
UNIVERSITÄT DER BUNDESWEHR MÜNCHEN

Fakultät für Elektrotechnik und Technische Informatik

# WDF-Messgerät und Parametrisierung von Messgeräten für die labAlive Simulationsumgebung

Masterarbeit

| | |
|---|---|
| Bearbeiter: | Peter Tholl |
| Matrikelnummer: | 1141308 |
| Abgabetermin: | 31.08.2018 |
| Prüfer: | Prof. Dr. Erwin Riederer |
| Erstellt bei: | Universität der Bundeswehr München |

# Eidesstattliche Erklärung

Ich versichere, dass ich diese Masterarbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Weiterhin versichere ich, dass diese Arbeit weder bei einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht wurde.

_____ _____

Ort, Datum Unterschrift

# 1. Einleitung

## 1.1 Motivation

In meinem Bachelorstudium konnte gewählt werden, ob man seinen Schwerpunkt auf die Informatik oder die Kommunikationstechnik und Datenübertragung legen möchte. Ich habe mich damals dazu entschieden die Informatik weiter zu vertiefen, da es mich zu dem Zeitpunkt mehr angesprochen hat. Nach erfolgreichem Abschluss des Bachelorstudiums und der Abschlussarbeit konnte ich am Anfang des Masterstudiums zwischen vielen verschiedenen Bereichen wählen. Unter anderem standen wieder die Bereiche Informatik und Kommunikationstechnik zur Verfügung. Aufgrund meiner Schwerpunktwahl im Bachelorstudium bot sich eine Vertiefung dieser Fachbereiche im Masterstudiengang an.

Als geeigneten Abschluss des kompletten Studiums in der Masterarbeit sollte ein Projekt dienen, welches beide Teilaspekte aus vier Jahren Studium vereinigt. Ein Messgerät der Kommunikationstechnik für die labAlive Simulationsumgebung zu programmieren erschien mir als guter Abschluss des Masters. Die eigentliche Applikation ist dem Bereich der Informatik zugehörig, lässt sich allerdings nur mit zusätzlichem Verständnis aus dem Fachbereich der Kommunikationstechnik programmieren.

LabAlive ist ein Online Lern-Programm, welches hauptsächlich von Studenten zum Lernen in Praktika oder von zu Hause genutzt wird. Es veranschaulicht viele komplizierte Dinge der Kommunikationstechnik und lässt die Studenten selber daran arbeiten und lernen. Die Applikation wird in der Programmiersprache Java geschrieben. Schon meine Bachelorarbeit habe ich in der Programmiersprache Java geschrieben und dabei sehr viel Freude gehabt. Deshalb wollte ich auch nach Möglichkeit in meiner Masterarbeit dabeibleiben. Schließlich habe ich mich dazu entschlossen einen Beitrag zu dieser Applikation zu leisten, da sie viele interessante Aspekte vereint.

## 1.2 Aufgabenstellung

LabAlive ist eine Simulationsumgebung, die viele verschiedene Bereiche aus der Kommunikationstechnik abbildet und veranschaulicht. Es wird von Studenten im Praktikum genutzt oder von zu Hause als Lernunterstützung herangezogen. Diese Masterarbeit ist in zwei Teilaufgaben gegliedert, die nacheinander abgearbeitet werden. Die erste Aufgabe besteht darin, sich in dem Programmcode zurechtzufinden und ein neues Messgerät zu implementieren.

LabAlive stellt viele verschiedene Messgeräte zur Verfügung, mit denen die Signale und Schaltungen analysiert werden können. Das neue Messgerät misst die Wahrscheinlichkeitsverteilung des Signales, welches auf der ausgewählten Leitung anliegt. Das Ergebnis der Messung wird in Form eines Graphen in deinem XY-Diagramm dargestellt. Wie bei allen anderen Messgeräten kann die Verteilungsfunktion über verschiedene Parameter verändert werden, um ein möglichst breites Spektrum an Funktionalität abbilden zu können. Für die Programmierung stehen schon einige Basisklassen zur Verfügung, die übernommen werden. Das Gerüst des Messgerätes entspricht dem der anderen Messgeräte, um auch nachfolgend dieses mit weiterentwickeln zu können.

Im zweiten Teil der Masterarbeit geht es darum, die verschiedenen Parameter der Messgeräte zu serialisieren und wieder zu deserialisieren. Bei der Benutzung der Messgeräte werden die Parameter so lange verstellt, bis sie das Ergebnis optimal darstellen. Hierbei ist es hilfreich, dies nicht jedes Mal von Neuem vornehmen zu müssen, sondern ein schon abgespeichertes eingestelltes Parameterprotokoll laden zu können, um sich nicht länger mit der Parametereinstellung befassen zu müssen. Es gibt mehrere Möglichkeiten die Parameter abzuspeichern. Als Dateiformate stellt Java beispielweise *.ser*, *.json* oder *.bson* zur Verfügung, die für die Funktionalität der Serialisierung herangezogen werden. Json bietet die Möglichkeit zur Implementierung einer manuellen Serialisierung und ist für labAlive am Geeignetsten.

## 1.3 Gliederung der Arbeit

Am Anfang der Masterarbeit soll in das Projekt labAlive eingeführt werden. Dazu stehen mehrere Dokumente aus Wahlpflichtmodulen zur Verfügung, um sich in der umfangreichen Thematik zurechtzufinden. Zum einen ein Skript, in dem alle Funktionen von labAlive erklärt sind und zum anderen eine Anleitung, wie man ein neues Messgerät entwickeln kann. Darin stehen die Klassen, die benötigt werden und wie sie miteinander verlinkt werden müssen, damit sie im Gesamtsystem erkannt werden und benutzt werden können.

Nach der Implementierung des Gerüstes muss die Logik für das eigentliche Messinstrument programmiert werden. Hierbei handelt es sich um eine statistische Auswertung von gesammelten Messpunkten. Diese werden auf ihre Auftrittswahrscheinlichkeit hin analysiert und dann als Graph in einem Diagramm angezeigt. Hierbei können verschiedenen Parameter eingestellt werden. Die X- und Y-Achse können entsprechend dem Graphen fein oder gröber ausgewählt werden. Des Weiteren kann die Anzahl der zu analysierenden Signale einstellen, die Auflösung der Wahrscheinlichkeitsdichte und den Nullpunkt verschieben.

Im zweiten Schritt der Masterarbeit geht es darum, diese einstellbaren Parameter zu speichern. Für den Benutzer ist es handlicher, wenn er für bereits bekannte Signale das Messgerät nicht wieder neu einstellen muss, sondern ein bereits abgespeichertes Parameterprotokoll wieder laden kann. Dazu müssen die einzelnen Parameter zunächst serialisiert und über einen Stream in einer Datei abgespeichert werden. Später sollen sie auf einem Server abgelegt werden, um sie wieder aufrufen zu können. Bei diesem Vorgang wird die Datei serialisiert, im Programm ausgelesen und in die laufende Applikation übernommen. Für diese Serialisierung und Deserialisierung soll das Format *.json* verwendet werden. Der Mechanismus zur Serialisierung und Deserialisierung soll dann auf andere Messgeräte angewandt werden können.

## 1.4 Software

Die labAlive Simulationsumgebung mit allen nötigen Klassen und Packages wird von Prof. Dr. Erwin Riederer zur Verfügung gestellt. Da die Applikation bisher in der Entwicklungsumgebung Eclipse entwickelt wurde, wird diese auch bei dieser Masterarbeit verwendet. Die verwendete Eclipse Version ist die Oxygen.2 Release (4.7.2). Für die Serialisierung wird die „javax.json“-Bibliothek verwendet, die online zur Verfügung steht und heruntergeladen werden kann.

Als Java Compiler wird die Version JavaSE-1.8 verwendet.

# 2. Grundlagen

## 2.1 Simulationsumgebung labAlive

Die Simulationsumgebung labAlive ist ein Online Lernprogramm für Studenten und Interessierte von außerhalb. Es wurde von Prof. Dr. Erwin Riederer entwickelt und immer wieder um neue Projekte erweitert und veranschaulicht nahezu jede Möglichkeit einer Messstrecke. Von Schaltungen mit einfachen Pulsformern und Basisfiltern bis hin zu komplexen Signalen in umfangreichen Verschaltungen, um auch komplizierte Situationen aus der Praxis abbilden zu können. Beinahe jedes Konstrukt welches in der Kommunikationstechnik gibt, kann man so nachbilden und eigenen Simulationen unterziehen. Aber auch für Studenten zum Lernen ist es ein interessanter Beitrag, da sie gelerntes Wissen an praktischen Beispielen in Echtzeit nachvollziehen und verstehen können.

Es sind nahezu alle Parameter, die in solchen Schaltungen relevant sind, einstellbar, was eine sehr flexible Simulation schafft. So können während der Laufzeit verschiedene Signale angeschlossen werden und auf die Leitungen gelegt werden.

In labAlive gibt es zwei wichtige Fenster, die dem Benutzer zur Verfügung stehen. Zum einen das Blockdiagramm, dass dem Nutzer hauptsächlich zur Verfügung steht. Hier kann auf Anhieb die komplette Verschaltung sehen und erkennen, um was für einen Typ Schaltung es sich handelt.

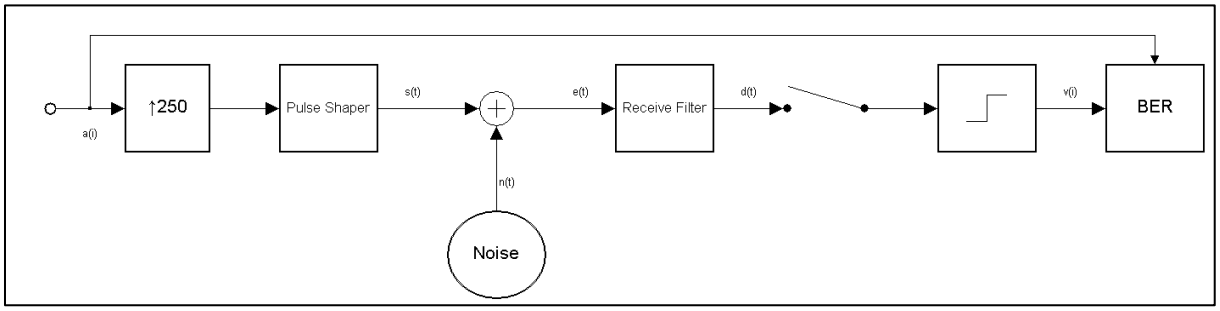


*Abb. 1: Blockdiagramm für eine Digitalbasisband Übertragung*

Im Blockdiagramm kann die einzelnen Komponenten und Leitungen der Schaltung sehen. Mit einem Klick auf die Komponente können die Einstellungen verändert werden.

Mit einem Rechtsklick auf eine Leitung können sich verschiedene Messgeräte angezeigt werden. Dies können zum Beispiel ein Spektrumanalysator oder ein Oszilloskop sein. So können direkte Zusammenhänge zwischen Komponentenänderungen und Auswirkungen auf das Signal betrachtet werden.

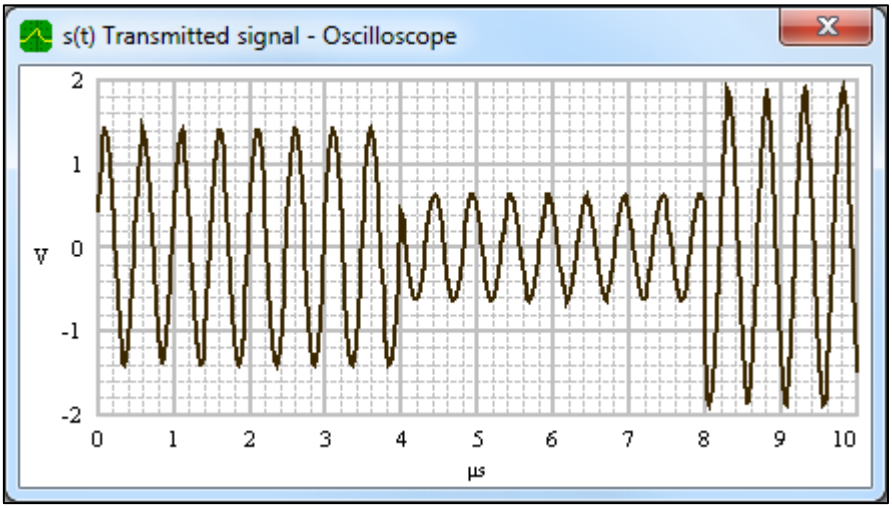


*Abb. 2: Aufgenommenes Signal im Oszilloskop (Quelle: http://www.etti.unibw.de/labalive/about)*

LabAlive ist für jeden frei zugänglich über die Website www.etti.unibw.de/labalive/. Dort werden viele kommunikationstechnische Aspekte von labAlive mit anschaulichen Videos erklärt und vertieft.

### 2.1.1 Java Applikation

Java Applets sind kleine Applikationen, die in einem Webbrowser lauffähig sind. Sie sind eines der ersten Java-Programme, mit denen gearbeitet wurde. Zum Starten eines Applets startet ein Browser eine virtuelle Maschine und führt das Applet aus.

In der anfänglichen Zeit der Applets waren die Browser-Hersteller für die Funktionsfähigkeit verantwortlich. Mit der Plug-In Technologie bietet Oracle alles, was der Browser benötigt, um den aktuellen Stand aufrufen zu können. Mittlerweile läuft die virtuelle Maschine im Browser selbst, aber mit dem Aufkommen neuerer Java-Versionen läuft die Anwendung in einem eigenen Prozess.

Beim Aufrufen eines Java Applets ruft der Browser automatisch verschiedene Methoden auf. Zunächst wird zur Initialisierung die Methode *init()* aufgerufen, während die Seite geladen wird. Ist die Initialisierung abgeschlossen wechselt die Methode zu *start()* und *stop()*. Diese Methoden werden entsprechend aufgerufen, ob das Applet im Browser sichtbar ist oder ein anderer Bereich im Vordergrund abgebildet wird. [1]

### 2.1.2 Funktionsweise von labAlive

Das gesamte Projekt für die Simulationsumgebung labAlive besteht aus vier Packages[1], in denen die einzelnen Bereiche aufgeteilt sind.

Das Package *Core* ist das wohl Wichtigste. Wie die Übersetzung es beschreibt, befindet sich darin der Kern-Code der Simulationsumgebung. Diese Klassen stellen die Basis für labAlive dar, auf denen die anderen Packages und die sich darin befindenen Klassen aufbauen. Da diese Masterarbeit lediglich ein neues Modul zu der Simulationsumgebung hinzufügt, aber nicht an grundlegenden Funktionen des Programms Veränderungen vornehmen möchte, wird auf dieses Package nicht weiter eingegangen.

Im Package *Wiring* befinden sich die erstellten Schaltungen, die direkt als Java Applet gestartet werden können. Sie sind nochmal unterteilt, je nach Thema in der Kommunikationstechnik. Die einzelnen Schaltungen variieren zwischen einfachen Filteranwendungen oder Pulsformern bis hin zu Schaltungen mit komplexen Signalen und komplizierten Bausteinen. Es gibt beispielsweise Schaltungen, die eine Audio-Datei analysieren, eine digitale Modulation simulieren oder die Fourier-Transformation veranschaulichen. Für die ersten Versuche des Messgerätes, welches unter anderem im Rahmen dieser Masterarbeit entwickelt wird, wurde eine bestimmte Schaltung herangezogen. Die Schaltung *Probeklausur* trennt aus einem Eingangssignal die tiefen und hohen Frequenzen mittels Filter heraus und gibt diesen dann in zwei Signalleitungen aus. Da dies ein recht einfaches Beispiel der Simulationsumgebung ist, war es für die ersten Versuche sowohl beim Entwickeln des Messgerätes als auch für die Serialisierung geeignet.

[1] Java Pakete sind vergleichbar mit Verzeichnissen in einem Dateisystem und organisieren so untergeordnete Klassen.

### 2.1.3 Anwendung von labAlive

LabAlive bietet ein breites Spektrum an Bereichen, für die es herangezogen werden kann. Da das Hauptaugenmerk auf universitärer Unterstützung für Studenten liegt, sind die meisten Themen auch dort zu finden. Alle Applikationen werden auf der Website des Instituts („http://www.etti.unibw.de/labalive/“) zur Verfügung gestellt und stehen jedem Interessenten zur Benutzung frei. Es können also auch Personen von Extern sich über die Simulationsumgebung informieren und weiterbilden. In der Simulationsumgebung werden fertige Schaltungen zur Verfügung gestellt.

Beim Start einer Applikation von labAlive öffnet sich zunächst ein Fenster, in dem der Status des zu ladenden Applets steht. Sind alle Klassen und dazugehörigen Bibliotheken korrekt geladen, öffnet sich automatisch ein weiteres Fenster, in dem die Messstrecke angezeigt wird.

Diese Schaltungen simulieren Messstrecken, wie sie in der Praxis vorkommen. Solche Messstrecken können einfache Hoch- und Tiefpass Filterungen sein oder kompliziertere Schaltungen darstellen. In der ersten unten liegenden Schaltung ist ein Eingangssignal, dass in seine Ober- und Unterwellen gefiltert wird.

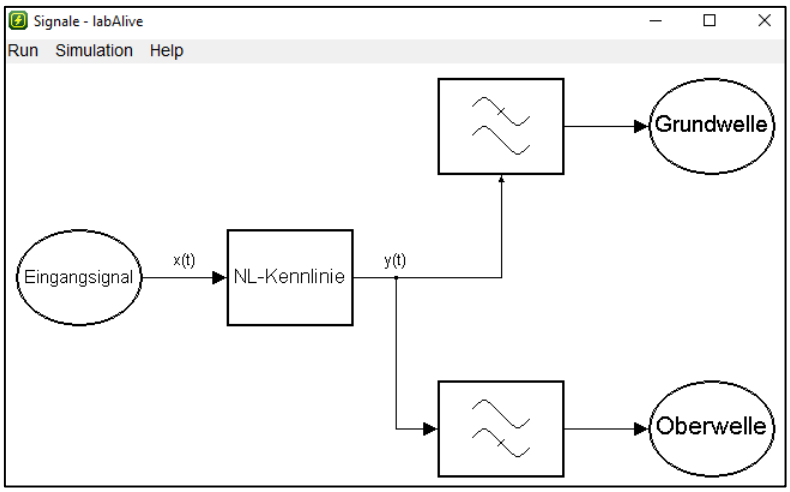


*Abb. 3: Einfache Schaltung in labAlive*

Es werden aber nicht nur einfache Filterungen dargestellt, sondern die Palette von Möglichkeiten reicht bis zu komplizierten Verfahren aus der Kommunikationstechnik. Beispielsweise sieht die Strecke eines Matched Filters in labAlive wie folgt aus.

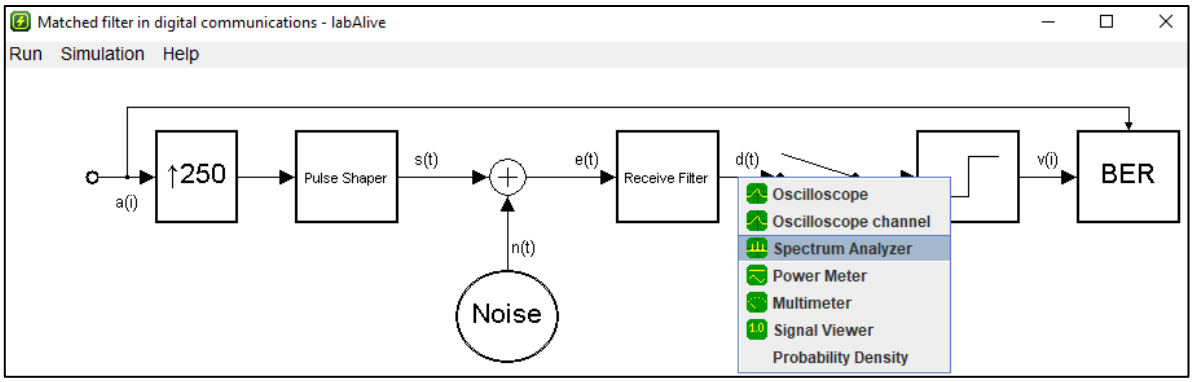


*Abb. 4: Schaltung in labAlive für ein Matched Filter mit geöffneter Messpalette*

Auf der linken Seite kommt das Eingangssignal in die Simulation und wird mittels Verbindungen von Modul zu Modul weitergeleitet. Mit der Messgerätepalette, wie man sie in der Abbildung (4) sieht, kann man an jede Stelle das Signal nach belieben analysieren und sich anzeigen lassen. Über sogenannte Kippschalter, kann man das Signal an bestimmten Stellen unterbrechen. Jedes Modul für sich stellt entsprechend Parameter zur Verfügung die vorgegeben werden können.

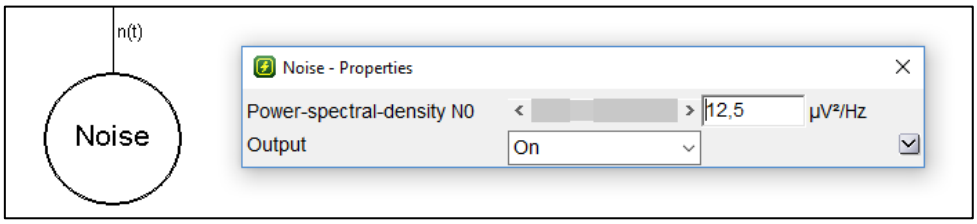


*Abb. 5: Einstellungsfenster für die zugefügte Rauschleistung*

In der obigen Abbildung sind die Einstellungen für das Rauschen aufgelistet. Das Rauschen kann zu oder -weggeschaltet werden und die Leistung pro Spektralbereich kann angepasst werden.

In der obigen Menüleiste können Einstellungen für die ganze Simulation verändert werden.

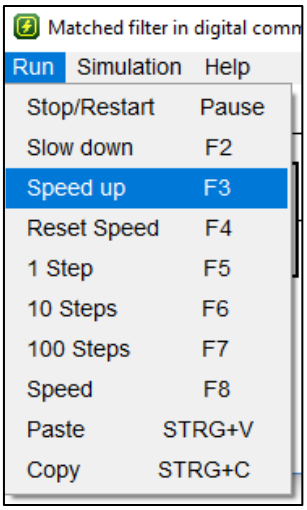


*Abb. 6: Einstellungen für die Simulation*

Die Veränderung dieser Parameter kann entweder über den Menüreiter vorgenommen werden, oder wie dort angezeigt wird, mit den entsprechenden Tasten. Im Reiter „Simulation“ können weitere Einstellungen für Messgerätefenster oder die Simulation an sich vorgenommen werden. Bei „Help“ kann die verwendete labAlive Version angeschaut werden.

## 2.2 Beschreibung von stochastischen Signalen

Bei deterministischen Signalen ist deren zeitlicher Verlauf vollständig mathematisch beschreibbar. Nicht-deterministische Signale sind sogenannte Zufallssignale, deren exakter Verlauf nicht bekannt ist. Solche Signale sind durch statistische Mittelwerte und Wahrscheinlichkeitsverteilungen beschreibbar. Neben dem klassischen Rauschsignal, welches komplett zufällig ist, gibt es auch Nutzsignale, die zufälliger Natur sind. Bei der Übermittlung von einem Sender zum Empfänger, ist dem Empfänger nicht bekannt, wie das Signal aussieht, dass er empfangen wird, da ihm die Nachricht nicht bekannt ist. Führt man nun an einem solchen stochastischen Prozess Messungen durch, so erhält man die Musterfunktionen. Werden mehrere Musterfunktionen aufgenommen, so weisen diese alle einen unterschiedlichen Verlauf auf, wobei aber jeweils die gleiche Statistik zugrunde liegt. [2]

Aus jeder Musterfunktion kann ein Mittelwert gebildet werden und aus einer Reihe von Mittelwerten kann ein Zufallsprozess beschrieben werden. [3]

Diese Mittelwerte hängen alle vom Beobachtungszeitpunkt ab. Geht man von einem stationären Zufallsprozess aus, als wesentliche Vereinfachung, so sind die statistischen Eigenschaften eines stationären Prozesses unabhängig vom Beobachtungszeitpunkt. In den meisten Fällen ist nur eine Musterfunktion bekannt. Es wird die Hypothese aufgestellt, dass eine Musterfunktion für den ganzen Prozess steht und diesen repräsentiert. Trifft dies für einen Prozess zu, so nennt man diesen per Definition ergodisch. Ist ein Prozess ergodisch, so genügt es zur Bestimmung der Erwartungswerte eine einzelne Musterfunktion $x(t)$ nacheinander an verschiedenen Zeitpunkten zu beobachten.

Wird ein Prozess zu einem gegebenen Zeitpunkt $t_1$ betrachtet, führt dies zu der Zufallsvariable $x(t_1)$. Um diese Zufallsvariable zu beschreiben, kann eine Amplitudenverteilung angegeben werden. Diese ist als Wahrscheinlichkeitsdichtefunktion definiert. [2]

## 2.3 Wahrscheinlichkeitsdichte und Häufigkeit

Bei einer relativen Häufigkeit verändert sich die Höhe des Graphen, je nach dem wie grob oder fein das Delta X angegeben wird. Normiert man die Häufigkeit auf das Delta X, bleibt die Höhe immer gleich. Delta X ist die Größe einer Stufe auf der X-Achse, die den Abstand zwischen den einzelnen Punkten eines Graphen widergibt. Dadurch bekommt man aus der Häufigkeit eine Dichte, welche in der Darstellung wesentlich angenehmer ist. Im realen Programm ist beim Delta X die Auflösung, also Resolution gemeint. In den unteren zwei Abbildungen kann man erkennen welchen Einfluss das Delta X auf das Ergebnis einer Wahrscheinlichkeitsdichte bei einem Gauß-verteilten Signal hat. In der linken Abbildung ist ein grobes Delta X ausgewählt und in der rechten Abbildung ein feines Delta X.

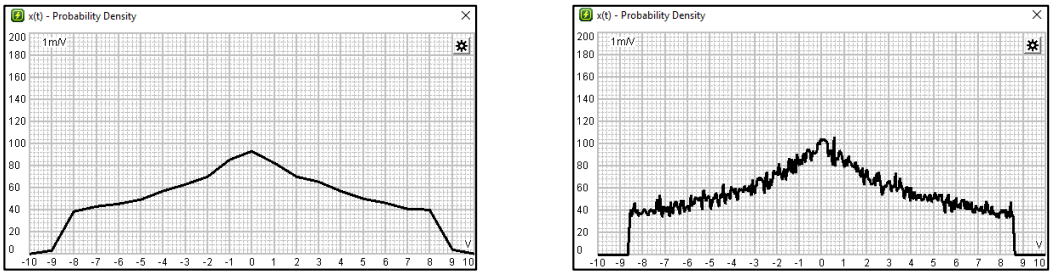


*Abb. 7: Gauß-verteiltes Signal mit niedriger(1V) und hoher(50mV) Auflösung*

In der rechten Abbildung ist die Gaußglocke etwas deutlicher und detaillierter zu erkennen als in der linken. Des Weiteren fällt auf, dass die Abstände auf der Y-Achse bei veränderter Resolution gleichbleiben.

# 3. Anforderungen

## 3.1 WDF-Messgerät

Das Messgerät zur Bestimmung der Wahrscheinlichkeitsdichtefunktion stellt den ersten Schritt in der Masterarbeit dar. Das Projekt labAlive umfasst mittlerweile viele einzelne Klassen und Teilprojekte, die miteinander verknüpft sind. Bei der Entwicklung eines neuen Messgerätes können Zusammenhänge am Sichtbarsten und Leichtesten erklärt werden. Die Wahrscheinlichkeitsverteilung der Signale soll, wie beim Oszilloskop oder dem Spektrum Analysator, auf einem XY-Diagramm dargestellt werden. Wobei beim WDF-Messgerät der Nullpunkt in der Mitte sein soll und nicht links beginnen. Hierfür gibt es schon vorbereitete Klassen, die lediglich angepasst und verwendet werden können.

Auf der X-Achse wird die Amplitude in Volt angezeigt, auf der Y-Achse die Dichte in 1/Volt. Die beiden Achsen sollen einstellbar sein, um flexibel verschiedene Signalgrößen analysieren zu können. In diesem Zusammenhang muss auch die Auflösung anpassbar sein.

Für die Aussage über die Wahrscheinlichkeit müssen mehrere Signale gesammelt und dann ausgewertet werden. Hierbei soll als Ergebnis keine Häufigkeit angezeigt werden, sondern die Wahrscheinlichkeitsdichte der gesammelten Signalwerte. Die Anzahl der zu analysierenden Signalwerte soll ebenfalls einstellbar sein.

## 3.2 Parametrisierung von Messgeräten

Jedes einzelne Messgerät hat mehrere Parameter, die eingestellt werden können, um möglichst genau das betrachtete Signal analysieren zu können. Die Werte, die den Parametern hinterlegt sind, sollen abgespeichert und wieder aufgerufen werden können. Dazu wird das Json-Format verwendet.

Bevor der String abgespeichert wird, wird dieser manuell erstellt, sodass das Format auf jedes Messgerät angewendet werden kann. Der String soll nach der Erstellung im Json-Format vorliegen. Im String soll der Schlüssel eines jeden Parameters mit dem dazugehörigen Wert enthalten sein. Idealerweise werden der Schlüssel, sein Wert und das nächste Parameterpaar durch Zeichen gemäß Json getrennt. Danach wird dieser String über ein standardisiertes Verfahren serialisiert und bei Bedarf wieder deserialisiert.

Nach der Deserialisierung wird die Information aus dem zusammenhängenden String extrahiert und wieder in das System eingegeben, um die gewünschte Anzeige wiederherzustellen.

Der Mechanismus zur Serialisierung der Parameter soll nicht nur für das Messgerät der Wahrscheinlichkeitsdichte funktionieren, sondern universal auf andere Messgeräte anwendbar sein. Das heißt sowohl der Spektrumanalysator und das Oszilloskop, welche ähnlich aufgebaut sind, sollen die gleichen Methoden und das gleiche Format verwenden, um ihre Parameter serialisieren zu können. Um keine Verwechslungen zu verursachen, dass das falsche Messgerät die verkehrten Parameter lädt, muss überprüft werden, ob diese die geladenen Parameter akzeptiert. In dem Json-String muss also ein Identifikationsmerkmal eingefügt werden, um dies zu überprüfen.

Im einem weiteren Schritt, sollen die Parameter auch flexibel zur Laufzeit kopiert und eingefügt werden. Mit den Tasten „C“ und „V“ sollen die Parameter eines geöffneten Messgerätefensters kopiert und dann in ein weiteres eingefügt werden. Dieses Copy und Paste Prinzip soll ebenfalls auf alle zur Verfügung stehenden Messgeräte mit einem Anzeigefenster angewendet werden. Hierbei sollen die gleichen Methoden verwendet werden, die vorher zur Serialisierung und Deserialisierung implementiert wurden.

# 4. Entwicklung des WDF-Messgerätes

## 4.1 Basismethoden eines Messgerätes

Das neue Messgerät hat den Namen *ProbabilityDensity* und wird auch in den folgenden Ausführungen als solches geführt. In Zusammenhang mit dem Messgerät *ProbabilityDensity* gibt es vier Basisklassen, die das Gerüst darstellen. Im untenstehenden Klassendiagramm sind alle wichtigen zugehörigen Klassen dargestellt. Pfeile mit einem Dreieck-förmigen Kopf bedeuten Vererbung von einer Super-Klasse[2]. Die anderen Pfeile zeigen an welche Klassen, Objekte oder Verlinkungen in welche Richtung besitzen.

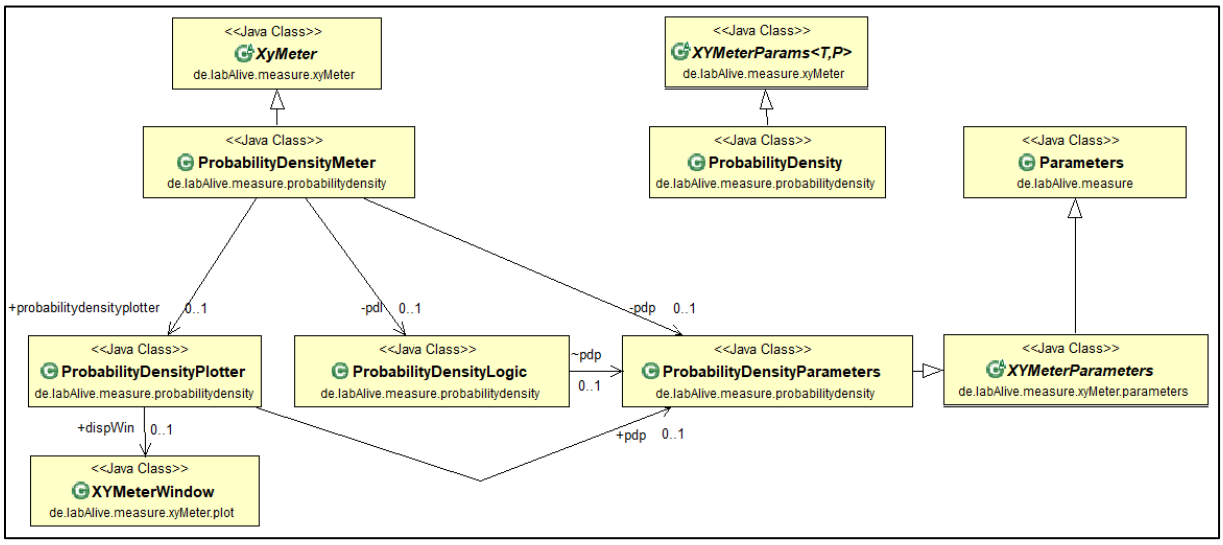


*Abb. 8: Klassendiagramm der wichtigsten Klassen für ProbabilityDensity*

Die Klasse *ProbabilityDensity* erbt von der Klasse *XYMeterParams*, die abstrakt ist, das heißt nicht instanziiert werden kann. In ihr kommen alle wichtigen Klassen zusammen. Über eine HashMap[3] ist jedem Messgerät *ProbabilityDensity* ein Parameter-Set *XYMeterParameters* zugeordnet. Über diese Parameterklasse hängen alle weiteren Klassen zusammen. Bei der Implementierung von Anwendungen werden in der Regel zur Vereinfachung nur eine Fassade genutzt, was bei Entwicklungsschemas von komplexeren Messgeräteeinstellungen üblich ist.

[2] Es wird bei der Vererbung zwischen Super-Klassen und Sub-Klassen unterschieden. Die Sub-Klasse erbt hierbei die Eigenschaften und Methoden der Super-Klasse.
[3] Um ein Messgerät mehrmals neu instanziieren, werden alle erstellten in einer Map abgelegt.

Die Fassade stellt dann nur die Setter-Methoden für die einzelnen Einstellungen bereit und verbirgt so die weiteren Bestandteile der Einstellmöglichkeiten vor dem Anwender. So ist *XYMeterParams* die Fassade für *XYMeterParameters,* hinter der sich der Rest des Messgerätes verbirgt.

Die Klasse *ProbabilityDensityParameters* erbt viele Funktionen ihrer beiden Superklassen *XYMeterParameters* und *Parameters*. Jedes Messgerät verfügt über individuelle Einstellmöglichkeiten, die in dieser Klasse deklariert werden. Hauptsächlich enthält die Klasse verschiedene Getter-Methoden und eine Methode zur Erstellung des eigentlichen Messgerätes.

Die eigentliche Klasse, in der die Logik des Messgerätes implementiert ist, heißt in diesem Fall *ProbabilityDensityMeter*. Hier ist die Messfunktion programmiert und sie enthält die Auswertung der erhaltenen Signale. Die Methode *meter()* wird bei jedem neuen Signalwert aufgerufen. Mit diesen Signalwerten kann später die Wahrscheinlichkeitsdichte berechnet werden. Diese Werte werden an *ProbabilityDensityLogic* gegeben, die die Auswertung vornimmt.

Die letzte wichtige Klasse ist *ProbabilityDensityPlotter*, die für die Erstellung und Darstellung der Grafik zuständig ist. Java Applet stellt keine Methoden zur Verfügung, um ein flexibles Koordinatensystem zu erstellen, in dem ein Graph eingezeichnet werden kann. Für genau diese Umsetzung zwischen analysierten Werten und Zeichnen der Ergebnisse ist *ProbabilityDensityPlotter* zuständig. *ProbabilityDensityMeter* übergibt seine berechneten Werte an *ProbabilityDensityPlotter*, welche die entsprechenden Koordinaten zugehörig zu den Werten ermittelt und den Graphen zeichnet. Der Graph wird im *XYMeterWindow* gezeichnet und auf dem Bildschirm dargestellt.

## 4.2 Erstellen der einzelnen Parameter

Jedes Messgerät verfügt über mehrere Parameter, die es möglich machen, das Messgerät auf den gegebenen Signaltypen und Stärke möglichst genau einzustellen. Die ersten beiden Parameter, die bei der Messung der Wahrscheinlichkeitsdichte relevant sind, sind die für die X- und Y-Achse. Die Parametereinstellungen für die Achsen und alle Weiteren können über ein Zahnrad im oberen rechten Bereich des Fensters per Mausklick geöffnet werden. Zunächst werden nur die Einstellungen für die beiden Achsen angezeigt. Per Klick auf einen nach unten gerichteten Pfeil kann das Einstellungsfenster weiter aufgeklappt werden, um sich alle Parameter anzeigen zu lassen.

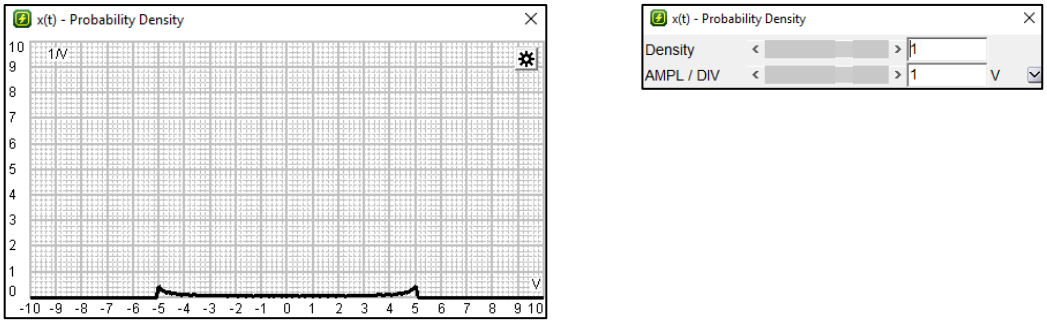


*Abb. 9: Messgerät mit Parameterfenster, dass über das Zahnrad geöffnet wird*

Die meisten Elemente der grafischen Anzeige bedienen sich an Basisklassen, die bereits implementiert wurden. Standardmäßig ist bei allen Parametern ein Schieberegler integriert, dem zusätzlich ein Eingabefeld beigefügt ist. So lässt sich bei der Nutzung sowohl mit der Maus arbeiten als auch über eine direkte Zahleneingabe beispielsweise die Auflösung der einzelnen Achsen verändern. Der Wert, der bei den Einstellungen beisteht, gibt nicht die komplette Länge der Achse an. Er bezieht sich lediglich auf ein Segment entlang der Achse. In der Regel ist das betrachtete Fenster in zehn Segmente aufgeteilt, die über Gitternetzlinien voneinander getrennt sind. Dies erleichtert zusätzlich die Betrachtung der Wahrscheinlichkeitsverteilung.

Auf der X-Achse wird die Amplitude aufgetragen, wobei der Nullpunkt zentriert ist. Als Standard ist eine Bandbreite von 20 Volt, also -10V bis +10V eingestellt. Im Fenster für die Einstellungen heißt dieser Parameter „AMPL/DIV“.

Die Y-Achse wäre eigentlich einheitenlos, da aber das Ergebnis durch die Amplitude geteilt wird, um eine Dichte zu berechnen, hat sie die Einheit „1/V“. Auch hier ist ein Schieberegler mit dem Namen „Density“ integriert mit dem die Segmentierung in Y-Richtung verändert werden kann.

Die implementierten Schieberegler sind vom Oszilloskop übernommen und können bei der Wahrscheinlichkeitsdichte ohne Einschränkung verwendet werden. Mit dem Übergabeparameter wird festgelegt, von welchem Minimalwert bis zu welchem Maximalwert der Regler die vorgegebenen Werte annimmt.

```
@Override
protected DoubleParameter createParameter() {
	DoubleParameter parameter = new IntDoubleParameter("Density", "",
	0d,new MinMaxIncrOsci(1e-6, 1e3));	parameter.value(1.0);
	return parameter;
}
```

*Abb. 10: Erstellen eines Parameters mit Werten vom Typ Double*

Die Implementierung eines solchen Parameters folgt immer dem gleichen Aufbau. Beide Klassen, die die Achsen darstellen, erben von der Klasse *DoubleProperty4Measure*. Dem Namen kann man entnehmen, dass sich die Werte der Paramater vom Typen *double* sind und somit ein sehr feines Zahlenspektrum abdecken. Mit der Implementierung der X-Achse besteht die Forderung, dass der Nullpunkt nicht als Ursprung des Koordinatensystems, sondern innerhalb des Anzeigefensters mittig der X-Achse angezeigt werden soll.

Die Klasse *CenterAmplitudeDens* ist eine aus einem anderen Messgerät abgewandelte Klasse, die dafür sorgt, dass der Nullpunkt sich immer mittig im Koordinatensystem befindet. Es ist zwar möglich bei den Parametereinstellungen den Nullpunkt zu verschieben, jedoch ist dies nicht von Vorteil für den Nutzer. Das Schaubild wird dann schnell unübersichtlich.

Zusätzlich zu den beschriebenen Achsen wird dem Parameter der Auflösung der Wahrscheinlichkeitsdichte eine hohe Bedeutung zugesprochen. Im Projekt heißt die entsprechende Klasse *ResolutionAmpl* und ist ebenfalls vom Typen *DoubleParameter*. Der Wert der *ResolutionAmpl* kann zwischen einem µVolt und einem Volt liegen. Die Auflösung meint, wie genau die einzelnen Werte gerundet und zugeordnet werden sollen. Dies ist insbesondere bei der Auswertung der Signalwerte wichtig, welche im nächsten Kapitel behandelt wird

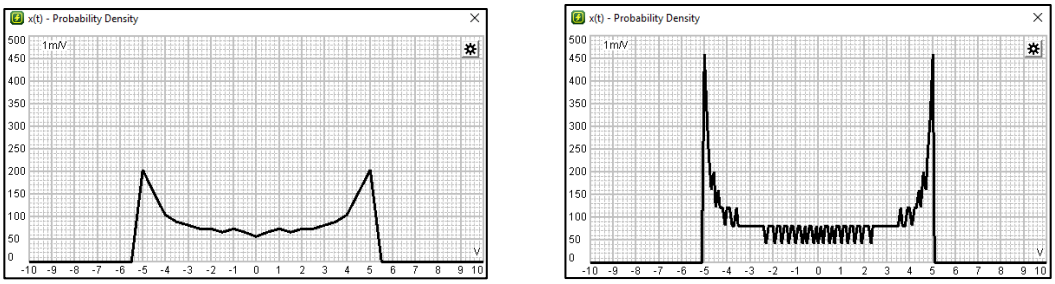


*Abb. 11: Wahrscheinlichkeitsdichte in niedriger und hoher Auflösung*

Die letzten beiden zugehörigen Parameter sind der Minimalwert *y-min* und die Anzahl der zu analysierenden Werte *Number of Values*. Mit dem Minimalwert y-min kann der Nullpunkt auf der Y-Achse nach oben geschoben werden und dadurch der negative Bereich ebenfalls sichtbar gemacht werden. Diesem Parameter ist allerdings im Vergleich zu *Number of Values* eine deutlich geringere Bedeutung zugesprochen, da bei der Wahrscheinlichkeitsdichte keine negativen Wahrscheinlichkeiten auftreten können. Der resultierende Graph wird sich ausschließlich im positiven Bereich befinden.

Mit dem Parameter *Number of Values* kann eingestellt werden, wie viele Signalwerte das Programm sammeln soll, bevor sie ausgewertet werden. Der Parameter ist als Standard auf 1.000 Werte eingestellt, was in der Regel ausreichend ist. Der Wert ist vom Typ *Double* und kann über einen Schieberegler zwischen 50 und 100.000 Signalwerten verändert werden. Bei einem niedrigen Wert fängt der Graph an zu wandern, da die Anzahl der Werte nicht ausreicht um die Wahrscheinlichkeitsdichte komplett zu bestimmen. Wird der Parameter zu hoch eingestellt, muss die Applikation eine große Menge an Daten analysieren und wird dadurch entsprechend gebremst und das Programm verzögert sich.

```
private void initParameterOrder() {
	getAmplDiv();
	getXDiv();
	getResolutionAmpl();
	getCenterFrequency();
	getNumberOfValues();
	getSerialization();
}
@Override
public MultiInstanceDoubleProperty getAmplDiv() {
	return new YDivDens(this);
}
```

*Abb. 12: Initialisierung der Parameter und der richtigen Reihenfolge*

Initialisiert werden alle Parameter im Konstruktor von *ProbabilityDensityParameters*. Dort wird die Funktion *initParameterOrder()* aufgerufen, die alle Getter-Methoden aufruft und so die einzelnen Parameter instanziiert. Des Weiteren wird so die korrekte Reihenfolge der Anzeigen im Parameterfenster gewährleistet.

Durch einen weiteren Klick auf den Pfeil unten links, wird das Einstellungsfenster ganz ausgeklappt und erscheinen noch zwei relevante Parameter, die standardmäßig von sogenannten Super-Klassen implementiert sind. Dies ist zum einen die Möglichkeit zur Veränderung der Fenstergröße. Zum anderen kann die Anzeige des Graphen verändert werden. Der klassische Graph kann auch als aneinandergereihte Diracs dargestellt werden. Eine Darstellung in Diracs macht bei einem Gleichsignal oder einem Rechtecksignal Sinn.

## 4.3 Implementierung der Logik

In der Datei *ConfigModel.java* werden die Messgeräte erstellt und dann in *Config.java* deklariert. Gleichzeitig werden in der Config-Datei die Initialisierungs-Methoden aus *ProbabilityDensity.java* aufgerufen.

```
density=new
ProbabilityDensity().size(AspectRatio._16_9).AmplDiv().onetoVolt().resolutionAm
pl(0.1).numberValues(1000).centerFrequency();
```

*Abb. 13: Instanziierung von ProbabilityDensity in config.java*

Hier werden die ersten Einstellungen nach dem Klon-Prinzip für das Messgerät vorgenommen.

```
public ProbabilityDensity AmplDiv(){
	ProbabilityDensity clone = clone();
	clone.getParameters().setXRange(new Range(-10,10));
	return clone;
}
```

*Abb. 14: Klon vom Typ ProbabilityDensity wird erzeugt und zurückgegeben*

Die Klasse *ProbabilityDensity* wird geklont und der Parameter für die X-Achse, also die Amplitude auf die Skala von -10 bis +10 eingestellt. Anschließend wird der Klon als Rückgabewert ausgegeben und übernommen.

Nach dem gleichen Prinzip werden auch weitere Initialisierungen vorgenommen. Es wird die Größe des Fensters für das Schaubild festgelegt und die Skala für die Y-Achse eingestellt. Abschließend werden noch die Standardwerte für weitere Parameter festgelegt.

Das Klon Prinzip ist erforderlich, da bei den meisten Schaltungen mehrere Leitungen existieren. Ein bestehendes Set, also bereits geöffnetes Messgerät, wird nicht verändert, sondern neu erstellt. Dies dient dazu Komplikationen zu vermeiden, wenn die gleiche Instanz für mehrere Leitungen verwendet wird.

In der dritten Config-Datei *ConfigPropagator.java* wird festgelegt für welche Signaltypen das Messgerät eingesetzt werden kann. Grundsätzlich gibt es drei Signaltypen: Analog, Digital und Komplex. Das Messgerät zur Messung der Wahrscheinlichkeitsdichte ist bei allen hinterlegt und somit frei nutzbar.

```
measuresPerSignalType.put(DigitalSignal.class, digitalScope, signalViewer,
density);
```

*Abb. 15: Das Messgerät density wird zu Klassen mit digitalen Signalen hinzugefügt*

Über einen Rechtsklick auf eine Leitung und dem Auswählen von *ProbabilityDensity* wird das Messgerät gestartet und weitere Initialisierungen werden zunächst ausgeführt. Die Funktion des eigentlichen Messgerätes ist in der Klasse *ProbabilityDensityMeter* implementiert. Die Klasse verfügt über vier wichtige Instanzvariablen, die maßgebliche für die Funktion des Messgerätes zuständig sind. Die erste wichtige Variable ist die der Parameter. Die zweite Variable ist für die Logik nötig, da in ihr die Signalwerte gesammelt und ausgegeben werden. Diese Signalwerte werden in einer weiteren Instanzvariable verarbeitet und das Ergebnis mit einem Plotter angezeigt. Die genaue Funktion wird im weiteren Verlauf beschrieben.

```
private ProbabilityDensityParameters pdp;
private SignalArrayDens signalArray;
private ProbabilityDensityLogic pdl;
public ProbabilityDensityPlotter probabilitydensityplotter;
```

*Abb. 16: Die vier lokalen Variablen der Klasse ProbabilityDensityMeter*

In der Methode *init()* werden diese Instanzvariablen zunächst initialisiert, damit diese zur Funktion der Messung benutzt werden können.

Die Funktion *meter(Signal signal)* wird bei jedem neuen Signalwert aufgerufen und übergibt das aktuelle Signal dem zugehörigen Wert. Da es sich bei einer Wahrscheinlichkeitsdichte um eine Auswertung von vielen Signalwerten handelt, müssen erst mehrere Signalwerte gesammelt werden, bevor diese ausgewertet werden können.

```
@Override
public void meter(Signal signal) {
	// TODO Auto-generated method stub
	signalArray.addSignal(signal);
	if (signalArray.isFull()) {
		probabilitydensityplotter.setBusy(true);
		double[] density = pdl.getDensity(signalArray.getArray());
```

*Abb. 17: Funktion meter() sammelt Signalwerte um sie dann auszuwerten*

In der Klasse *SignalArrayDens* werden zunächst so viele Signalwerte gesammelt, wie vom Benutzer vorgegeben werden. Der einstellbare Wert ist über den Parameter *Number of Values* veränderbar, welcher sich im Parameterfenster befindet.

Befinden sich genug Werte in dem internen Feld der Klasse *SignalArrayDens*, signalisiert es über eine boolesche Variable, die mit *isFull()* abgefragt wird, dass jetzt die Auswertung beginnen kann. Mithilfe des Plotters wird dem Fenster mitgeteilt, dass der Graph verändert wird.

Das Array[4] mit den Signalen wird dann an die Klasse *ProbabilityDensityLogic* übergeben, welche die Signalwerte analysiert.

*ProbabilityDensityLogic* ist eine eigenständige nicht verknüpfte Klasse, die alleine die Auswertung der gesammelten Signalwerte vornimmt, was im folgenden Kapitel beschrieben wird.

[4] Arrays sind Felder von Datentypen, die es möglich machen mehrere Werte diesen Types zu speichern.

## 4.4 Auswertung der Signalwerte

In der Klasse *ProbabilityDensityLogic* wird das übergebene Array mit den Signalwerten ausgewertet. Zuständig dafür ist die Methode *getDensity()* die als einzige Funktion dieser Klasse für andere Packages sichtbar ist. Die weiteren zwei relevanten Funktionen werden nur innerhalb der Klasse benötigt. Als globale Variable werden bei der Initialisierung dem Objekt die Parameter verlinkt, da diese für die Auswertung wichtig sind. Das heißt, dass bei einer Veränderung der Parameter durch den Nutzer, diese Information automatisch weitergeleitet wird.

Der erste Schritt zur Bestimmung der Wahrscheinlichkeitsdichte ist die Festlegung der aktuellen Skala auf der X-Achse. Diese kann einfach über das Objekt der Parameter ausgelesen werden. Im Folgenden besteht die Methode aus zwei For-Schleifen[5], die jeweils eine eigene Funktion haben.

```
double[] tmp = new double[signal.length];

for(int j = 0; j<signal.length; j++) {
	tmp[j] = round(signal[j].analogValue(), resolution);
}
double[] density = new double[(int)Math.round((range*(1/resolution)+1))];

for(int i = 0; i < tmp.length; i ++) {

	double place = Math.round((tmp[i]/resolution)+((range/2.0)/resolution));
	int position = (int) place;

	if(position < density.length && position >= 0) {
		density[position]++;
	}else {
	}
}
```

*Abb. 18: Auswertung der übergeben Signalwerte mittels zwei For-Schleifen*

Die erste For-Schleife entnimmt die Werte aus dem übergebenen Array. Da diese Werte vom Typen Double sind und in der Regel viele Nachkommastellen haben, die Auflösung aber nicht genau so hoch oder niedrig eingestellt ist, müssen diese Werte entsprechend angepasst werden. Die Auflösung ist beim Start auf 0.1 eingestellt, so dass jeder Wert auf 1/10 gerundet wird und in ein neues Array geschrieben.

[5] Ist eine Schleife, die so lange ausgeführt wird, bis eine Bedingung erfüllt ist.

```
private static double round(double value, double resolution) {
	double tmp = Math.round(value * (1/resolution)) / (1/resolution);
	return tmp;
}
```

*Abb. 19: Funktion rundet den übergebenen Wert auf die gewünschte Auflösung*

Sind alle Werte der Auflösung entsprechend gerundet ist der nächste Schritt die Berechnung der Amplitudenverteilung. Die zweite For-Schleife ermittelt die Häufigkeitsverteilung der Amplitudenwerte und zählt entsprechend die Stelle in einem neuen Array nach oben.

Dieses neue Array hat genau so viele Stellen wie die X-Achse entsprechend ihrer Auflösung besitzt. Hierbei ist zu beachten, dass die X-Achse links nicht mit null beginnt, sondern einem negativen Wert, da die X-Achse zentriert ist. Dies muss sowohl bei der Berechnung im Array als auch später beim Zeichnen des Graphen berücksichtigt werden. Das heißt jede Position im Array stellt ein Punkt auf der X-Achse dar, mit seinem zugehörigen Y-Wert. Der Y-Wert ist nach der zweiten For-Schleife noch die Anzahl, wie oft dieser Wert vorgekommen ist. Im nächsten Schritt wird die Häufigkeitsverteilung auf die Dichte heruntergerechnet.

Die Methode *normalize()* in der Klasse *ProbabilityDensityLogic* berechnet aus den hochgezählten Werten des übergeben Arrays die Wahrscheinlichkeitsdichte. Neben dem Array, welches übergeben wird, wird der Methode noch die aktuelle Auflösung der Amplitude *resolution,* in der Einführung auch Delta X genannt, mitübergeben. Diese ist notwendig um die Berechnung korrekt auszuführen.

```
private synchronized double[] normalize(double[] density, double resolution) {
	int numberValues = (int)this.pdp.getNumberOfValues().value();
	for(int i = 0; i < density.length ; i++) {
		density[i] = ((density[i]/numberValues)/(resolution));
	}
	return density;
}
```

*Abb. 20: Berechnung der Dichte in der Funktion normalize()*

Der gezählte Wert wird zunächst durch die Gesamtzahl der Signale geteilt und dann noch durch die eingestellte Auflösung. Da die Auflösung der Amplitude in Volt ist, resultiert aus dem Ergebnis die Einheit 1/Volt. Ohne diese Teilung durch die Auflösung würde ein Ergebnis in prozentualer Darstellung entstehen.

Die ermittelten Werte werden an die aufrufende Klasse *ProbabilityDensityMeter* zurückgegeben. Dort werden sie temporär in ein Array geschrieben um sie an den Plotter weiterzugeben, dessen Aufgabe es ist den resultierenden Graphen zu zeichnen.

## 4.5 Anzeigen des Ergebnisses

Für die Darstellung der Ergebnisse, also das was der Nutzer dann am Bildschirm sieht, wird die Klasse *ProbabilityDensityPlotter* benötigt. Diese besitzt zwei lokale Variablen und erbt von keiner Super-Klasse.

Bei der ersten lokalen Variable handelt es sich um die Klasse für die Parameter, wodurch die aktuellen Einstellungen im Parameterfenster abgefragt werden können. Die andere lokale Variable stellt das Fenster für die Grafik dar. Dieses Objekt ist vom Typ *XYMeterWindow* und ist die Basis-Klasse für alle grafisch arbeitenden Messgeräte. Diese grafisch arbeitenden Messgeräte wie das Oszilloskop und der Spektrumanalysator stellen ihre Ergebnisse auch in einem Koordinatensystem dar. Daher ist es möglich die Grundfunktionen der Mechanismen für die Anzeige der Schaubilder zu verstehen und zu übernehmen. Es muss allerdings das Koordinatensystem bearbeitet werden, da die Achsen andere Einheiten besitzen. Zudem befindet sich der Nullpunkt bei diesem Messgerät nicht im Ursprung, sondern mittig im Koordinatensystem.

Die Grafik des Messgerätes wird von der errechneten X-Koordinate ausgerichtet. Die berechneten Werte, die im Array abgespeichert sind, sind ebenfalls an den X-Werten ausgerichtet. Das heißt, die Stelle im Array an der der Wert steht beschreibt die X-Koordinate und der zugehörige Wert die Y-Koordinate. Die Klasse *AmplitudeXCoordinate* beschreibt die X-Koordinaten von der aus über den errechneten Y-Wert der Punkt im Schaubild gezeichnet werden kann.

Die Methode *plotDensity()* hat als Übergabeparameter das errechnete Array und die Auflösung, welche beide vom Typen Double sind. Mit der globalen Variable *XYMeterParameters* können alle aktuellen Einstellungen abgefragt werden. Diese sind wichtig, um die Grafik anhand der Parameter korrekt darstellen zu können.

```
dispWin.beams.reset();

double indexMin = pdp.getXMinValue();
double indexMax = (pdp.getXMinValue() + pdp.getXDivisions().getValue() *
pdp.getFrequencyDiv());

double range = round(Math.abs(indexMin) + Math.abs(indexMax), resolution);
```

*Abb. 21: Bestimmung der Indizes*

Zunächst wird der Index für den Minimal- und Maximalwert auf der X-Achse errechnet. Aus diesen beiden Indizes und unter Berücksichtigung der Auflösung wird die Abbruchbedingung für die folgende For-Schleife bestimmt. Diese stimmt überein mit der der Größe des Arrays.

```
for (double a = 0; a <= range; a = round(a + resolution, resolution)) {
	AmplitudeXCoordinate x = new AmplitudeXCoordinate(round(indexMin+a,
	resolution), pdp, dispWin);
	try {
		double tmp_2 = Math.round(a/resolution);
		int tmp = (int) tmp_2;
		dispWin.beams.addPoint(x, density[tmp]);
	} catch (ArrayIndexOutOfBoundsException e) {
		System.out.println("Plotter out of window: " + e.getMessage());
	}
}
```

*Abb. 22: Erstellen der einzelnen Koordinaten im Fenster*

Die Klasse *AmplitudeXCoordinate* implementiert das Interface[6] *CoordinateI*. Die einzelnen Punkte werden durch die obenstehende Schleife gezeichnet und automatisch durch bereits existierende Funktionen verbunden, wodurch der Graph entsteht. Wichtig ist hier, dass genau gerundet wird. Das liegt daran, dass bei einer hohen Auflösung die Zählvariable *a* erhöht wird und im weiteren Verlauf mit dieser gerechnet wird. Dabei ist zu vermeiden, dass durch Rundungsfehler die falsche Stelle im übergebenen Array erreicht wird und somit ein falscher Graph ausgegeben.

Die Variable *dispWin* stellt das Fenster dar, in dem mit der Methode *addPoint()* die einzelnen Punkte eingefügt werden. Damit die Applikation während der Laufzeit bei einem Fehler nicht anhält und abstürzt, findet der Teil in einem Try-Catch Block statt. Insbesondere werden hier Fehler abgefangen, bei denen Werte im Array versucht werden abzufragen die außerhalb der Array-Bandbreite liegen und somit nicht existieren.

[6] Schnittstelle, die vorgibt welche Funktionen die Klasse enthalten muss.

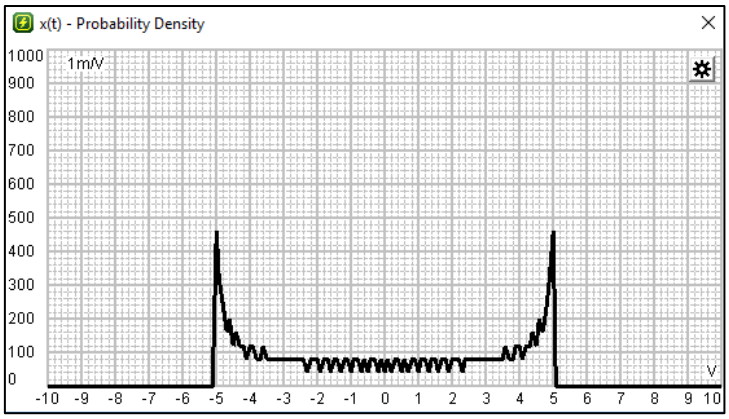


*Abb. 23: Wahrscheinlichkeitsdichte bei einem Sinussignal*

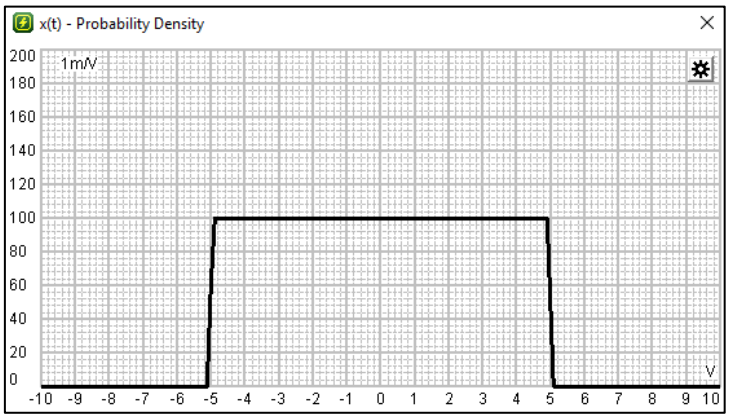


*Abb. 24: Wahrscheinlichkeitsdichte bei einem Sägezahn Signal*

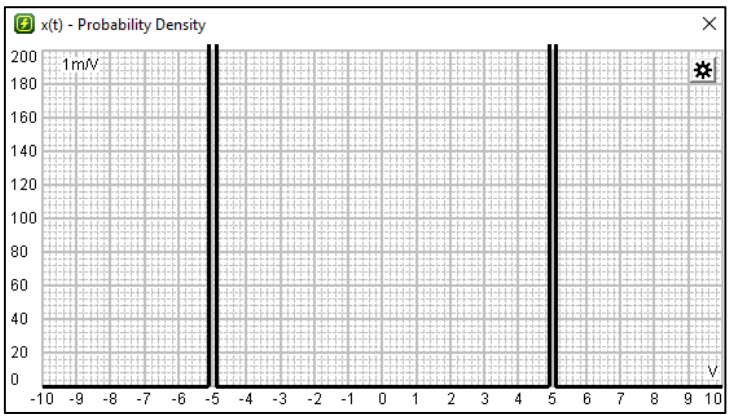


*Abb. 25: Wahrscheinlichkeitsdichte bei gleichverteiltem Rechteck Signal(-5V,5V)*

# 5. Serialisierung und Deserialisierung von Parametern

## 5.1 Vorgehensweise für die Serialisierung und Deserialisierung

Die Speicherung und das wieder aufrufen von Parametereinstellungen hat keine genauen Zielvorstellungen, da in der labAlive-Umgebung noch keine ähnlichen Mechanismen implementiert sind. Die Umsetzung dieser Aufgabe lässt daher Raum, mehrere Möglichkeiten auszuprobieren und zu evaluieren.

Bevor die Implementierung der Logik beginnen kann muss ein einfacher Weg gefunden werden, das Ergebnis zu überprüfen. Da in dem Projekt zur Laufzeit etliche Verlinkungen existieren, die wichtig für die Funktionsweise sind, muss auch die Entwicklung direkt im Projekt stattfinden und kann nicht zunächst in einem Projekt außerhalb geprobt werden.

Die Umstellung im Parameterfenster zwischen durchlaufendem Graphen und der Anzeige in Diracs erfolgt über einen ausklappbaren Reiter. Für einen Versuch, um sich an diesem komplexen Thema zu probieren, ist das ein passendes Modul, um den aktuellen Code schnell und einfach zu testen. Für die spätere fertige Version ist möglicherweise ein besserer Weg zu finden.

Die Klasse für den beschriebenen Reiter heißt *SerializationProperty* und erbt von der Super-Klasse *SelectProperty4Measure*.

```
@Override
protected SelectParameter createParameter() {
       SelectParameter    parameter    =    new    SelectParameter("Save/Load",
Serialization.RUNNING);
      parameter.addOptions(Serialization.values());
      parameter.detailLevel(ParameterDetailLevel.DETAIL_LEVEL1);
      return parameter;
      }
```

*Abb. 26: Erstellen des Parameters für die Serialisierung*

Die einzelnen Optionen werden in einer gesonderten Klasse vom Typ *enum*[7] definiert und dann bei der Initialisierung hinzugefügt. Es gibt drei Optionen, die zur Verfügung stehen: running, save, load. Mit *addOptions()* werden die Werte aus der Enumeration zugefügt. Der Rückgabewert ist automatisch vom Typ *SelectParameter.*

[7] Enum, zu Deutsch Enumeration (Aufzählung), ist ein Datentyp, der nur bestimmte Werte aufnehmen kann.

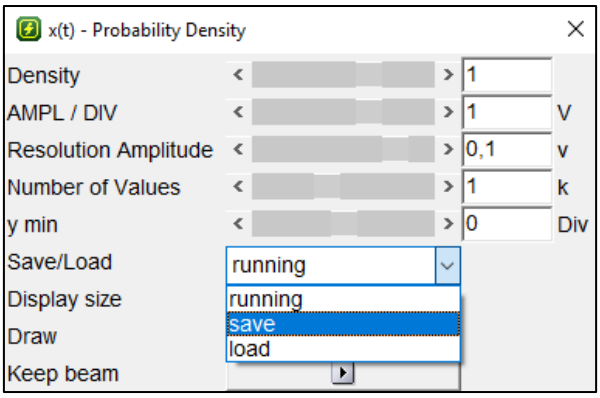


*Abb. 27: Paramater-Reiter zum Speichern und Laden im Einstellungsfenster*

Bei normaler Laufweise der Applikation ist immer *running* eingestellt. Der aktuelle Status dieses Parameters wird über eine If-Else Struktur in der Funktion *meter()* abgefragt.

```
if(pdp.getSerialization().getValue() == Serialization.SAVE) {
	super.saveParameter();
}else if(pdp.getSerialization().getValue() == Serialization.LOAD) {
	super.loadParameter();
}
```

*Abb. 28: Abfrage der Reiterstellung im Parameterfenster*

Die weitere Klasse, die für die eigentliche Funktion zuständig ist, heißt *JString* und befindet sich im gleichen Package *Serialization*. Sie bietet zwei Funktionen einen String zu serialisieren. Den übrigen Klassen werden nur die wichtigsten Funktionen angezeigt, da sie als *private* deklariert und somit nach außen unsichtbar sind. Des Weiteren verfügt diese Funktion über die Methoden, um den String aus dem Programm heraus an einem Speicherort abzulegen und von dort wieder aufzurufen. Die Klasse bietet ebenso entsprechende Methoden, um die Informationen aus dem String zu filtern und dann in die aktuelle Klasse für die Paramater zu laden.

## 5.2 Erstellen des Strings und Serialisierung

Bei der Serialisierung und Deserialisierung vom Java Haus aus, werden Objekte in Bytes und wieder zurückgewandelt. So ist es eigentlich gewollt, Daten in einem Netzwerk oder Datenbank abzulegen und wiederaufzurufen. Will ein solches Objekt serialisieren, muss die Klasse *java.io.Serializable* implementieren und so kann die Instanz serialisiert werden [4].

Das funktioniert nur wenn die Klasse keine Verlinkungen zu anderen Klassen besitzt, also von keiner Super-Klasse erbt.

Sollte es der Fall sein, muss auch diese das Interface implementieren, da diese mit serialisiert werden muss. Wegen dieser Problematik war der Weg im Projekt labAlive sehr umständlich und nicht umzusetzen. Allein die Klasse *XYMeterParameters* erbt über drei Ebenen, genauso wie die einzelnen Objekte, die in dieser HashMap[8] hinterlegt sind. Daher musste sich mit einem alternativen Weg beholfen werden.

Im Wesentlichen sind zwei Wege implementiert, um einen String zu erstellen. Dies ist darauf zurückzuführen, dass im Laufe der Entwicklung eine andere Vorgehensweise für sinnvoller erachtet wurde.

Zunächst wird versucht einen String automatisch von Java erstellen zu lassen. Hierbei gibt es die zwei Möglichkeiten, die Klasse für die Parameter direkt mit der Methode *.toString()* einen String zu erstellen oder erst mit einem Zwischenschritt über eine Map. In beiden werden alle Parameter mit den zugehörigen Werten getrennt mit verschiedenen Zeichen aufgelistet. Jedoch stellt dieser String kein standardisiertes Format dar und ist auf den ersten Blick auch sehr unübersichtlich. Die Klasse *JString* stellt aber trotzdem eine Methode zur Verfügung, um diesen String zu extrahieren. Da dies ein erster Versuch war, der funktioniert sind beide Methoden, also für die Erstellung des Strings und Extraktion in der Klasse verblieben.

```
public String createJString(XYMeterParameters parameter) {
	Set<Entry<Object, Parameter>> map = parameter.entrySet();

	String string = map.toString();
	System.out.println(string);
```

*Abb. 29: Einfache Methode um serialisierbaren String zu erstellen*

Der resultierende String sieht dann wie folgt aus:

```
Density=Density: 1 , AMPL / DIV=AMPL / DIV: 1 V, Resolution Amplitude=Resolution
Amplitude: 0,1 v,
```

Es sind in diesem alle Informationen die wichtig sind enthalten, allerdings etwas umständlich. So werden die einzelnen Parameter überflüssigerweise doppelt aufgeführt. Des Weiteren fehlt eine eindeutige Identifikation des Messgerätes. Es steht zwar ganz am Anfang, dass es sich um das Messgerät *Density* handelt, wesentlich präziser wäre es aber hier die exakte Klasse zu nennen, um eventuellen Verwechslungen vorzubeugen. Im Weiteren werden die Kommata, die Paare voneinander trennen durch Semikolons ersetzt.

[8] Dort werden Schlüssel mit zugehörigen Werten angegeben. Mit dem Schlüssel kann der Wert später wieder abgefragt und verändert werden.

Da auch die Werte Kommata verwenden, um ihre Nachkommastellen zu trennen, ist das Semikolon ein eindeutiges Trennzeichen, um die Wertpaare zu identifizieren. Der weitere anschaulichere String ist im Format Json[9], welches ein schlankes Datenaustauschformat ist.

Der Wesentliche Vorteil von JSON liegt darin, dass es für den Betrachter einfach zu lesen ist und für den Computer einfach zu parsen und zu generieren. Json ist komplett unabhängig von Programmiersprachen, hat aber viele Wurzeln aus den C-basierten Sprachen übernommen. Mit diesen Eigenschaften ist Json ein geeignetes Format für den Austausch von Daten. JSON besitzt zwei Strukturen. Zum einen sind das die „Name/Wert“ Paare. Dies können zum Beispiel Objekte oder Hash-Tabellen sein. Oder zum anderen eine geordnete Liste von Werten, also sogenannte Arrays, Vektoren oder Listen. Es handelt sich dabei um universelle Datenstrukturen, die fast alle modernen Programmiersprachen unterstützen. [5]

Alle Parameter eines Messgerätes werden als Objekte in einer Hash-Tabelle gespeichert. Der String stellt hierbei den Namen des Parameters dar. Der aktuell eingestellte Wert wird hinter dem String mit einem Doppelpunkt eingetragen. Der ganze resultierende String befindet sich in geschweiften Klammern, worin die einzelnen Paare durch Kommata getrennt werden.

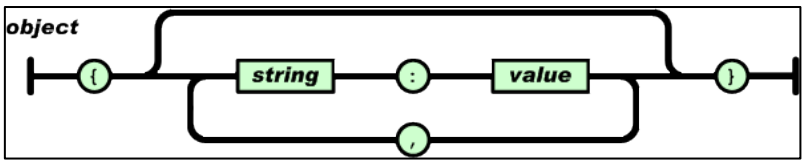


*Abb. 30: Zusammensetzen des Strings bei JSON (Quelle: https://www.json.org/json-de.html)*

Es gibt von Json eine online zur Verfügung gestellte Bibliothek, die in das Projekt in Eclipse eingebunden werden kann. Leider stellt diese lediglich Schnittstellen zur Verfügung und keine Funktionalität. Im Internet findet man noch weitere Bibliotheken, die Methoden zur Verfügung stellen sollen, um mit Json arbeiten zu können. Diese befinden sich auf nicht vertraulichen Webseiten und wurden deswegen aufgrund der Virengefahr nicht getestet.

Das heißt, die Erstellung des Json Strings musste genauso manuell erfolgen wie das spätere zurückwandeln in eine Hash-Tabelle. Für die Erstellung des Strings im Json-Format ist die Methode *createJsonString()* zuständig. Diese Funktion fügt in einer For-Schleife die einzelnen Schlüssel und Werte der Hash-Tabelle von den Parametern zusammen, nach den Kriterien des JSON Format. Die einzelnen Strings werden aufeinander aufaddiert, woraus der serialisierte String automatisch entsteht.

[9] Abkürzung: JavaScript Object Notation

```
List keys = new ArrayList(parameter.keySet());
for(int i = 0; i < keys.size(); i++) {

	Object obj = keys.get(i);
	double val = (double) parameter.get(obj).getValue();
	string = string + ",\"" + obj.toString() + "\": " + val ;
}
```

*Abb. 31: Erstellen des Strings im Json Format*

Als erstes String/Wert-Paar wird der genaue Klassentyp des Messgerätes eingefügt, um später sicher zu gehen, die richtigen Parameter für das richtige Messgerät zu laden.

Der entstandene Json String sieht dann wie folgt aus:

```
{"Measure": "de.labAlive.measure.probabilitydensity.ProbabilityDensityParameters",
"Density": 1.0,"AMPL / DIV": 1.0,"Resolution Amplitude": 0.1,"Center frequency":
0.0,"Number of Values": 1000.0,"Delay": -10.0,"x": 20.0,"y min": 0.0,"y":
10.0,"Offset": 0.0}
```

Bis hierhin wurde lediglich die Erstellung des zu speichernden Strings besprochen, allerdings noch nicht den eigentlichen Weg, diesen String außerhalb der Applikation abzulegen. Dieser Mechanismus wird im Folgenden beschrieben. Beide String Typen werden nach dem gleichen Verfahren serialisiert. Es ist wichtig einen angenehmen Weg zu implementieren, der es dem Nutzer erlaubt, möglichst einfach zu Speichern und zu Laden.

Java bietet deshalb die Klasse *JFileChooser*. Diese Klasse eröffnet dem Nutzer einen Datenexplorer, in dem er flexibel den gewünschten Pfad auswählen kann, in dem die Datei gespeichert werden soll. Wichtig hierbei ist dann nur am Ende einen Namen anzugeben mit der Endung .json um den Typen dieser Datei eindeutig zu definieren.

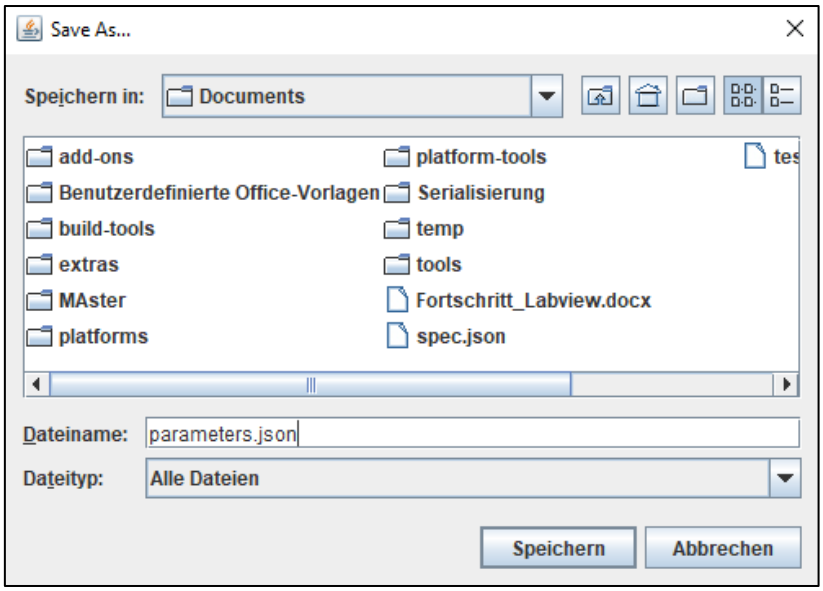


*Abb. 32: Java JFileChooser zum Auswählen des Speicherortes*

Dadurch wird ebenfalls gewährleistet, dass andere Applikationen diese Datei als geeignet erkennen. Mithilfe eines *FileWriter* wird die Datei beschrieben und dann dort abgelegt. War die Serialisierung erfolgreich, gibt die Funktion *true* zurück, beziehungsweise *false*, sollte ein Fehler aufgetreten sein. Das gleiche Fenster öffnet sich auch, wenn ein String geladen werden möchte.

```
private static boolean serialize(String string) {

          File file = null;
          JFileChooser chooser = new JFileChooser();
          chooser.setDialogTitle("Save As...");

          int result = chooser.showSaveDialog(null);
          if (result == JOptionPane.OK_OPTION)
            file = chooser.getSelectedFile();
          else
            return false;
          System.out.println(file.getAbsolutePath());
          try {
            FileWriter fw = new FileWriter(file);
            fw.write(string);
            fw.flush();
            fw.close();
          }catch(Exception e) {
            System.out.println(e.getMessage());
            return false;
          }
            return true;
      }
```

*Abb. 33: Funktion in Klasse JString um serialisierten String abzulegen*

## 5.3 Schritte zur Deserialisierung

Die Datei wird wie bei der Speicherung über *JFileChooser* in einem Explorer ausgewählt und über einen Button bestätigt. Sollte die Datei nicht gelesen oder geöffnet werden können, gibt die Applikation automatisch einen Fehler aus, der dem Nutzer in der Konsole angezeigt wird. Diese wird dann zunächst in einem *File* geschrieben. Mithilfe eines *FileReaders* werden die Zeichen aus der externen Datei in ein Array vom Typ *char* geschrieben. Der Typ *char* wird insbesondere wichtig, wenn später Informationen ausgelesen werden sollen.

Zunächst ist der Applikation allerdings unbekannt wie groß das Array ist, in welches der Json String geschrieben wird. Deshalb wurde ein Array implementiert, welches die feste Größe von 400 hat, was in der Regel ausreicht. In einer folgenden While-Schleife[10] wird die Länge des geladenen Char-Arrays ermittelt und dann in einer weiteren For-Schleife in ein weiteres Array kopiert, welches die exakte Größe des Char-Array hat.

```
if(result == JOptionPane.OK_OPTION)
	file = chooser.getSelectedFile();
else
	return null;
char[] tmp = new char[400];
try {
	FileReader fr = new FileReader(file);
	fr.read(tmp);
	fr.close();
}catch(Exception e) {
	System.out.println(e.getMessage());
}
```

*Abb. 34: Öffnen der zu ladenden Parameter-Datei*

Die Klasse *JString* stellt eine Methode zur Verfügung, mit der beliebig ein String in ein Char-Array und zurück gewandelt werden kann. Für die weitere Analyse ist es nötig, den Typ Char beizubehalten, da dieser die weitere Verarbeitung erheblich erleichtert.

[10] While-Schleife wiederholt Anweisungen, so lange ihre Laufzeitbedingung gültig ist (ähnlich wie For-Schleife).

## 5.4 Extrahieren des geladenen Strings

Da keine vertrauensvolle Json-Bibliothek zu finden war, mit der man automatisch den String in ein Java Objekt umwandeln kann, muss die Lösung manuell eingepflegt werden. Die Vorgehensweise für die Extraktion der Information bei dem geladenen String folgt sowohl bei dem Json-Format, als auch bei dem ersten Versuch dem gleichen Prinzip. Im Weiteren wird lediglich das Json-Format betrachtet, da dies auch in Zukunft verwendet werden soll.

Für die weitere Analyse des geladenen Arrays gibt es die Methode *extractJsonInfo()*, welche eine HashMap als Rückgabewert besitzt. Die HashMap wird zu Beginn der Funktion initialisiert und dann im Weiteren mit Daten befüllt. Übergeben wird der Funktion ein Array vom Typ Char und die aktuellen Parameter. Diese müssen mit übergeben werden, um im ersten Schritt zu überprüfen, ob die richtigen Parameter für das richtige Messgerät geladen wurden.

Die ganze Methode arbeitet nach dem Prinzip der Analyse der einzelnen Character. Die Tatsache, dass jedes Zeichen, also Buchstaben, Zahlen oder Sonderzeichen einen eigenen Wert besitzen, macht es sehr angenehm diese zu vergleichen oder bestimmte Zeichen zu ermitteln. Um sicher zu gehen, dass es sich um einen Json String handelt werden die ersten zwei Zeichen überprüft, da diese immer gleich sein müssen.

```
if(!compareChar(chars[0], '{') || !compareChar(chars[1], '"'))
      return null;        // No Json String
```

*Abb. 35: Überprüfung auf Json String*

Im zweiten Prüfzyklus wird dann der erste Eintrag betrachtet. Dort steht immer um welchen Messgerätetypen es sich handeln muss, für den diese Parameter bestimmt sind. Um sicher zu gehen, wird gleich der komplette Klassenname hinterlegt.

```
// Check Right Measure
i_skey = searchChar(chars,'"',i_skey)+1;                    // +1 & -1 because of
"M... set start at M
i_ekey = searchChar(chars,'"',i_skey)-1;
String key_m = chartoString(copy(chars,i_skey,i_ekey));
i_sval = searchChar(chars,'"', i_ekey+2)+1;
i_eval = searchChar(chars,'"', i_sval)-1;
String val_m = chartoString(copy(chars, i_sval, i_eval));

if(!key_m.equals("Measure")  ||  !val_m.equals(parameter.getClass().getName()))
      // Wenn der JSON String nicht zu den Parametern passt
      return null;
```

*Abb. 36: Überprüfung auf korrekte Parameter für korrektes Messgerät*

Wurden beide Überprüfungen erfolgreich durchlaufen, folgt das eigentliche Lesen der Information. Dies geschieht in einer Schleife, die so lange ausgeführt wird, wie das Char-Array lang ist.

Mit den einzelnen Trennzeichen können Schlüssel und Werte getrennt gefunden und danach gemeinsam abgespeichert werden. Der Schlüssel steht immer in Anführungszeichen und ist vom Typen String. Nach dem Schlüssel kommt gefolgt hinter einem Doppelpunkt der Wert, der dann grundsätzlich in den Typen Double gewandelt werden. Die ganze Suche, benutzt Integer Werte als Indizes um die genauen Stellen zu ermitteln. Mit der Funktion *copy()* die die Indizes als Übergabeparameter und das gesamte Char Array bekommt, werden die entsprechenden Ausschnitte aus dem Array kopiert. Die Schlüssel werden mit der Methode *chartoString()* in einen String gewandelt. Mit ihnen können später die aktiven Parameter des Messgerätes ermittelt werden. Die Methode *chartoDouble()* ermittelt aus den Charactern die entsprechenden Double Werte, damit auch später verwendete Integer Werte bequem wieder zurück gecastet werden können.

```
while(forward) {

	i_skey = searchChar(chars,'"',i_eval)+1;
	i_ekey = searchChar(chars,'"',i_skey)-1;
	char[] ch_key= copy(chars,i_skey,i_ekey);

	i_sval = searchChar(chars,':',i_ekey)+2;
	i_eval = searchChar(chars,',', i_sval)-1;
	char[] ch_val= copy(chars,i_sval,i_eval);

	System.out.println(chartoString(ch_key)+ " " + chartoDouble(ch_val));
	map.put(chartoString(ch_key), chartoDouble(ch_val));

	if((i_eval+1)-(chars.length-1)==0)
		forward = false;
}
```

*Abb. 37: Suchen der Trennzeichen, um Schlüssel und zugehörige Werte herauszukopieren*

Mit der Funktion *searchChar()* werden die Trennzeichen im String gesucht und mit den Indizes gespeichert. Danach wird mithilfe der Grenzen, also den gefundenen Indizes, die entsprechende Stelle herauskopiert. Dies geschieht sowohl bei den Schlüsseln als auch bei den Werten gleich. Danach werden Schlüssel und Werte in einer Map abgelegt.

Die fertige Map wird zurück an die aufrufende Klasse übergegeben, von wo aus der nächste Schritt eingeleitet wird. Die gefundenen Parameter müssen jetzt in die laufende Applikation eingepflegt werden, was mit der Methode *loadParams()* geschieht, der die fertige Map übergeben wird. Die Liste der Schlüssel und Werte werden ausgelesen, um sie temporär in einer List zu speichern. In einer Schleife werden die Parameter der aktuell laufenden Applikation ermittelt und dann mit neuen Werten hinterlegt. Bei einem ersten Versuch wurden die Parameter vom Graphen direkt angenommen und neu gezeichnet. Jedoch behielt das Fenster für die Einstellungen die alten Werte, die vor dem Laden der Parameter dort angezeigt wurden. Wenn der Nutzer dann eine Einstellung vornimmt, werden wieder alle Werte aus dem Einstellungsfenster übernommen. Das heißt, dem zuständigen Objekt für die Parameter die neuen Werte zu geben reicht nicht, da noch im Hintergrund die alten vorhanden sind und das laufende Messgerät sich auf diese bezieht.

Es gab nun die zwei Möglichkeiten, die Stelle in dem großen Projekt zu suchen, wo die alten Daten noch liegen oder eine Aktualisierung bzw. Neuinitialisierung der entsprechenden Objekte. Zunächst wurde nach dem Ort gesucht, an dem alte Werte hinterlegt waren. Das Projekt umfasst auch mittlerweile eine riesige Menge an Klassen, Verlinkungen und Zeilen Code, so dass die weitere Arbeit in dieser Richtung nicht aussichtsreich war. Daher musste ein neuer Weg gesucht werden, die Anzeige zu aktualisieren.

Über die Klasse *ProbabilityDensityPlotter* können sich in einem Array alle aktuell geöffneten Fenster angezeigt und bearbeitet werden.

```
public static void closePropertyWindows(java.awt.Window[] windows) {

	for(int i = 0; i < windows.length; i++) {
	if(windows[i].getClass().getName().equals("de.labAlive.core.window.prope
rty.propertyWindow.PropertyWindow")) {
		windows[i].setVisible(false);
		PropertyWindow pw = (PropertyWindow) windows[i];
		pw.closeWindow();
		}
	}
}
```

*Abb. 38: Funktion in JString, um das Parameterfenster zu schließen*

Da sich irgendwo in der Klasse für die Parameter die alten Werte befinden müssen, muss auch nur diese Klasse zurückgesetzt werden. Das Fenster für die Einstellungen ist vom Typen *PropertyWindow* und erbt von der Super-Klasse *Window*. Mit einem Vergleich der Klassennamen kann das zuständige Fenster ermittelt und dann geschlossen werden. Nach dem Schließen des Parameterfensters wird das Messgerät neu initialisiert und dann wieder geöffnet. Die geladenen Parametereinstellungen sind nun fest hinterlegt und der Graph wird dementsprechend angezeigt.

## 5.5 Übertragung auf andere Messgeräte

Es gibt mehrere Gründe dafür, dass die implementierte Klasse *JString* und damit ihre Funktion der Serialisierung und Deserialisierung auf jeglichen anderen Messgeräten übertragbar ist. Gemeint sind hierbei Messgeräte, die wie *ProbabilityDensity* eine Anzeige in einem Koordinatensystem haben.

Zum einen ist die baugleiche Implementierung aller Messgeräte entscheidend für die Wiederverwendbarkeit der Serialisierung. Die Klasse *JString* ist hauptsächlich an dem Messgerät zur Messung der Wahrscheinlichkeitsdichte entwickelt und getestet worden. Das Oszilloskop, als auch der Spektrumanalysator sind ziemlich identisch aufgebaut. Die wichtigen Klasse für die Parameter erben von den gleichen Super-Klassen und bei den übrigen verwendeten Klassen verhält es sich ähnlich.

Der Reiter für die Serialisierung, mit den drei Optionen „running“, „save“ und „load“, ist in der Klasse *XYMeterParameters* deklariert. Da alle Messgeräte mit einer grafischen Anzeige von dieser Klasse erben, stellen auch alle diese Option zum Speichern zur Verfügung. Implementiert ist der Reiter in dem Package *Serialization*, in dem auch die weiteren Klassen für die Serialisierung hinterlegt sind. Die Klasse *JString* ist für die Logik die den ganzen Teil der Serialisierung und Deserialisierung besitzt.

Lediglich das Aufrufen der Funktionen zum Ausführen der Serialisierung findet mit einer Klasse, die für das Messgerät spezifiziert ist. In Der Funktion *meter()*, die ständig aufgerufen wird, um den aktuellen Signalwert weiterzugeben, ist eine If-Else Struktur implementiert mit der die aktuelle Auswahl des Serialisierung Reiter im Parameterfenster abgefragt wird.

```
if(pdp.getSerialization().getValue() == Serialization.SAVE) {
	super.saveParameter(); //<- Aufruf einer Funktion aus Super-Klasse
}else if(pdp.getSerialization().getValue() == Serialization.LOAD) {
	super.loadParameter(); //<- Aufruf einer Funktion aus Super-Klasse
}
```

*Abb. 39: If-Else Struktur zum Ausführen entsprechender Funktion aus Super-Klasse*

Die Instanz ruft die Methoden *saveParameter()* und *loadParameter()* aus der Klasse *XYMeter*.

```
public final void saveParameter() {

       JString.saveParams((XYMeterParameters) getParams());
        ((XYMeterParameters)
       getParams()).getSerialization().setValue(Serialization.RUNNING);
       JString.closePropertyWindows(getMeterWindow().getWindows(),
       (XYMeterParameters) getParams());
}

public synchronized final void loadParameter() {

       JString.loadParams((XYMeterParameters) getParams());
        ((XYMeterParameters)
       getParams()).getSerialization().setValue(Serialization.RUNNING);
       JString.closePropertyWindows(getMeterWindow().getWindows(),
       (XYMeterParameters) getParams());
       System.out.println("Parameters loaded!");
}
```

*Abb. 40: Funktion, die die Serialisierung und Deserialisierung ausführt*

Die beiden Methoden sind relativ synchron aufgebaut. Als erstes wird die Methode aus *JString* aufgerufen, der die aktuellen Parameter des aufrufenden Messgerätes mit übergeben werden. Das sind beim Speichervorgang die Parameter, die dann serialisiert werden. Beim Ladevorgang dienen diese Parameter zur Sicherstellung, dass die richtigen Parameter für das richtige Messgerät aufgerufen werden. In jeder Parameterliste, steht an erster Stelle der Messgerätetyp und kann so eindeutig identifiziert und verglichen werden. Im nächsten Schritt wird der Wert des Reiters wieder auf „running“ gestellt, um wieder in den normalen Messmodus zu wechseln. Beim letzten Schritt wird das Fenster für die Parametereinstellungen geschlossen. Das ist deswegen wichtig, um die Parameterwerte, die im Hintergrund geladen und übernommen wurden, auch im Einstellungsfenster für die Parameter zu aktualisieren. Dies geschieht am einfachsten, wenn es geschlossen wird, so dass beim erneuten Öffnen die neuen Werte übernommen werden.

## 5.6 Flexibles Kopieren und Einfügen der Parameter

Bei einer Schaltung, in der mehrere Verbindungen bestehen und diese alle gleichzeitig mit dem einem Messgerät beobachtet werden wollen, ist es etwas umständlich für jedes geöffnete Fenster die gespeicherten Parameter neu zu laden. Daher ist es von Vorteil einen Mechanismus zu integrieren, mit dem dies beschleunigt werden kann. Hier bietet sich ein Prinzip an, wie man es aus Windows kennt, das so genannte „copy & paste“. Mithilfe einer Tastenkombination werden die eingestellten Parameterwerte in der Applikation gespeichert und über eine weitere Tastenkombination in das gewünschte Messgerätefenster eingepflegt. Dies erspart den umständlichen Weg zum Laden eines externen Json-Strings und geht deutlich schneller. Für die Lösung dieses Weges, sind die gleichen Funktionen herangezogen, wie sie auch in den vorherigen Kapiteln zur Serialisierung und Deserialisierung beschrieben sind. Wichtig hierbei ist das korrekte Fenster ansprechen zu können und die Tastenkombination korrekt zu erfassen.

Für die Verarbeitung der Tastenkombination gibt es eine bereits gefertigte Klasse in labAlive, *WindowKeyEventListener*. Diese Klasse erbt von *KeyEventListener* und sorgt dafür, dass bei jedem Tastenklick in ein Fenster diese Klasse mit ihrer Methode *processKeyEvent()* aufgerufen wird. In einer Switch-Case Struktur wird anhand des übergebenen Events, in welchem sich eine Zahl für jede Taste befindet, herausgelesen um welche gedrückte Taste es sich handelt. Eigentlich gäbe es auch eine Tastenzahl für die Tastenkombination „copy“(STRG+C) und „paste“(STRG+V). Diese Tastenkombinationen werden vom System allerdings nicht erkannt und deswegen reagiert die Applikation in labAlive stattdessen auf die Tasten „C“ für kopieren und „V“ für einfügen.

```
case KeyEvent.VK_C:
	if(window.isFocused()	&&	window.getClass().toString().equals(new
	String("class de.labAlive.measure.xyMeter.plot.XYMeterWindow"))) {
		XYMeterWindow xywindow = (XYMeterWindow) this.window;
		xywindow.copyParameter();
	}
	break;
case KeyEvent.VK_V:
	if(window.isFocused()	&&	window.getClass().toString().equals("class
	de.labAlive.measure.xyMeter.plot.XYMeterWindow")) {
		XYMeterWindow xywindow = (XYMeterWindow) this.window;
	xywindow.pasteParameter();
}
break;
```

*Abb. 41: Switch-Case Struktur für die Tasten „C“ und „V“*

In der If-Abfrage wird überprüft ob das Fenster, welches das Event für den *KeyEventListener* ausgelöst hat, sich im Vordergrund befindet und ob es sich um ein XY-Fenster handelt. Dadurch wird sichergestellt, dass die Anwendung zum Kopieren und Einfügen nur bei Fenstern funktioniert, die zu einem Messgerät gehören. Danach wird das *window* temporär zu einem *XYMeterWindow* gecastet, um die entsprechenden Funktionen aus dieser Klasse aufrufen zu können.

```
public void copyParameter() {
	jstring.copyPara(getParams());
}

public void pasteParameter() {
	jstring.pastePara(getParams());
	JString.closePropertyWindows(
	(java.awt.Window[])plotter.window.getWindows());
}
```

*Abb. 42: Aufgerufene Methoden in der Klasse XYMeterWindow*

Die Variable *jstring* ist global und ist in der Klasse *Config* initialisiert. Dadurch wird gewährleistet, dass sie in der Applikation nur einmal instanziiert wird. Jedes Messgerätefenster greift also durch die Tasten „C“ und „V“ immer auf das gleiche Objekt vom Typ JString zu. Die Klasse JString wurde durch eine statische Variable vom Typen String erweitert, der die Parameter im Json-Format beinhaltet. Es werden bei der Serialisierung der Parameter und dem Extrahieren des entsprechenden Strings die gleichen Methoden wiederverwendet, wie sie auch beim Speichern und Laden schon vorhanden sind.

Wird die Taste „C“ gedrückt, wenn ein *XYMeterWindow* fokussiert ist, ruft diese die Funktion *copyPara()* auf, in der die Parameter in einen Json-String serialisiert werden und dann in der statischen Variable *info* hinterlegt.

```
public static void copyPara(XYMeterParameters parameter) {
	resetString();
	info = creatJsonString(parameter);
}
```

*Abb. 43: Funktion zum Kopieren der Parameter mit Aufruf bereits existierender Methoden*

Genau gleich verhält es sich bei der Funktion *PastePara()*. Hier wird mit bereits vorhandenen Methoden der Json-String ausgelesen und dann in die Parameterliste des aktuellen Messgerätes eingepflegt.

# 6. Evaluierung

## 6.1 Fazit

In dieser Masterarbeit sind zwei Hauptthemen behandelt worden. Als erstes das Messgerät zur Messung der Wahrscheinlichkeitsdichte. Im zweiten Schritt ist die Serialisierung und Deserialisierung von Parametern und deren flexibles kopieren und einfügen mit „Copy & Paste“ in die laufende Applikation, implementiert.

Der Programmierung in der Simulationsumgebung geht eine große Zeit an Vorbereitung voraus. Das Projekt labAlive ist mittlerweile sehr umfangreich, was es am Anfang schwer gemacht hat sich darin zurechtfinden. Die vielen Verlinkungen, das Java Modul Applet und die teilweise vielen Vererbungen über mehrere Ebenen sind am Anfang kompliziert zu durchschauen. Hat man sich aber erstmal darin zurechtgefunden ist die Implementierung eines Messgerätes ein geeigneter Einstieg, um die Simulationsumgebung zu verstehen. Bei der Programmierung von *ProbabilityDensity* war es sehr hilfreich sich an den anderen Messgeräten wie dem Spektrum Analystaor oder dem Oszilloskop zu orientieren. Des Weiteren war das Skript zur allgemeinen Entwicklung eines Messgerätes eine anfängliche gute Unterstützung. Schwierig wurde es dabei beispielsweise die X-Achse zu implementieren. Da es vorher nicht bekannt ist, dass schon eine solche Struktur besteht um die X-Achse mittig auszurichten, hat der erste Versuch zur selbständigen Implementierung Zeit und Aufwand in Kauf genommen. Ein weiteres größeres Problem war die Rundungsarithmetik in Java. Hier ist es oft nötig, mehrere Umwege zu gehen und mehrmals zu casten. Es war hier anspruchsvoll am Ende zu gewährleisten, dass immer die richtige Stelle und die korrekte Auflösung berechnet wird.

Für den Fortschritt des Messgerätes war es von Vorteil zunächst das Gerüst zu bauen, dass heißt sicher zu stellen, dass die Applikation läuft, ein Messgerätefenster angezeigt wird und das verändern der Parameter Wirkung zeigt. Danach konnte der ganze Fokus auf die eigentliche Logik der Applikation gelegt werden. Dadurch, dass im Messgerätefenster der errechnete Graph direkt angezeigt wird, hat eine direkte Kontrolle des Ergebnisses.

Im Gegensatz zur Implementierung eines neuen Messgerätes, wofür schon Baupläne und Beispiele bestehen, ging es im zweiten Schritt um eine neue Komponente, die es in labAlive noch nicht gibt. Hier war es sehr hilfreich, dass durch Bewältigung des ersten Schrittes ein Grundverständnis der Simulationsumgebung vorhanden war. So wusste man, an welchen Stellen im Code man anknüpfen kann.

Der Start für die Serialisierung von Parametern war stockend. Zuerst ist versucht worden mit Mechanismen und Bibliotheken, die von Java zur Verfügung stehen, vor zu gehen. Im ersten Schritt wurde versucht mittels der Schnittstelle „Serializable“ von Java einzelne Klassen direkt serialisieren zu können. Allerdings hatte das zur Folge, dass alle dazugehörigen und verlinkten Klassen auch mit dem Interface verlinkt werden müssen. Da aber labAlive ein sehr mächtiges Projekt ist, war dieser Weg wenig aussichtsreich. Der nächste Schritt bestand darin, mittels einer externen Bibliothek, den Json-String zu erstellen. Diese Bibliothek war im Internet frei verfügbar, besaß allerdings nur die Schnittstellen der einzelnen Funktionen, aber nicht deren Inhalt. Der Fokus wurde daher auf das eigene Implementieren der Serialisierung und Deserialisierung gesetzt und von einer Nutzung bestehender Bibliotheken abgesehen. Ab da war es ein angenehmes Programmieren mit einem stetigen Fortschritt. Es war sehr interessant sich mit einer Aufgabe, zu beschäftigen, von der man noch nicht genau wusste wie das Ergebnis aussehen wird und ob es funktioniert. Bei der Auswahl des Speicherortes und dem wieder Aufrufen einer gespeicherten Datei, ist der JFileChooser[11] von Java eine hilfreiche Entdeckung. Er ist einfach zu implementieren und funktioniert auf Anhieb fehlerfrei. Der Reiter im Parameterfenster war aufgrund vieler Beispiele einfach zu programmieren und auch für die Entwicklung sehr angenehm, stellt allerdings keine schöne Lösung für die Zukunft dar. Durch das Einfügen der wenigen Funktionen in die entsprechenden Superklassen der Messgeräte, konnten die Mechanismen zum Speichern und Laden auf alle Messgeräte übertragen werden. Im Nachhinein war es hilfreich die Serialisierung zunächst an einem Messgerät zu etablieren und dann auf die Anderen zu übertragen. Ähnlich verhielt es sich bei der Programmierung von Copy & Paste für die Parameter. Hierbei war es wichtig die richtige Stelle im Projekt zu finden, wo angesetzt werden kann. Die Schwierigkeit hierbei bestand darin, den Listener[1] für die Tasten richtig zu handhaben. Da es in einem solch großen Projekt Listener[12] für die Menüs und die Messgeräte gibt, war es zunächst schwer mit den unterschiedlichen Hierarchien umzugehen. Einen weiteren Listener nur für den Copy & Paste Mechanismus zu integrieren stellte sich als zu umständlich in der Umsetzung heraus. Bei der weiteren Entwicklung konnten fast alle Funktionen, die aus der Serialisierung und Deserialisierung bereits bestanden, wiederverwendet werden.

[11] JFileChooser öffnet ein Dialogfenster zum Speichern und Laden.

[12] Die Funktion eines Listener wird ausgeführt, wenn ein bestimmtes Ereignis (in diesem Beispiel wird eine Taste gedrückt) stattfindet.

## 6.2 Erweiterungsmöglichkeiten

Aufgrund der limitierten Zeit, die für ein solches Projekt zur Verfügung steht, können nicht alle Details und Aspekte ausgeführt werden, die die Simulationsumgebung verbessern würden. Jedoch können theoretisch einige Gedanken angestellt werden, wie die Lösungen für solche Probleme aussehen könnten.

Im Moment werden bei dem Messgerät zur Messung der Wahrscheinlichkeitsdichte alle Signalwerte in ein Array geschrieben. Ist dieses Array voll, werden die Werte analysiert und dann wird das Array wieder neu befüllt. Das Problem hierbei ist, dass immer nur die Werte betrachtet werden, die aktuell im Array stehen, aber nicht die die bevor schonmal drinstanden und schon ausgewertet wurden. Das heißt das Ergebnis der Wahrscheinlichkeitsdichte ist immer nur eine Momentaufnahme, die keinen Bezug auf die Ergebnisse vor dieser Messung nimmt. Besser wäre es bei einer neuen Messung das alte Array mit Werten behalten und die beiden dann vor der Analyse zu einem größeren Array zusammenzufügen. So könnte gewährleistet werden, dass alle Werte betrachtet werden und am Ergebnis Einfluss haben. Bei einer Änderung der Parameter im Einstellungsfenster müsste das Array zurückgesetzt werden und wieder von vorne befüllt, zusammengefügt und analysiert werden. Des Weiteren hat das Messgerät noch kein eigenes Symbol, welches anzeigt, wenn sich die Messgerätepalette öffnet.

Zum Speichern und Laden der Parameter muss das Parameterfenster geöffnet und ausgeklappt werden, um den entsprechenden Reiter bedienen zu können. Die Bedienung der Serialisierung könnte etwas einfacher funktionieren. Besser wäre zu überlegen hier mit einer weiteren Tastenkombination zu arbeiten, die die externe Serialisierung und Deserialisierung aufruft. Eine weitere Möglichkeit ist die Integration eines Reiters im Menüfenster. Dort könnte man schnell auf ihn zugreifen.

Der Copy & Paste-Mechanismus kann sich bis jetzt nur einen Parameter für ein Messgerät merken und diesen einfügen. Durch eine Erweiterung der Klasse *JString* könnten sich Parameter verschiedener Messgeräte gleichzeitig gemerkt werden. Die Klasse *JString* besitzt eine statische Variable des Typen String, worin die serialisierten Parameter hinterlegt sind. Durch eine Erweiterung um drei Variablen vom Typen String, könnten die Parameter von allen Messgeräten gespeichert werden. Damit dies funktioniert müssen noch weitere Funktionen implementiert werden, die bei Copy und Paste erkennen welcher String zu welchem Messgerät gehört.

# 7. Literaturverzeichnis

[1] Rheinwerk Verlag, „Rheinwerk," Rheinwerk Verlag GmbH, 2012. [Online]. Available: http://openbook.rheinwerk-verlag.de/java7/1507_15_001.html. [Zugriff Juli 2018].

[2] Karl-Dirk Kammeyer, Nachrichtenübertragung, Springer Vieweg, 2017.

[3] C. Roppel, Grundlagen der digitalen Kommunikationstechnik Carl Hanser Verlag, 2006.

[4] A. Haase, „JAXenter," JAXenter, 7 März 2016. [Online]. Available: https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558. [Zugriff August 2018].

[5] E. i. Json, „https://www.json.org/json-de.html," www.json.org, [Online]. Available: https://www.json.org/json-de.html. [Zugriff August 2018].

[6] Prof. Dr. E. Riederer, „Simulation kommunikationstechnischer Prozesse," München, 2016.

[7] „JavaBeginners," 31 Juli 2017. [Online]. Available: https://javabeginners.de/Arrays_und_Verwandtes/Array_deklarieren.php. [Zugriff Juli 2018].

# 8. Abbildungsverzeichnis