand der hier erörterten Projektphase war die Entwicklung eines ersten funktionsfähigen Prototypen unter Verwendung dieser Komponenten.

Ein besonderer Fokus lag dabei auf der Implementierung eines Fit-Algorithmus zur Optimierung der mathematischen Modelle und einer darauf zugeschnittenen interaktiven Bedienoberfläche, die eine nutzerfreundliche Bedienung und Manipulation aller relevanten Parameter erlauben sollte.

Die Funktionsfähigkeit der implementierten Anwendung wurde mit dem erfolgreichen Test mit einem zuvor dafür bestimmten Beispieldatensatz gezeigt. Der erstellte Prototyp bietet somit einen gute Ausgangspunkt für die Weiterentwicklung zu einer für Materialforscher*innen hilfreichen Analyse-Software für zeitaufgelöste Daten.

Inhaltsverzeichnis

Abstract **I**

Inhaltsverzeichnis **III**

Abbildungsverzeichnis **IV**

Abkürzungsverzeichnis **V**

1 Einleitung **1**
1.1 Das Projekt UDATTS 1
1.2 Physikalischer Hintergrund 2
1.3 Mathematischer Hintergrund 4
1.4 Zielsetzung 5

2 Anforderungsanalyse **7**
2.1 Funktionale Anforderungen 7
2.2 Nichtfunktionale Anforderungen 9

3 Entwurf **10**
3.1 Aufbau und Gestaltung der Bedienoberflächen 10
3.2 Anwendungs-Workflow 12
3.3 MVC-Architektur 14

4 Implementierung **17**
4.1 Haupt-Modul 18
4.2 Projekt-Modul 20
4.2.1 Der Projektbaum 20
4.2.1.1 Datenimport und -darstellung 21
4.2.1.2 Aufruf der Model-Designer 21
4.2.2 Speichern und Laden von Projektdateien 22
4.2.2.1 Abspeichern einer Projektdatei 23
4.2.2.2 Öffnen einer Projektdatei 25
4.3 Fit-Modul 27
4.3.1 Das Fit-Control-Dashboard 27
4.3.1.1 Der Model-Parameter-Tree-View 28
4.3.1.2 Der Solver-Parameter-View 31
4.3.1.3 Der Fit-Parameter-View 31

4.3.2 Die Fit-Klasse 32
4.3.2.1 Evaluieren der Models 32
4.3.2.2 Der Fit-Algorithmus 34
4.3.3 PyQt-Signale beim Fitten 38

5 Test mit Beispieldatensatz **41**

6 Fazit **44**
6.1 Evaluation 44
6.2 Ausblick 46

Literaturverzeichnis **47**

Ehrenwörtliche Erklärung **49**

Abbildungsverzeichnis

1 GUI der Software Glotaran 11
2 Anwendungs-Workflow 13
3 MVC-Architektur - Ursprünglicher Entwurf 16
4 MVC-Architektur - Tatsächliche Umsetzung 18
5 Hauptfenster von UDATTS 19
6 Datenimport über den Projektbaum 21
7 Signal-Austausch beim Evaluieren oder Fitten 38
8 Vergleich Datensatz vs. Modell 41
9 Ergebnisse des ersten Fit-Testdurchlaufs 42
10 Fit-Ergebnisse nach Fehlerkorrektur 43

Abkürzungsverzeichnis

API Application Programming Interface

GUI Graphical User Interface

HDF Hierarchical Data Format

HZB Helmholtz-Zentrum Berlin für Materialien und Energie

IRS Instrument Response Function

JSON JavaScript Object Notation

MVC Model-View-Controller

NetCDF Network Common Data Format

ODE Ordinary Differential Equation

UDATTS Unified Data Analysis Tool for Transient Spectroscopy

XUV Extreme-Ultraviolet

1 Einleitung

Am Helmholtz-Zentrum Berlin für Materialien und Energie (HZB) werden am Institut für Angewandte Materialforschung unter anderem auch Messexperimente mit den Methoden der zeitaufgelösten Spektroskopie durchgeführt, deren Ziel es ist, mehr über die physikalischen Eigenschaften bestimmter untersuchter Materialien zu erfahren.

Die Ergebnisse solcher Untersuchungen sind allgemein für die theoretische Materialwissenschaft, aber auch für viele praktische Anwendungen relevant, beispielsweise Weiterentwicklungen in der Solarzellentechnologie oder die Entwicklung und Optimierung von Materialien für die Photokatalyse.

Für die Analyse der Spektroskopiemessdaten werden mathematische Modelle erstellt und diese dann mittels eines Fit-Algorithmus optimiert, so dass sie eine möglichst exakte Modellierung der tatsächlich gemessenen Daten ergeben.

Zur Berechnung dieser Modelle und dem anschließenden Fitten werden von den Wissenschaftler*innen oft eigene Skripte z.B. in Python oder MATLAB geschrieben. Es gibt allerdings bislang keine vereinheitlichte, vom konkreten Experiment unabhängige Standardsoftware für diese Art von Experimenten. Zudem ist insbesondere das Fitten relativ zeitintensiv. Somit geht den Forschenden aktuell viel Zeit durch das Programmieren geeigneter Skripte und deren Ausführung verloren.

1.1 Das Projekt UDATTS

Ausgehend von dieser Problematik hat Dr. Christoph Merschjann bereits 2018 die Idee zu einer interaktiven Analyse-Software für zeitaufgelöste Spektroskopiemessdaten entwickelt, die den Arbeitstitel Unified Data Analysis Tool for Transient Spectroscopy (UDATTS) trägt.

Diese Applikation wird seitdem von ihm gemeinsam mit Mitarbeiter*innen des HZB, darunter anderen Studierenden aus der IT-Abteilung unter Verwendung der Programmiersprache Python entwickelt.

Aufgrund der Komplexität dessen, was die Software können soll, sowie mehrerer Refaktorisierungen und der mehrfachen Übergaben unter den Entwickler*innen hat sich eine beträchtliche Menge an Legacy Code angesammelt.

Der Stand zu Beginn der in der vorliegenden Bachelorarbeit beschriebenen Projektphase ist, dass zwar einzelne Bestandteile der UDATTS-Software bereits implementiert sind, es aber noch keine funktionsfähige Gesamtanwendung gibt. Insbesondere wird der Fit-Algorithmus bislang noch in einem von der Applikation völlig separaten MATLAB-Skript ausgeführt.

1.2 Physikalischer Hintergrund

In der zeitaufgelösten Spektroskopie wird allgemein die Änderung einer spektralen Messgröße (z.B. Lichtabsorption oder Emission von Elektronen mit unterschiedlicher kinetischer Energie) in Abhängigkeit der Zeit gemessen. Dazu wird die zu untersuchende Probe zu einem definierten Zeitpunkt mit einem Anregungspuls (engl. „pump pulse“) angeregt. Häufig kommen hierbei sehr kurze Laserpulse zum Einsatz (ultraschnelle oder Femtosekundenspektroskopie). Die durch den Anregungspuls verursachte Änderung des Materials wird zeitlich relativ dazu mit einem sogenannten Abfragepuls („probe pulse“) vermessen. (Vgl. zu diesem Abschnitt [Lak17])

Im Folgenden wird dies am Beispiel des designierten Testdatensatzes (vgl. [BWKA16]) stark vereinfacht erläutert. In diesem Fall handelt es sich um ein Experiment der zeitaufgelösten Photoelektronenspektroskopie.

Dabei werden durch den Anregungspuls (hier ein Femtosekunden-Laserpuls aus dem sichtbaren Spektralbereich) Elektronen im Material auf ein energetisch höheres Niveau angehoben. Durch den zeitlich verzögerten Abfragepuls (ein Laserpuls mit sehr hoher Photonenenergie im Extreme-Ultraviolet (XUV)-Bereich) werden Elektronen aus der Materialoberfläche gelöst, deren kinetische Energie mittels eines Photoelektronenspektrometers gemessen wird. Genauer gesagt wird zu jedem Messzeitpunkt ein Energiespektrum, also eine Häufigkeitsverteilung von Energiewerten bei den Elektronen gemessen. Ein Messdatensatz lässt sich daher als eine 2D-Matrix mit den Achsen *Zeit* und *Energie* darstellen.

Nach den Prinzipien der chemischen Kinetik, einem Teilgebiet der physikalischen Chemie, können die Energielevel der Elektronen hier als diskrete Anzahl von Zuständen postuliert werden (vgl. [KFW09], S. 735 ff.). Zu Beginn der Messung (t_0) befinden sich alle Elektronen im Grundzustand und durchlaufen dann nach Beschuss

mit dem Anregungspuls mehrere (hier 5) weitere Energiezustände („States“). Zu jedem gemessenen Zeitpunkt t_x befindet sich also ein gewisser Anteil der Elektronen in Zustand $Z_0, Z_1 \dots Z_5$, was als Anteile von 1 bzw. Dezimalzahlen, die in Summe 1 ergeben, dargestellt werden kann.

Die Veränderung dieser Anteile über die Zeit kann mathematisch durch ein Gleichungssystem sogenannter Ratengleichungen modelliert werden, die als Ergebnis eine 2D-Matrix mit den Dimensionen *Zustände* und *Zeit* liefert, das „kinetische Modell“. Bei den Ratengleichungen handelt es sich um Differentialgleichungen, mit denen die Übergangsraten von einem Zustand in den anderen berechnet werden (vgl. [KFW09], S. 735 ff.). Im Fall des Testdatensatzes sieht das Gleichungssystem des „kinetischen Modells“ beispielsweise folgendermaßen aus:

$$\frac{\mathrm{d}[^1\mathrm{A}_1]}{\mathrm{d}t} = -P(t)[^1\mathrm{A}_1] + k_1[^1\mathrm{MLCT}_{\mathrm{cold}}] \tag{1.1}$$

$$\frac{\mathrm{d}[^1\mathrm{MLCT}]}{\mathrm{d}t} = P(t)[^1\mathrm{A}_1] - k_1[^1\mathrm{MLCT}] \tag{1.2}$$

$$\frac{\mathrm{d}[^3\mathrm{MLCT}_{\mathrm{hot}}]}{\mathrm{d}t} = k_1[^1\mathrm{MLCT}] - k_2[^3\mathrm{MLCT}_{\mathrm{hot}}] \tag{1.3}$$

$$\frac{\mathrm{d}[^3\mathrm{MLCT}_{\mathrm{cold}}]}{\mathrm{d}t} = k_2[^3\mathrm{MLCT}_{\mathrm{hot}}] - k_3[^3\mathrm{MLCT}_{\mathrm{cold}}] \tag{1.4}$$

$$\frac{\mathrm{d}[\mathrm{Q}_0]}{\mathrm{d}t} = -k_{\mathrm{Q}_0}[\mathrm{Q}_0] + P(t)[^1\mathrm{A}_1] \tag{1.5}$$

$$\frac{\mathrm{d}[\mathrm{CC}]}{\mathrm{d}t} = \mathrm{A}_{\mathrm{CC}}\frac{\mathrm{d}[P(t)]}{\mathrm{d}t} \tag{1.6}$$

Eckige Klammern bezeichnen hier die Konzentration von Elektronen in einem bestimmten Zustand. A_1 ist der Grundzustand, $MLCT$ bedeutet hier „metal to ligand charge transfer“, also ein Ladungstransfer von Metall zu Ligand; diese Art Zustand kann wiederum als sogenannter „Singlet“- (hochgestellte 1) oder „Triplet“-Zustand (hochgestellte 3) vorkommen und letzterer „heiß“- oder „kalt“ sein (vgl. [BWKA16]). Bei den indexierten k_n handelt es sich um die Ratenkoeffizienten. Die Gleichungen 1.5 und 1.6 spielen jeweils eine Sonderrolle. Erstere modelliert einen in diesem Fall auftretenden zusätzlichen Zustand, der gewissermaßen parallel neben den anderen herläuft. Letztere stellt die Kreuzkorrelation („cross correlation“) mit dem Anregungspuls dar.

Im obigen Fall ist das „kinetische Modell“ ausschließlich durch Differentialgleichungen gegeben, es gibt jedoch auch andere Fälle in denen es durch geschlossene Ausdrücke gegeben ist. Im Kontext „kinetischer Modelle“ in UDATTS wird der erste Fall als „implizit“, der zweite als „explizit“ bezeichnet. In der aktuellen Implementierung des `KineticModel` kann dieses bisher nur implizit eingegeben und verarbeitet werden.

Ausgehend von der Annahme, dass jeder Zustand mit einem bestimmten Spektrum korreliert, kann ein weiteres mathematisches Modell erstellt werden, das „spektrale Modell“. Dabei wird jedem Zustand eine bestimmte energetische Verteilung zugeschrieben, die aus mehreren Funktionskomponenten bestehen kann, z.B. Gauß'schen Normalverteilungen, Cauchy-Verteilungen (Lorentzkurve) oder Gleichverteilungen. Das „spektrale Modell“ liefert als Ergebnis eine 2D-Matrix mit den Dimensionen *Zustände* und *Energie*.

Beim „spektralen Modell“ wird in UDATTS zwischen „impliziten“ und „expliziten“ Komponente unterschieden. Diese Begriffe bedeuten hier jedoch etwas Anderes als beim „kinetischen Modell“: explizit heißt hier, dass die Komponenten als Funktionswerte berechnet werden, implizit dagegen bedeutet, dass sie aus dem Datensatz und dem „kinetischen Modell“ berechnet werden. Aktuell sind im `SpectralModel` nur explizite Komponenten eingebbar und berechenbar.

Nach Anwendung weiterer mathematischer Verfahren, die die aus den Messbedingungen herrührenden Ungenauigkeiten beseitigen sollen, kann durch Mutliplikation der beiden Modellmatrizen ein vereinheitlichtes, spektrotemporales Modell errechnet werden, dass die gleiche Form hat wie der Messdatensatz: eine 2D-Matrix mit den Dimensionen *Zeit* und *Energie*.

Da es sich dabei um ein sogenanntes parametrisiertes Modell handelt, kann dieses durch geeignete Manipulation der Modellparameter mittels eines Fit-Algorithums optimiert werden. Das Ziel ist dabei eine möglichst akkurate mathematische Darstellung der Messdaten, aus der dann materialwissenschaftliche Erkenntnisse gewonnen werden können.

1.3 Mathematischer Hintergrund

Im Kontext dieser Bachelorarbeit ist mit Fit-Algorithmus eine Form der mathematischen Optimierung gemeint. Die Bezeichnung „Fit“ (vom engl. Wort für Anpassung) bezieht sich dabei allgemein auf das Anpassen der Parameter einer Funktion, so

dass diese möglichst nah an eine Menge von (Mess-)Datenpunkten kommt, also mit anderen Worten der Fehler dieser Funktion gegenüber den Daten minimiert wird. In der Mathematik geläufige Beispiele dafür sind die lineare und non-lineare Regression, bei der eine Ausgleichsgerade bzw. Ausgleichskurve gesucht wird, deren durchschnittlicher Abstand von den Datenpunkten am geringsten ist. In der mathematischen Optimierung wird die Funktion, die zu optimieren ist, das heißt für die ein (globales oder lokales) Minimum oder Maximum gesucht wird, als Zielfunktion (engl. „objective function“) bezeichnet. (Vgl. zu diesem Abschnitt [BZ11], [Pie17] und [uJS04].

Im Fall von UDATTS ist das, was minimiert werden soll, die Differenz zwischen Datensatz(-Matrix) und Modell(-Matrix), die als Residuen(-Matrix) bezeichnet wird. Allerdings kann mit den gängigen mathematischen Optimierungsverfahren nur eine Funktion, die ein skalares Ergebnis liefert minimiert werden. Die Residuenmatrix muss daher noch mittels einer sogenannten Verlustfunktion (engl. „loss function“) auf eine nicht-negative reelle Zahl abgebildet werden. Eine gängige Art von Verlustfunktion ist die „Summe der Quadrate“: das heißt alle Elemente der Residuenmatrix werden quadriert und dann aufsummiert.

Im Kontext von UDATTS kann die Zielfunktion („objective function“) daher als Komposition aus der Residuenfunktion („residual function“) und der Verlustfunktion(„loss function“) vorgestellt werden:

$$objective\_fun(x_0) = loss\_fun(residual\_fun(x_0)$$

Dabei stellt x_0 den Parametervektor dar, in dem alle Modell-Parameter zusammengefasst sind. Dieser Vektor soll vom Fit-Algorithmus solange iterativ verändert werden, bis ein lokales Minimum der gefunden ist. Für diese Optimierung kommt die SciPy-Funktion `opimize.minimize` (siehe [The08]) zum Einsatz, wie später noch erläutert wird.

1.4 Zielsetzung

Das primäre Ziel der hier behandelten Projektphase ist, eine erste Version der UDATTS-Applikation zu implementieren, die den gesamten Soll-Workflow ermöglicht: vom Import und der visuellen Anzeige der Messdaten über die mathematische Modellierung und den Fit-Algorithmus bis hin zur Darstellung und dem Abspeichern der Ergebnisse.

Dazu ist es erstens notwendig die bisher entwickelten Programmkomponenten in einer Hauptanwendung einschließlich einer Haupt-Graphical User Interface (GUI) miteinander zu verknüpfen und interagieren zu lassen.

Zweitens müssen einige entscheidende funktionale Teile völlig neu implementiert werden: ein Projektmodul, dass das Anlegen, Verwalten und Abspeichern eines Projekts ermöglicht, ein interaktives „Fit Control Dashboard“, welches das einfache Anpassen der Startkonfiguration der Modelparameter ermöglicht und nicht zuletzt eine damit verknüpfte Implementierung des Fit-Algorithmus.

Drittens soll die Funktionsfähigkeit der Anwendung mit einem Beispieldatensatz und Fit dokumentiert und ein Performance-Vergleich zwischen dem bisherigen Fit-Algorithmus mit MATLAB und der neuen Python-Implementierung durchgeführt werden.

2 Anforderungsanalyse

Da das Projekt UDATTS bereits besteht und im Zuge der hier beschriebenen Projektphase zwar substanziell erweitert, nicht aber von Grund auf neu entwickelt werden soll, sind die Programmiersprache Python und auch viele der weiterhin zu verwendenden Bibliotheken weiter zu verwenden. Dies betrifft insbesondere die Python-Pakete PyQt5, PyQtGraph, NumPy, SciPy, SymPy und Xarray.

Darüber hinaus wurden vom Auftraggeber und Projektleiter Dr. Christoph Merschjann eine Reihe von Anforderungen gestellt, die im Folgenden aufgelistet sind.

2.1 Funktionale Anforderungen

1. Der Import und die grafische Darstellung mehrdimensionaler Messdaten in der Nutzeroberfläche müssen möglich sein.

2. Es muss möglich sein, geeignete mathematische Modelle zur Beschreibung des zeitlichen und spektralen Verhaltens der Messdaten zu erstellen und zu speichern.

3. Ein nichtlinearer Fit-Algorithmus muss in Python implementiert werden und es muss eine geeignete Schnittstelle geschaffen werden, so dass diesem Algorithmus eine geeignete Kombination aus Datensätzen und mathematischen Modellen übergeben werden kann.

4. Die Ergebnisse des Fit-Algorithmus müssen grafisch darstellbar und speicherbar sein.

5. Wenn nutzerseitig versucht wird, den Fit-Algorithmus auszuführen, kann eine Überprüfung stattfinden, ob alle dafür notwendigen Komponenten (Datensätze, mathematische Modelle) vorhanden sind, und der Fit-Algorithmus nur gestartet werden, wenn dies der Fall ist. Andernfalls kann den Nutzer*innen darüber ein Feedback gegeben werden, dass das Fitten noch nicht möglich ist.

6. Eine Kombination aus Datensätzen, mathematischen Modellen, Fit-Parametern und Fit-Ergebnissen soll als benennbares Projekt gespeichert und aus einer Projektdatei wieder in die Anwendung geladen werden können. Zum Abspei-

chern müssen allerdings nicht zwangsläufig alle genannten Elemente bereits vorhanden sein.

7. Alle Elemente eines Projekts müssen in einem Projektbaum in der Nutzeroberfläche angezeigt werden.

8. Über ein Kontextmenü sollen dem Projektbaum Datensätze hinzugefügt werden können, die aus vorverarbeiteten Network Common Data Format (NetCDF)-Dateien geladen werden sollen.

9. Abhängig von der Anzahl der im Projekt vorhandenen Datensätze sollen im Projektbaum auch mathematische Modelle (ein kinetisches Modell und so viele spektrale Modelle wie Datensätze) und Fit-Ergebnisse (gleiche Anzahl wie Datensätze) angezeigt werden, auch wenn die davon repräsentierten Objekte noch nicht erstellt sind.

10. Durch den Klick auf einen Datensatz oder ein Fit-Ergebnis (sofern vorhanden) im Projektbaum soll die grafische Repräsentation des entsprechenden Objekts angezeigt werden.

11. Durch den Klick auf eines der mathematischen Modelle soll sich in einem neuen Fenster der entsprechende „Model-Designer“ öffnen. Ist dieses Modell bereits im Projekt vorhanden, sollen dessen Funktionen und Parameter angezeigt werden; andernfalls soll ein leeres Modell angezeigt werden. Über diese Modell-Bedienflächen sollen die mathematischen Modelle neu angelegt oder verändert werden können.

12. Alle für den Fit-Algorithmus relevanten Parameter (die sich in Model-Parameter, ODE-Solver-Parameter und Fit-Parameter unterteilen) müssen in einem Unterbereich der Haupt-GUI, dem sogenannten `FitControlDashboard` dargestellt werden und interaktiv manipulierbar sein. Interaktiv bedeutet hierbei konkret: es muss möglich sein, die Modelle auf Basis der aktuellen Parameterkonfiguration zu evaluieren, das Ergebnis visuell darzustellen und aufgrund dessen dann einzelne Parameter direkt im `FitControlDashboard` nachzujustieren bevor der eigentliche Fit-Algorithmus ausgelöst wird.

13. Zukünftig kann es auch möglich sein, den Fit-Algorithmus während der Ausführung anzuhalten und die aktuelle Parameterkonfiguration anzusehen und gegebenenfalls nachzujustieren.

2.2 Nichtfunktionale Anforderungen

1. Die Funktionsfähigkeit der Anwendung soll mit mindestens einem Beispieldatensatz und dem daraus erstellten Fit-Resultat dokumentiert werden. Der dafür designierte Datensatz stammt aus einem spektroskopischen Experiment, das von Forschern des HZB, der Freien Universität Berlin und dem Institute for Molecular Science in Okazaki, Japan durchgeführt und dessen Ergebnisse bereits 2016 publiziert wurden (Vgl. [BWKA16]).

2. Dabei kann auch eine Performanzanalyse im Vergleich zur bisherigen Implementation des Fit-Algorithmus in MATLAB durchgeführt und gegebenenfalls Verbesserungsansätze aufgezeigt werden.

3. Die Gestaltung der Bedienoberflächen soll so benutzerfreundlich und intuitiv wie möglich sein.

4. Numerische Eingaben im `FitControlDashboard` sollen auch in wissenschaftlicher Notation (Exponentialdarstellung) möglich sein.

5. Für alle Eingaben im `FitControlDashboard` sollen geeignete Bedienelemente genutzt werden, die die Eingabe erleichtern und fehlerhafte Eingaben ausschließen (z.B. Checkboxen für Boolean-Werte, Spinboxen für Float- oder Integer-Werte etc.). Dies erleichtert außerdem die programmatische Behandlung der Werte, da diese so im jeweils korrekten Datentyp erfasst werden.

6. Die Gestaltung und der Aufbau der Hauptbedienoberfläche können sich an der Anwendung Glotaran (siehe [SLS+12]) orientieren.

7. Bei der Programmierung soll stets beachtet werden, dass die zukünftige Wartung und Erweiterungen einfach möglich sind.

8. Daher sollen weitestgehend aktiv entwickelte Bibliotheken (z.B. NumPy, SciPy, Xarray) und Datenformate (HDF5, NetCDF4) genutzt werden.

9. Der Code soll in englischer Sprache kommentiert werden.

10. Eine Dokumentation für Nutzer*innen kann erstellt werden.

3 Entwurf

Um die Implementierung der Anforderungen und somit die Entwicklung einer ersten funktionsfähigen Version der UDATTS-Software strukturiert angehen zu können wurden mehrere Vorüberlegungen angestellt.

Erstens wurden Vorüberlegungen zum Aufbau, zur Gestaltung und zur Hierarchie der Bedienoberflächen („Fenster“) der Anwendung gemacht.

Zweitens wurde der Anwendungs-Workflow aus Nutzersicht in Form eines Flowchart skizziert. Anhand dieser Skizze kann auch der Datenfluss innerhalb der Anwendung nachvollzogen werden, was eine Orientierung bei der Reihenfolge der Implementierung der einzelnen Programmteile und deren Abhängigkeiten untereinander bietet.

Drittens wurde unter Berücksichtigung der bisher implementierten Programmteile ein Schema zur Visualisierung der Model-View-Controller (MVC)-Architektur der Programmkomponenten erstellt.

3.1 Aufbau und Gestaltung der Bedienoberflächen

Ein Vorbild von UDATTS ist die von der Vrije Universiteit Amsterdam entwickelte Software Glotaran, bei welcher es sich um ebenfalls um eine Analysesoftware für zeitaufgelöste Spektroskopie- und Mikroskopiedaten handelt (vgl. [Sne13]). Bei der Konzeption der Haupt-Benutzeroberfläche von UDATTS diente die GUI von Glotaran als Inspiration:

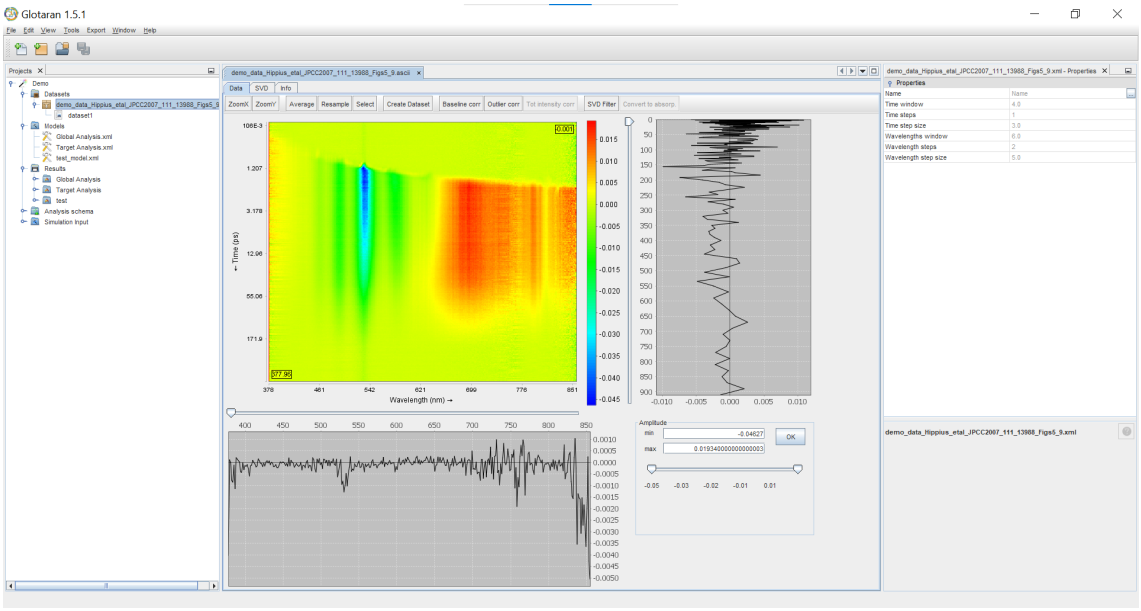

Abbildung 1: GUI der Software Glotaran

Auch die Haupt-GUI von UDATTS soll mehrere Unter-GUIs enthalten, die verschiedene Funktionen übernehmen und miteinander interagieren. Ähnlich wie in Abbildung 1 soll in einem Projektfenster am linken Rand der Haupt-GUI ein Projektbaum das Anlegen und die Verwaltung eines Projekts ermöglichen. Im Projektbaum sollen ein oder mehrere Projekte angezeigt werden können, die jeweils Datensätze, Modelle und Fit-Ergebnisse als Unterpunkte enthalten. Über das Projektfenster soll ein Projekt angelegt und diesem Datensätze und Modelle hinzugefügt werden können. Außerdem soll durch Anklicken der Datensätze und Fit-Resultate (sofern vorhanden) deren jeweilige visuelle Repräsentation angezeigt werden.

Eine Benutzeroberfläche zur grafischen Darstellung eines Datensatzes inklusive Optionen zur Manipulation derselben war für UDATTS bereits implementiert worden, das sogenannte `DataWidget`. Von diesem werden drei Instanzen benötigt: eine zum Anzeigen des Datensatzes, eine zum Anzeigen eines Modells der Daten und eine zum Anzeigen der Residuen, also der Differenz zwischen *Dataset* und *Model*. Diese seien im Folgenden zur besseren Unterscheidung hinsichtlich ihres darzustellenden Inhalts benannt als `DataWidget`, `ModelWidget` und `ResidualsWidget`. Diese sollen als drei Reiter im Hauptfenster den Platz in der unteren Hälfte des Bereichs rechts vom Projektfenster einnehmen.

Der obere Bereich darüber soll vom zu implementierenden `FitControlDashboard` ausgefüllt werden, das wiederum drei Unterfenster enthalten soll: eines zur baumartigen Darstellung der Modellparameter des kinetischen, sowie des spektralen Modells, eines zur tabellarischen Darstellung der Parameter des Differentialgleichungs-

lösers im `KineticModel` und schließlich eines zur Darstellung der Parameter des Fit-Algorithmus. Diese sollen hier auch direkt manipulierbar sein. Außerdem soll das `FitControlDashboard` zwei Buttons `Evaluate` und `Fit` enthalten, die das Ausrechnen des aktuellen Modells respektive das Fitten und die Anzeige der jeweiligen Ergebnisse auslösen soll, ebenso wie ein Fit Log, in welchem Nachrichten des Fit-Solvers angezeigt werden sollen.

3.2 Anwendungs-Workflow

In Abbildung 2 wurde der wesentliche Anwendungs-Workflow bei der Benutzung der UDATTS-Applikation, wie er aus den Anforderungen hervorgeht, in Form eines Flowchart dargestellt, um ein besseres Verständnis des Datenflusses und der programmatischen Abhängigkeiten zu gewinnen. Dabei wurde von dem Fall ausgegangen, dass ein Projekt neu angelegt wird. Das Laden, ebenso wie das Speichern eines Projekts wurden in dieser schematischen Darstellung also außen vor gelassen.

Nach dem Start der Software öffnet sich das `MainWindow`, die Haupt-GUI, in der der Projektbaum, das `FitControlDashboard` und die Visualisierung der Messdaten, sowie der Ergebnisse des Fit-Algorithmus angezeigt werden sollen. Zu Beginn sind die dafür zuständigen Widgets zwar bereits sichtbar, enthalten aber noch keine Daten.

Es kann nun ein neues Projekt angelegt werden, indem ein oder mehrere Datensätze hinzugefügt werden. Durch das Hinzufügen eines Datensatzes werden zwei Dinge ausgelöst: zum Einen wird der jeweils zuletzt hinzugefügte Datensatz im dafür zuständigen `DataWidget` angezeigt, das Teil des `MainWindow` ist; zum Anderen aktualisiert sich der Projektbaum im `Project Tree View`. Im Projektbaum wird jeweils ein `Dataset`, ein `SpectralModel` und ein `Result` mehr durchnummeriert angezeigt. Nur beim ersten Hinzufügen eines Datensatzes wird ein `Kinetic Model` hinzugefügt.

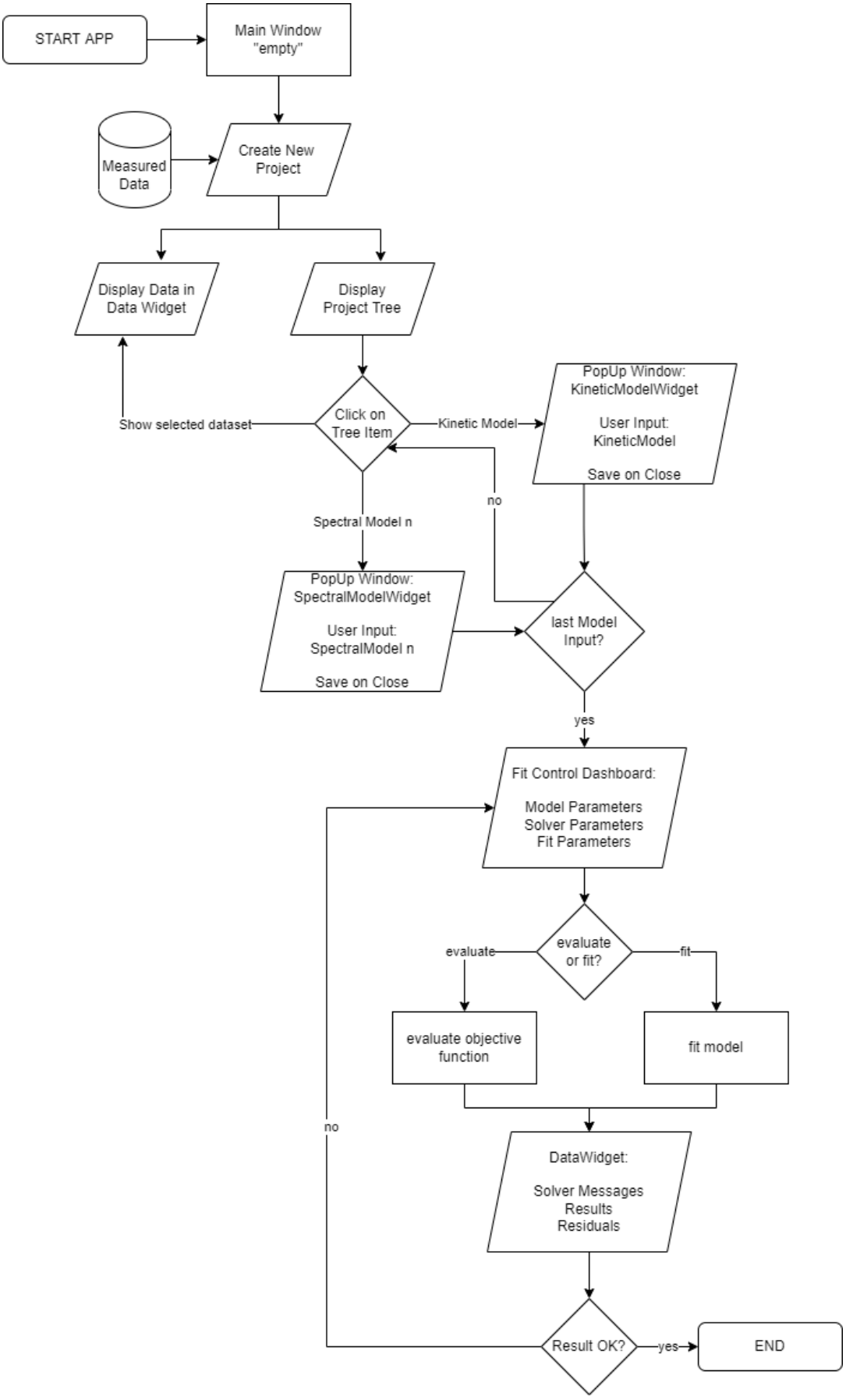

Abbildung 2: Anwendungs-Workflow

Durch Klicken auf eines der Elemente im Projektbaum wird Verschiedenes ausgelöst. Der Klick auf ein `Dataset` bewirkt, dass dieses im `DataWidget` angezeigt wird. Der Klick auf eines der Modelle führt zum Öffnen des zugehörigen `Model Designers` als Pop-Up-Fenster, also des `KineticModelWidget` oder `SpectralModelWidget`.

In diesen können dann mathematische Modelle mit allen notwendigen Parametern eingegeben oder aus zuvor gespeicherten JavaScript Object Notation (JSON)-Dateien geladen werden. Diese Modelle können dann auch dem Projekt hinzugefügt werden. Dadurch aktualisiert sich jedesmal auch die Anzeige der Model-Parameter und im Fall des `KineticModel` auch der Solver-Parameter im `FitControlDashboard`.

Wenn alle Modelle eingegeben bzw. dem Projekt hinzugefügt wurden, können im `FitControlDashboard` die Fit-Parameter eingestellt und gegebenenfalls auch die Model- und Solver-Parameter nachjustiert werden. Durch einen Klick auf einen der Buttons `Evaluate` oder `Fit` werden entweder die Modelle direkt ohne Fit-Algorithmus evaluiert oder erst der Fit-Algorithmus ausgelöst und die gefitteten Modelle evaluiert.

In beiden Fällen wird das Ergebnis der Berechnung (sozusagen der idealisierte Datensatz, wie er sich aufgrund der Modelle darstellt), sowie die Residuen zwischen Ergebnis und realem Datensatz in entsprechenden Reitern des `DataWidget` angezeigt. Wurde gefittet, dann zeigt ein Log im `FitControlDashboard` Log Messages des Fit-Solvers an, z.B. ob der Fit-Algorithmus konvergiert ist.

Der Prozess des Nachjustierens der Parameter im `FitControlDashboard` und des anschließenden Evaluierens und Fittens kann nun solange wiederholt werden, bis das Resultat für gut befunden wird.

3.3 MVC-Architektur

Die bereits vor der aktuellen Projektphase implementierten Programmteile zur Eingabe und Speicherung der kinetischen und spektralen Modelle verwenden jeweils das Entwurfsmuster MVC.

> „Das MVC-Paradigma wird durch drei Objekte umgesetzt. Das Model-Objekt stellt das Anwendungsobjekt dar, das View-Objekt seine Bildschirmpräsentation, und das Controller-Objekt bestimmt die Möglichkeiten, mit denen die Benutzungsschnittstelle auf Benutzungseingaben reagieren kann.“ [GHJV01]

In UDATTS bilden jeweils ein *Model*, ein *View* und ein *Controller* zusammen eine funktionale Einheit, im Folgenden auch „Modul“ genannt. Zum Beispiel bilden die Klassen `KineticModelWidget` (View), `KineticModelWidgetController` (Controller) und `KineticModel` (Model) ein solches Modul, wobei der View eine Instanz des Controllers als Attribut besitzt, der wiederum eine Instanz des Models als Attribut hat.

Ein Hintergedanke bei der Verwendung von MVC war, dass dieses Entwurfsmuster erlaubt, die gesamte Logik in die Model-Klassen zu lagern, die dann als Bibliothek zusammengefasst auch ohne ihre Controller und Views in andere Zusammenhänge programmatisch eingebunden werden können. Außerdem kann auch ein MVC-Modul einzeln gestartet und getestet werden, also eine Form von Unit-Testing durchgeführt werden, was die Robustheit und Wartbarkeit der Anwendung erhöht.

Aus diesem Grund wurde beschlossen, auch bei der Implementierung neuer Programmteile das MVC-Muster zu verwenden und es außerdem auch auf der Makroebene der Gesamtanwendung zu benutzen. Zur Konzeption wurde ein Diagramm erstellt (Abbildung 3).

Abbildung 3: MVC-Architektur - Ursprünglicher Entwurf

Auf der Makroebene soll UDATTS aus einem *Main Controller*, einem *Main Widget* und einem *Main Model* bestehen. An den drei Komponenten dieses Haupt-Moduls hängen die MVC-Objekte der einzelnen Module, die auch jedes für sich als programmatische Einheit funktionieren.

Wie in Abbildung 3 zu erkennen, wurde die MVC-Hierarchie bei den neu hinzu kommenden Modulen anders konzipiert, als bei den bisherigen: der *Controller* ist hier die oberste Instanz, die jeweils ein *Model* und einen *View* als Attribute besitzt. Dies ist mehr im Sinne des MVC-Entwurfsmusters ist, da so sichergestellt ist, dass *Model* und *View* nie direkt aufeinander zugreifen, sondern die Kommunikation über den *Controller* geregelt wird.

4 Implementierung

Die Reihenfolge der Implementierung orientierte sich grob an dem im Entwurfskapitel 3.2 skizzierten Workflow. Ausgenommen davon waren naturgemäß bereits bestehende Programmkomponenten wie die zwei „Model-Designer“ `KineticModelWidget` und `SpectralModelWidget`, sowie das `DataWidget`. Außerdem kam es wegen der Abhängigkeiten aller Programmteile voneinander immer wieder zu Überarbeitungen in allen relevanten Skripten.

Die UDATTS-Applikation ist vollständig objektorientiert programmiert und verwendet zur grafischen Darstellung der Benutzeroberflächen das Qt-Framework in Form der Python-Anbindung PyQt5 (siehe [Pyt22] und [Wei18], S. 775 ff.). Die Architektur der Objektklassen orientiert sich, wie im Entwurf geschildert, am MVC-Entwurfsmuster. Allerdings stellte sich im Verlauf der Implementierung heraus, dass es aus verschiedenen Gründen vorteilhaft war vom ursrpünglichen MVC-Entwurf abzuweichen. Abbildung 4 zeigt die tatsächlich implementierte MVC-Architektur.

Alle verwendeten Klassen können entweder als *Model*, *View* oder *Controller* klassifiziert werden. Allerdings gibt es nicht zu jedem MVC-Objekt immer die beiden anderen, es lassen sich also nicht immer MVC-Dreiergruppen bilden. So erwies sich ein `MainModel` bisher als überflüssig, da alle Models von eigenen Controllern gesteuert werden und ein zusätzliches Haupt-Model nur eine unnötige Verkomplizierung des Codes bedeutet hätte.

Darüber hinaus kann ein Model weitere Model-Unterkomponenten als Attibute haben, für die keine eigenen Controller und Views gebraucht werden; gleiches gilt analog für Views mit View-Unterkomponenten. Da aus Zeitgründen in dieser Projektphase keine Refaktorisierung des `DataWidget` möglich war, existiert dafür kein `DataController`. Um Objekte der *Model*-Klasse `Dataset` in einem `DataWidget` anzuzeigen, werden diese aktuell in das Format `SpectroData` konvertiert, das in einer früheren Implementierung von UDATTS für Datensätze verwendet wurde. Dieses `SpectroData`-Objekt wird dann als Parameter einer Funktion zur grafischen Darstellung übergeben (gestrichelte Linie in Abbildung 4).

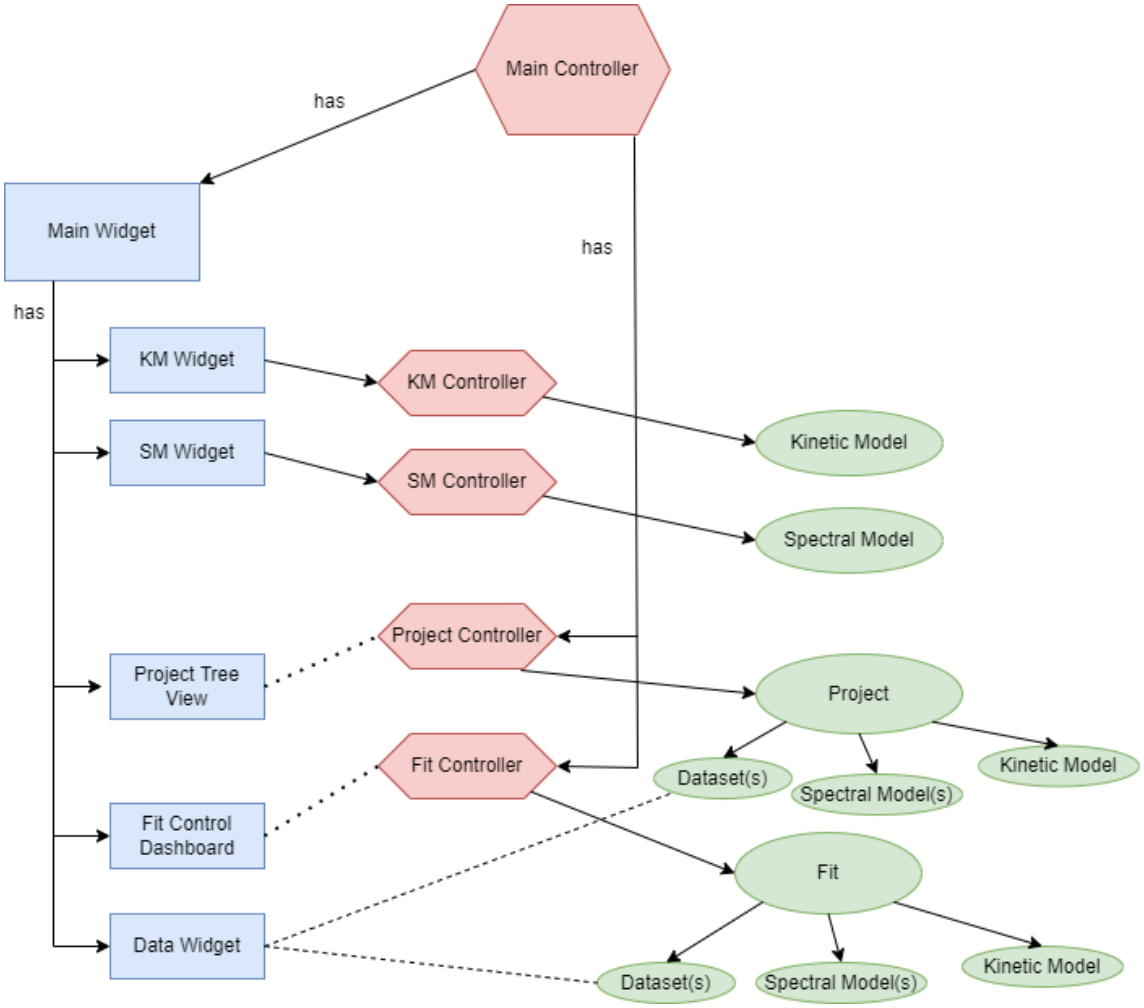

Abbildung 4: MVC-Architektur - Tatsächliche Umsetzung

Alle *View*- und *Controller*-Klassen erben von PyQt-Klassen und können daher mit *PyQt-Signalen* miteinander kommunizieren (gepunktete Linie in Abbildung 4). Die *Models* dagegen sind reine Pythonklassen und können daher nur von ihren jeweiligen *Controllern* angesteuert werden, welche jeweils ein *Model* als Membervariable besitzen.

4.1 Haupt-Modul

Auf der Makrobene besteht die Anwendung aus einem `MainController`, über den sie gestartet wird, und einem `MainWindow`, der Haupt-GUI, welche vom `MainController` instanziiert und geöffnet wird. Das `MainWindow` ist der zugehörige *View*, für ein `MainModel` bestand wie gesagt bislang keine Notwendigkeit.

Stattdessen besitzt der `MainController` weitere Unter-*Controller* als Attribute, die wiederum eigene *Models* besitzen und kontrollieren: den `ProjectController` mit dem *Model* `Project` und den `FitController` mit dem *Model* `Fit`.

Im Sinne eines konsisten Aufbaus sollte der `MainController` eigentlich auch einen `KineticModelWidgetController` und einen `SpectralModelWidgetController` mit ihren jeweiligen *Models* besitzen. Da diese jedoch Teil der bereits früher implementierten Strukturen sind, die zu refaktorisieren den Rahmen der hier beschriebenen Projektphase gesprengt hätte, wurde dies aufgeschoben.

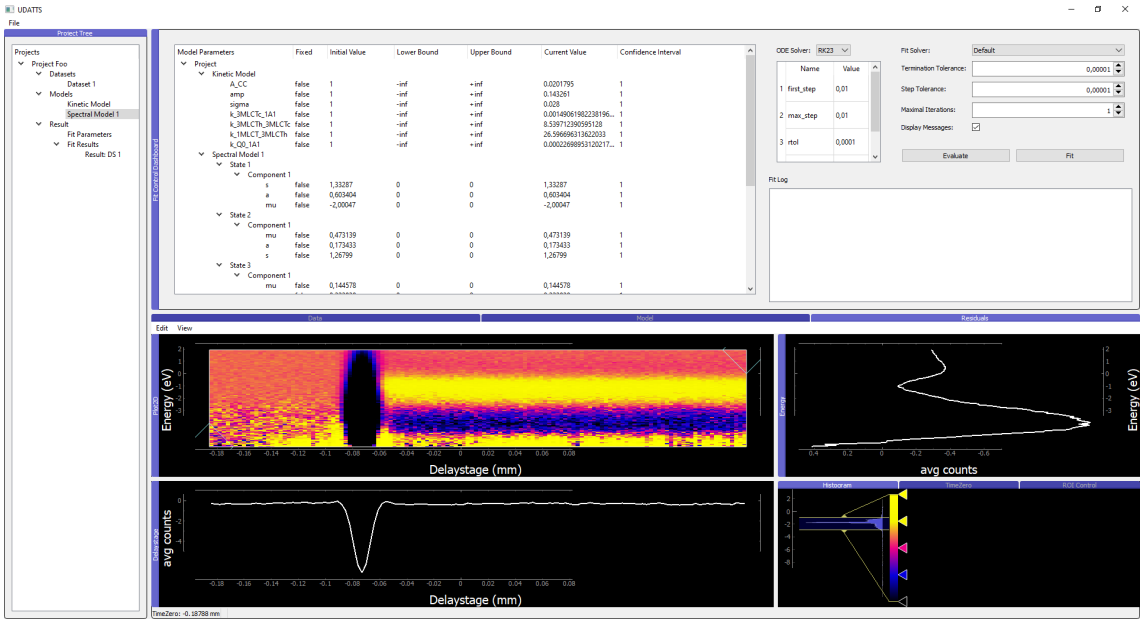

Abbildung 5: Hauptfenster von UDATTS

Das `MainWindow` als Haupt-View besitzt weitere Views als Klassenattribute: dies sind einerseits der `ProjectTreeView` (in Abbildung 5 links), das `FitControlDashboard` (obere Hälfte) und die drei `DataWidget`-Objekte als Reiter (untere Hälfte) „Data“, „Model“ und „Residuals“, die alle im `MainWindow` eingebettet und sichtbar sind, andererseits `KineticModelWidget` und `SpectralModelWidget`, die beiden „Model-Designer“, die als eigene Dialoge in Popup-Fenstern aufgerufen werden.

Der `MainController` stellt Verbindungen von *PyQt-Signals* mit *PyQt-Slots* her. Dadurch können verschiedene PyQt-Objekte des Programms miteinander kommunizieren und sich Daten senden. Der Signal-Slot-Mechanismus ist integraler Bestandteil des Qt-Frameworks und ist das Standardmittel für die Kommunikation zwischen `QObjects`. Wenn ein `QObject` ein *PyQt-Signal* emittiert, wird es von Qt der Event Queue hinzugefügt. Ein anderes `QObject` (oder mehrere) kann dann das Signal empfangen, wenn eine entsprechende Signal-Slot-Verbindung konfiguriert wurde. *PyQt-*

Signals können auch (typisierte) Parameter haben, wodurch eine einfache Datenübertragung zwischen den Objekten möglich wird (vgl. [The22] und [Wei18], S. 784 f.).

4.2 Projekt-Modul

Das Projekt-Modul besteht aus den Klassen `Project` (*Model*), `ProjectController` (*Controller*) und `ProjectTreeView` (*View*). Es verwaltet und repräsentiert ein Projekt in UDATTS. Ein Projekt-Modul ist prinzipiell auch unabhängig von der Gesamtanwendung ausführbar, in dem der `ProjectController` gestartet wird, der die anderen beiden Komponenten als Attribute besitzt. Eingebunden in die Gesamtanwendung wird der `ProjectTreeView` des `ProjectController` allerdings nicht genutzt, sondern die Darstellung des Projekts erfolgt im `ProjectTreeView` des `MainWindow` über *PyQt-Signale*.

Ein `Project`-Objekt hat sechs Attribute: einen Namen sowie fünf Dictionaries, in denen Datensätze, Kinetische Modelle, Spektrale Modelle, Fit-Parameter und Fit-Resultate gespeichert und verwaltet werden. Für das im Rahmen dieser Arbeit getestete Beispiel wird nur jeweils ein `Dataset`, `KineticModel` und `SpectralModel` benötigt und folglich vom Fit-Algorithmus genau ein `FitResult` erwartet. Die Anzahl der Fit-Parameter ist für jedes Projekt konstant: es handelt sich um die fünf im `FitControlDashboard` konfiguierbaren Fit-Parameter, die auch im Projekt gespeichert werden können.

4.2.1 Der Projektbaum

Die korrespondierende Benutzerfläche eines Projekts wird vom `ProjectTreeView` bereitgestellt. Da schon in der Entwurfsphase auch die Darstellung der Model-Parameter im `FitControlDashboard` als baumartig konzipiert wurde, wurde zunächst eine Elternklasse `TreeView` implementiert, von der `ProjectTreeView` und `ModelParameterTreeView` erben.

`TreeView` selbst erbt von `QWidget`, der PyQt-Standardklasse für Widgets und besitzt als Attribut ein Objekt `self.tree` vom Typ `QTreeView`, welches die baumförmige Darstellung von Daten erlaubt. Bei der Implementierung dieser Klasse und ihrer Methoden diente ein Tutorial aus dem Internet als Vorlage (Vgl. [T.Y20]).

Für den `ProjectTreeView` wurden zusätzliche Methoden implementiert, um das Hinzufügen von Datensätzen, sowie kinetischen und spektralen Modellen zum Projekt zu ermöglichen.

4.2.1.1 Datenimport und -darstellung

Durch Rechtsklick auf den Projektbaum öffnet sich ein Kontextmenü, über das es möglich ist, Datensätze im `NetCDF`-Format in das Projekt zu laden. Die `NetCDF`-Datei wird dazu in ein Objekt der Klasse `Dataset` konvertiert, die eine für die anwendungsspezifischen Zwecke erweiterte Kindklasse von `xarray.Dataset` (siehe [xar14a]) ist. Dieses wird dem Dictionary `self.data` des Projekts hinzugefügt.

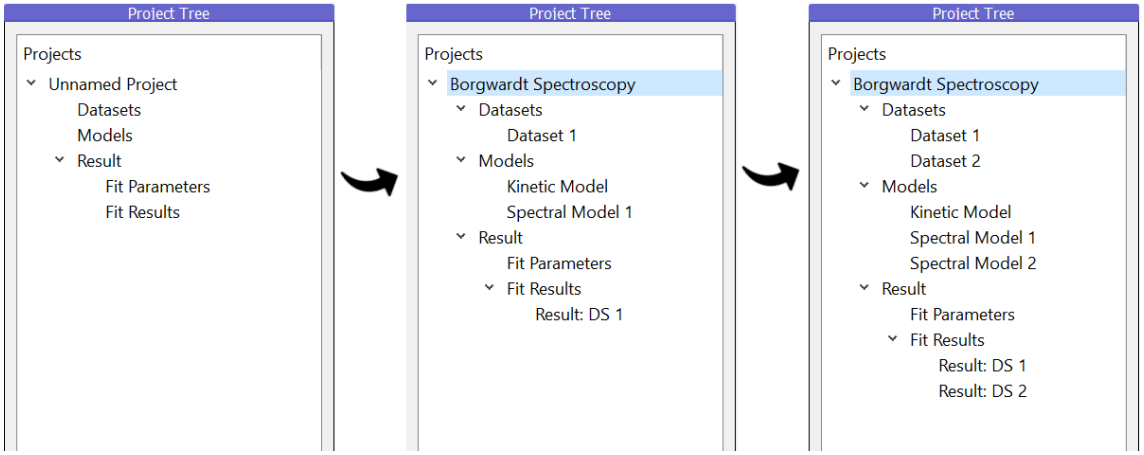

Abbildung 6: Datenimport über den Projektbaum

Für jedes hinzugefügte `Dataset` werden dem Projektbaum neue Unterpunkte bei „Datasets“, „Models“ und „Results“ hinzugefügt wie in Abbildung 6 ersichtlich. Der Unterpunkt „Kinetic Model“ wird nur beim ersten `Dataset` hinzugefügt.

Außerdem wird der zuletzt hinzugefügte Datensatz im `DataWidget` dargestellt. Bei mehreren Datensätzen kann die Anzeige durch Klicken auf den jeweiligen Datensatz im Projektbaum geändert werden.

4.2.1.2 Aufruf der Model-Designer

Der Klick auf ein Modell im Projektbaum öffnet den jeweiligen Model-Designer, also das `KineticModelWidget` respektive `SpectralModelWidget` als neuen Dialog. In diesen bereits früher implementierten *Views* können die mathematischen Model-

le entweder neu eingegeben, oder aus JSON-Dateien geladen und auch als solche abgespeichert werden.

Neu hinzugefügt wurde die Funktionalität, die Modelle nun auch im Projekt abzuspeichern. Dazu werden Kopien des dem *View* zugehörigen *Models* (im MVC-Sinne) sowohl dem `Project`, als auch dem `Fit` als Attribute zugewiesen.

Außerdem werden die `ModelParameter` und im Falle des `KineticModel` auch die `SolverParameter` per PyQt-Signal ans `FitControlDashboard` übermittelt und dort dargestellt.

4.2.2 Speichern und Laden von Projektdateien

Bei der Frage wie ein UDATTS-Projekt gespeichert und wieder in die Anwendung geladen werden kann, wurden zwei grundsätzliche Optionen erwogen: ein Projektverzeichnis mit mehreren Dateien oder eine einzelne Projektdatei.

Für eine Speicherung in mehreren Dateien sprach der deutlich geringere Programmieraufwand, da für `KineticModel` und `SpectralModel` bereits Export und Import als JSON-Dateien implementiert waren und das Speichern eines Xarray-`Dataset` als NetCDF-Datei trivial ist. Eventuelle Fit-Ergebnisse eines Projekts hätten dann einfach als zusätzliche Textdateien abgespeichert werden können. Ein großer Nachteil einer Speicherung in Form eines Ordners mit mehreren Dateien wäre allerdings die Gefahr eines unvollständigen Umkopierens oder versehentlichen Löschens einzelner Dateien. In einem solchen Fall wäre es dann auch nicht in allen Fällen möglich sicher festzustellen, ob ein Projekt vollständig ist oder nicht.

Aus diesem Grund wurde eine Entscheidung für das Abspeichern eines Projekts als einzelne Projektdatei gefällt. Dazu wurden die Möglichkeiten der Bibliothek Xarray und des Formats NetCDF ausgenutzt.

Sowohl das Speichern, wie auch das Laden einer Projektdatei sind entweder über das Hauptmenü im `MainWindow`, als auch über das Kontextmenü des Projektbaums, also vom `ProjectTreeView` aus aufrufbar.

4.2.2.1 Abspeichern einer Projektdatei

Unabhängig davon in welchem Menü sie aufgerufen wird löst der Klick auf die Option „Save Project“ ein Signal `sigSaveProject` aus, das im `ProjectController` von der Methode `save_project(self)` empfangen wird (siehe Listing 4.1). Diese öffnet einen `QFileDialog`, der die Auswahl eines Speicherpfads und die Benennung der Projektdatei ermöglicht. Dabei wird standardmäßig der UDATTS-Unterordner „projects“ geöffnet und falls noch nicht vorhanden neu erzeugt. Ein voreingestellter Filter sorgt dafür, dass die Datei die korrekte NetCDF-Endung „.nc“ erhält.

```
1  def save_project(self):
2      # make project directory if it doesn't exist yet
3      p_dir = "..\\..\\projects"
4      if not os.path.exists(p_dir):
5          os.mkdir(p_dir)
6      # open file dialog
7      fd = QFileDialog()
8      fd.setNameFilter("*.nc")
9      fd.selectFile(self.project.name.replace("␣", "_") + ".nc")
10     file_path = fd.getSaveFileName(caption="Save␣project",
           directory=p_dir, filter="NetCDF␣files␣(*.nc)")[0]
11     self.project.save(file_path)
```

Listing 4.1: `ProjectController`-Methode zum Speichern einer Projektdatei

Nach der Eingabe eines Dateinamens und dessen Bestätigung im Dialogfenster, schließt sich dieses und der Dateipfad wird der nun aufgerufenen Methode `save(self, file_path)` des `Project` übergeben (siehe Listing 4.2). In dieser findet das eigentliche Speichern des Projekts statt. Sie ist so implementiert, dass sie theoretisch auch ohne den Parameter `file_path` aufrufbar ist, sollte dies zukünftig gewünscht sein. In diesem Fall wird ein Standardpfad aus dem aktuellen Projektnamen generiert.

Als erster Schritt der Projektspeicherung wird ein `xarray.Dataset` instanziiert, das als Container der Projektdaten dient. Diesem werden dann nacheinander die `KineticModels`, `SpectralModels` und `FitResults` in Form von Strings als globale Attribute hinzugefügt. Bei den Modellen wird hier die bereits implementierte Umwandlung in JSON genutzt, die allerdings leicht adaptiert werden musste. Zur Separierung der Modell-Namen und der JSON-Struktur, sowie der Modelle unter-

einander, werden in die jeweiligen Zeichenketten die Zeichen § bzw. # eingefügt; analoges gilt für die Fit-Resultate.

```
def save(self, file_path=None):
    if file_path is None:
        # make project directory if it doesn't exist yet
        p_dir = "..\\..\\projects"
        if not os.path.exists(p_dir):
            os.mkdir(p_dir)
        # make standard file path
        file_path = p_dir + "\\" + self.name.replace("␣", "_") + ".
            nc"
    # prepare export as NetCDF file
    DS = xr.Dataset()
    # add kinetic models to DS
    kms_as_str = ""
    if len(self.kinetic_models) > 0:
        for km_name, km in self.kinetic_models.items():
            kms_as_str += km_name + "  " + json.dumps(km.
                convertToJsonString(), indent=4) + "#"
    DS.attrs["kinetic␣models"] = kms_as_str
    # add spectral models to DS
    sms_as_str = ""
    if len(self.spectral_models) > 0:
        for sm_name, sm in self.spectral_models.items():
            sm.update_serializable_properties()
            sms_as_str += sm_name + "  " + json.dumps(sm.
                toJsonString(sm), indent=4) + "#"
    DS.attrs["spectral␣models"] = sms_as_str
    # add fit results to DS
    frs_as_str = ""
    if len(self.fit_results) > 0:
        for fr_name, fr in self.fit_results.items():
            frs_as_str += fr_name + "  " + str(fr) + "#"
    DS.attrs["fit␣results"] = frs_as_str
    # store number of datasets in global attributes
    DS.attrs["number␣of␣datasets"] = len(self.data)
    # create NetCDF file from DS
    DS.to_netcdf(file_path)
    # add datasets to NetCDF file
    if len(self.data) > 0:
        for i, (ds_name, ds) in enumerate(self.data.items()):
            ds_name = ds_name.replace("␣", "_")
            ds.save(path=file_path, mode='a', group=ds_name)
```

Listing 4.2: `Project`-Methode zum Speichern einer Projektdatei

Dann wird das Projekt-`xarray.Dataset` als NetCDF-Datei exportiert und erst danach werden die Datensätze des Projekts dieser Datei hinzugefügt. Das liegt daran, das hier ein Feature von `xarray.Datasets` ausgenutzt wird: mit der Methode `xarray.Dataset.to_netcdf()` können nämlich auch in eine bereits existente NetCDF-Datei beliebig viele `xarray.Datasets` als benannte „groups“ (Gruppen) geschrieben werden. Dazu wird als Parameter `path` immer der gleiche Dateipfad übergeben, der Parameter `mode` wird auf `‚‚a‘‘` für „append“ (engl. für „anhängen“) gesetzt und als Parameter `group` jeweils der Name des Datensatzes geschrieben (vgl. [xar14b]).

4.2.2.2 Öffnen einer Projektdatei

Genau wie das Speichern, kann auch das Laden einer Projektdatei über das Hauptmenü oder das Kontextmenü des Projektbaums per Klick auf „Open Project“ aufgerufen werden. Das Signal `sigOpenProject` löst dann im `ProjectController` die Methode `open_project(self)` aus (siehe Listing 4.3). Falls er exisitert, wird in einem `QFileDialog` der UDATTS-Unterordner „projects“ geöffnet, falls nicht, stattdessen das Laufwerk „C:“.

```
1  def open_project(self):
2      # open project directory; if it doesn't exist open C:
3      p_dir = "..\\..\\projects"
4      if not os.path.exists(p_dir):
5          p_dir = "C:\\"
6      # open dialog to pick a project file
7      netcdf_path = QFileDialog.getOpenFileName(caption="Pick␣a␣.
           netcdf-File", directory=p_dir)[0]
8      if netcdf_path == "":
9          return
10     # set project name
11     project_name = netcdf_path.split("/")[-1].split(".")[0].replace
           ("_", "␣")
12     self.set_project_name(project_name)
13     self.sigShowProjectName.emit(project_name)
14     # load project-xr.Dataset (possibly containing udatts.Datasets)
15     DS = xr.load_dataset(netcdf_path)
16     # get datasets
17     for n in range(1, DS.attrs["number␣of␣datasets"] + 1):
18         ds = xr.load_dataset(netcdf_path, group="Dataset_" + str(n)
               )
19         # convert xarray Dataset to UDATTS Dataset
```

```
20          ds = Dataset(data_vars=ds.data_vars, coords=ds.coords,
                attrs=ds.attrs)
21          self.project.data["Dataset " + str(n)] = ds
22          self.sigAddedDataset.emit(ds)
23          self.sigAddDatasetToFit.emit(ds)
24      # get kinetic models
25      kms_str_list = DS.attrs["kinetic models"].split("#")
26      del kms_str_list[-1]
27      for km_str in kms_str_list:
28          km_name = km_str.split("  ")[0]
29          km_json_str = km_str.split("  ")[1]
30          # temporary hot fix: preprocessing JSON string to avoid
                problems
31          km_json_str = km_json_str.strip("\""). \
32              replace("'", "\""). \
33              replace("False", "false"). \
34              replace("True", "true"). \
35              replace("\\\\\"", "'"). \
36              replace("\\\\n", "\n")
37          km = KineticModel()
38          km_json_dict = km.fromJson(km_json_str, strict=False)
39          try:
40              km.remakeLoadedData(km_json_dict)
41          except Exception as e:
42              print(f"\nError when trying to remake Kinetic Model:\t{
                    e}")
43          self.project.kinetic_models[km_name] = km
44          self.sigAddKineticModelToDashboard.emit(km, km_name)
45          self.sigAddKineticModelToFit.emit(km)
46      # get spectral models
47      sms_str_list = DS.attrs["spectral models"].split("#")
48      del sms_str_list[-1]
49      for sm_str in sms_str_list:
50          sm_name = sm_str.split("  ")[0]
51          sm_json_str = sm_str.split("  ")[1]
52          sm = SpectralModel()
53          try:
54              sm.fromJson(sm_json_str)
55              print("\nSuccessfully reloaded Spectral Model!")
56              sm.Show()
57          except Exception as e:
58              print(f"\nError when trying to remake Spectral Model:\t
                    {e}")
59          self.project.spectral_models[sm_name] = sm
60          self.sigAddSpectralModelToDashboard.emit(sm, sm_name)
61          self.sigAddSpectralModelToFit.emit(sm)
62      # get fit results
```

```
63      frs_str_list = DS.attrs["fit results"].split("#")
64      del frs_str_list[-1]
65      for fr_str in frs_str_list:
66          fr_name = fr_str.split("  ")[0]
67          fit_result = fr_str.split("  ")[1]
68          self.project.fit_results[fr_name] = fit_result
69      if DS.attrs["fit results"] != "":
70          self.sigShowFitResult.emit(self.project.fit_results["Fit
                Result 1"])
71          self.sigEvaluate.emit()
```

Listing 4.3: `ProjectController`-Methode zum Öffnen einer Projektdatei

Wurde eine NetCDF-Datei ausgewählt, wird diese nun wieder Schritt für Schritt in ein `Project`-Objekt entpackt. Zunächst wird der Projektname aus dem Dateinamen generiert und dem Projekt vergeben sowie im Projektbaum angezeigt. Dann wird die gesamte NetCDF-Datei in ein `xarray.Dataset` geladen und die einzelnen Datensätze werden in einer Schleife einzeln aus den Gruppen der Datei geladen. Zudem müssen sie noch von Objekten der Klasse `xarray.Dataset` wieder in Objekte der Kindklasse `udatts.Dataset` konvertiert werden. Aus dem `xarray.Dataset`, das das gesamte Projekt darstellt, werden die Attribute „kinetic models“, „spectral models“ und „fit results“ als Strings ausgelesen, die jeweils zerlegt und in Schleifen weiterverarbeitet werden, um die entsprechenden Objekte wieder herzustellen und dem Projekt zuzuweisen.

Wenn die geöffnete Projektdatei mindestens ein Fit-Ergebnis enthielt, werden die Signale `sigShowFitResult` und `sigEvaluate` emittiert und dadurch, das Fit-Ergebnis im Fit-Log angezeigt und die Modelle des Projekts evaluiert (vgl. Kapitel 4.3.2.2).

4.3 Fit-Modul

Das Fit-Modul ist zuständig für den Fit-Algorithmus und die Nutzerinteraktion mit diesem. Es umfasst die Klassen `Fit` (*Model*), `FitControlDashboard` (*View*) und `FitController` (*Controller*).

4.3.1 Das Fit-Control-Dashboard

Das `FitControlDashboard` stellt die Benutzeroberfläche des Fits dar und umfasst die drei Unterbereiche `ModelParameterTreeView`, `SolverParameterTableView` und `FitParameterView`, die es als Attribute besitzt.

Über diese Unter-GUIs können alle für den Fit relevanten Parameter interaktiv angepasst werden. Mit dem Button `Evaluate` kann das aktuelle Modell ausgerechnet und im `ModelWidget` dargestellt werden. Der `Fit`-Button löst den Fit-Algorithmus aus. Ist dieser terminiert (und gegebenenfalls konvergiert), wird das gefittete Modell angezeigt und in einem Fit Log Nachrichten des Fit-Solvers ausgegeben.

4.3.1.1 Der Model-Parameter-Tree-View

Die Klasse `ModelParameterTreeView` dient zur baumartigen Darstellung aller Model-Parameter aller mathematischen Modelle eines Projekts. Sie erbt von der oben erwähnten Klasse `TreeView` und erweitert diese um Methoden zum Hinzufügen und zum Auslesen von Parametern.

Die Methode `add_model_parameters_to_tree(self, model, m_name)` (siehe Listing 4.4) wird jedes mal aufgerufen, wenn über die Model-Designer dem Projekt ein `KineticModel` respektive `SpectralModel` hinzugefügt wird. Das jeweilige Modell und sein Name werden über ein PyQt-Signal, z.B. `sigSaveKMtoProject` für das `KineticModel`, an die PyQt-Eventqueue gesendet. Die Signale beider Modelltypen sind im `MainController` mit dieser Methode verknüpft, wo die Parameter empfangen und weiterverarbeitet werden.

Dabei ist eine Fallunterscheidung aufgrund der unterschiedlichen Struktur der Modelle nötig: Während beim `KineticModel` die Parameter direkt an diesem als Blätter hängen, verzweigt sich ein `SpectralModel` zunächst in `SpectralStates`, die sich weiter in `SpectralComponents` verzweigen, an denen die Parameter hängen. Zudem ist abhängig vom Modellnamen zu beachten, an welcher Stelle im Parameterbaum die neuen Parameter eingefügt werden müssen und gegebenenfalls müssen Parameter zunächst gelöscht werden, wenn ein Modell gleichen Namens bereits im Parameterbaum existierte und nun überschrieben wird.

Der Parameterbaum hat im Unterschied zum Projektbaum nicht nur eine sondern sieben Spalten, um alle Eigenschaften der Model-Parameter darzustellen. Dabei sind die erste und letzte Spalte (Name und Konfidenzintervall) als nicht editierbar eingestellt. In den anderen fünf Spalten können die Werte der Parameter verstellt werden, bis auf die Spalte *Fixed*, die einen Boolean-Wert enthält, handelt es sich dabei um Float-Werte.

```
@QtCore.pyqtSlot(object, str)
def add_model_parameters_to_tree(self, model, m_name):
    if m_name.startswith('Kinetic'):
        # clear old Kinetic Model parameters
        idx_km = self.find_idx_by_short_name(m_name)  # "Kinetic Model"
        idx_sm1 = self.find_idx_by_short_name("Spectral Model 1")
        del self.parameter_items[idx_km + 1:idx_sm1]
        # write new Kinetic Model parameters
        model_parameters = model.model_parameter_list
        new_items = []
        for i, mp in enumerate(model_parameters):
            new_item = {'unique_id': 111 + i, 'parent_id': 11,
                        'short_name': mp.name, 'fixed': mp.fixed, 'ini_value': float(mp.initial_value),
                        'low_bound': float(mp.lower_bound), 'up_bound': float(mp.upper_bound),
                        'cur_value': float(mp.current_value), 'conf_interval': float(mp.confidence_interval)}
            new_items.append(new_item)
        self.parameter_items[idx_km + 1:idx_km + 1] = new_items
    else:
        # clear old Spectral Model n parameters
        idx_sm = self.find_idx_by_short_name(m_name)  # i.e. "Spectral Model 3"
        if not idx_sm:
            idx_sm = len(self.parameter_items)
        num_sm = int(m_name.split()[-1])
        idx_nxt_sm = self.find_idx_by_short_name("Spectral Model " + str(num_sm + 1))
        if not idx_nxt_sm:
            idx_nxt_sm = len(self.parameter_items)
        del self.parameter_items[idx_sm + 1:idx_nxt_sm]
        # write new Spectral Model n parameters
        spectral_model_tree_items = model.spectral_model_tree_data.values()
        new_items = []
        model_id = 11 + num_sm
        if idx_sm == idx_nxt_sm:
            model_item = {'unique_id': model_id, 'parent_id': 1, 'short_name': 'Spectral Model ' + str(num_sm)}
            new_items.append(model_item)
        for i, state in enumerate(spectral_model_tree_items):
            short_name = correct_state_name(state.name)
            state_id = 10 * model_id + 1 + i
```

```
38              state_item = {'unique_id': state_id, 'parent_id':
                    model_id, 'short_name': short_name}
39              new_items.append(state_item)
40              components = state.spectral_component.values()
41              for j, component in enumerate(components):
42                  short_name = correct_component_name(component.name)
43                  component_id = 10 * state_id + 1 + j
44                  component_item = {'unique_id': component_id, '
                        parent_id': state_id, 'short_name': short_name}
45                  new_items.append(component_item)
46                  model_parameters = component.ModelParameters.values
                        ()
47                  for k, mp in enumerate(model_parameters):
48                      param_id = 10 * component_id + 1 + k
49                      new_item = {'unique_id': param_id, 'parent_id':
                            component_id, 'short_name': mp.name,
50                                  'fixed': mp.fixed, 'ini_value': mp.
                                      initial_value, 'low_bound': mp.
                                      lower_bound,
51                                  'up_bound': mp.upper_bound, '
                                      cur_value': mp.current_value,
52                                  'conf_interval': mp.
                                      confidence_interval}
53                      new_items.append(new_item)
54          self.parameter_items[idx_sm + 1:idx_sm + 1] = new_items
55
56      self.importData(self.parameter_items)
57      self.tree.expandAll()
```

Listing 4.4: Methode zur Darstellung der Model-Parameter

Um den Anforderungen für die Darstellung und Eingabe dieser Dezimalzahlen zu genügen (unter anderem in wissenschaftlichem Format und sowohl mit deutschem Komma wie auch englischem Punkt als Dezimalseparator), wurde ein sogenanntes „ItemDelegate“ implementiert, eine Klasse die für bestimmte `QStandardItems` im `QTreeView` eine bestimmte Darstellung und Funktionalität bereitstellt. Diese von der Klasse `QStyledItemDelegate` abgeleitete Klasse `DoubleSpinBoxDelegate` nutzt wiederum ein Objekt einer eigens implementierten Kindklasse `SciDoubleSpinBox`, die von `QDoubleSpinBox` abgeleitet ist.

Wird durch `Evaluate` oder `Fit` eine Modellberechnung ausgelöst, werden die Model-Instanzen des `Fits` mit den Werten aus dem `ModelParameterTreeView` geupdated. Dazu wird mit der Methode `get_model_parameters_from_tree(self)` der gesamte Parameterbaum traversiert und als mehrfach verschachteltes Dictionary zurückgege-

ben, das per *PyQt-Signal* an den `FitController` gesendet wird, wo es dem `Fit` zur weiteren Verarbeitung übergeben wird.

4.3.1.2 Der Solver-Parameter-View

In der aktuellen Implementierung berechnet sich das `KineticModel` ausschließlich aus impliziten *States*, d.h. mittels eines Algorithmus zum Lösen von Differenzialgleichungen, englisch Ordinary Differential Equation (ODE)-Solver. Für die dafür verwendete SciPy-Funktion `integrate.solve_ivp` können im `KineticModelWidget` einige Parameter konfiguriert werden, etwa die Lösungsmethode, die initiale und die maximale Schrittgröße und die relative sowie absolute Toleranz. Diese sollten auch im `FitControlDashboard` angezeigt werden und verstellbar sein, wofür die Klasse `SolverParameterTableView` erstellt wurde. (Der Begriff „Solver-Parameter“ bezieht sich im Folgenden ausschließlich auf den ODE-Solver, *nicht* den Fit-Solver.)

Der `SolverParameterTableView` umfasst eine `QComboBox` und eine Tabelle mit vier Zeilen. Die `QComboBox` bietet ein Dropdownmenü über das die ODE-Solver-Methode eingestellt werden kann; mögliche Werte sind RK45 (der Default-Wert), RK23, Radau, BDF und LSODA. In der Tabelle können Werte für die oben erläuterten Parameter `first_step`, `max_step`, `rtol`, und `atol` eingegeben werden. Für die Formatierung und Editierbarkeit dieser Zellen wurde wieder das `DoubleSpinBoxDelegate` verwendet.

4.3.1.3 Der Fit-Parameter-View

Die dritte Unterkomponente des `FitControlDashboard` ist der `FitParameterView` und betrifft die Parameter des Fit-Algorithmus. Vorläufig wurden in dieser Bedienoberfläche allerdings nicht alle möglichen Parameter der verwendeten Scipy-Funktion `optimize.minimize` dargestellt, sondern nur eine Auswahl. Außerdem enthält der `FitParameterView` auch die bereits erwähnten Buttons `Evaluate` und `Fit` als Trigger zur Berechnung.

Die hier einstellbaren Fit-Parameter sind: *Fit-Solver*, *Termination Tolerance*, *Step Tolerance*, *Maximal Iterations* und *Display Messages*. Die Auswahl des Fit-Solvers erfolgt über eine `QCombobox`. Darin sind alle Werte einstellbar, die als gültige Werte des optionalen Parameters *method* von `scipy.optimize.minimize` genommen werden. Es handelt sich dabei um verschiedene mathematische Verfahren zur Op-

timierung wie *Nelder_ Mead*, *Powell*, *CG*, *dogleg* und weitere. Voreingestellt in der `QCombobox` ist der Wert *Default*, dann wird der Parameter `method` leer gelassen. In diesem Fall nimmt `optimize.minimize` abhängig davon, ob der Methode noch `constraints` und / oder `bounds` als Argumente mitgegeben wurden die Optimierungsverfahren BFGS, L-BFGS-B oder SLSQP (vgl. [The08]).

Für die weiteren Parameter wurden als Anzeigeelemente `QDoubleSpinBox` (*Termination Tolerance* und `Step Tolerance`), *QSpinBox* (*Maximal Iterations*) und `QCheckBox` (*Display Messages*) gewählt.

4.3.2 Die Fit-Klasse

Wie bei den anderen MVC-Modulen wurde auch beim Fit-Modul die eigentliche Logik in die zugehörige Model-Klasse `Fit` gelagert. Hier sind die zentralen Berechnungsmethoden `evaluate_models(self)` und `fit(self, x0, *args, **kwargs)` implementiert mit denen, je nachdem welcher Button geklickt wurde, entweder das aktuelle Modell ausgerechnet und angezeigt wird oder der Fit-Algorithmus ausgeführt und das gefittete Modell angezeigt wird. Diese Methoden greifen wiederum auf diverse Hilfsmethoden zurück.

Eine grobe Orientierung bei der Implementierung bot dabei der bisher verwendete Fit-Algorithmus in MATLAB. Ziel dieser Projektphase war eine Minimalimplementierung, mit der der Testdatensatz fehlerfrei gefittet werden kann. Aus Zeitgründen wurden deshalb noch nicht alle in MATLAB implementierten Schritte umgesetzt und die Programmierung funktioniert noch nicht für Projekte mit mehr als einem Datensatz.

4.3.2.1 Evaluieren der Models

Die Methode `evaluate_models(self)` (siehe Listing 4.5) wird sowohl beim bloßen Evaluieren, als auch beim Fitten ausgeführt. Zunächst werden dazu aus dem Datensatz die temporale und die spektrale (energetische) Achse übernommen. Es folgt ein konditionaler Block, in dem der Fit ausgelöst wird, wenn der `Fit`-Button geklickt und dadurch im `Fit`-Objekt das Flag `run_fit` gesetzt wurde.

Danach werden die beiden Fit-Modelle in Abhängigkeit der aus dem Datensatz übernommenen Achsen ausgerechnet, das Ergebnis ist jeweils eine 2D-Matrix. Im Fall des `KineticModel`, dessen Achsen Zeit und Energiezustände sind, wird mit einer Hilfsmethode von allen Zuständen (Spalten) der Grundzustand (die 0. Spalte) abgezogen.

Mit einer weiteren Hilfsmethode `calc_model_and_resid(self, km_mat, sm_mat)` (siehe Listing 4.6) wird aus den beiden Matrizen dann die Modellmatrix und die Residuenmatrix ausgerechnet, aus denen neue Datensatz-Objekt erstellt werden, um sie im `ModelWidget` und `ResidualsWidget` anzeigen zu können.

```
def evaluate_models(self):
    ds = self.datasets["Dataset␣1"]
    # set time axis from dataset
    self.get_and_set_t_axis_for_km_from_ds(ds)
    # set spectral axis from dataset
    self.s_axis = ds.coords["kinetic_energy"]
    # if fit button has been clicked
    if self.run_fit:
        x0 = np.zeros_like(self.initial_values)
        self.fit_result = self.fit(x0)
    # Evaluate Kinetic Model
    km_mat = self.kinetic_model.evaluate(self.t_axis)
    # subtract ground state from all states
    km_mat = self.subtract_gs_from_km_traces(km_mat)
    # Evaluate Spectral Model
    sm_mat = self.spectral_model.evaluate(self.s_axis)
    model, residuals = self.calc_model_and_resid(km_mat, sm_mat)
    model_ds = ds.copy(deep=True, data={"data": model, "weight": ds
        .weight})
    residuals_ds = ds.copy(deep=True, data={"data": residuals, "
        weight": ds.weight})
    return model_ds, residuals_ds
```

Listing 4.5: `Fit`-Methode zum Evaluieren der Modelle

In `calc_model_and_resid(self, km_mat, sm_mat)` wird die im Einleitungskapitel 1.2 geschilderte Matrixmultiplikation der kinetischen und der spektralen Modellmatrizen durchgeführt, deren Ergebnis die (vereinte) Modellmatrix bildet, während die Residuenmatrix durch die Differenz der Datensatzmatrix und der Modellmatrix gegeben ist.

```
def calc_model_and_resid(self, km_mat, sm_mat):
    explicit_model = xr.DataArray(np.transpose(km_mat) @ sm_mat,
        dims=self.datasets["Dataset␣1"].dims)
    flipped_data = np.flipud(self.datasets["Dataset␣1"].data)
    residuals = (flipped_data - explicit_model)
    return explicit_model, residuals
```

Listing 4.6: `Fit`-Methode zur Matrizenberechnung

4.3.2.2 Der Fit-Algorithmus

Bevor evaluiert oder gefittet wird, werden zunächst die beiden Fit Modelle in Abhängigkeit möglicher Änderungen im `FitControlDashboard` mit neuen Parameterwerten geupdated. Ebenfalls werden die Fit-Parameter im engeren Sinne in einem Dictionary als Klassenattribut `self.fit_parameters` gespeichert. Außerdem wird mit Hilfe der Funktion `get_param_vec(mp_dict)` ein Initialwertvektor, ein Faktorvektor und ein Indexvektor aus den Modell-Parametern erstellt und diese als `Fit`-Klassenattribute gespeichert.

In Zeile 9 im Fit-Block von `evaluate_models(self)` wird ein Nullvektor der gleichen Länge des Initialwertvektors erstellt und der Methode `fit(self, x0)` übergeben. Prinzipiell würde der Fit auch funktionieren, wenn man ihm direkt den Initialwertvektor übergäbe. Durch die Normung des Parameterraums werden jedoch numerische Probleme bei der Optimierung vermieden, die bei Parametern sehr unterschiedlicher Größenordnungen auftreten können.

```
1  def fit(self, x0, *args, **kwargs):
2      options = {
3          "maxiter": self.fit_parameters["maxiter"],
4          "disp": self.fit_parameters["disp"]
5      }
6      if self.fit_parameters["method"] in ["Powell", "Newton-CG", "
           TNC"]:
7          options["xtol"] = self.fit_parameters["xtol"]
8      print("STARTING␣FIT...")
9      start = time.time()
10     res = minimize(self.obj_fun, x0, args=(),
11                    method=self.fit_parameters["method"], jac=None,
                          hess=None,
12                    hessp=None, bounds=None, constraints={},
13                    tol=self.fit_parameters["tol"], callback=None,
14                    options=options)
15     end = time.time()
16     print(f"\nFIT␣RESULT\n\n{res}")
17     print(f"\nTime␣the␣Fit␣Algorithm␣took:\t{timedelta(seconds=end␣
           -␣start)}")
18     return res
```

Listing 4.7: `Fit`-Methode `fit(self, x0, *args, **kwargs)`

Die eigentliche Fit-Funktion (siehe Listing 4.7) ist im Wesentlichen ein „Wrapper“ der SciPy-Funktion `optimize.minimize`. Dieser werden hier die Fit-Parameter aus

dem erwähnten Dictionary sowie der normierte Vektor `x0` übergeben. Zudem wird die Zeit gemessen, die der Fit-Algorithmus braucht.

Die Optimierungsfunktion `optimize.minimize` minimiert eine Zielfunktion indem in Iterationen diese immer wieder in Abhängigkeit des Parametervektors `x0` berechnet und dann kleine Veränderungen an den Parametern vorgenommen werden. Dabei können verschiedene Methoden ausgewählt werden, von denen die meisten eine Form des Gradientenabstiegsverfahren nutzen.

Die Zielfunktion („objective function“, siehe Listing 4.8) ist wie bereits in Kaptitel 1.3 erwähnt, in diesem Fall eine Komposition aus der Residuenfunktion („residual function“) und der Verlustfunktion(„loss function“).

```
1 def obj_fun(self, x0, *args):
2     residuals = self.residual_fun(x0, *args)
3     return self.loss_fun(residuals)
```

Listing 4.8: Zielfunktion des `Fit`, die minimiert wird

Was bei UDATTS minimiert werden soll, ist der Unterschied zwischen dem Datensatz und dem mathematischen Modell des Datensatzes. Mathematisch wird dieser Unterschied durch eine Residuenmatrix dargestellt, die in der Residuenfunktion berechnet wird. Allerdings kann `scipy.optimize.minimize` nur eine Funktion mit skalarem Funktionswert minimieren. Aus der Residuenmatrix muss daher noch mittels der Verlustfunktion ein Skalarwert berechnet werden, der den Fehler oder Verlust zwischen Modell und Datensatz beschreibt.

Zukünftig sollen dafür verschiedene Verlustfunktionen auswählbar sein, vorläufig wurde die Verlustfunktion als Summe der Quadrate der Elemente der Residuenmatrix implementiert (siehe Listing 4.9).

```
1 def loss_fun(residuals, name="summed_squares"):
2     loss = 0
3     if name == "summed_squares":
4         loss = np.sum(abs(residuals ** 2))
5     return loss
```

Listing 4.9: Verlustfunktion im `Fit`: Summe der Quadrate

In der Residuenfunktion `residual_fun(self, x0, *args)` (siehe Listing 4.10) muss zunächst der oben erwähnte Normierungsschritt rückgängig gemacht werden, das

heißt aus dem mit einem Nullvektor initialisierten Parametervektor `x0`, mit dem `scipy.optimize.minimize` operiert, wird hier in jeder Iteration wieder ein Vektor `current_values` mit physikalisch sinnvollen Werten errechnet. Hier kommen die vor dem Fitten gespeicherten Initialwertvektoren `self.initial_values` und `self.factors` ins Spiel. Bei letzterem handelt es sich um eine Kopie des ersteren, bei der allerdings alle Werte die gleich 0 sind, durch den Wert 1e-16 ersetzt wurden. Dies ist nötig, damit bei der Multiplikation in Zeile 3 der Residuenfunktion das Ändern dieses Parameters im Fit überhaupt eine Auswirkung hat.

```
1  def residual_fun(self, x0, *args):
2      # unmake norm-step: normalized x0 ==> physical x0
3      current_values = x0 * self.factors + self.initial_values
4      # update models from x0-vector
5      self.update_models_from_vec(current_values)
6      # evaluate kinetic model
7      km_traces = self.kinetic_model.evaluate(self.t_axis)
8      # subtract ground state from all states
9      km_traces = self.subtract_gs_from_km_traces(km_traces)
10     # evaluate spectral model
11     sm_exp = self.spectral_model.evaluate(self.s_axis)
12     # calculate model and residuals
13     model, residuals = self.calc_model_and_resid(km_traces, sm_exp)
14     # calculate weighted residuals
15     ds = self.datasets["Dataset␣1"]
16     weighted_residuals = residuals / ds.weight
17     return weighted_residuals
```

Listing 4.10: Residuenfunktion im `Fit`

Mit dem denormalisierten Parametervektor werden dann die `Fit`-Modelle mit der Methode `update_models_from_vec(self, current_values)` (siehe Listing 4.11) aktualisiert. Im Anschluss werden ganz ähnlich wie in `evaluate_models(self)` die einzelnen Modellmatrizen und aus diesen die vereinte Modellmatrix und die Residuenmatrix ausgerechnet. Diese wird in `residual_fun(self, x0, *args)` allerdings noch gewichtet, indem sie durch eine im Datensatz gegebene Gewichtematrix dividiert wird. In einer zukünftigen Version der Anwendung soll es noch weitere Optionen zur Gewichtung geben.

Von äußerster Wichtigkeit für das Funktioneren des Fit-Algorithmus ist, dass alle Parameter in jeder Iteration wieder an die korrekte Stelle in `KineticModel` und `SpectralModel` zurückgeschrieben werden. Für diese Zuordnung wird der bereits erwähnte Indexvektor `self.mp_indices` benötigt. Dieser wird immer zusammen mit

dem Initialwertvektor aus dem verschachtelten Model-Parameter-Dictionary `mp_dict` erzeugt, welches beim Updaten der `Fit`-Modelle aus dem `ModelParameterTreeView` generiert und dem `Fit` übergeben wird.

```
def update_models_from_vec(self, current_values):
    km_count = 0
    sm_count = 0
    for i, (idx, cv) in enumerate(zip(self.mp_indices,
        current_values)):
        idx_key_list = idx.split(";")
        # Kinetic Model
        if idx.startswith("Kinetic"):
            self.kinetic_model.model_parameter_list[i].
                current_value = cv
            km_count += 1
        # Spectral Model
        elif idx.startswith("Spectral"):
            self.spectral_model.model_parameter_list[i - km_count].
                current_value = cv
            state_name = idx_key_list[1]
            component_name = idx_key_list[2]
            mp_name = idx_key_list[3]
            mp_key = None
            for key, mp in self.spectral_model.
                spectral_model_tree_data[
                    'Spectral' + state_name].spectral_component[
                    'Spectral' + component_name].ModelParameters.
                        items():
                if mp.name == mp_name:
                    mp_key = key
            if mp_key is None:
                print("ERROR: Couldn't find parameter in Spectral
                    Component!")
            self.spectral_model.spectral_model_tree_data['Spectral'
                + state_name].spectral_component[
                'Spectral' + component_name].ModelParameters[mp_key
                    ].current_value = cv
            sm_count += 1
        else:
            print("ERROR IN INDEX-VECTOR!")
```

Listing 4.11: Funktion zum Aktualisieren der Modelle in jeder Fit-Iteration

Bei den Indizes handelt es sich um Strings, die die Position eines Parameters im Dictionary bzw. im Parameterbaum codieren. Dazu werden die Namen von a) Mo-

dell, b) gegebenenfalls Zustand, c) gegebenenfalls Komponente und d) Parameter aneinandergereiht und durch Semikolons separiert, z.B. „*Kinetic Model;sigma*“ oder „*Spectral Model 1;State 3;Component 1;mu*“. Da der Indexvektor und der Initialwertvektor die gleiche Parameterreihenfolge zugrunde legen, ist so eine eindeutige Zuordnung gegeben und die aktualisierten Parameterwerte können in jedem Iterationsschritt korrekt in die Modelle „einsortiert“ werden.

4.3.3 PyQt-Signale beim Fitten

Wenn `Evaluate` oder `Fit` ausgelöst wird, wird dadurch in der UDATTS-Applikation jeweils ein Mechanismus ausgelöst, der auf einer Kette von *PyQt-Signalen* basiert und dafür sorgt, dass die einzelnen Programmteile miteinander kommunizieren und einander die gerade notwendigen Informationen liefern.

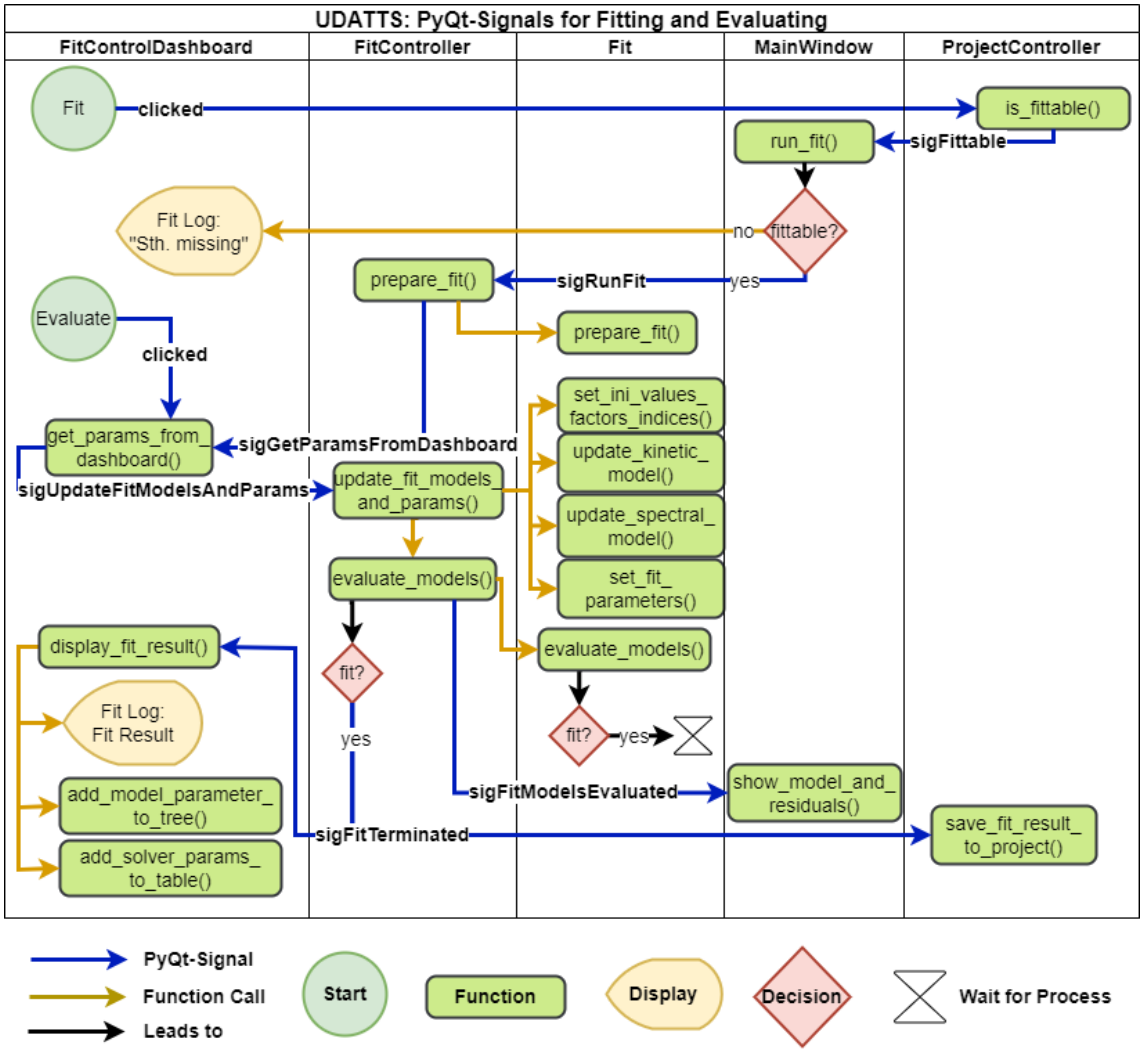

Abbildung 7: Signal-Austausch beim Evaluieren oder Fitten

Abbildung 7 stellt dies in einem Ablaufdiagramm mit *Swimlanes* dar, wobei jede *Swimlane* für eines der beim Evaluieren und Fitten beteiligten Objekte steht. Aus Übersichtsgründen wurden die Unterkomponenten des `FitControlDashboard`, sowie das `Project`, `ModelWidget` und `ResidualsWidget` hier nicht dargestellt. Auslöser der hier dargestellten Abläufe ist der Klick auf einen der beiden Buttons `Evaluate` oder `Fit`, wobei die im letzteren Fall ausgelöste Signalkette, die des ersteren vollständig inkludiert, allerdings um `Fit`-spezifische Routinen erweitert. Der Übersichtlichkeit halber wurden die Parameter der Funktionen (hellgrüne gerundete Kästen) im Diagramm weggelassen.

Wird der `Fit`-Button geklickt, überprüft der `ProjectController` zunächst, ob dafür überhaupt eine konsistente Anzahl von Datensätzen und Modellen im Projekt existiert. Diese Information wird über das Signal `sigFittable` emittiert, das im `MainWindow` von der Methode `run_fit(self)` empfangen wird. Falls die Antwort „nein“ ist, wird im Log-Bereich des `FitControlDashboard` eine Nachricht darüber ausgegeben, dass noch Datensätze oder Modelle fehlen. Falls „ja“, wird das Signal `sigRunFit` gesendet. Dieses löst im `FitController` die Methode `prepare_fit(self)` aus, die eine gleichnamige Methode im `Fit`-Objekt aufruft. Dadurch wird das Flag `self.run_fit` gesetzt. Darüber hinaus emittiert der `FitController` nun das Signal `sigGetParamsFromDashboard`. Dieses löst im `FitControlDashboard` die Methode `get_params_from_dashboard(self)` aus. An dieser Stelle beginnt der `Evaluate` und `Fit` gemeinsame Teil des Ablaufs, da die letztgenannte Methode auch direkt durch den Klick auf `Evaluate` ausgelöst wird.

Von `get_params_from_dashboard(self)` werden alle Parameter aus den Views `ModelParameterTreeView`, `SolverParameterTableView` und `FitParameterView` gelesen und für den `Fit` vorbereitet. Diese werden in Form von sieben Parametern mit dem Signal `sigUpdateFitModelsAndParams` versendet und dann im `FitController` von `update_fit_models_and_params(self, ...)` entgegen genommen. Diese Methode übergibt die jeweils relevanten und unterschiedlich vorverarbeiteten Parameter an vier Methoden im `Fit`, die dort erstens die Initialwert-, Faktor- und Indexvektoren setzen, zweitens das kinetische Modell und drittens das spektrale Modell aktualisieren und viertens die Fit-Parameter setzen. Danach wird im `FitController` die Methode `evaluate_models` aufgerufen, die wiederum gleichnamige und oben diskutierte `Fit`-Methode aufruft.

Wenn zuvor das `Fit`-Flag gesetzt wurde, wird nun der Fit-Algorithmus ausgelöst, was eine Weile dauert, im Diagramm an der Sanduhr erkennbar. Unabhängig davon ob gefittet oder nur evaluiert wurde, erhält der `FitController` die `Dataset`-

Objekte für das Modell und die Residuen vom `Fit` zurück und sendet diese mit dem Signal `sigModelsEvaluated` an das `MainWindow`. Dort werden diese mit Hilfe der Methode `show_model_and_residuals(self, model_ds, residuals_ds)` in den entsprechenden Widgets angezeigt und im unteren Bereich der GUI der Reiter mit dem `ModelWidget` in den Vordergrund geholt.

Wenn gefittet wurde, emittiert der `FitController` nun zusätzlich noch das Signal `sigFitTerminated` und schickt mit diesem die gefitteten Models und das Fit-Ergebnis (vom Typ `scipy.optimize.OptimizeResult`) als Parameter. Dieses Signal wird an zwei Stellen der Anwendung empfangen: vom `FitControlDashboard` wird das Fit-Ergebnis im Log angezeigt und die Model- und Solver-Parameter in der GUI aktualisiert; vom `ProjectController` werden die Fit-Ergebnisse im `Project` gespeichert. Dort werden die alten Models mit den gefitteten überschrieben und im Dictionary `fit_results` ein neuer Eintrag vorgenommen.

5 Test mit Beispieldatensatz

Zum Testen der Anwendung wurde wie in den nicht-funktionalen Anfoderungen in Kapitel 2.2 erwähnt ein spektroskopischer Datensatz von Forschern aus dem HZB verwendet (vgl. [BWKA16]). Das bei diesem physikalisch-chemischen Experiment mit XUV-Photoelektronen-Spektroskopie untersuchte Material war eine ionische Lösung mit sogenannten Tris(bipyridine)ruthenium(II)-Kationen (chemische Formel: $[Ru(bpy)_3]^{2+}$), deren energetisch angeregte Zustände Gegenstand der Untersuchung waren.

Dieser Datensatz wurde von Dr. Christoph Merschjann zum Testen von UDATTS ausgesucht, da der Versuchsaufbau analog war zu jenem für den UDATTS konzipiert ist und die Forscher ebenfalls einen Fit ihrer mathematischen Modelle durchgeführt und publiziert haben. Ein besonderer Vorteil besteht dabei, dass hierbei die Visualisierungen des Datensatzes und des gefitteten Modells eine relative klar erkennbare Form haben, so dass auch ein wissenschaftlich ungeschultes Auge darin einen deutlichen Amplitudenausschlag erkennen und die Ähnlichkeit von Datensatz und Fit-Modell begutachten kann:

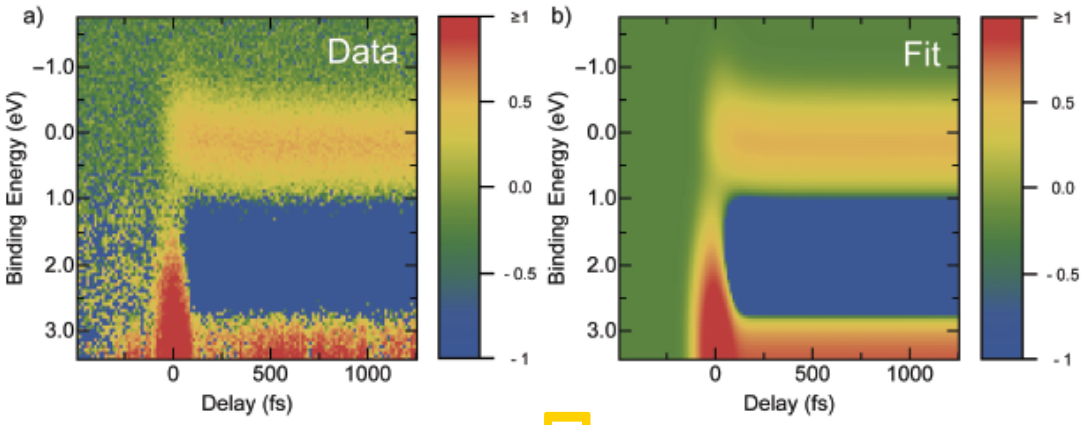

Abbildung 8: Vergleich Datensatz vs. Modell. Bildquelle: Fig. 4 in: [BWKA16]

Ein erfolgreicher Test der UDATTS-Applikation würde sich unter anderem dadurch auszeichnen, dass das damit errechnete Fit-Modell in seiner grafischen Darstellung eine so starke Ähnlichkeit mit dem Datensatz zeigen würde, wie dies in Abbildung 8 der Fall ist. Zur Vorbereitung des Tests wurden von Dr. Merschjann anhand der Daten aus der Publikation ein passendes kinetisches und spektrales Modell in dem JSON-Format angelegt, welches von den Model-Designern importierbar ist. Der erste

Testdurchlauf dauerte ca. 20 Minuten, durchlief 346 Iterationen und erbrachte ein visuell halbwegs zufrieden stellendes Ergebnis:

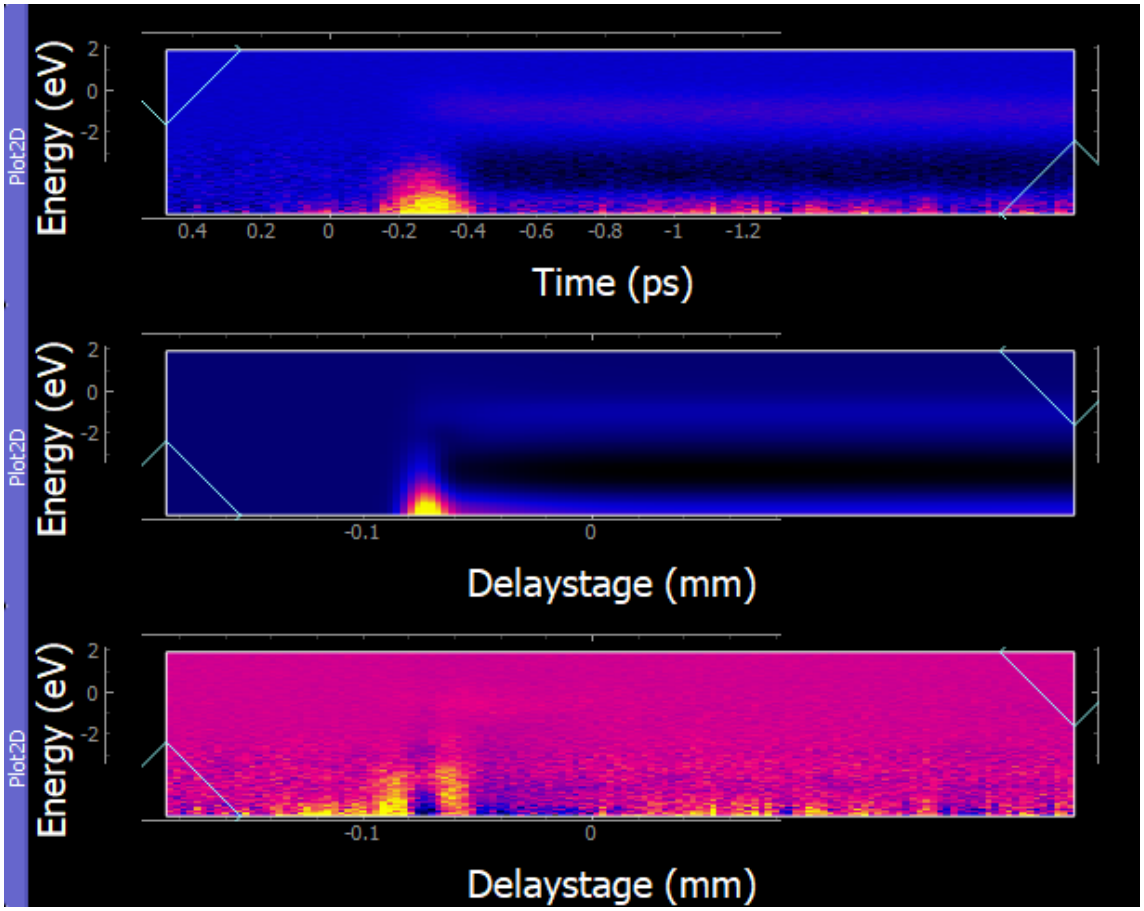

Abbildung 9: Ergebnisse des ersten Fit-Testdurchlaufs

In Abbildung 9 ist erkennbar, dass zwar eine deutliche Ähnlichkeit zwischen Datensatz (oben) und gefittetem Modell (Mitte) besteht, dass aber die energetische Amplitude in letzterem deutlich schmaler ist. Diese Differenz zeigt sich auch in den Residuen (unten), wo in gelb deutlich „überstehende“ Ränder zu sehen sind, die vom Fit-Modell scheinbar nicht „erfasst“ wurden.

Dieser visuellen Auffälligkeit entsprach eine numerische im Fit-Resultat: der Fit-Algorithmus hatte nur die Parameter des `SpectralModel` verändert, nicht aber die des `KineticModel`. Da es hochgradig unwahrscheinlich schien, in einem der beiden Modelle auf Anhieb die exakt optimalen Parameterwerte getroffen zu haben, lieferte dies einen starken Grund zur Annahme eines programmatischen Fehlers.

Zunächst wurde davon ausgegangen, dass die jeweils in einer Fit-Iteration geänderten Parameter nicht korrekt an das `KineticModel` des `Fit` übergeben wurden. Dies

war jedoch nicht die Ursache. Tatsächlich fand sich diese in einer unsauberen Implementierung der Evaluationsmethode im `KineticModel`. Diese produzierte nämlich bei jedem Ausführen einen Seiteneffekt, der das Attribut `self.st_pop_fcn_data` veränderte. Beim Testen eines kinetischen Models im `KineticModelWidget` wird dieser Seiteneffekt nach jedem Evaluieren des Modells aufgrund einer Signalverkettung wieder rückgängig gemacht, beim Aufruf der Methode aus dem `Fit` hingegen nicht. Da es sich hierbei um eine Altlast handelte und das Refactoring der Model-Designer kein Gegenstand der aktuellen Projektphase war, wurde das Problem vorläufig durch einen Hotfix überbrückt. Der anschließende Testdurchlauf erbrachte ein deutlich zufriedenstellenderes Ergebnis.

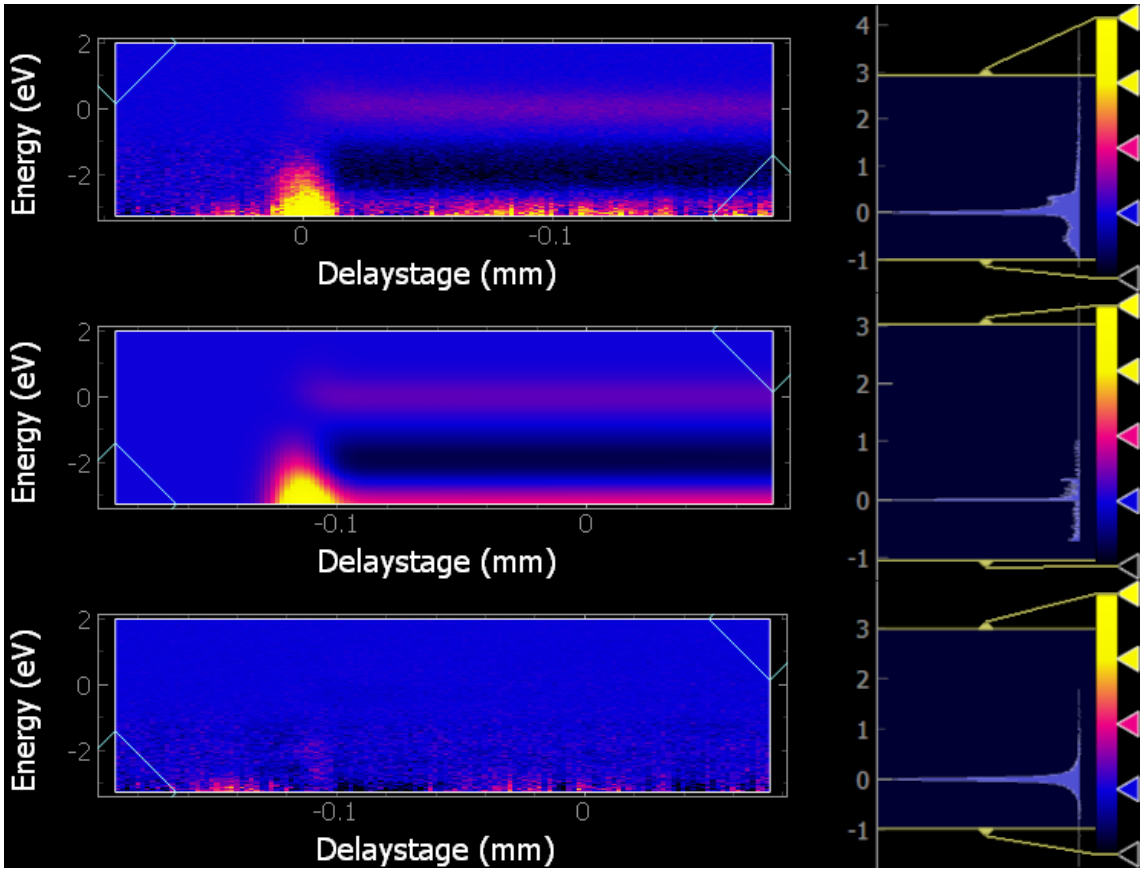

Abbildung 10: Fit-Ergebnisse nach Fehlerkorrektur

Wie Abbildung 10 zeigt, stimmt nach Behebung des Fehlers die Amplitudenbreite zwischen Datensatz und Fit-Modell nun deutlich besser überein und auch die Residuen lassen an der Position der Amplitude weniger überstehende Reste erkennen. Aus dem Fit-Ergebnis auf der Konsole ließ sich zudem ablesen, dass nun auch die Parameter des `KineticModel` vom Fit verändert worden waren. Außerdem benötigte der Fit-Algorithmus nun nur noch ca. 6 Minuten und 85 Iterationen.

6 Fazit

6.1 Evaluation

Hauptziel der in dieser Arbeit diskutierten Projektphase war es, die bisher implementierten, unverbundenen Teile der Applikation so zu verknüpfen und um weitere Programmkomponenten zu erweitern, dass als Ergebnis ein erster funktionsfähiger Prototyp von UDATTS geschaffen wird (vgl. Kapitel 1.4), der den gesamten Soll-Workflow (vgl. Kapitel 3.2) ermöglicht, was an einem Beispieldatensatz gezeigt und dokumentiert werden sollte (vgl. Kapitel 5). Dies kann als im Wesentlichen erfolgreich gelungen gelten.

Aufgrund des hohen Einarbeitungsaufwands und diverser aus Altlasten des Legacy Codes resultierender Probleme, die zusätzlich repariert oder umgangen werden mussten, war jedoch eine vollständige Erfüllung aller gestellten Anforderungen (vgl. Kapitel 2) zeitlich nicht möglich.

Insbesondere wurde beim Testen aus Zeitgründen die ursprünglich vorgesehene Performanzanalyse des neu in Python programmierten Fit-Algorithmus im Vergleich zur bisher verwendeten MATLAB-Implementierung in die Zukunft verschoben. Zur Performanzanalyse ist geplant, mit Hilfe eines geeigneten Python-Profilers eine ausführliche Analyse des Programmcodes vorzunehmen, um Flaschenhälse zu identifizieren (vgl. [Hei22]). Allerdings können bereits jetzt einige Flaschenhälse benannnt und mögliche Maßnahmen zur Optimierung der Performanz von UDATTS benannt werden. Verbesserungspotentiale können etwa in einer effizienteren Zusammenfassung von *PyQt-Signalen* und von unnötig vielen Funktionsaufrufen insbesondere in den den Modellen bzw. Modell-Designern vermutet werden.

Gerade bei den rechnerisch aufwändigen Funktionen in der `Fit`-Klasse, wie der Residuenfunktion verspricht der Einsatz von Python-Erweiterungen wie Cython oder Numba, die dazu dienen Funktionen vorzukompilieren, einen deutlichen Performanzgewinn. Weitere Maßnahme könnten Multiprocessing und / oder Threading sein; diese wäre in jedem Fall sinnvoll, um auch Nutzeraktionen mit der GUI während der Fit-Algorithmus läuft zu ermöglichen, was aktuell nicht möglich ist. Auch die Auslagerung besonders rechenintensiver Prozesse auf die Grafikkarte wäre eine vielversprechende Option zur Steigerung der Performanz.

Unabhängig davon ist auch festzuhalten, dass die aktuelle Implementierung des Fits noch nicht alle möglichen Fälle potentieller spetroskopischer Datensätze abdeckt. Ganz grundlegend muss die `Fit`-Klasse so umgeschrieben werden, dass ein Fit auch mit Projekten mit mehr als einem Datensatz möglich ist. Außerdem werden in der aktuellen Implementierung beim Fitten noch nicht die Spalten *Fixed*, *Lower Bound* und *Upper Bound* aus dem `FitControlDashboard` berücksichtigt. Zudem sind aus physikalischen Gründen weitere Rechenschritte in der oben diskutierten Residuenfunktion `residual_fun(self, x0, *args)` zu ergänzen. Dies sind unter anderem folgende:

1. Beim `KineticModel` ist zu unterscheiden, ob dieses in *impliziter* oder *expliziter* Form vorliegt (siehe Kapitel 1.2) und je nachdem in der Rechnung unterschiedlich zu behandeln. Dafür muss allerdings auch das `KineticModelWidget` erweitert werden, da aktuell nur *implizite* kinetische Modelle möglich sind.

2. Analog dazu muss in der Implementierung berücksichtigt werden, dass ein `SpectralModel` nicht nur *explizite*, sondern auch *implizite* Komponenten haben kann (siehe Kapitel 1.2), die in der Modellberechnung vom `SpectralModel` abgezogen und gesondert behandelt werden müssen. Auch dafür muss zunächst der entsprechende Model-Designer, das `SpectralModelWidget` erweitert werden.

3. Eine in der Spektroskopie sogenannte Instrument Response Function (IRS) ist zu implementieren. Mit dieser IRS muss das `KineticModel` mathematisch gefaltet werden, um den Messumständen Rechnung zu tragen. Zuvor muss dafür noch auf Grundlage der IRS eine optimale linearisierte Zeitachse für die Faltung errechnet und das `KineticModel` auf dieser evaluiert werden. Nach der Faltung wird das `KineticModel` zurück auf die korrekte Zeitachse interpoliert.

4. Die in Kapitel 4.3.2.2 diskutierte Subtraktion des Grundzustands von den Zustandsspalten der `KineticModel`-Matrix, ist abhängig von einem *Flag* zu machen, der im `KineticModelWidget` gesetzt werden können muss.

5. Für die Gewichtung der Residuenmatrix soll es drei Optionen geben: a) eine mit dem Datensatz explizit gelieferte Gewichte-Matrix, b) eine Gleichgewichtung, c) eine aufgrund einer Funktion aus der Datenmatrix errechnete Gewichte-Matrix.

Außerdem sollen nach einem erfolgreichen Fit-Durchlauf noch die Konfidenzintervalle aller Model-Parameter ausgerechnet und in der entsprechenden Spalte im `ModelParameterTreeView` angezeigt werden.

6.2 Ausblick

Ausgehend vom nun geschaffenen Prototypen von UDATTS können diese Korrekturen und Erweiterungen nun in Angriff genommen werden.

Mittelfristig sind bereits weitere Features geplant. So soll z.B. das Fit-Modul derart erweitert werden, dass im `FitControlDashboard` weitere optionale Fit-Parameter, die von der SciPy-Funktion `optimize.minimize` verarbeitet werden können, konfigurierbar sind. Idealerweise sollen diese in Abhängigkeit davon, welche Fit-Solver-Methode eingestellt ist, ein- oder ausgeblendet werden.

Eine weitere zukünftige Aufgabe ist, die `Dataset`-Klasse so zu überarbeiten, dass ein aktuell im `Fit` provisorisch implementiertes Preprocessing entfällt.

Schließlich ist langfristig eine vollständiges Refaktorisierung der alten Programmteile, also des `DataWidget` und der MVC-Module des `KineticModel` und `SpectralModel` geplant.

Die Möglichkeiten zur Weiterentwicklung von UDATTS sind mannigfaltig; die Basis dafür wurde nun in Form eines ersten Prototypen geschaffen.

Literaturverzeichnis

[BWKA16] Borgwardt, M., Wilke, M., Kiyan, I. Y., and Aziz, E. F. (2016). Ultrafast excited states dynamics of $[Ru(bpy)_3]^{2+}$ dissolved in ionic liquids. *Physical Chemistry Chemical Physics*, 41(18):28893–28900.

[BZ11] Burkard, R. E. and Zimmermann, U. T. (2011). *Einführung in die Mathematische Optimierung*. 1 Edition. Springer Verlag, Berlin, Heidelberg.

[GHJV01] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2001). *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. 1 Edition. Addison-Wesley, München.

[Hei22] Heinz, M. (2022). Profiling and Analyzing Performance of Python Programs. https://towardsdatascience.com/profiling-and-analyzing-performance-of-python-programs-3bf3b41acd16. Abgerufen: 2022-08-03.

[KFW09] Kuhn, H., Försterling, H.-D., and Waldeck, D. H. (2009). *Principles of Physical Chemistry*. 2. Edition. Wiley-Interscience, Hoboken, New Jersey.

[Lak17] Lakowicz, J. R. (2017). *Principles of Fluorescence Spectroscopy*. 3. Edition. Springer, Berlin.

[Pie17] Pieper, M. (2017). *Mathematische Optimierung : Eine Einführung in die kontinuierliche Optimierung mit Beispielen*. 1 Edition. Springer Verlag, Wiesbaden.

[Pyt22] Python Software Foundation (2022). PyQt5 5.15.7. https://pypi.org/project/PyQt5/. Abgerufen: 2022-08-02.

[Sne13] Snellenburg, J. (2013). Glotaran. https://glotaran.org/. Abgerufen: 2022-07-25.

[SLS+12] Snellenburg, J., Laptenok, S., Seger, R., Mullen, K., and Stokkum, I. (2012). Glotaran: A Java-Based Graphical User Interface for the R Package TIMP. *Journal of Statistical Software*, 49(3):1–22.

[The22] The Qt Company Ltd. (2022). Signals Slots. https://doc.qt.io/qt-5/signalsandslots.html. Abgerufen: 2022-07-29.

[The08] The Scipy community (2008). scipy.optimize.minimize. https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.optimize.minimize.html. Abgerufen: 2022-07-25.

[T.Y20] T.Y. (2020). Common Manipulation of QTreeview using PyQT5. http://pharma-sas.com/common-manipulation-of-qtreeview-using-pyqt5/. Abgerufen: 2022-07-15.

[uJS04] und Josef Stoer, F. J. (2004). *Optimierung*. 2 Edition. Springer Verlag, Berlin, Heidelberg.

[Wei18] Weigend, M. (2018). *Python 3. Lernen und professionell anwenden. Das umfassende Praxisbuch*. 7. Edition. MITP Verlag, Frechen.

[xar14a] xarray Developers (2014a). xarray.Dataset. https://docs.xarray.dev/en/stable/generated/xarray.Dataset.html. Abgerufen: 2022-08-01.

[xar14b] xarray Developers (2014b). xarray.Dataset.to_netcdf. https://docs.xarray.dev/en/stable/generated/xarray.Dataset.to_netcdf.html. Abgerufen: 2022-08-01.

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich:

1. dass ich meine Bachelor-Thesis selbstständig verfasst habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe,

3. dass ich meine Bachelor-Thesis bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum	Alexander Matthias Schmidt