

---

ABSCHLUSSPRÜFUNG SOMMER 2022  
ENTWICKLUNG EINES SOFTWARESYSTEMS

---

VORGELEGT VON: SVEN BERGMANN  
VORGELEGT AM: 12. MAI 2022  
PRÜFLINGSNUMMER: 101 20541  
AUSBILDUNGSBETRIEB: CAE GMBH

# Inhaltsverzeichnis

<b>1 Aufgabenbeschreibung</b>	<b>4</b>
1.1 Aufgabenanalyse . . . . .	4
1.2 Beschreibung der mathematischen Methoden . . . . .	4
1.2.1 Algorithmus 1 . . . . .	4
1.2.2 Algorithmus 2 . . . . .	5
1.2.3 Algorithmus 3 . . . . .	5
1.2.4 Algorithmus 4 . . . . .	5
1.3 Konzepterstellung zur Nebenläufigkeit von Einlesen, Verarbeiten und Ausgeben . . . . .	6
1.4 Einlesen und Initialisieren der Daten . . . . .	7
1.5 Ausgabe gemäß Aufgabenstellung . . . . .	7
<b>2 Objektorientierter Entwurf</b>	<b>7</b>
2.1 Framework . . . . .	7
2.2 Problem . . . . .	8
2.3 ProblemSimple . . . . .	9
2.4 Utils . . . . .	10
<b>3 Änderungen zur schriftlichen Ausarbeitung</b>	<b>10</b>
3.1 Umbenennungen . . . . .	10
3.2 Modifikation der Klassenstruktur . . . . .	11
3.3 Modifikation der Logik . . . . .	11
<b>4 Allgemeines zu Multithreading in Java</b>	<b>12</b>
<b>5 Auswertung, Interpretation und Fehlererkennung</b>	<b>13</b>
<b>6 Benutzeranleitung</b>	<b>13</b>
6.1 Ordnerstruktur . . . . .	14
6.2 Benötigte Programme . . . . .	14
6.3 Ausführen als Kommandozeilenprogramm . . . . .	14
6.3.1 Ausführen der JAR . . . . .	15
6.3.2 Ausführen des Python-Skripts . . . . .	15
<b>7 Zusammenfassung und Ausblick</b>	<b>15</b>
<b>A UML-Diagramme</b>	<b>17</b>
A.1 Klassendiagramme . . . . .	17
A.1.1 Übersicht . . . . .	17
A.1.2 Klassen im Paket „framework“ . . . . .	18
A.1.3 Klassen im Paket „problem“ . . . . .	20
A.1.4 Klassen im Paket „problemsimple“ . . . . .	23
A.1.5 Klassen im Paket „utils“ . . . . .	24

A.2 Nassi-Shneidermann Diagramme . . . . .	25
A.3 Sequenzdiagramme . . . . .	32
<b>B Bilder der resultierenden Funktionen</b>	<b>34</b>
<b>C Eigenständigkeitserklärung</b>	<b>39</b>

# 1 Aufgabenbeschreibung

## 1.1 Aufgabenanalyse

Eine Firma möchte einen optischen Autokorrelator<sup>1</sup> bauen, welcher mit einem Laser, halbdurchlässigen und voll reflektierenden Spiegeln, einem Kristall, einem Filter und einem Detektor funktioniert. Der Detektor misst hierbei die Autokorrelationsfunktion des Eingangssignals, die Aufschluss über die Breite des Laserpulses oder die Ursprungsfrequenz gibt. Die Messdaten werden allerdings vom Detektor permanent und nebenläufig geliefert, was mit einem Thread modelliert werden soll, welcher mit  $20Hz$ , also alle  $0,05$  Sekunden eine neue Messdatei zur Verfügung stellt und die alte überschreibt. Der Lese-Thread ist hierbei Thread A und läuft ständig weiter, indem die Daten "0.txt" bis "9.txt" immer wieder eingelesen werden und den aktuellen Datensatz überschreiben. Der Thread B soll dann diese Messdaten weiterverarbeiten und mit den später erklärten mathematischen Methoden die Funktion modellieren, bzw. die Messwerte umrechnen und glätten. Schließlich wird die fertige Berechnung dann an Thread C weitergegeben, um die Daten in seinem auch später erklärten Format in eine Datei auszugeben, welche als Präfix "out" besitzt und ansonsten gleich heißt. Das Programm wird dann beendet, wenn alle Messdaten verarbeitet wurden. Es kann durchaus vorkommen, dass eine Messdatei häufiger eingelesen wird. Diese muss dann allerdings nicht erneut verarbeitet werden, es wird dann einfach auf eine noch nicht verarbeitete Datei gewartet.

## 1.2 Beschreibung der mathematischen Methoden

### 1.2.1 Algorithmus 1

Als erstes werden alle  $\hat{x}$ -Werte in Pikosekunden umgerechnet, was mit der Formel

$$\hat{x}_k = \frac{\tilde{x}_k}{2^{18} - 1} \cdot 266,3 - 132,3$$

passieren soll. Zur Zeitersparnis und Verbesserung der Genauigkeit ist hierbei eine Umstellung sinnvoll, sodass der Bruch  $\frac{266,3}{2^{18}-1}$  nicht immer wieder erneut berechnet werden muss. Also wird es eine Variable  $\alpha$  geben, welche diesen Bruch beschreibt und die Formel wird zu

$$\hat{x}_k = \alpha \cdot \tilde{x}_k - 132,3$$

Des Weiteren werden noch alle  $y$ -Werte normiert, sprich jeder Wert  $y_k$  wird durch das Maximum aller  $y$ -Werte geteilt. Die Formel ist in Abbildung 17 auf Seite 26 in einem Nassi-Shneidermann Diagramm beschrieben.

---

<sup>1</sup><https://de.wikipedia.org/wiki/Autokorrelator>

### 1.2.2 Algorithmus 2

Danach sollen die Daten geglättet werden, indem der sogenannte gleitende Mittelwert berechnet werden soll. Die Formel hierfür ist

$$x_k = \frac{1}{n} \sum_{i=0}^n \hat{x}_{k-\tau+i}$$

mit  $\tau = \frac{n-1}{2}$  und

$$n = \begin{cases} \lfloor 0,002 \cdot N \rfloor - 1 & \text{für } \lfloor 0,002 \cdot N \rfloor \text{ gerade} \\ \lfloor 0,002 \cdot N \rfloor & \text{für } \lfloor 0,002 \cdot N \rfloor \text{ ungerade} \end{cases}$$

wobei  $n$  die Größe des Mittelungsfensters beschreibt, ungerade ist und 0,2% von  $N$  entspricht. Hierbei wird wieder die Optimierung vorgenommen, dass der Bruch  $\frac{1}{n}$  nicht für jede Iteration erneut berechnet wird, sondern anfangs durch  $m = \frac{1}{n}$  dargestellt wird. Es soll zudem noch geeignete Werte für  $k < \tau$  und  $k > N - 1 - \tau$  bestimmt werden, sprich den linken und rechten Rand der Messungen. Hierbei habe ich mich dafür entschieden, dass ich Fallunterscheidungen für den index des zu summierenden Elements genommen habe, sprich

$$index = \begin{cases} 0 & k - \tau + i < 0 \\ N - 1 & k - \tau + i > N - 1 \end{cases}$$

Das Symbol  $\lfloor x \rfloor$  heißt untere Gaußklammer und beschreibt, wie auf ganze Zahlen gerundet werden soll. Laut Definition gilt also  $\lfloor x \rfloor := \max\{k \in \mathbb{Z} \mid k \leq x\}$ , was in Java zu „Math.floor(double a)“ wird. Die Formel ist in Abbildung 18 auf Seite 27 in einem Nassi-Shneidermann Diagramm beschrieben.

### 1.2.3 Algorithmus 3

Als nächstes wird die obere Einhüllende bestimmt, welche die Autokorrelationsfunktion oben komplett einschnürt. Da dies aber numerisch sehr aufwändig ist, reicht es aus, diese Einhüllende einfacher zu approximieren und zwar indem zuerst von links beginnend jedem Positionswert der zuletzt höchste Intensitätswert zugeordnet wird, solange bis man am Maximum ist, dann wird das gleiche Verfahren von rechts wiederholt. Es werden also die lokalen Maxima im Bereich von  $[0, x]$  für  $x$  gesucht, so lange, wie der  $x$  Wert noch nicht das globale Maximum erreicht hat. Auf der anderen Seite wird dann im Bereich von  $[y, 1]$  so lange lokale Maxima gesucht, wie der  $y$  Wert noch nicht das globale Maximum erreicht hat. Am Ende wird noch der  $x$  Wert des globalen Maximums gesetzt. Die Formel ist in Abbildung 19 auf Seite 28 in einem Nassi-Shneidermann Diagramm beschrieben.

### 1.2.4 Algorithmus 4

Zuletzt wird noch die Pulsbreite  $b$  berechnet, welche sich über den Abstand der beiden Punkte  $L$  und  $R$  bestimmen lässt. Diese sind Punkte auf der oberen Einhüllenden, wobei

die Gerade die durch  $L$  und  $R$  geht, genau auf der Hälfte der Grundlinie und des Maximums liegt. Die Grundlinie stellt dabei die mittlere Höhe des äußersten linken Prozents der Intensitätswerte dar. Es müssen also die gemittelten Werte der ersten Berechnung im Intervall  $[0; 0, 01]$  genommen werden. Diese werden dann ebenfalls gemittelt und der daraus resultierende Wert beschreibt die Grundlinie. Realisiert wird das Ganze, indem zuerst die Anzahl der Punkte berechnet wird, die in diesem Bereich liegen. Unter Benutzung der oberen Gaußklammer mit der analogen Definition zur unteren,  $\lceil x \rceil := \min\{k \in \mathbb{Z} | k \geq x\}$ , ist die Anzahl dann  $n = \lceil N \cdot 0, 01 \rceil$ . Aus dieser Anzahl an Punkten wird dann der Mittelwert gebildet und dann mit  $a_{12} = \frac{1 - \text{mittelwert}}{2} + \text{mittelwert}$  die halbe Höhe zwischen Grundlinie und Maximum bestimmt. Daraus kann dann der index von  $L$  und  $R$  bestimmt werden, indem jeweils von links, bzw. rechts beginnend, solange der Wert des Arrays der oberen Einhüllenden ausgelesen wird, bis  $a_{12}$  erreicht ist. In meinem Algorithmus wird der erste Wert genommen, der darüber ist, man könnte allerdings auch den letzten Wert nehmen, welcher darunter ist. Die Formel ist in Abbildung 20 auf Seite 29 in einem Nassi-Shneidermann Diagramm beschrieben.

### 1.3 Konzepterstellung zur Nebenläufigkeit von Einlesen, Verarbeiten und Ausgeben

Bei dem gegebenen Problem kann ganz strikt nach dem IPO (Input-Process-Output) oder EVA (Eingabe-Verarbeitung-Ausgabe) Prinzip gearbeitet werden, wobei jedoch nur genau ein Thread für das Liefern der Daten, also den Input zuständig ist. Hierfür wird ein Interface bereitgestellt werden, welches ein Runnable implementiert. Dieses Interface gibt eine „read“ Methode vor, welche Daten einliest. Die zu implementierende Klasse dieses Threads wird dann diese Methode mit  $20Hz$  aufrufen. Der Thread B wird dann, wie auch der Thread C, so lange laufen, wie noch Daten nicht verarbeitet wurden. Wenn Thread A also eine Datei bereitstellt, startet Thread B und berechnet das Ergebnis, welches dann vom Thread C geschrieben werden kann. Thread A, B und C können somit gleichzeitig ausgeführt werden, da die Verarbeitung unabhängig voneinander passiert. Für Thread B wird dann also ebenfalls ein Interface bereitgestellt, welches ein Runnable implementiert und die Methode „process“ mit den Parametern aus Thread A bereitstellen muss. Der Thread C sieht dann ebenfalls ein Interface vor, welches ein Runnable implementiert und die Methode „write“ bereitstellt. Um die Kommunikation zwischen den einzelnen Thread-Klassen zu realisieren, wird das Observer-Observable Pattern genutzt werden. Der Thread A wird also von Thread B beobachtet und Thread B von Thread C.

In meiner Ausarbeitung finden sich zwei verschiedene Implementierungen der Threads B und C wieder, eine Implementierung mit dem Master-Worker Pattern und eine ohne. Die Implementierung ohne das Pattern beschreibt genau das gegebene Problem und den (wahrscheinlich) gewünschten Ablauf und die andere Implementierung ist eine etwas optimiertere Lösung, bei welcher beim Programmstart die Größe des Threadpools und der Warteschlange, bzw. die Anzahl der Worker und die Anzahl der später zu bearbeitenden Elemente angegeben werden kann. Dies findet dann in Thread B statt. Beim Thread C wird angenommen, dass das Schreiben der Daten schneller passiert, als die Verarbeitung und daher wird dort auch das Master-Worker Pattern angewandt, jedoch ohne Begren-

zung der Anzahl der Worker, oder der Anzahl der Elemente in der Warteschlange. Die Elemente werden dann asynchron geschrieben.

Falls von Thread B alle Messdaten bearbeitet und von Thread C geschrieben wurden, wird das Programm beendet.

## 1.4 Einlesen und Initialisieren der Daten

Das Einlesen der Daten passiert über die „read“ Methode<sup>2</sup> in ThreadA<sup>3</sup>. Besonders große Schwierigkeiten gibt es hierbei nicht, da alle Zeilen mit einem „#“ startend ignoriert werden können. Die weiteren Zeilen bezeichnen die Messwerte, welche mit „\t“ getrennt sind. Hierbei ist die erste Zahl der y-Wert, also die Intensität des Signals und die zweite Zahl der x-Wert, also die Position des Spiegels. Falls während des Einlesevorgangs eine Exception geschmissen wird, wird der Vorgang einfach abgebrochen und null zurückgegeben. Das habe ich so realisiert, da der ThreadA<sup>4</sup> die Daten schnell lesen muss und keine lange Fehlerbehandlung machen kann, falls diese korrupt sind. Die Werte von  $x$  und  $y$  werden zuerst in Paare eingelesen, welche in einer Liste gespeichert werden. Laut Aufgabenstellung sind dies immer positive, ganzzahlige Werte. Diese Liste von Paaren wird dann in zwei Arrays, xStart und yStart, umgewandelt und mit dem Dateinamen in ein Data Objekt gegeben und von der Methode zurückgegeben.

## 1.5 Ausgabe gemäß Aufgabenstellung

Die Ausgabe der fertigen Autokorrelationsfunktion<sup>5</sup> erledigt dann ThreadC<sup>6</sup>. Hierfür habe ich eine Methode<sup>7</sup> in der Klasse AKF geschrieben, welche einen formatierten String für die Ausgabe zusammenbaut und zurückgibt. Da in der Klasse der Wert für die Pulsbreite, den IndexL und den indexR gesetzt ist, ist die erste Zeile kein Problem. Die weiteren Zeilen werden dann mit den Arrays „xTransformiert“, „yNormiert“ und „obereEinhuellende“ zusammengebaut.

Wenn der String fertig ist, wird das Ganze in eine Datei mit dem Namen „out“ + „fileName“ aus der Klasse AKF geschrieben.

# 2 Objektorientierter Entwurf

## 2.1 Framework

Das Paket Framework beinhaltet alle wichtigen Interfaces und eine abstrakte Klasse zur Lösung des gegebenen Problems. Somit wird die geforderte Austauschbarkeit des Threads A erreicht, man kann aber auch alle anderen Threads austauschen.

---

<sup>2</sup>s. Abbildung 16 auf Seite 25

<sup>3</sup>s. Abbildung 9 auf Seite 21

<sup>4</sup>s. Abbildung 9 auf Seite 21

<sup>5</sup>s. Abbildung 7 auf Seite 20

<sup>6</sup>s. Abbildung 11 auf Seite 22

<sup>7</sup>s. Abbildung 21 auf Seite 30

Das vorher genannte Observer-Observable-Pattern<sup>8</sup> habe ich mit einem Interface „Observer“ und einer abstrakten Klasse „Observable“ vorgegeben. Der „Observer“ ist generisch und muss die Methode „void update(T t)“ bereitstellen, welche vom „Observable“ aufgerufen werden kann. Ebenfalls generisch gestaltet sich die „Observable“ Klasse, die eine Liste von „Observern“ hält. In diese Liste kann man über die Methode „public void registerObserver(Observer observer)“ einen „Observer“ hinzufügen. Diese werden dann über die Methode „public void notifyObservers(T t)“ über ein verändertes Objekt informiert.

Das Threading ist durch die drei Interfaces „Runnable“<sup>9</sup>, „WriteRunnable“<sup>10</sup> und „ProcessRunnable“<sup>11</sup> implementiert, welche selbst das Interface „Runnable“ erweitern. Der Lesethread soll durch das Interface „ReadRunnable“ implementiert werden, welches die Methode „T read(Path pathToFile)“ bereitstellt und generisch ist. „ProcessRunnable“ stellt das Interface für den Verarbeitungsthread dar und gibt die Methode „R process(P p)“ vor, wobei R der Typ des Ergebnisparameters ist und P der Type des Eingabeparameters. Schlussendlich wird der Schreibthread durch das Interface „WriteRunnable“ dargestellt, wobei man die Methode „boolean write(T t)“ implementieren muss, die einen generischen Parameter schreiben soll.

## 2.2 Problem

Problem beinhaltet alle Klassen zur (optimierten) Lösung des gegebenen Problems, welche unter anderem von der Klasse und den Interfaces aus „Framework“ erben.

Die Klasse „Algorithms“<sup>12</sup> beinhaltet alle vier Algorithmen, welche über die Methode „public static AutoKorrelationsFunktion solve(Data data)“ ausgeführt werden können. Die einzelnen Algorithmen sind daher auch private, sodass Zugriffe von Außen nicht möglich sind. Als Input Parameter nimmt die Funktion ein Objekt der Klasse „Data“<sup>13</sup>, welches als Record<sup>14</sup> implementiert ist und den Namen der Datei, das Array der eingelesenen x-Werte und das Array der eingelesenen y-Werte beinhaltet. Diese Werte werden dann intern durch die Algorithmen zu einem neuen Objekt vom Typen „AutoKorrelationsFunktion“ zusammengebaut. Dieser Typ ist ebenfalls als Record realisiert und hält den Dateinamen, die Pulsbreite, den indexL, den indexR, das double-Array der transformierten x-Werte, das double-Array der normierten y-Werte und das double-Array der Werte der oberen Einhüllenden.

Die Klasse „ThreadA“<sup>15</sup> implementiert das Interface „Runnable<Data>“, erweitert die abstrakte Klasse „Observable< Pair<Data, Integer>>“ und stellt eben genau den Thread A aus der Aufgabenstellung dar. Als Konstruktor-Parameter wird der relative Pfad zum Ordner der Eingabedateien und der sleepTime Wert als long angegeben. Die Methode „public void run()“ des Runnable Interfaces wird so überschrieben, dass zuerst alle Pfade

---

<sup>8</sup>s. Abbildung 2 auf Seite 18

<sup>9</sup>s. Abbildung 4 auf Seite 19

<sup>10</sup>s. Abbildung 5 auf Seite 19

<sup>11</sup>s. Abbildung 3 auf Seite 18

<sup>12</sup>s. Abbildung 6 auf Seite 20

<sup>13</sup>s. Abbildung 8 auf Seite 21

<sup>14</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html>

<sup>15</sup>s. Abbildung 9 auf Seite 21



im Eingabeordner ausgelesen werden und dann eine Endlosschleife beginnt. In dieser Endlosschleife wird immer wieder durch die Pfade iteriert, die jeweilige Datei eingelesen<sup>16</sup> und alle Observer mit einem Paar aus dem erstellten Data Objekt und der Anzahl der Daten im Ordner weitergegeben. Danach wartet der ThreadA so lange, wie es in sleeptime angegeben ist, laut Aufgabenstellung 0,05s, wonach der Schleifendurchlauf weitergeht, oder von vorne beginnt. Die Methode zum Einlesen der Daten in das Data Objekt ist in Abschnitt 1.4 beschrieben.

Klasse „ThreadB“<sup>17</sup> erweitert die abstrakte Klasse „Observable<AutoKorrelationsFunktion>“, implementiert die Interfaces „Observer<Pair<Data, Integer>>“ und „ProcessRunnable<Data, AutoKorrelationsFunktion>“ und stellt den Thread B der Aufgabenstellung dar. Der Konstruktor fordert „maxPoolSize“ und „maxQueueSize“ als Parameter und erstellt dadurch einen neuen ThreadPoolExecutor<sup>18</sup> mit den gegebenen Parametern. Dieser Executor wird dann für das Master-Worker Pattern genutzt, indem die von „ThreadA“ eingelesenen Daten an verfügbare Worker zur Weiterbearbeitung verteilt werden. Falls kein Worker mehr zur Verfügung steht, wird die aktuelle Datei verworfen und die nächste eingelesene Datei bearbeitet, falls bis dahin wieder Worker frei sind. Sobald eine Datei fertig bearbeitet ist und das Objekt vom Typen „AutoKorrelationsfunktion“ erstellt wurde, werden wieder alle Observer mit dem neuen Objekt informiert.

„ThreadC“<sup>19</sup> implementiert die Interfaces „Observer<AutoKorrelationsFunktion>“ und „WriteRunnable<AutoKorrelationsFunktion>“ und stellt den Thread C der Aufgabenstellung dar. Der Konstruktor nimmt hier nur den Pfad des Ausgabeordners. In der Methode „public void update(AutoKorrelationsFunktion akf)“ wird für jedes neu geschickte Objekt ein Runnable erstellt, welches mit „CompletableFuture.supplyAsync(()->this.write(akf))“ ausgeführt und in ein HashSet gepackt wird. Somit kann jede Schreiboperation asynchron ausgeführt werden. Falls der Thread C dann ein null Objekt im Update übergibt, so wird auf jedem Worker „join“ aufgerufen, also auf die Ausführung gewartet und danach das Programm beendet. Die Methode „public boolean write(AutoKorrelationsFunktion akf)“<sup>20</sup> wurde bereits in Abschnitt 1.5 beschrieben.

Die Zusammenarbeit dieser Threads wird auch nochmal in Abbildung 24 auf Seite 33 grafisch beschrieben.

## 2.3 ProblemSimple

In „problemsimple“ findet sich eine alternative Implementierung des gegebenen Problems wieder, es wurden allerdings nur die Klassen „ThreadB“<sup>21</sup> und „ThreadC“<sup>22</sup> neu geschrieben, alle anderen Klassen werden weiter benutzt. Wie der Name schon sagt, wird hier naiv und mit einer einfachen Fehlerbehandlung an das Problem herangegangen. Wäh-

---

<sup>16</sup>s. Abbildung 16 auf Seite 25

<sup>17</sup>s. Abbildung 10 auf Seite 22

<sup>18</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html>

<sup>19</sup>s. Abbildung 11 auf Seite 22

<sup>20</sup>s. Abbildung 22 auf Seite 31

<sup>21</sup>s. Abbildung 12 auf Seite 23

<sup>22</sup>s. Abbildung 13 auf Seite 23

rend dieser Ausführung existieren nur genau diese drei Threads A, B und C, wobei das Master-Worker Pattern nicht implementiert ist.

„ThreadB“ erweitert nun die abstrakte Klasse „Observable<Pair<AutoKorrelationsFunktion, Integer>>“ und implementiert die Interfaces „Observer<Pair<Data, Integer>>“ und „ProcessRunnable<Pair<Data, Integer>, AutoKorrelationsFunktion>“. Dies ist nötig, da nun die Anzahl der zu bearbeitenden Elemente an „ThreadC“ weitergeschickt werden muss. Die update Methode dieses Threads wird nun so ausgeführt, dass zuerst geprüft wird, ob der Thread gerade eine Datei bearbeitet. Wenn ja, wird die Datei einfach übersprungen und aus der Funktion herausgesprungen. Falls der Thread frei ist, wird die übergebene Datei bearbeitet und das Ergebnis inklusive der Anzahl der Dateien in dem Eingabeordner an den Thread C zur Weiterbearbeitung geschickt.

„ThreadC“ implementiert nun die zwei Interfaces „Observer<Pair<AutoKorrelationsFunktion, Integer>>“ und „WriteRunnable<AutoKorrelationsFunktion>“, um die Anzahl der Daten geschickt bekommen zu können. Die Methode „public boolean write(AutoKorrelationsFunktion akf)“ ist gleichgeblieben, allerdings wird nun die update Methode anders implementiert. Falls ein Observable ein Update schickt, wird zuerst geschaut, ob der Thread schreibend oder frei ist. Falls der Thread schreibend ist, wird die Datei einfach nicht weiterbearbeitet, andernfalls wird zuerst geprüft, ob die Anzahl der Daten der Größe der Menge von Dateinamen entspricht. Ist dem so, so wird das Programm beendet, andernfalls wird geprüft, ob der Dateiname in der Menge der Dateinamen vorhanden ist, bzw. schon geschrieben wurde. Trifft das zu, wird einfach aus der Methode herausgesprungen, ansonsten wird das Objekt der Methode write übergeben und der Dateiname in der Menge der Dateinamen gespeichert.

Die Zusammenarbeit dieser Threads wird auch nochmal in Abbildung 23 auf Seite 32 grafisch beschrieben.

## 2.4 Utils

Das Paket „utils“ beinhaltet zwei Klassen, „CmdLineParser“<sup>23</sup> und „Pair“<sup>24</sup>. Der „CmdLineParser“ bekommt als Konstruktorparameter das String-Array der Kommandozeilenargumente des Programmaufrufs übergeben, versucht diese auszulesen und zu setzen. Mit den Getter Methoden können diese dann z. B. von der Main-Klasse abgerufen, und in die Threads übergeben werden. Des Weiteren existiert noch das Record „Pair“, welches ein nicht veränderliches generisches Objekt von zwei Typen K und V darstellt.

# 3 Änderungen zur schriftlichen Ausarbeitung

## 3.1 Umbenennungen

Es sind fast alle der Benennungen so wie in der schriftlichen Ausarbeitung geblieben, es wurden nur alle Benennungen an das Deutsche angepasst. Die Java-spezifischen Be-

---

<sup>23</sup>s. Abbildung 14 auf Seite 24

<sup>24</sup>s. Abbildung 15 auf Seite 24

nennungen wie z. B. „getAttribut“ oder „setAttribut“ sind geblieben. Zudem wurden natürlich noch die Umlaute umschrieben.

### 3.2 Modifikation der Klassenstruktur

In der schriftlichen Ausarbeitung wurden noch keine Paketnamen angegeben. Wie bereits genannt, existieren die Pakete „framework“, „problem“ und „utils“. Des Weiteren wollte ich jede Thread Klasse sowohl von Observable, Observer, als auch einem der drei Thread-Interfaces erben lassen. Hier kommt aber das Diamond-Problem zum tragen und Java verbietet daher das Erben mehrerer abstrakter Klassen. Ich habe mich also für mehrere Interfaces entschieden und benutze jetzt das „Runnable“ Interface anstatt die Thread Klasse. Ansonsten ist die Klassenstruktur im Groben gleich geblieben.

### 3.3 Modifikation der Logik

Wie schon gesagt werden mittlerweile Runnables<sup>25</sup> anstatt Threads<sup>26</sup> genutzt. Dies hat den Vorteil, dass nun die Klassen Thread A, B und C von mehreren Interfaces wie z. B. „Runnable“ und „Observer“ erben können und zusätzlich die abstrakte Klasse „Observable“ erweitern können. Jede Threadklasse überschreibt also nur die „run()“ Methode und benutzt die Methoden des Observer-Observable Patterns. Eine weitere Modifikation besteht darin, dass nun der Thread A den Thread B als Observer registriert hat und der Thread B den Thread C als Observer. Das heißt, dass die Threads nun in einer Reihe verkettet sind und nicht mehr wie bei der schriftlichen Ausarbeitung in einem Ring.

Es wurden zwei mögliche Implementierungen dargestellt, einmal mit dem Master-Worker Pattern und einmal ohne. Die Implementierung des Master-Worker Patterns befindet sich im Paket „problem“, während die einfache Implementierung im Paket „problemsimple“ zu finden ist. Beide Implementierungen nutzen die gleiche Klasse ThreadA, unterscheiden sich aber in den Klassen ThreadB und ThreadC und vor allem auch in der Abbruchbedingung des Programms.

In „problem“ wird das ganze Programm erst von Thread C beendet, falls dieser in der „update(T t)“ Methode null zurück bekommt, da nur der ThreadB weiß, wann keine Daten mehr eingelesen, bzw. verarbeitet werden. Sowohl ThreadB, als auch ThreadC implementieren das Master-Worker Pattern und bei Thread B kann zusätzlich angegeben werden, wie groß der Threadpool sein soll und wie viele Objekte in die Warteschlange gepackt werden dürfen. ThreadC schreibt einfach alle gepushten Daten asynchron, da davon ausgegangen wird, dass die Verarbeitung länger dauert, als das Schreiben.

In „problemsimple“ wird nur mit genau diesen drei Threads gearbeitet und keinen Weiteren. Um dies zu realisieren, gibt es in beiden Klassen nun eine Variable „processing“ oder „writing“, welche beide vom Typen AtomicBoolean<sup>27</sup> sind, da diese Variablen threadsafe sind. Falls nun der ThreadA zu schnell Daten pusht, sodass ThreadB in einem

<sup>25</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html>

<sup>26</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.State.html>

<sup>27</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/AtomicBoolean.html>

bearbeitenden Zustand ist, so überspringt ThreadB einfach diese Datei und bearbeitet die nächste, sobald er fertig ist. ThreadC macht das ähnlich, woraus auch resultiert, dass ThreadC die Abbruchbedingung in diesem Fall verwalten muss. Die Anzahl der Input Dateien wird also von ThreadA über ThreadB nach ThreadC geschleust und in ThreadC wird dann immer geprüft, ob die aktuelle Datei schon geschrieben wurde und ob schon alle Daten geschrieben wurden. Ist dies der Fall, wird das Programm beendet.

Die Änderungen habe ich vorallem vorgenommen, da ich für meine Lösung zwingend das Observer-Observable Pattern brauche und daher ein Observable die Klasse erweitern muss. In Java ist es wie schon gesagt nicht möglich, mehrere Klassen zu erweitern, wodurch Threads ausgeschlossen wurden. Ich habe mich daher zur Implementierung von Runnables entschieden. Weiterhin habe ich zwei Lösungen für das gegebene Problem implementiert, da ich denke, dass es – vorallem auf einem Multicoresystem – sinnvoll sein könnte, das Master-Worker Pattern zu implementieren. So kann nämlich der Thread B nur die Aufgaben verteilen und überwachen, ob noch Kapazitäten frei sind. In der „simplen“ Lösung können nicht alle Ressourcen genutzt werden und der Thread B ist einfach blockiert. Ähnlich läuft das mit Thread C, bei dem allerdings asynchrone Methodenaufrufe genutzt wurden. Da heutzutage nahezu jedes System ein Multicoresystem ist wäre das denke ich auch die bessere Lösung.

## 4 Allgemeines zu Multithreading in Java

In Java kann Multithreading auf verschiedene Arten realisiert werden. Das etwas ältere, aber durchaus noch genutzte Konzept besteht darin, die Thread Klasse zu erweitern und die Methoden, vor allem die „run()“ Methode, zu überschreiben. Der Vorteil hierbei ist, dass von Haus aus Basisfunktionalitäten, wie „yield()“, oder „sleep()“ bereitgestellt werden. Eine weitere Methode ist es, das Interface „Runnable“ zu implementieren, was den Vorteil mit sich bringt, dass man mehrere Interfaces implementieren kann, aber nur eine Klasse erweitern kann. „Runnable“ wird auch von der Klasse „Thread“ implementiert, stellt aber nur Basisfunktionalitäten bereit, wie die „run()“ Methode. Ein Objekt vom Typen „Runnable“ kann aber dafür genutzt werden dieses von mehreren Threads ausführen zu lassen. Das Ausführen der beiden Objekte verhält sich auch unterschiedlich. Ein Objekt des Typs „Thread“ kann einfach mit „new Thread“ erstellt werden und dann mit „Thread.start()“ gestartet werden, wodurch jedes Mal eine neue Instanz eines Thread Objektes erstellt wird. „Runnables“ hingegen werden entweder einem vorher erstellten „new Thread()“ Objekt als Parameter mitgegeben und dann gestartet, oder aber einem Executor Service übergeben, welcher entweder selbst erstellt und verwaltet werden kann, oder einem bereits vorgegebenen Service zugeordnet werden kann. Vorgegebene Services sind zum Beispiel der „newCachedThreadPool()“, welcher neue Threads erstellt, solange diese gebraucht werden und alte wieder nutzt, oder der „newFixedThreadPool(int nThreads)“, welcher eine fixe Anzahl an Threads nutzt, um die Aufgaben auszuführen und ansonsten eine Queue benutzt, in welcher die Aufgaben gespeichert und dann nach und nach abgearbeitet werden. Je nach Programm oder Problemstellung kann man also diese Services nutzen. In meinem Programm nutze ich in der „Main“ zur Ausführung der

selbstgeschriebenen Thread Klassen, welche das Interface „Runnable“ implementieren, einen „newFixedThreadPool“ mit 3 Threads. Diese Runnables werden dann von diesem Threadpool verwaltet. Falls man nun nicht nur wie in dieser Aufgabenstellung laufende Threads haben möchte, sondern auch ein Ergebnis aus diesen Threads erwartet, mit dem dann weitergemacht werden soll, kann man die sogenannten „CompletableFutures“ nutzen, welche über den Funktionsaufruf „CompletableFuture.supplyAsync(<Runnable>)“ erstellt und asynchron ausgeführt werden. Mit dem „CompletableFuture“<sup>28</sup> Objekt kann dann mit Methoden wie z. B. „thenApply()“ weitergearbeitet werden. Diese werden in dem ThreadC der Methode mit der Implementierung des Master-Worker Patterns genutzt. Um das Ganze allerdings etwas lebensechter zu machen habe ich in ThreadB einen eigenen ThreadPoolExecutor erstellt, welcher die gewünschte Anzahl an Threads und die gewünschte Größe der Warteschlange setzt. Somit kann man gut simulieren, was passiert, wenn keine Worker mehr verfügbar sind und die Queue zudem auch voll ist.

## 5 Auswertung, Interpretation und Fehlererkennung

Da keine wirklichen Prüfdaten gegeben waren, gegen die man die Algorithmen hätte testen können und zudem jeder Algorithmus eine andere Bedingung hatte, was mit dem Index passiert, falls  $k < \tau$  oder  $k > N - 1 - \tau$ , war die Auswertung und vor allem auch die Fehlerbewertung nur bedingt sinnvoll. Es konnte also anhand der resultierenden Bilder<sup>29</sup> ungefähr geprüft werden, ob die Funktion, bzw. die Datenpunkte ähnlich zueinander sind. In diesem Fall sind die resultierenden Bilder ähnlich zu den Gegebenen. Des Weiteren war in der Problembeschreibung festgelegt, dass die Messdaten immer von der Länge  $N$  sind, jedoch wurde  $N$  nicht explizit angegeben. In einem Live System würde  $N$  feststehen und die Daten könnten direkt auf Korrektheit geprüft werden. Dies wird hier nicht gemacht, da einfach angenommen wird, dass die eingelesenen Messdaten die gleiche Länge haben. Wenn ein Ausreißer in den Messdaten vorhanden ist, so wird dies auch vom Algorithmus nicht überprüft, da kein Referenzrahmen existiert oder gegeben ist, wodurch man die Messwerte auf Korrektheit prüfen könnte. Vielmehr ist es der Fall, dass die Kurve durch nur einen Ausreißer mit einem Unterschied von 2 Stellen, egal ob bei den x-, oder y-Werten, sich sehr verzerrt und überhaupt nicht mehr der ursprünglichen Kurve entspricht.

## 6 Benutzeranleitung

Generell befindet sich die Gesamtdokumentation der Klassen und Methoden als javadoc im Ordner „javadoc“.

---

<sup>28</sup><https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CompletableFuture.html>

<sup>29</sup>s. Bilder der resultierenden Funktionen

## 6.1 Ordnerstruktur

An sich sollten die fertig gebaute „jar“ Datei zusammen mit dem ausführbaren Python Skript in einem Ordner liegen. Die Testbeispiele sollten ebenfalls auf der gleichen Ebene in einem Ordner vorhanden sein, wobei der Ordnername beim Programmstart oder in dem Skript angegeben werden kann. Der Ordner für alle Ausgabe Dateien wird dann ebenso in dieser Ebene erstellt. Die Ausgabedateien werden pro Eingabedatei jeweils in eine Datei namens „out<Name der Eingabedatei>.txt“ im output Ordner gespeichert, welcher entweder über die Programmzeile vorher angegeben wird, oder einfach „output“ heißt.

## 6.2 Benötigte Programme

Für die Ausführung der „jar“ Datei benötigt der Zielrechner zwingend eine Installation des Open JDK 17<sup>30</sup>, da das Java-Sprachlevel auf 17 gesetzt wurde. Um das Python Skript auszuführen wird eine Python Installation gebraucht, wobei ich die Version 3.10<sup>31</sup> benutzt habe. Um das Projekt zu bauen, wird Gradle<sup>32</sup> genutzt.

## 6.3 Ausführen als Kommandozeilenprogramm

Es gibt zwei Wege das Programm auszuführen, einmal direkt über Java und einmal über das Python Skript, welches intern die „jar“ Datei aufruft.

Das Programm nimmt Argumente entgegen, welche sind:

- -inputfolder {„Name des Ordners mit den Eingabedateien“}, default: „input“
- -outputfolder {„Name des Ordners, wohin die Ausgabedateien geschrieben werden“}, default: „output“
- -log {„true“ oder „false“ oder „file“}, default: „file“
- -loglvl {„warning“ oder „info“}, default: „all“
- -poolsize {„Anzahl der gewünschten Threadpools“}, default: „1“
- -sleep {„Zeit in Millisekunden mit welcher der ThreadA Daten pusht“}, default: „50“
- -queuesize {„Größe der Schlage der Tasks“}, default: „1“
- -mw {„true“ oder „false“}, default: „false“

Man kann jede dieser Optionen setzen, muss das aber nicht tun. Falls eine Option nicht gesetzt wird, wird der default Wert angenommen.

---

<sup>30</sup><https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

<sup>31</sup><https://www.python.org/downloads/release/python-3100/>

<sup>32</sup><https://gradle.org/>

Die Namen der Ordner werden immer relativ zum derzeitigen Ordnerpfad ausgewertet. Die Logoption beschreibt das Verhalten des Loggers, beziehungsweise wohin die Logs geschrieben werden. Der LogLevel setzt das Level der geloggten Nachrichten, sprich welche Nachrichten tatsächlich geloggt werden. Poolsize gibt an, wie groß die Anzahl der gewünschten Threads sein soll, die der ThreadPool von Thread B maximal bereitstellt. Sleep gibt die Zeit in Millisekunden an, die der ThreadA wartet, nachdem dieser Daten gepusht hat und QueueSize beschreibt die Größe der Schlange des Services des ThreadsB. Die Option „-mw“ gibt an, ob das Programm die Thread Klassen mit dem intern implementierten Master-Worker Pattern nutzen soll, oder die Threads ohne das Master-Worker Pattern.

### 6.3.1 Ausführen der JAR

Um die „.jar“ ausführen zu können, muss zumindest unter Windows der Pfad zum JDK 17 in den Umgebungsvariablen gesetzt sein. Mit einem Doppelklick auf die Datei wird der Code mit Standardargumenten ausgeführt. Ansonsten könnte der Benutzer auch über die Kommandozeile gehen und die Programmargumente selbst setzen. Das würde dann beispielsweise so aussehen:

```
java -jar IHK_Abschlusspruefung.jar -inputfolder input -outputfolder  
output -log file -loglvl warning -poolsize 1 -sleep 50 -queueSize 1
```

### 6.3.2 Ausführen des Python-Skripts

Der Benutzer kann zusätzlich auch noch das Python-Skript „execute\_gro\_pro.py“ ausführen, um das Programm mit den Standardargumenten zu starten. Hierfür ist eine Installation von Python notwendig, sowie die Verlinkung zur Umgebungsvariablen. Der Kommandozeilencode sieht dann beispielsweise so aus:

```
python execute_gro_pro.py
```

## 7 Zusammenfassung und Ausblick

In dieser großen Programmieraufgabe musste ein Softwaresystem erstellt werden, welches theoretisch aus drei unabhängigen Systemen besteht. Gegeben war die Aufgabe, einen Autokorrelator zu simulieren, welcher mit 20Hz neue Messdaten liefert, was durch einen eigenständigen Thread realisiert werden sollte. Um die Daten zu verarbeiten wurde ein weiterer Thread benötigt, der neue Messdaten überspringen sollte, falls diese zu schnell geschickt werden und dieser Thread die Daten nicht schnell genug verarbeiten kann. Die verarbeiteten Daten mussten danach vom letzten Thread noch geschrieben werden. Um die Übergabe einer unbegrenzten Anzahl von Daten simulieren zu können, wurde ein Ordner mit Dateien von Messwerten gegeben, welche ständig eingelesen und Thread B zur Verfügung gestellt werden sollten. Falls nun jede Datei bearbeitet und geschrieben wurde, kann das Programm beendet werden.

Die Algorithmen im verarbeitenden Thread zu implementieren war keine wirklich große Schwierigkeit, einzig für das Verhalten des Indexes bei Algorithmus 2 für  $k < \tau$  und  $k > N - 1 - \tau$  musste eine geeignete Wahl getroffen werden.

Falls das Programm später tatsächlich in einem Produktivsystem genutzt werden würde, könnte man sich überlegen, die Daten welche letztendlich in Thread C ankommen, direkt anzeigen zu lassen. So hätte man immer einen grafischen Bezug zum aktuellen Versuchsaufbau. Des Weiteren könnte eine Client-Server Architektur implementiert werden, wobei der Server den Thread A darstellt und mit diesen 20Hz Daten über ein beliebiges Protokoll überträgt. Diese Daten könnten dann wiederum von unterschiedlichen Clients, sprich unterschiedlich implementierten Auswertungsprogrammen, abgeholt und verarbeitet werden. Somit hätte man eine weitere Ebene zur Austauschbarkeit geschaffen. Diese Clients könnten dann natürlich immer andere Algorithmen implementieren und selbst sozusagen wieder als Server funktionieren, da nun auch unterschiedliche Schreibthreads als Client fungieren könnten. Braucht der Benutzer beispielsweise nur die eben angesprochene grafische Ausgabe wäre es ja wenig sinnvoll, immer alle Daten zu schreiben, besser könnten die Daten dann einfach direkt vom Clienten angezeigt werden. Andersrum, wenn die Daten noch weiter ausgewerten werden sollten, oder auch nur abgespeichert werden sollten, wäre es auch nicht sinnvoll, diese immer wieder anzeigen zu lassen.





### A.1.2 Klassen im Paket „framework“

Abbildung 2: Observer-Observable

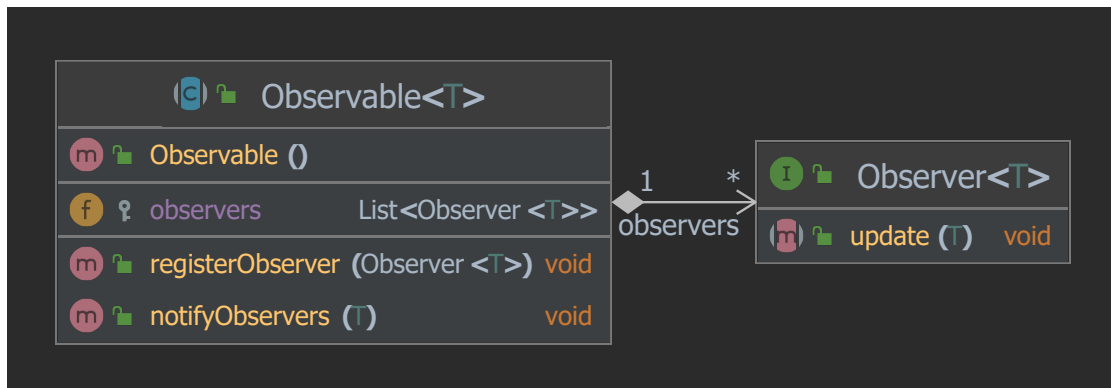


Abbildung 3: ProcessRunnable

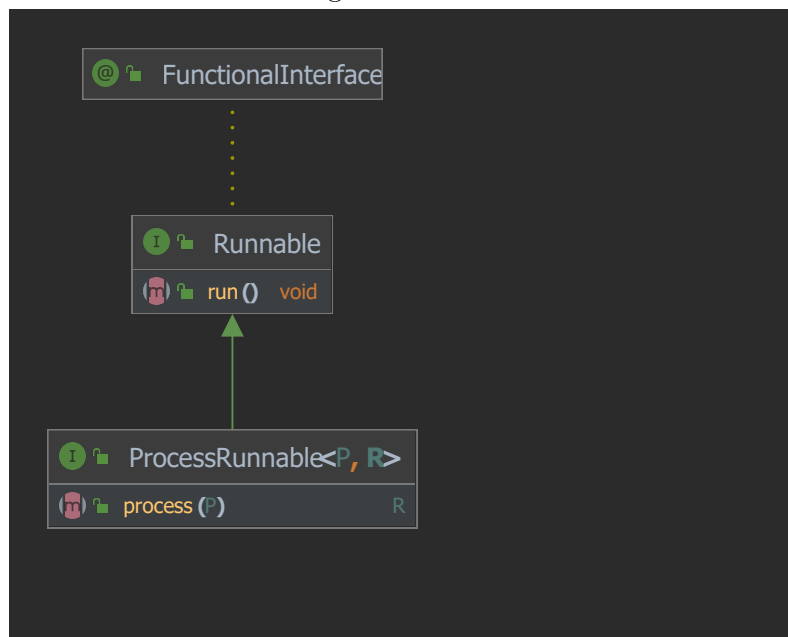


Abbildung 4: ReadRunnable

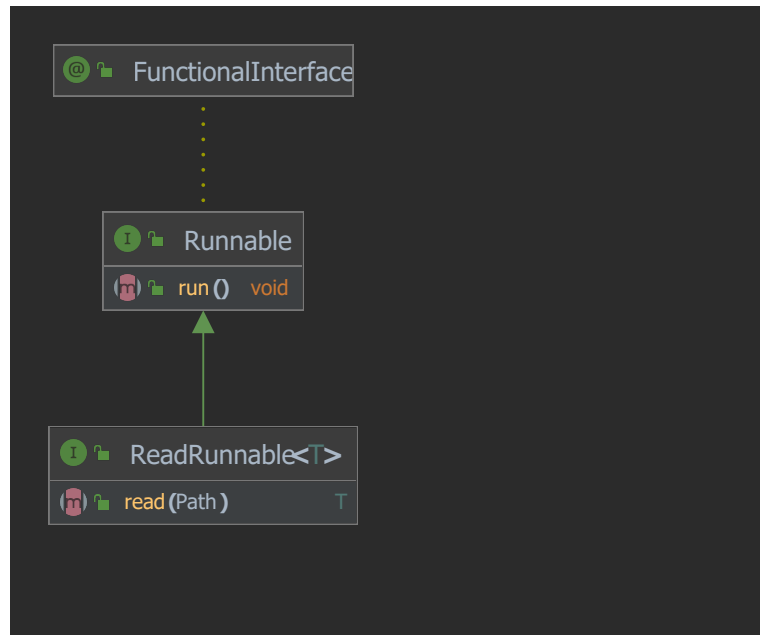
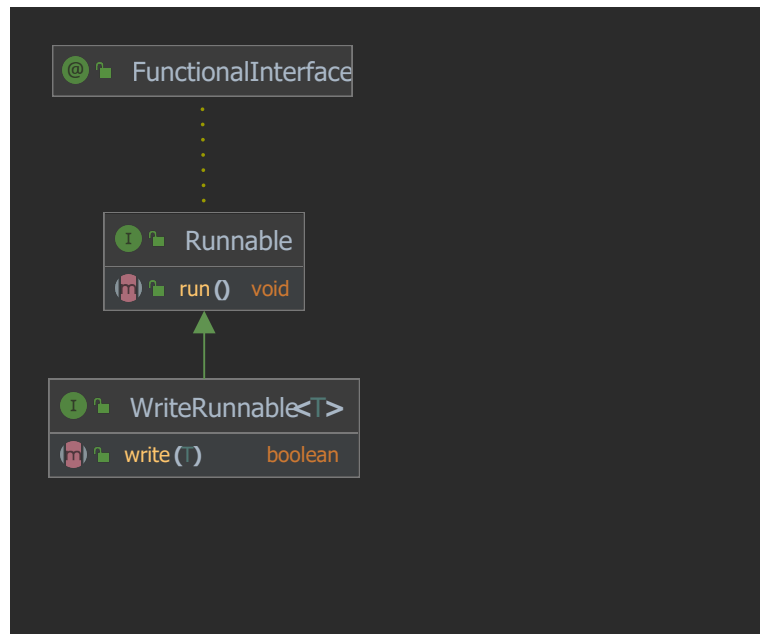


Abbildung 5: WriteRunnable



### A.1.3 Klassen im Paket „problem“

Abbildung 6: Algorithms

Algorithms	
Algorithms ()	
alg1 (int [], int [])	Pair<double [], double []>
alg2 (double [])	double []
alg3 (double [])	double []
alg4 (double [], double [], double [])	Pair<Float, Pair<Integer, Integer>>
max (int [])	int
solve (Data)	AutoKorrelationsFunktion

Abbildung 7: AutoKorrelationsFunktion

AutoKorrelationsFunktion	
AutoKorrelationsFunktion (String, float, int, int, double [], double [], double [])	
fileName()	String
fwhm()	float
indexL()	int
indexR()	int
obereEinhuellende()	double []
xTransformiert()	double []
yNormiert()	double []
outputString	String

Abbildung 8: Data

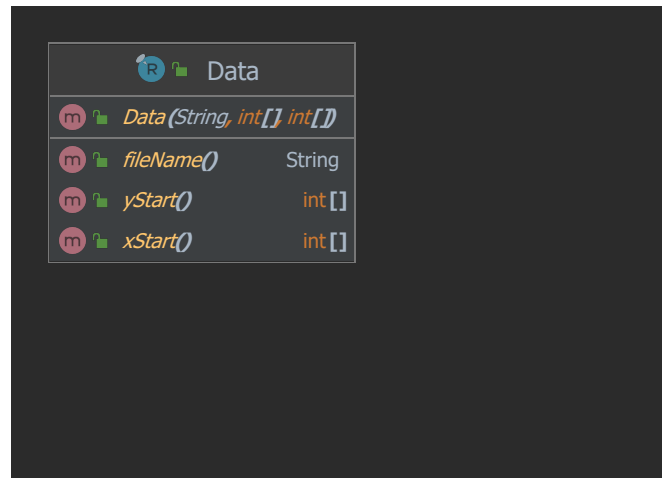


Abbildung 9: ThreadA

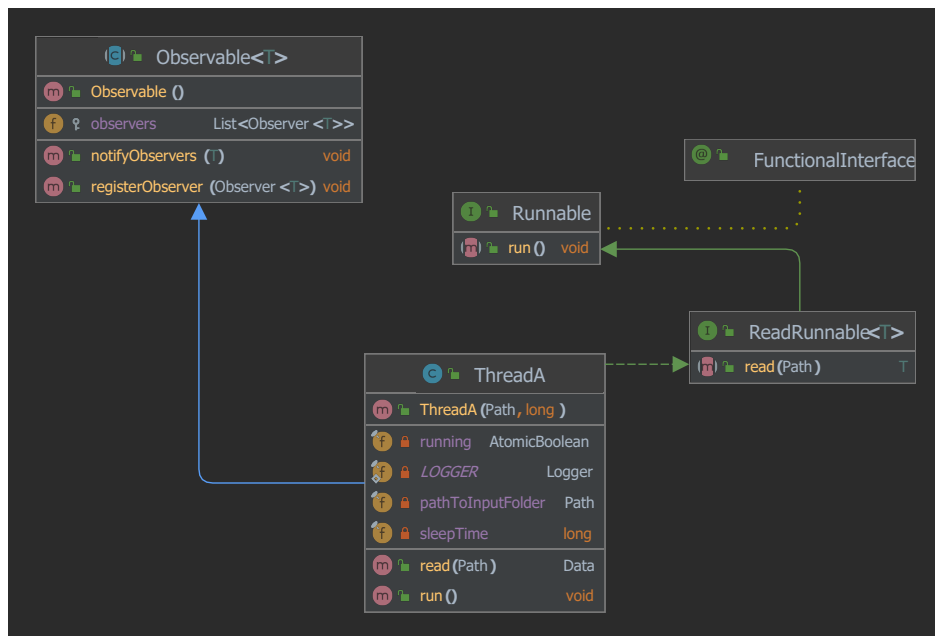


Abbildung 10: ThreadB

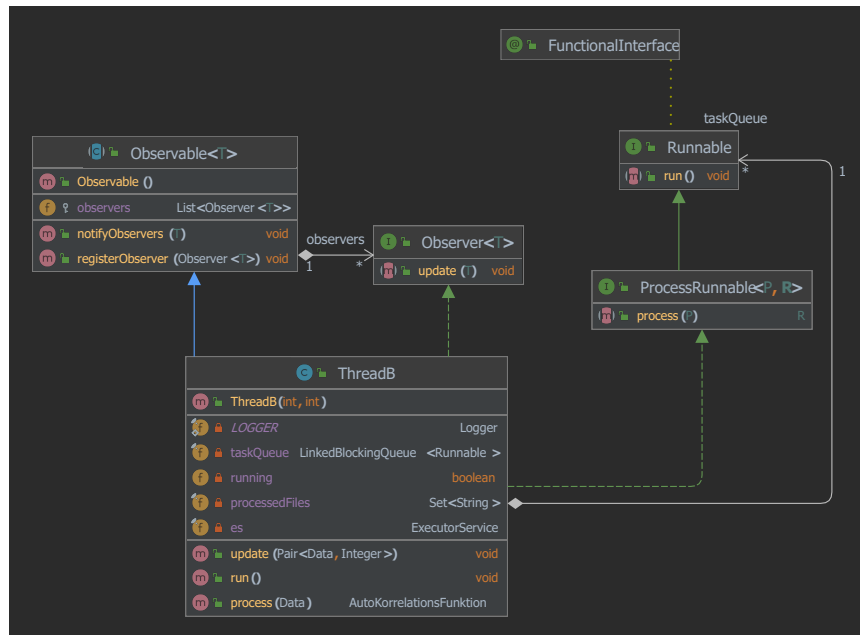
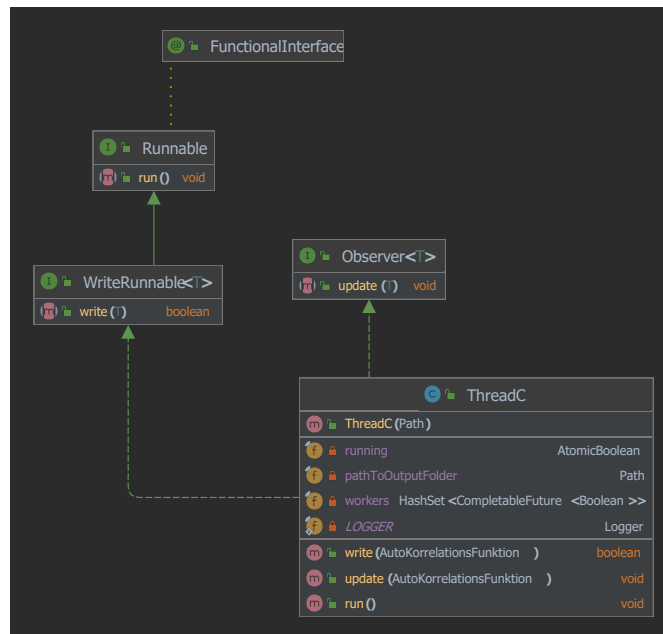


Abbildung 11: ThreadC



#### A.1.4 Klassen im Paket „problemsimple“

Abbildung 12: ThreadB\_simple

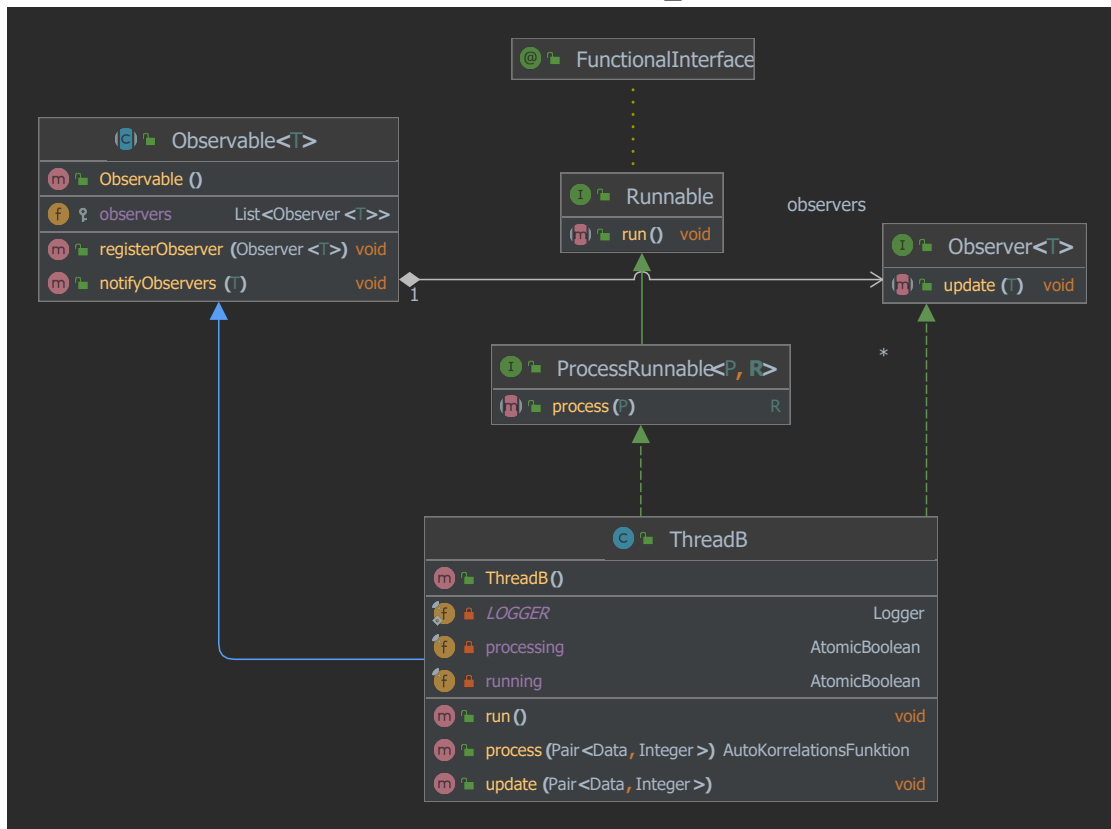
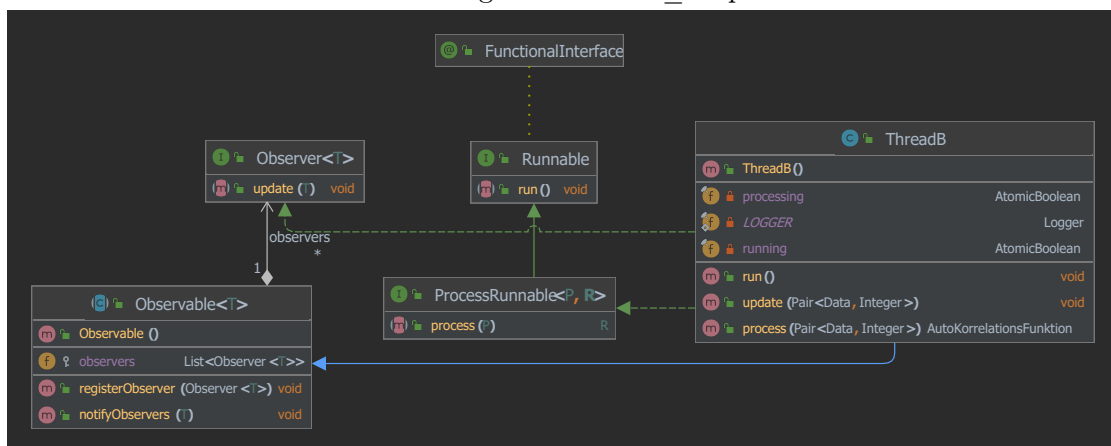
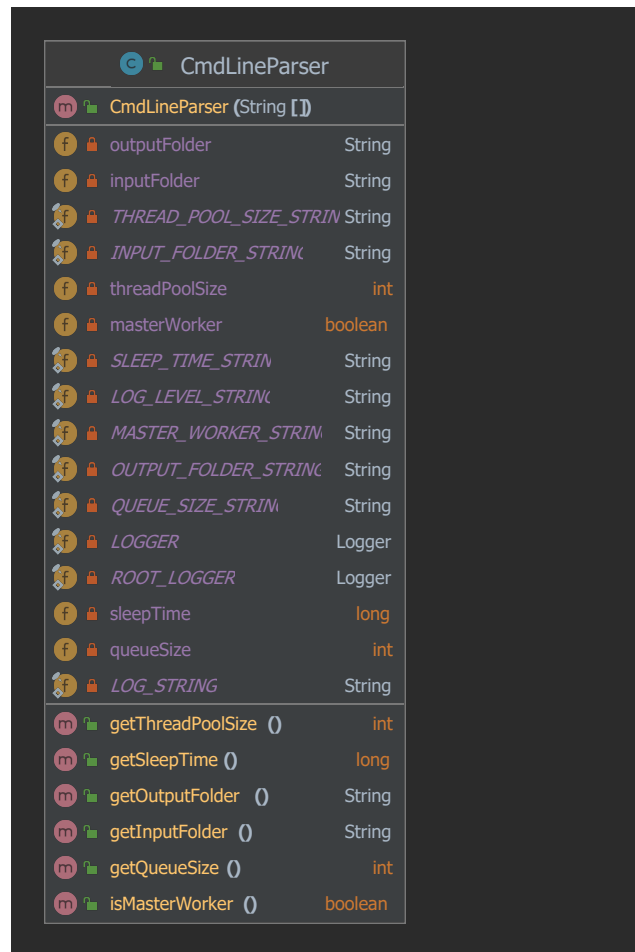


Abbildung 13: ThreadC\_simple



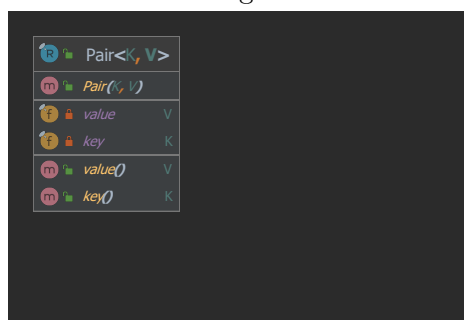
### A.1.5 Klassen im Paket „utils“

Abbildung 14: CmdLineParser



CmdLineParser		
CmdLineParser (String [])		
f	outputFolder	String
f	inputFolder	String
f	THREAD_POOL_SIZE_STRIN	String
f	INPUT_FOLDER_STRIN	String
f	threadPoolSize	int
f	masterWorker	boolean
f	SLEEP_TIME_STRIN	String
f	LOG_LEVEL_STRIN	String
f	MASTER_WORKER_STRIN	String
f	OUTPUT_FOLDER_STRIN	String
f	QUEUE_SIZE_STRIN	String
f	LOGGER	Logger
f	ROOT_LOGGER	Logger
f	sleepTime	long
f	queueSize	int
f	LOG_STRING	String
m	getThreadPoolSize ()	int
m	getSleepTime ()	long
m	getOutputFolder ()	String
m	getInputFolder ()	String
m	getQueueSize ()	int
m	isMasterWorker ()	boolean

Abbildung 15: Pair



Pair<K, V>		
Pair (K, V)		
f	value	V
f	key	K
m	value()	V
m	key()	K



## A.2 Nassi-Shneidermann Diagramme

Abbildung 16: Methode „read“ von ThreadA

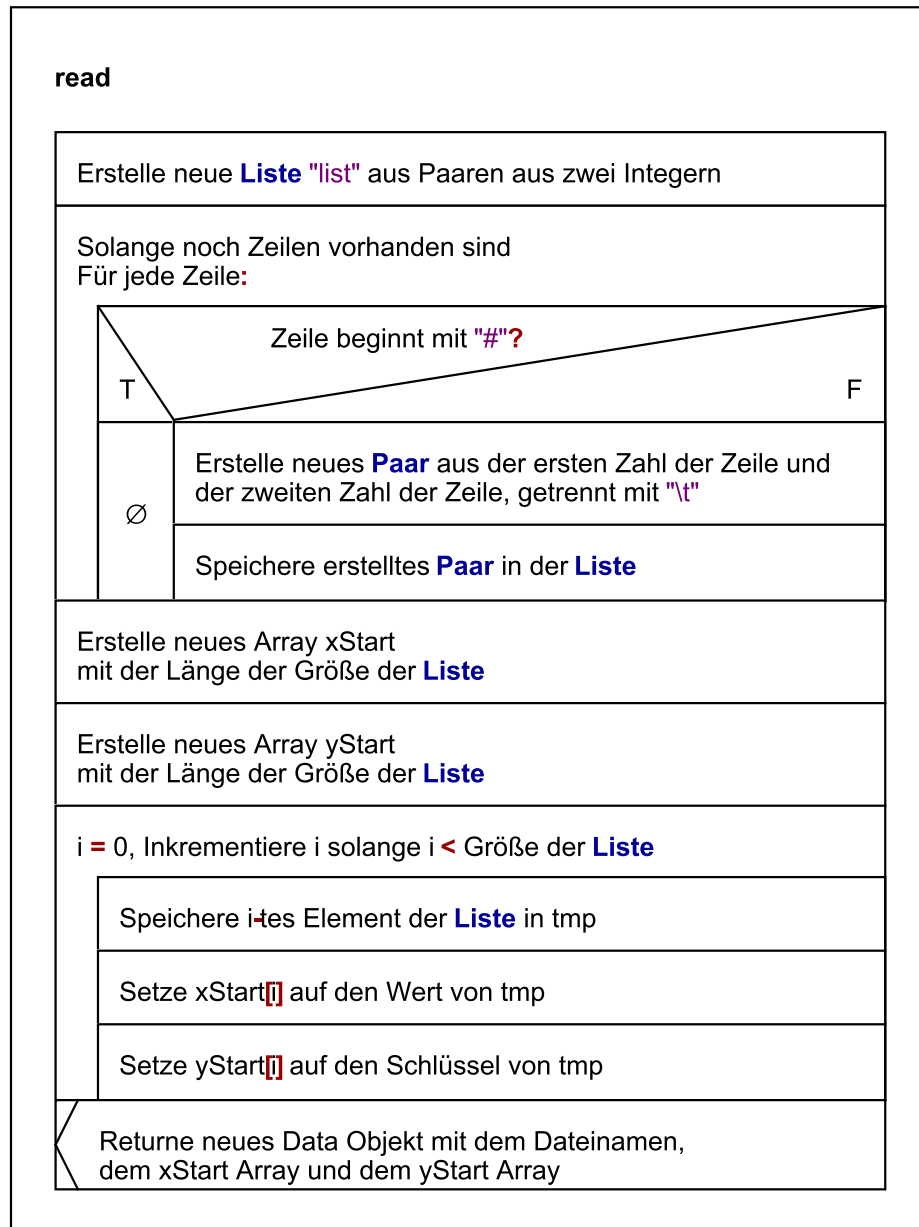


Abbildung 17: Algorithmus 1

### Algorithmus1

	yMax = max(yStart)	
	xDach = neues <b>double</b> Array mit der Länge von xStart	
	yNorm = neues <b>double</b> Array mit der Länge von yStart	
	alpha = $266.3 / (2^{18} - 1)$	
	i = 0, Inkrementiere i solange i < Länge von xStart	
	Setze xDach[i] auf (alpha * xStart[i] - 132,3	
	Setze yNorm[i] auf yStart[i]/yMax	
⬅	Returne ein neues Paar mit xDach und yNorm	

Abbildung 18: Algorithmus 2

### Algorithmus2

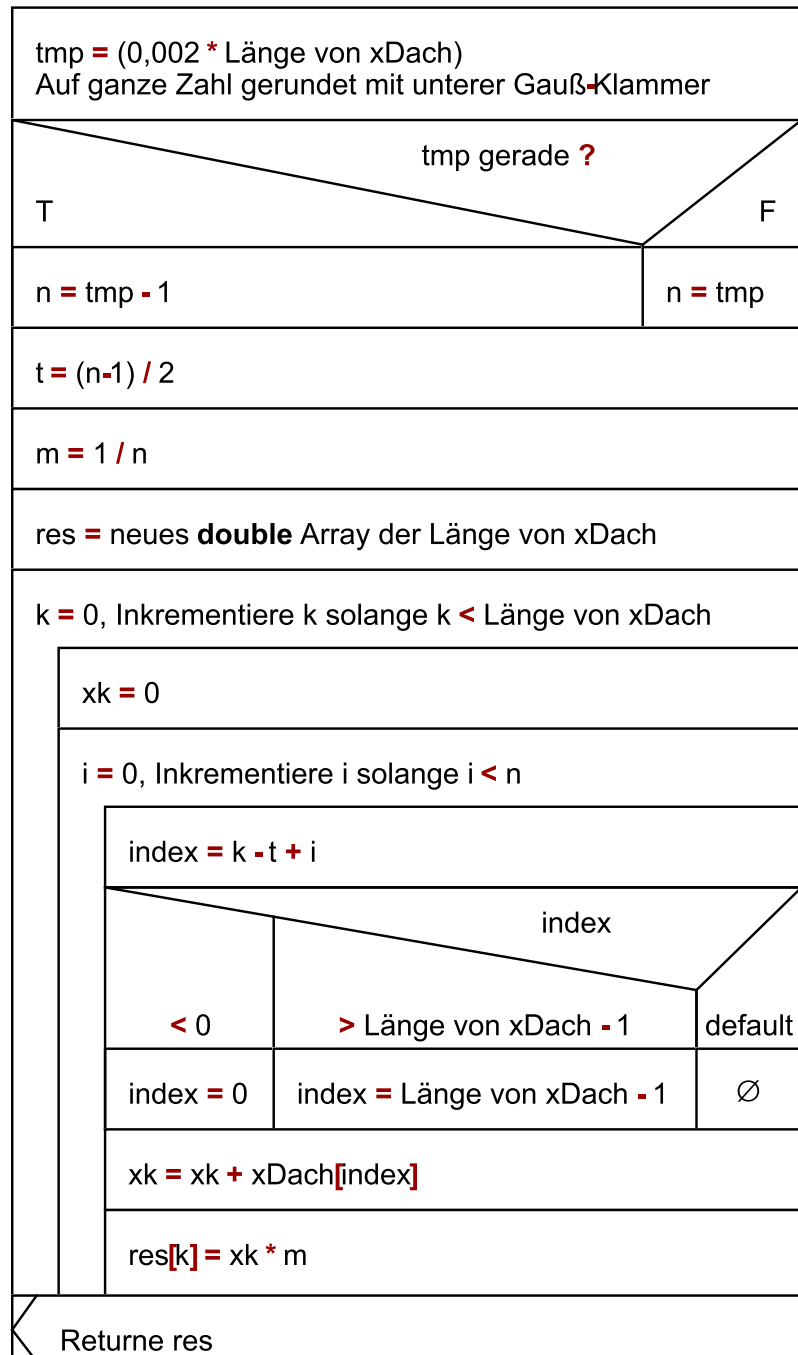


Abbildung 19: Algorithmus 3

### Algorithmus3

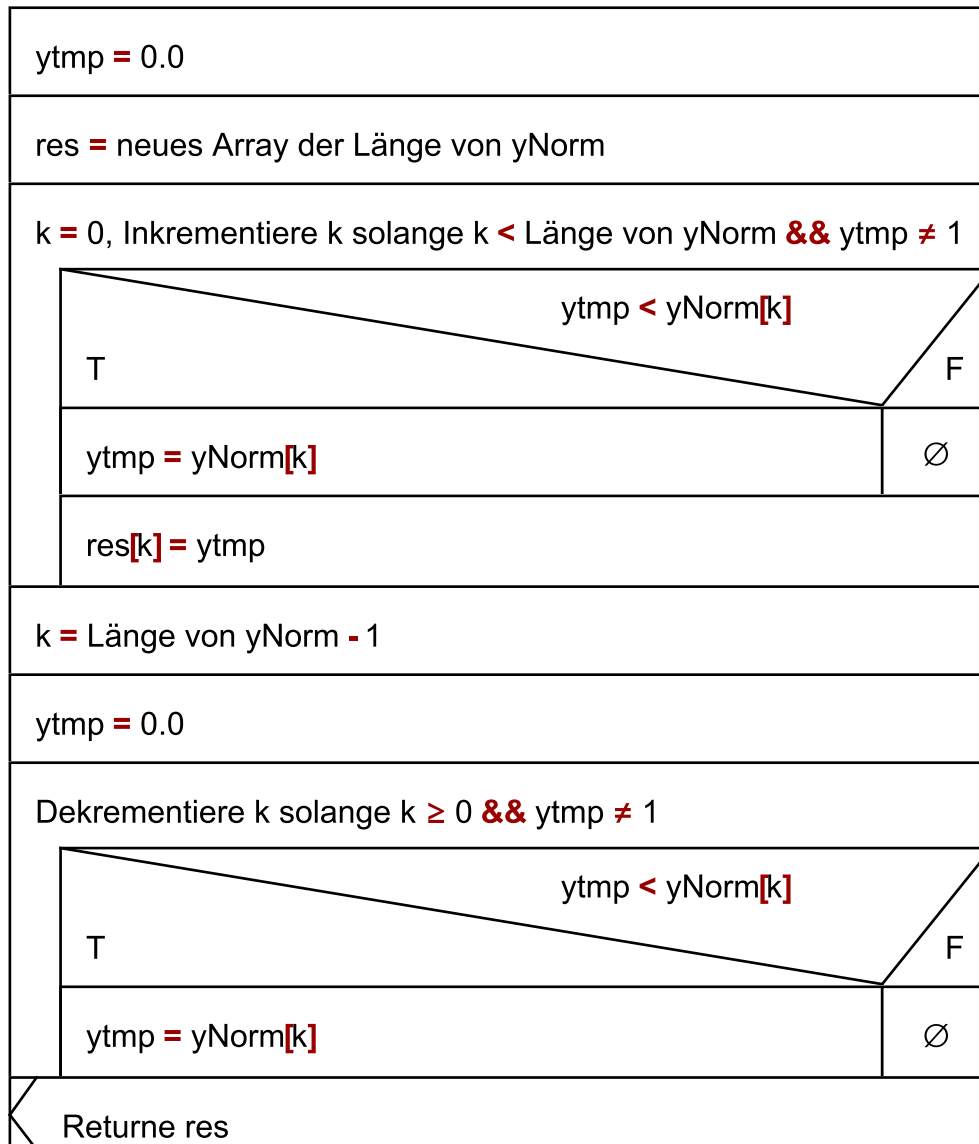


Abbildung 20: Algorithmus 4

#### Algorithmus4

anzahlPunkte = (Länge von yNorm \* 0.01)  
gerundet mit der oberen Gauß-Klammer

yGes = 0.0

i = 0, inkrementiere i solange i < anzahlPunkte

yGes = yGes + yNorm[i]

grundlinie = yGes / anzahlPunkte

a12 = ((1 - grundlinie) / 2) + grundlinie

indexL = 0

y = 0.0

i = 0, inkrementiere i solange i < Länge von yNorm && y < a12

indexL = i

y = obereEinhuellende[i]

indexR = 0

y = 0.0

i = Länge von yNorm - 1, dekrementiere i solange i ≥ 0 && y < a12

indexR = i

y = obereEinhuellende[i]

Returne neues Paar aus x[indexL] - x[indexR] und einem  
Paar aus indexL und indexR

Abbildung 21: Methode „getOutputString“ von AutoKorrelationsFunktion

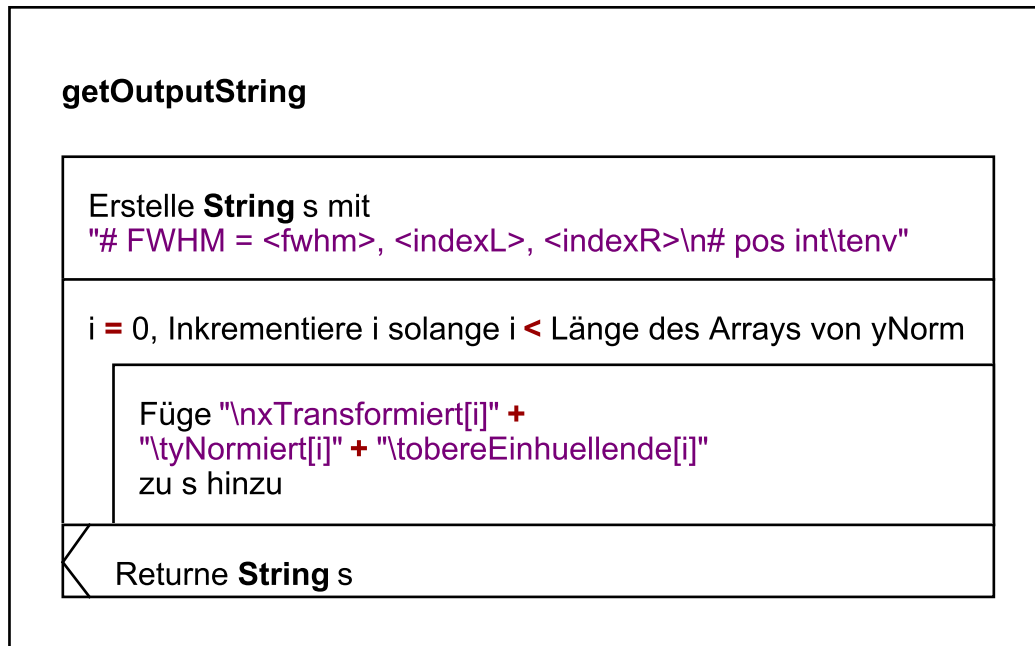
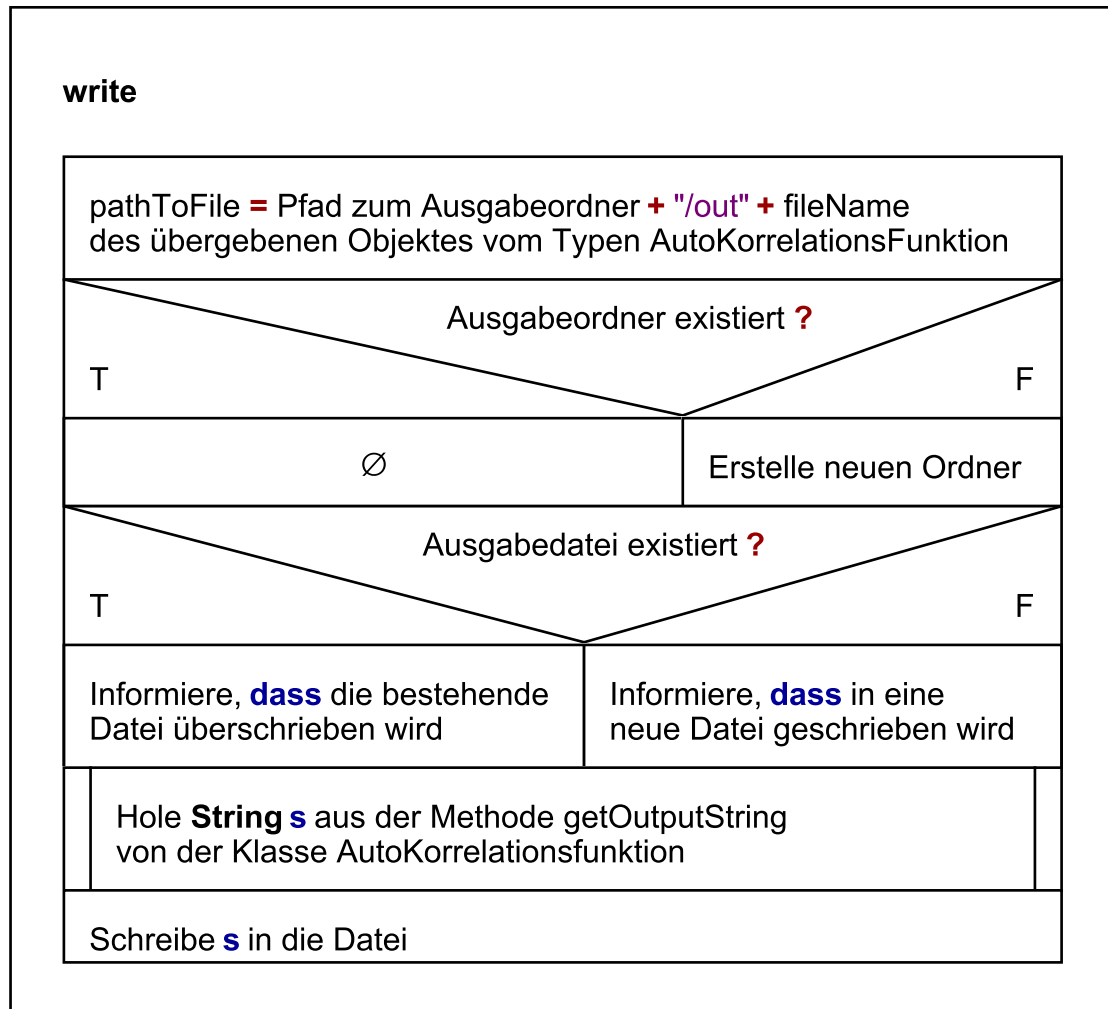


Abbildung 22: Methode „write“ von ThreadC



### A.3 Sequenzdiagramme

Abbildung 23: Darstellung der Zusammenarbeit der Threads in Problem Simple

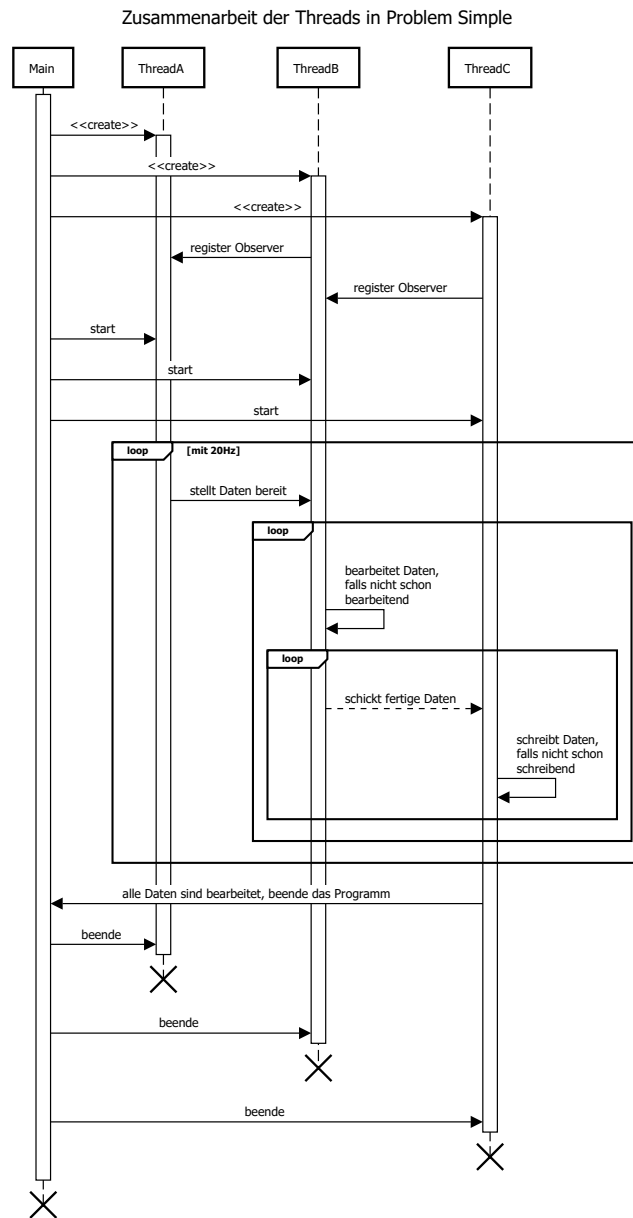
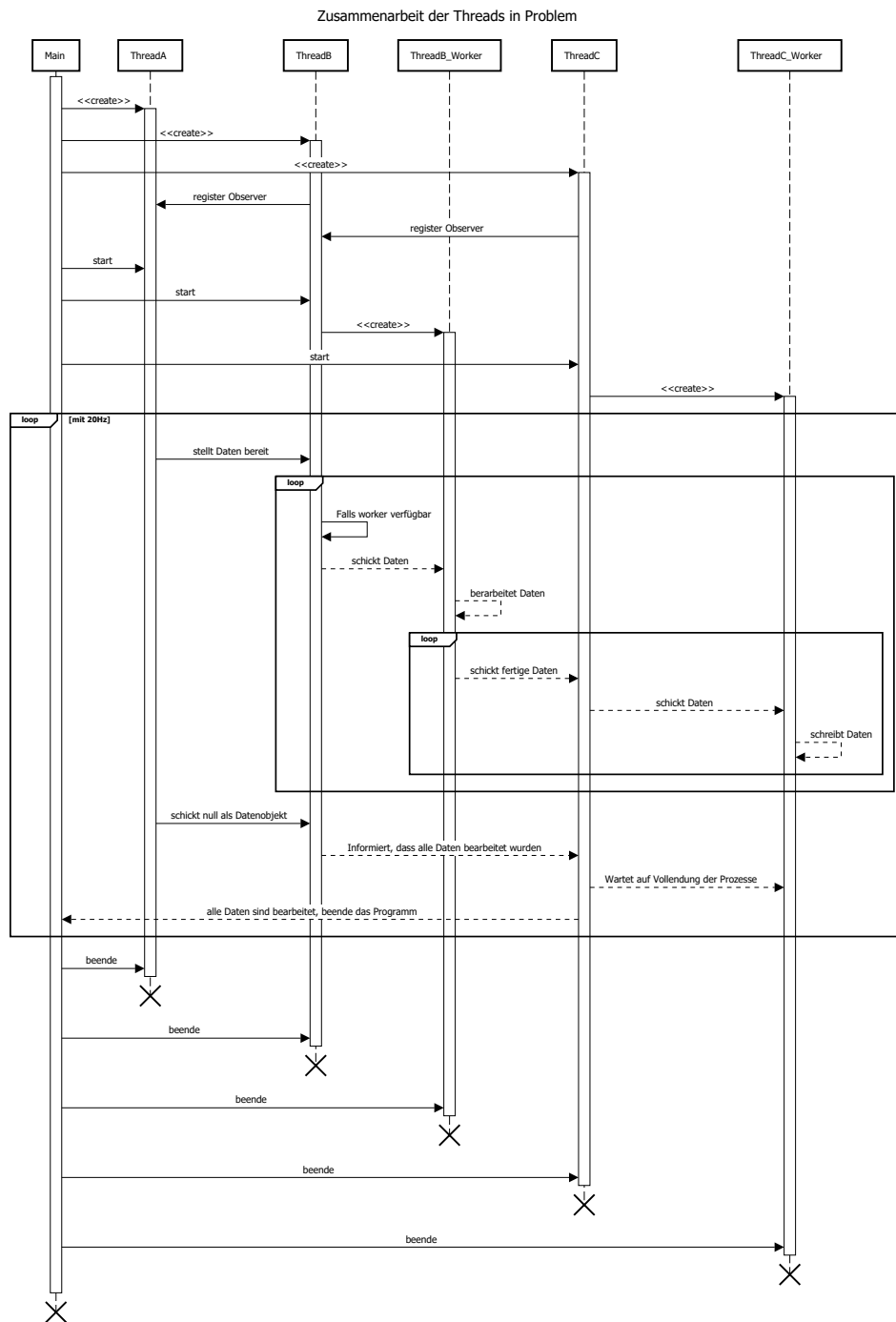




Abbildung 24: Darstellung der Zusammenarbeit der Threads in Problem



## B Bilder der resultierenden Funktionen

Abbildung 25: out0

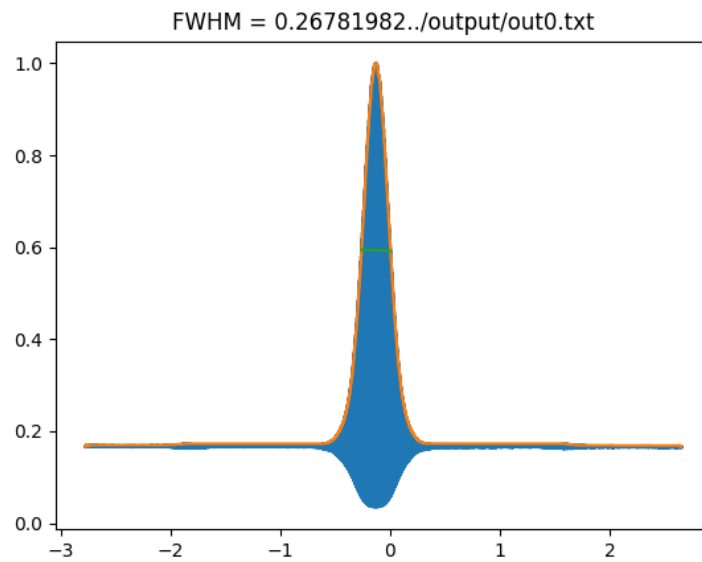


Abbildung 26: out1

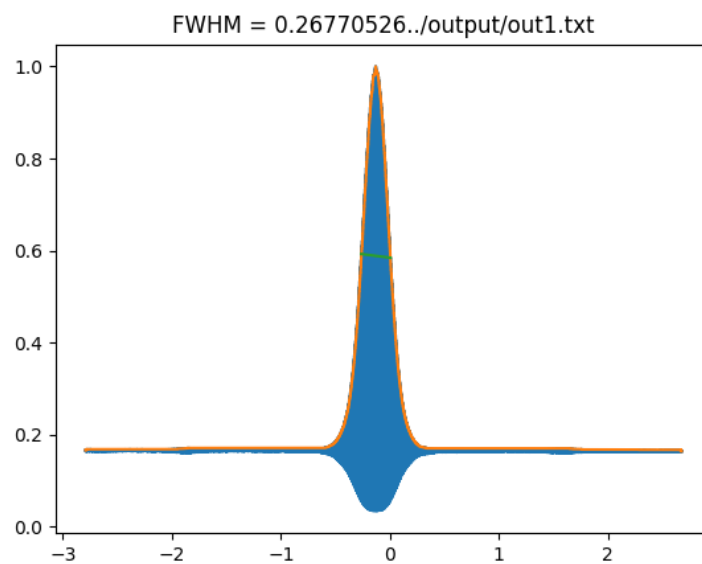


Abbildung 27: out2

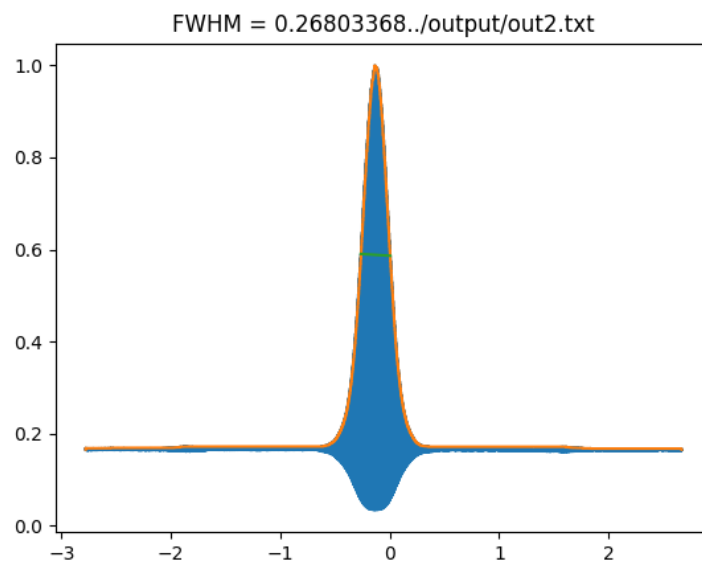


Abbildung 28: out3

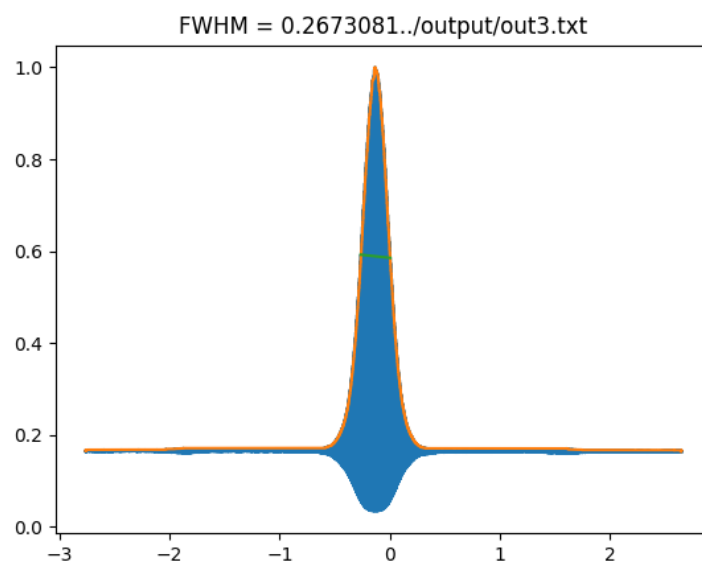


Abbildung 29: out4

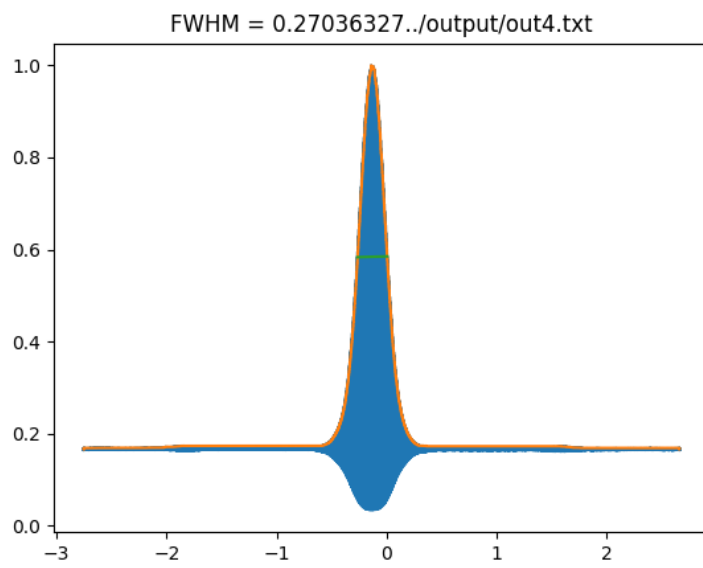


Abbildung 30: out5

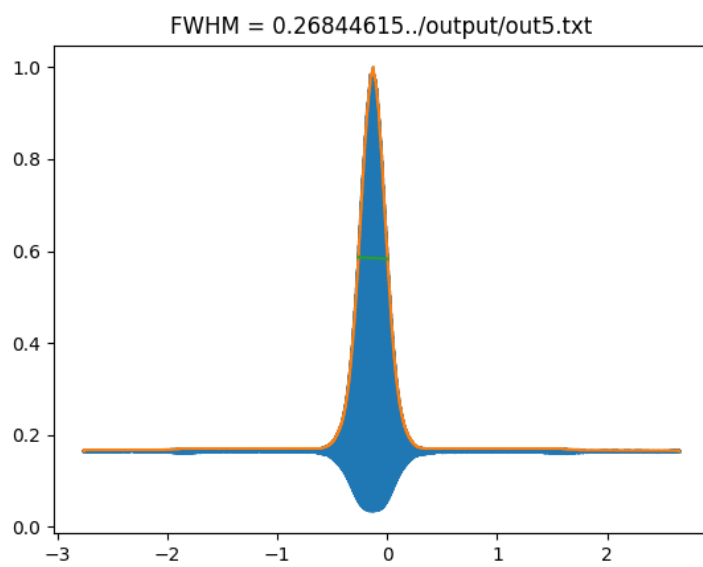


Abbildung 31: out6

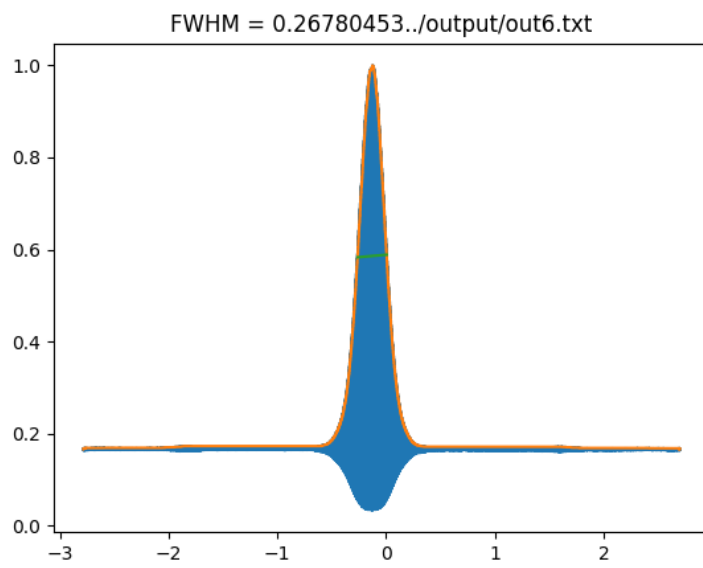


Abbildung 32: out7

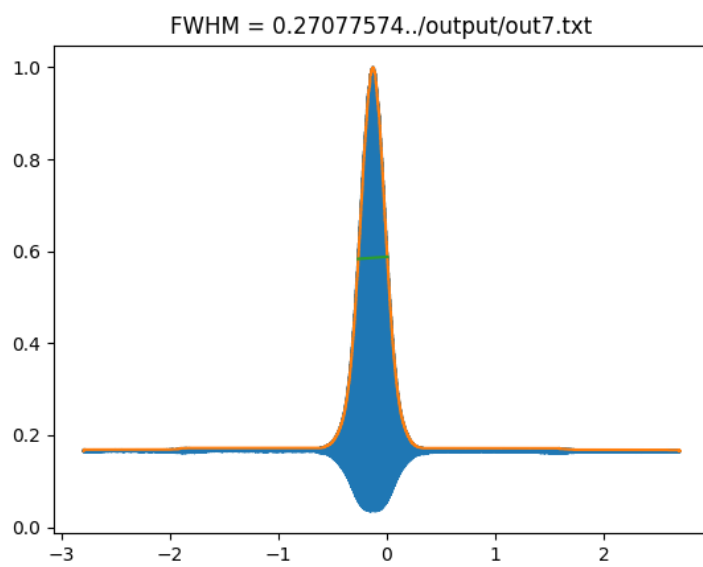


Abbildung 33: out8

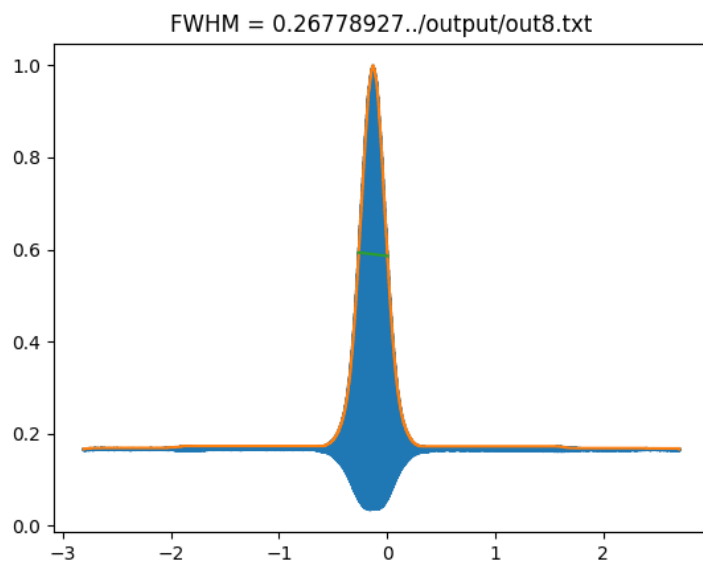
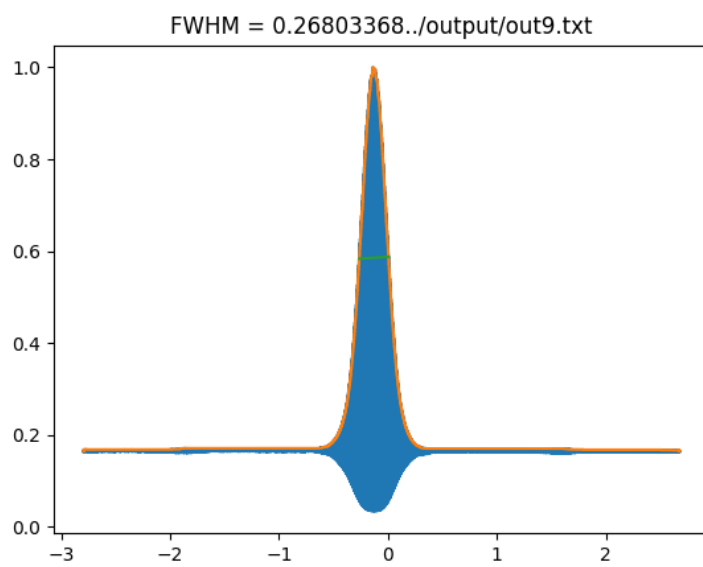


Abbildung 34: out9

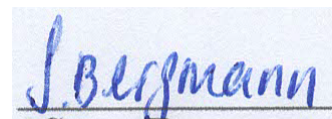


## C Eigenhändigkeitserklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Aachen, der 13. Mai 2022

Ort, Datum



Sven Bergmann