Fakultät Elektrotechnik und Informationstechnik
Institut für Nachrichtentechnik
Lehrstuhl Theoretische Nachrichtentechnik

Diplomarbeit

MIMO-Erweiterung einer Plattform für Software Defined Radio

Vorgelegt von:	Paul Machemehl geb. am 14.11.1985 in Halle(Saale)
Studiengang:	Elektrotechnik
Studienrichtung:	Informationstechnik
Betreuer:	Dipl.-Ing. Axel Schmidt
Prüfer:	Prof. Dr.-Ing. Eduard A. Jorswieck Prof. Dr.Ing. habil. Adolf Finger

Ausgehändigt am:	02.06.2014
Einzureichen bis:	02.03.2015

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angemessenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten:

Schmidt, Axel

Weitere Personen waren an der geistigen Herstellung der vorliegenden Arbeit nicht beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Diplomabschlusses führen kann.

Dresden, den Unterschrift

Glossar

Datensymbol

Ein komplexer Wert, der zur Darstellung von Daten genutzt wird.

Datenton

Siehe *Datenträger*.

Datenträger

Eine Trägerfrequenz, die mit einem *Datensymbol* moduliert worden ist.

Delay-Spread

Ist die zeitliche Differenz zwischen dem Signal übertragen über den direkten Übertragungspfad und dem längsten indirekten Übertragungspfad.

GNU Radio

Ist ein Framework zur Signalanalyse und -verarbeitung mittels Signalflussgraphen geschrieben in C++ und Python.

OFDM-Frame

Eine Aneinanderreihung von zwei *Synchronisationssymbole*n und mindestens einem *OFDM-Symbol*.

OFDM-Symbol

Eine bestimmte Aneinanderreihung von *Datenträger* und *Pilotträger*, die mittels der *IDFT*[1] aus dem Frequenzbereich in den Zeitbereich transformiert wurde.

Orthogonalität

Benennt das Phänomen, dass an der Stelle eines Trägers sich bis auf ihn selbst alle anderen Träger zu 0 aufsummieren..

[1] Inverse Discret Fourier Transform

Pilotsymbole

Ein komplexer Wert, der zur Phasen- und Kanalkorrektur verwendet wird.

Pilotton

Siehe *Pilotträger*.

Pilotträger

Eine Trägerfrequenz, die mit einem *Pilotsymbole* moduliert worden ist.

Schmidl-Cox-Korrelator

Bezeichnet eine Methode ein OFDM-Signal aufgrund eines vorangestellten Synchronisationssymbols zu erkennen..

Synchronisationssymbol

Ist eine Aneinanderreihung von komplexen Werten im Zeitbereich, die zur Erkennung und Synchronisation nach einem bestimmten Synchronisationsschema erstellt worden ist und dementsprechend einen besonderen Aufbau besitzt.

Ton

Siehe *Träger*.

Träger

Eine Frequenz in einem bestimmten Frequenzbereich, die mit einem komplexen Wert nach einer vorgegebenen Modulationsart moduliert worden ist.

Akronyme

8-PSK	Es werden $2^3 = 8$ Bit mit einem Symbol dargestellt.
BPSK	Es werden $2^1 = 2$ Bit mit einem Symbol dargestellt.
CFO	carrier frequency offset.
CP	Ein Verfahren, bei dem einem Signal dessen hinterer Abschnitt beliebiger Länge zusätzlich vorangestellt wird.
DFT	discrete Fourier transform.
ICI	Hierbei handelt es sich um eine Störung verursacht durch andere Träger innerhalb eines OFDM-Symbols. In diesem Fall ist dir Orthogonalität der einzelnen Träger untereinander zerstört..
IDFT	inverse discrete Fourier transform.
IFFT	inverse fast Fourier transform.
ISI	Hierbei handelt es sich um eine Störung verursacht durch andere OFDM-Symbole bspw. durch einen Multi-Path-Channel. In diesem Fall ist dir Orthogonalität der einzelnen Träger untereinander zerstört..
OFDM	Orthogonal Frequency-Devision Multiplexing.
PAP	Eine grafische Veranschaulichung zur Struktur eines Programms.
PAPR	Ist das Verhältnis von maximaler Leistung zur mittleren Leistung eines Signals.

PSK	Eine Modulationstechnik-bei die Phase der alleinige Informationsträger ist.
QAM	Quadrature amplitude modulation.
QPSK	Es werden $2^2 = 4$ Bit mit einem Symbol dargestellt.
SNR	Gibt das Verhältnis der mittleren Leistung des Nutzsignals zur mittleren Leistung des Rauschleistung des Störsignals an.
STO	Ist der Zeitoffset, der ein zu frühes oder zu spätes Erkennen eines OFDM-Symbols beschreibt.

Symbolverzeichnis

T_s	Ist die Zeitspanne eines einzelnen OFDM-Symbols ohne Cyclic Präfix. 11, 15, 18, 19, 24, 25
T_{CP}	Ist die Zeitspanne eines Cyclic Präfix. 11, 24, 25
T_{s+CP}	Ist die Zeitspanne von T_s und T_{CP} zusammen. 24
$\Delta\varphi$	Ist die Phasenverschiebung zwischen Sender und Empfänger. 21
ε	Ist die auf die eigentliche Sendefrequenz normierte Frequenzabweichung. 22

Inhaltsverzeichnis

1 Theoretische Grundlagen **15**

1.1 Orthogonal Frequency-Division Multiplexing 15
1.1.1 Übersicht 15
1.1.2 Grundlegende Funktionsweise 16
1.2 Störungen 22
1.2.1 Phasenverschiebung 22
1.2.2 Frequenzoffset 22
1.2.3 Kanal-Parameter 23
1.2.4 Rauschen 23
1.2.5 Mehrwegausbreitung 23
1.3 Schmidl-Cox-Korrelator 24
1.4 Cyclic Prefix 25

2 Präzisierung und Interpretation der Aufgabenstellung **29**

3 Entwurf und Implementierung **31**

3.1 Quelltexte anderer Arbeiten als Basis 31
3.1.1 `gr-jwdiplom` 31
3.1.2 OFDM-Beispielquelltext aus dem Modul `gr-digital` 31
3.1.3 Der Block `ofdm_txrx.py` 32
3.2 Das GNU Radio-Modul `gr-mimoots` 33
3.2.1 Blöcke 37
3.2.2 Das Modul `utils` 43
3.2.3 Programme 44
3.2.4 Generierung eines OFDM-Frames 47
3.3 Implementierung, Probleme und ihr Einfluss auf Design-Entscheidungen . 52
3.3.1 Generelle Prinzipien 52
3.3.2 Skalierung der Amplituden für das Device USRP2 N210 54

3.3.3 Samples aus dem Buffer des USRP2 N210 beim Aktivieren als Empfänger . . . 54
3.3.4 Häufiges Ignorieren des ersten OFDM-Frames durch den *Schmidl-Cox-Korrelator* . . . 55
3.3.5 Erhöhung der Fehlerrate bei der Nutzung von 8-PSK . . . 56
3.3.6 Das Design vom Block `ofdm_extract_frame_vcvc` . . . 56
3.3.7 `ofdm_extract_frame_vcvc` und der Thread-Per-Block Scheduler . 61
3.3.8 Notwendigkeit einer Pause, damit der Signalflussgraph abgearbeitet werden kann . . . 62

4 Nutzungs-Szenarien 65
4.1 Messaufbau und Umgebung . . . 65
4.2 SISO-Datenübertragung . . . 65
4.3 Gleichzeitige SIMO-Datenübetragung zur Messung der Diversität . . . 66
4.4 Zeitversetzte MIMO-Datenübertragung zur Messung der Diversität . . . 67

5 Bewertung und Interpretation der Messergebnisse 69
5.1 Definition Korrelationskoeffizient . . . 69
5.2 Aussagekraft . . . 70
5.3 Messergebnisse . . . 72
5.3.1 Gleichzeitige MISO-Datenübetragung zur Messung der Diversität . 72
5.3.2 Zeitversetzte MIMO-Datenübertragung zur Messung der Diversität 73

6 Zusammenfassung und Ausblick 75

Literatur 80

1 Theoretische Grundlagen

1.1 Orthogonal Frequency-Division Multiplexing

1.1.1 Übersicht

OFDM[1] lässt sich den linearen Mehrträger-Modulationsverfahren zuordnen, genauer noch den orthogonalen Mehrträger-Modulationsverfahren. Dabei werden aus einem Frequenzintervall bestimmte einzelne Frequenzen (die Trägerfrequenzen oder auch Töne genannt) genommen und unabhängig voneinander moduliert. Die Modulationsart kann für jeden *Träger* separat gewählt werden (bspw. *8-PSK*[2] oder 16-*QAM*[3]). *OFDM* schafft es *ICI*[4] zu vermeiden und damit ein Übersprechen zu verhindern, indem nur zueinander orthogonale Frequenzen als *Träger* gewählt werden, sodass sich diese nicht gegenseitig stören. Durch die *Orthogonalität* kann *OFDM* als eine Gruppe von mehreren unabhängigen Übertragungssystemen angesehen werden.

Aufgrund der *Orthogonalität* können zum einen aufwendige Realisierungen von Bandpassfilter für jeden *Träger* vermieden, zum anderen kann das Signal auf elegante Weise durch die *IDFT* generiert und auf Empfängerseite mittels der *DFT*[5] wieder in seine einzelnen *Träger* zerlegt werden. Vor allem die Nutzung von *IDFT* und *DFT* erlaubt eine effektive Implementierung von *OFDM*.

OFDM bietet vor allem in Szenarien mit einem durch Mehrwegausbreitung und Mehrfachreflexionen verursachten frequenzselektiven Übertragungskanal große Vorteile und findet aus diesem Grunde in modernen Funkübertragungssystemen Verwendung.

Durch die freie Wahl verschiedener *Träger* und die Art und Weise, wie diese genutzt werden, ergibt sich eine hohe Flexibilität, da je nach Bedingungen einzelne *Träger* abgeschaltet oder deren Modulationsverfahren geändert werden können. Dies erlaubt ein sehr dynamisches Reagieren auf äußere Einflüsse.

[1] Orthogonal Frequency-Devision Multiplexing
[2] 8 Phase Shift Keying
[3] Quadrature Amplitude Modulation
[4] Inter Carrier Interference
[5] Discret Fourier Transform

Vor allem aber der hohe Informationsgehalt durch die parallele Nutzung mehrerer *Träger* macht *OFDM* für diese und spätere darauf aufbauende Arbeiten äußerst nützlich.

1.1.2 Grundlegende Funktionsweise

1.1.2.1 Signalgenerierung im Basisband

Mehrträgermodulationssysteme basieren auf dem Prinzip, dass ein Satz Daten (komplexe Symbole) auf mehrere *Träger* verteilt werden. Diese *Träger* werden dann aufaddiert und mittels einer Antenne in die Umgebung abgestrahlt. Die Antenne des Empfängers kann diese wieder empfangen. Durch die Eigenschaft des Raums, dass komplexe Schwingungen linear überlagert werden können, ist es möglich diese Schwingungen nach dem Empfang wieder zu zerlegen. Das bedeutet, dass als *Träger* komplexe Schwingungen $\phi_i = e^{2\pi f t}$ moduliert durch komplexe Symbole $\{X_k\}_{k=0}^{N-1}$ zur Datenübertragung geeignet sind. Das Schwierige an diesem Prinzip besteht in der Art und Weise wie jeder *Träger* spektral geformt werden muss, damit diese sich nicht gegenseitig stören. *OFDM* greift dabei auf ein einfaches Prinzip zurück: Orthogonalität. Dies soll in diesem Kapitel erklärt werden.

Mathematisch lässt sich das Prinzip der Mehrfachträgermodulation folgendermaßen formulieren:

$$x(t) = \sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi f_k t} \qquad 0 \leq t \leq T_s \tag{1.1}$$

$x(t)$ stellt das komplexe Signal im Basisband dar und $X[k]$ das k-te komplexe Symbol aus der Folge $\{X_k\}_{k=0}^{N-1}$. Jedes dieser Symbole wird also einem *Träger* $\phi_k = e^{\mathrm{j}2\pi f_k t}$ zugeordnet. Zum Schluss wird alles aufaddiert. N stellt die Anzahl der Unterträger dar. Zugleich wird der Zeitabschnitt eines *OFDM*-Symbols $0 \leq t \leq T_s$ in N Samples zerlegt.

Da in dieser Arbeit die Signalgenerierung mithilfe von *GNU Radio* erfolgen soll, muss dieses Prinzip aus dem kontinuierlichen Bereich noch in den diskreten Bereich übertragen werden. Dazu werden folgende Substitutionen durchgeführt:

$$t = n\frac{T_s}{N} \qquad n = 0, 1, \ldots, N-1 \tag{1.2a}$$

$$f_k = k\Delta f = k\frac{1}{T_s} \qquad k = 0, 1, \ldots, N-1 \tag{1.2b}$$

Als Resultat ergibt sich folgender Ausdruck:

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi \frac{kn}{N}} \tag{1.3}$$

Dieser Ausdruck erinnert bereits stark an die Formel der *IDFT*:

$$\mathrm{IDFT}(\{X_k\})[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi \frac{kn}{N}} \tag{1.4}$$

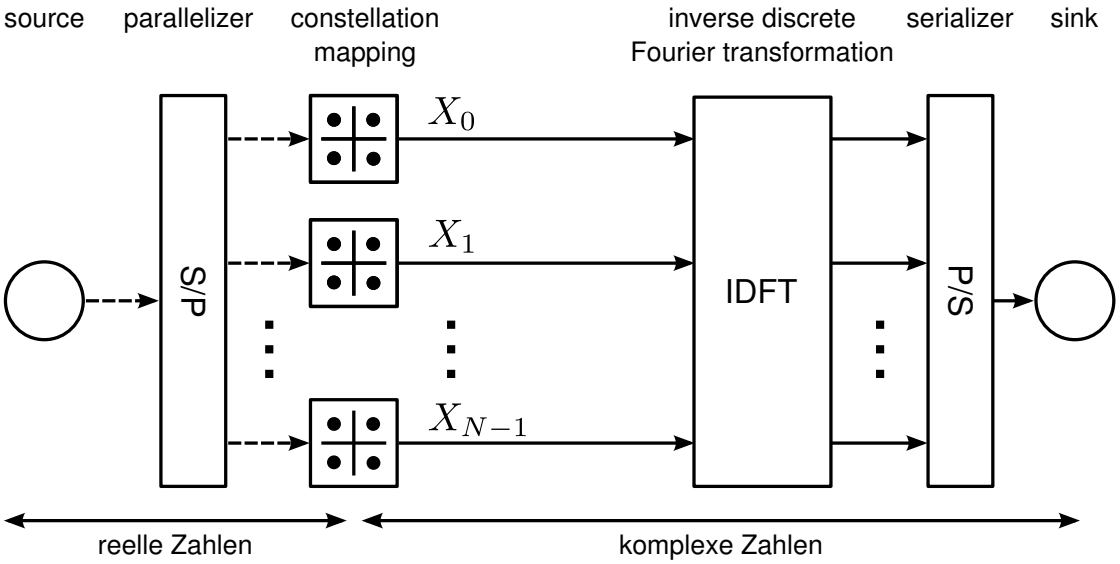

Abbildung 1.1: Generierung eines Signals mittels *OFDM*

Der einzige Unterschied zwischen Formel (1.4) und (1.3) besteht in dem Faktor $\frac{1}{N}$. Damit lässt sich das Signal $x[n]$ mithilfe der *IDFT* folgendermaßen ausdrücken:

$$x[n] = N \cdot \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi \frac{kn}{N}} \tag{1.5a}$$

$$x[n] = N \cdot \mathrm{IDFT}(\{X_k\}_{k=0}^{N-1})[n] \tag{1.5b}$$

Dieser Zusammenhang bringt einen großen Vorteil in der Implementierung von *OFDM*: Die *IDFT* kann effektiv durch die *IFFT*[6] implementiert werden und bringt für den Fall $N = 2^a$ einen enormen Geschwindigkeitsvorteil.

In Abbildung 1.1 wird die grobe Funktionsweise von *OFDM* dargestellt. Die Quelle (*source*) generiert eine Folge von verschiedenen Bits. Diese Folge wird durch einen *Seriell-Parallel-Wandler* (*parallelizer*) in N verschiedene Bit-Vektoren aufgeteilt, deren Länge jeweils den gewählten Modulationsverfahren entspricht. Alle N Bit-Vektoren werden nun durch das *Constellation-Mapping* parallel auf komplexe Symbole abgebildet und dann als komplexer Vektor bestehend aus den komplexen Symbolen $X_0, X_1, ..., X_{N-1}$ der *IDFT* zugeführt. Als Ergebnis erhalten wir das zeitdiskrete *OFDM*-Basisbandsignal aus Gleichung (1.3) in Form eines Vektors bestehend aus den Werten $x_0, x_1, \ldots, x_{N-1}$. Dieser Vektor muss nun noch durch den *serializer* in eine Folge von komplexen Werten $\{x_k\}_{k=0}^{N-1}$ umgewandelt werden.

Es ist wichtig zu verstehen, dass dem letzten Schritt aus der Sicht der Programmierung eine große Bedeutung zukommt. *GNU Radio* macht einen großen Unterschied zwischen

[6] Inverse Fast Fourier Transform

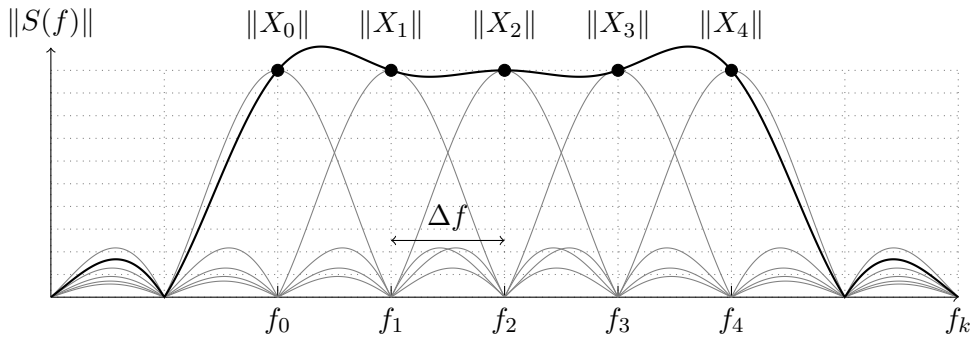

Abbildung 1.2: Betragsspektrum $\|S(f)\|$ der Fourier-Transformierten des Signals $x(t)$

einer Folge und einem Vektor von komplexen Werten. Ersteres ist eine Aneinanderreihung von skalaren Werten, die Schritt für Schritt und damit Werte für Wert an einen Block übergeben und verarbeitet werden. Letzteres hingegen ist ein Bündel von Werten, das mit einem mal an einen *GNU Radio*-Block übergeben und damit zusammenhängend verarbeitet wird. In sämtlichen Zusammenhängen soll aus diesem Grund von einer Folge gesprochen werden, wenn es sich um von *GNU Radio* Schritt für Schritt verarbeite Werte handelt und von einem Vektor, wenn dessen Elemente allesamt in einem Schritt verarbeitet werden können.

1.1.2.2 Amplitudenspektrum des *OFDM*-Basisbandsignals

Abbildung 1.2 stellt das Betragsspektrum im Frequenzbereich der zeitkontinuierlichen Funktion aus Gleichung (1.1) dar. Das Spektrum lässt sich mit Hilfe der kontinuierlichen Fouriertransformation berechnen:

$$\mathcal{F}(x)(t) = \int_{-\infty}^{+\infty} x(t) e^{-\mathrm{j}2\pi f t} \mathrm{d}t \tag{1.6}$$

Daraus folgt:

$$S(f) = \mathcal{F}(x)(t) \tag{1.7a}$$
$$= \int_{-\infty}^{+\infty} \left(\sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi f_k t} \right) e^{-\mathrm{j}2\pi f t} \mathrm{d}t \qquad 0 \le t \le T_s \tag{1.7b}$$
$$= \int_{-\infty}^{+\infty} \mathrm{rect}(\frac{t}{T_s} - \frac{1}{2}) \left(\sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi f_k t} \right) e^{-\mathrm{j}2\pi f t} \mathrm{d}t \tag{1.7c}$$
$$= \int_{0}^{T_s} \left(\sum_{k=0}^{N-1} X[k] e^{\mathrm{j}2\pi f_k t} \right) e^{-\mathrm{j}2\pi f t} \mathrm{d}t \tag{1.7d}$$
$$= \sum_{k=0}^{N-1} X[k] \int_{0}^{T_s} e^{\mathrm{j}2\pi f_k t} e^{-\mathrm{j}2\pi f t} \mathrm{d}t \tag{1.7e}$$
$$= \sum_{k=0}^{N-1} X[k] \int_{0}^{T_s} e^{\mathrm{j}2\pi (f_k - f) t} \mathrm{d}t \tag{1.7f}$$
$$= \sum_{k=0}^{N-1} X[k] e^{-\mathrm{j}2\pi (f - f_k) T_s} \mathrm{sinc}(\pi (f - f_k) T_s) \tag{1.7g}$$

Das Betragsspektrum besteht also aus einer Summe von sinc-Funktionen. Dies liegt daran, dass die Fouriertransformation in diesem Fall nicht im Intervall $(-\infty, +\infty)$ berechnet sondern auf das Intervall $(0, T_s)$ beschränkt wird. Dies ist gleichzusetzen mit einer harten Umtastung, da nach T_s das erste OFDM-Symbol übertragen wurde und darauf das nächste OFDM-Symbol folgt. Dies bewirkt ein Verschmieren der eigentlich zu erwartenden Dirac-Impulse bei den Frequenzen f_k. Daher finden wir an deren Stelle sinc-Funktionen vor, die einen Teil ihrer Energie in die benachbarten Frequenzbereiche abgeben. Das Besondere von OFDM liegt nun darin, dass die Träger-Frequenzen so gewählt worden sind, dass sie immer auf die Nullstellen der anderen sinc-Funktionen fallen (siehe (1.2b)) und sich damit nicht gegenseitig stören.

Für die Nullstellen gilt:

$$\mathrm{sinc}(x) = 0 \qquad x \in \{n\pi | n \in \{\pm 1, \pm 2, \dots\}\} \tag{1.8}$$

Damit lässt sich eine Bedingung für die zu wählenden Frequenzen ableiten:

$$\pi (f - f_k) T_s = n\pi \qquad f - f_k = \delta f \tag{1.9a}$$
$$f - f_k = \frac{n}{T_s} \tag{1.9b}$$
$$\Delta f = \frac{n}{T_s} \qquad n = \pm 1, \pm 2, \dots \tag{1.9c}$$

Die Differenz zwischen einer Trägerfrequenz f_k und der, an deren Stelle ein weiterer *Träger* liegen soll, muss ein Vielfaches der Frequenz $\Delta f = \frac{1}{T_s}$ betragen. Nur in diesem Fall treten keine *ICI*-Störungen auf.

1.1.2.3 Orthogonalität

Das Grundprinzip, auf welches das eben genannte Phänomen basiert, nennt sich *Orthogonalität* In Gleichung (1.7e) ist die Grundlage dieser mathematischen Eigenschaft zu finden[7]:

$$\frac{1}{T_s}\int_0^{T_s} e^{\mathrm{j}2\pi f_k t} e^{-\mathrm{j}2\pi f t}\mathrm{d}t = \frac{1}{T_s}\int_0^{T_s} e^{\mathrm{j}2\pi \frac{k}{T_s} t} e^{-\mathrm{j}2\pi \frac{i}{T_s} t}\mathrm{d}t \tag{1.10a}$$

$$= \frac{1}{T_s}\int_0^{T_s} e^{\mathrm{j}2\pi \frac{(k-i)}{T_s} t}\mathrm{d}t \tag{1.10b}$$

$$= \begin{cases} 1, & k = i \text{ oder } f_k = f_i \\ 0, & \text{sonst} \end{cases} \tag{1.10c}$$

Oder im diskreten Fall[8]:

$$\frac{1}{N}\sum_{n=0}^{N-1} e^{\mathrm{j}2\pi \frac{k}{T_s} T_s \frac{n}{N}} e^{-\mathrm{j}2\pi \frac{i}{T_s} T_s \frac{n}{N}} = \frac{1}{N}\sum_{n=0}^{N-1} e^{\mathrm{j}2\pi n \frac{(k-i)}{N}} \tag{1.11a}$$

$$= \frac{1}{N} e^{\mathrm{j}\pi(k-i)\frac{N-1}{N}} \cdot \frac{\sin(\pi(k-i))}{\sin(\pi\frac{(k-i)}{N})} \tag{1.11b}$$

$$= \begin{cases} 1, & k = i \\ 0, & \text{sonst} \end{cases} \tag{1.11c}$$

Bei Gleichung (1.11b) handelt es sich um den Dirichlet-Kern[9] für den Fall $n \geq 0$.

[7] Siehe [1, S. 121]
[8] Siehe [1, S. 121]
[9] Siehe [2, S. 171-173, 290-294]

1.1.2.4 Datenextraktion aus einem OFDM-Signal im Basisband

Um die einzelnen komplexen Symbole aus dem erzeugten Basisbandsignal wieder extrahieren zu können, soll folgende Gleichung näher betrachtet werden[10]:

$$Y[k] = \sum_{n=0}^{N-1} y[n] e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.12a}$$

$$= \sum_{n=0}^{N-1} \left\{ \frac{1}{N} \sum_{i=0}^{N-1} X[i] e^{-\mathrm{j}2\pi i \frac{n}{N}} \right\} e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.12b}$$

$$= \frac{1}{N} \sum_{n=0}^{N-1} \sum_{i=0}^{N-1} X[i] e^{-\mathrm{j}2\pi i \frac{n}{N}} e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.12c}$$

$$= \frac{1}{N} \sum_{n=0}^{N-1} \sum_{i=0}^{N-1} X[i] e^{-\mathrm{j}2\pi n \frac{i-k}{N}} \tag{1.12d}$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} \sum_{n=0}^{N-1} X[i] e^{-\mathrm{j}2\pi n \frac{i-k}{N}} \tag{1.12e}$$

$$= \sum_{i=0}^{N-1} X[i] \frac{1}{N} \sum_{n=0}^{N-1} e^{-\mathrm{j}2\pi n \frac{i-k}{N}} \tag{1.12f}$$

$$= X[k] \tag{1.12g}$$

$y[k]$ stellt die einzelnen Samples der empfangenen Zeitreihe $\{y_k\}_{k=0}^{N-1}$ dar (der Einfachheit halber wird an dieser Stelle angenommen, dass keinerlei Störung beim Übertragen eines OFDM-Symbols auftritt, d.h. es gilt $x[k] = y[k]$). Die *Orthogonalität* stellt sicher, dass bei freier Wahl von k auch dasjenige Symbol $X[i]$ herausgefiltert wird, wofür $i = k$ gilt, also $X[k]$.

Es stellt sich bei genauerer Betrachtung heraus, dass Gleichung (1.12a) mit der *DFT* übereinstimmt:

$$\mathrm{DFT}(\{y_n\}_{n=0}^{N-1})[k] = \sum_{n=0}^{N-1} y[n] e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.13}$$

Damit lässt sich folgende Gleichung formulieren:

$$X[k] = \mathrm{DFT}(\{y_n\}_{n=0}^{N-1})[k] \tag{1.14}$$

Abbildung 1.3 zeigt, wie die Symbole aus dem empfangenen OFDM-Signal wieder extrahiert werden können. Als erstes wird die empfangene Folge (*source*) einem *Seriell-Parallel-Wandler* übergeben der diese dann in einen Vektor mit N Elementen umwandelt und der *DFT* übergibt. Das Ergebnis ist wiederum ein Vektor aus komplexen Werten. Jeder dieser Werte wird einem Entscheider (*Estimator*) übergeben. Dieser legt nach einem bestimmten Verfahren fest, welchem Symbol aus dem vorher vereinbarten Symbolvorrat $X_0, X_1, \ldots, X_{N-1}$ dieser Wert am ehesten entspricht, da die empfangenen Werte durch Störungen verfälscht sein können. Daraus ergeben sich komplexe Schätzwerte $\{\hat{X}_k\}_{k=0}^{N-1}$, die im besten Fall mit den Werten $\{X_k\}_{k=0}^{N-1}$ übereinstimmen.

[10] Siehe [1, S. 124,125]

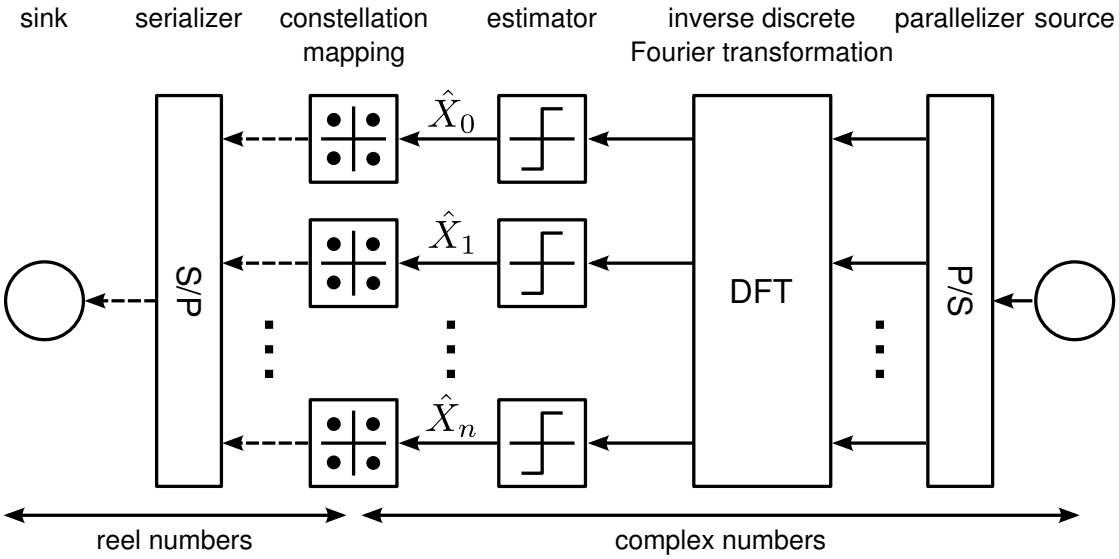

Abbildung 1.3: Extraktion der Daten aus einem *OFDM*-Signal im Basisbandbereich

1.2 Störungen

1.2.1 Phasenverschiebung

Über Sender und Empfänger hinweg kann es dazu kommen, dass eine Phasenverschiebung $\Delta\varphi$ zwischen dem gesendeten und dem empfangenen Signal auftritt. Diese ist von der betrachteten Trägerfrequenz f_k in Form eines linearen Zusammenhangs abhängig.

Zum einen kann diese Phasenverschiebung durch den im Aufbau von Sender und Empfänger liegenden geometrischen Zusammenhang begründet sein, zum anderen durch ein Verschieben des Abtastzeitpunkts eines *OFDM-Symbols* in den *CP*[11] hinein verursacht werden. In beiden Fällen muss dieser Phasenunterschied zur Rekonstruktion der Daten herausgerechnet werden.

1.2.2 Frequenzoffset

Sende und Empfänger haben beide einen Oszillator eingebaut um die Frequenz zum Senden und Empfangen der Signal zu erzeugen. Nun ist jedem Bauteil zu Eigen, dass es von vorgegebenen Größen in einem gewissen Umfang abweicht, da viele Herstellungsprozesse nur in bestimmten Grenzen unter Kontrolle gehalten werden können. So ist es auch bei den Oszillatoren. Dies hat zur Folge, dass der Empfänger mit einer vom Sender abweichenden Frequenz das Signal empfängt.

Abbildung 1.4 veranschaulicht den Fall, dass das Signal bei einer anderen Frequenz empfangen wird als es gesendet wurde. Dies hat zur Folge, dass die Symbole an der

[11] Cyclic Prefix
[12] Carrier Frequency Offset

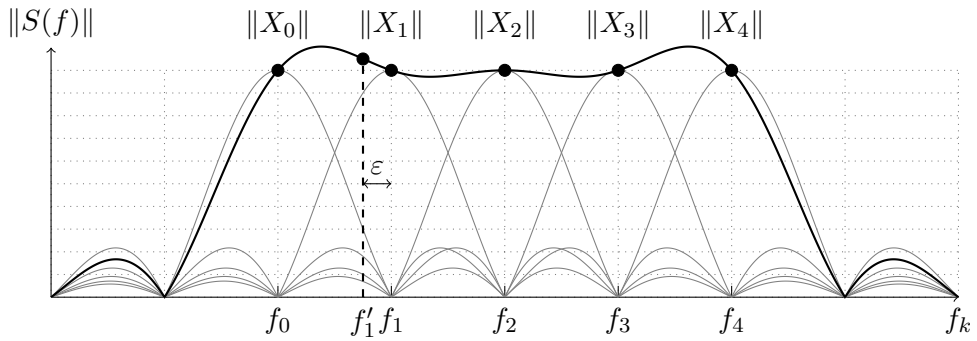

Abbildung 1.4: Auswirkung eines *CFO*[12]

falschen Stelle im Frequenzspektrum abgegriffen werden (in diesem Fall bei Frequenz f_1' statt bei Frequenz f_1). Somit ist die *Orthogonalität* nicht mehr gewährleistet und es kommt zum *ICI*. Dieses Phänomen wird *CFO* genannt.

Es lässt sich folgendes formulieren:

$$f_{offset} = f_c - f_c' \tag{1.15}$$

Hierbei ist f_c die eigentliche unverfälschte und f_c' die durch den Frequenzunterschied verfälschte Frequenz.

Damit lässt sich der normierte *CFO* als ε definieren[13]:

$$\varepsilon = \frac{f_{offset}}{\Delta f} \tag{1.16}$$

1.2.3 Kanal-Parameter

Durch den Kanal wird das Signal sozusagen gestört: Zum einen in seiner Amplitude und zum anderen in seiner Phase. Dies bewirkt auch ein Verfälschen der gesendeten Datensymbole. Um dies rückgängig machen zu können, muss dieser Einfluss beim Empfänger heraus gerechnet werden. Dazu müssen die Kanalparameter, welche die Störung verursachen, geschätzt werden (dafür bieten sich bspw. *Pilottöne* an).

1.2.4 Rauschen

Zusätzlich zur Verzerrung durch die Kanalparameter tritt ein additives Rauschen auf, das zum Verfälschen des gesendeten Signals beiträgt. Dieses kann nicht ohne Weiteres heraus gerechnet werden.

1.2.5 Mehrwegausbreitung

Durch die Mehrwegausbreitung entstehen wegen Reflexionen aus der Umgebung mehrere Echos des gesendeten Signals, die zeitversetzt beim Empfänger eintreffen. Sie können

[13] Siehe [1, S. 156-159]

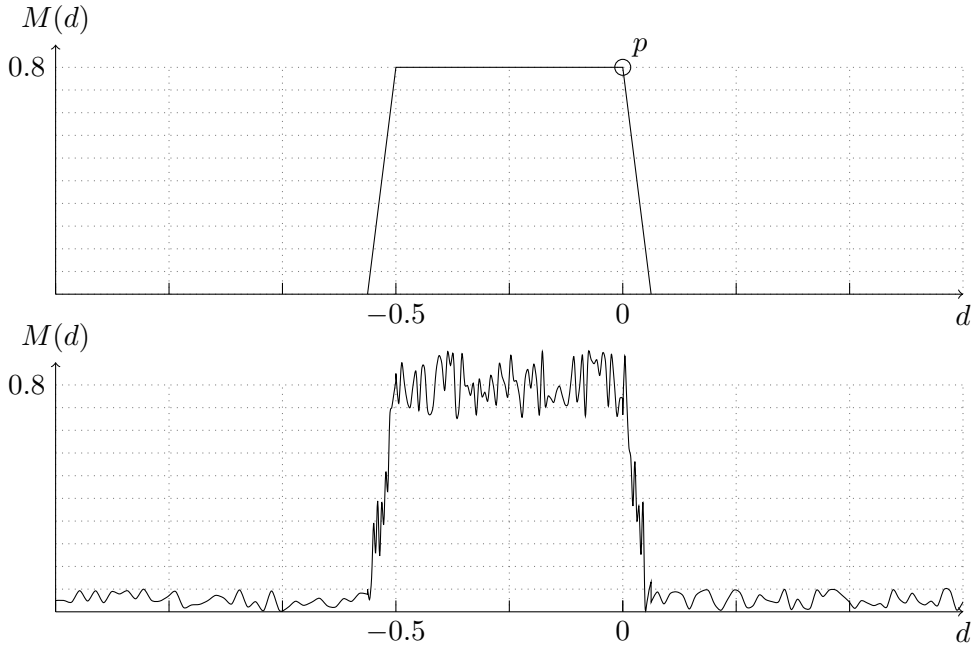

Abbildung 1.5: Darstellung der Metrik zum Erkennen des *OFDM*-Symbols ohne Rauschen (oben) und mit Rauschen (unten)

das Signal soweit beeinflussen, dass die *Orthogonalität* zwischen den einzelnen Trägern zerstört wird. Es entsteht *ISI*[14]. Um diesen Einfluss zu minimieren, sind weitere Maßnahmen notwendig.

1.3 Schmidl-Cox-Korrelator

Der Empfänger muss in der Lage sein, zu erkennen, wann ein OFDM-Symbol empfangen wird. In dieser Arbeit wird dazu der *Schmidl-Cox-Korrelator*[15] verwendet.

Das Prinzip zur Erkennung eines OFDM-Signals kann anschaulich am Beispiel des Cocktail-Party-Effekts erklärt werden. Dieser besteht darin, dass ein Mensch auf einer Party umgeben von mehreren Menschen, unter den vielen Störgeräuschen seinen Namen deutlich heraushören kann, obwohl dessen Lautstärke nicht größer ist als die der anderen Gäste.

Es ist also wünschenswert ein OFDM-Signal nicht an seiner Energie, sondern an seiner Struktur zu erkennen. Der *Schmidl-Cox-Korrelator* arbeitet nach diesem Prinzip. Er legt zwei hintereinander liegende Fenster über das empfangene Signal und schiebt diese Sample für Sample weiter. Mittels einer Metrik wird ein normierter Wert berechnet, der beim Übersteigen eines Schwellwerts das Eintreffen eines OFDM-Frames signalisiert.

[14] inter symbol interference
[15] Siehe [3]

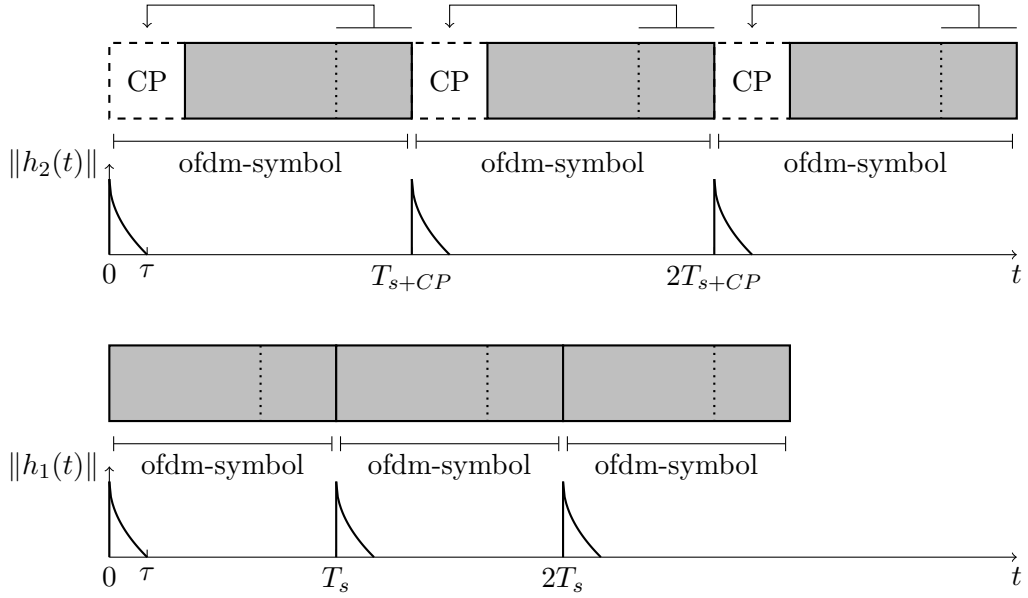

Abbildung 1.6: Darstellung mehrere *OFDM*-Symbole ohne *CP* (*unten*) und mit *CP* (*oben*)

Dazu werden zwei *Synchronisationssymbole* den OFDM-Symbolen vorangestellt. Sie weisen eine bestimmte Struktur auf, bei deren Eintreffen durch die Metrik ein charakteristischer Kurvenverlauf erzeugt wird. Dieser ist in Abbildung 1.5 abgebildet. Die obere Funktion stellt den idealen Verlauf dar. Es ist ein Trapez, dessen obere rechte Ecke (gekennzeichnet durch den Punkt p) ein *Synchronisationssymbol* anzeigt.

Das Problem besteht darin, dass ein zusätzliches Rauschen die Erkennung dieses Punktes erschwert. Dies ist in der unteren Funktion aus Abbildung 1.5 abgebildet. Der Korrelator irrt sich je nach Stärke des Rauschens um einige wenige oder mehrere Samples. Dieses Problem wird als *STO*[16] bezeichnet.

1.4 Cyclic Prefix

Wie bereits erwähnt, können durch die Mehrwegausbreitung des Signals Echos entstehen, die den Empfang des eigentlichen Signals stark stören können. Es stellt sich die Frage, wie deren Einfluss auf ein zu vernachlässigendes Maß reduziert werden kann.

Zu diesem Zwecke kommt ein *CP* zum Einsatz. Abbildung 1.6 zeigt eine Folge von OFDM-Symbolen, die hintereinander weg liegen. Im unteren Teil wird der Betrag des Kanal-Impulses $h_1(t)$ und darüber eine Aneinanderreihung von OFDM-Symbolen ohne *CP* dargestellt. Im oberen Teil wird der Betrag des Kanal-Impulses $h_2(t)$ und darüber wiederum eine Aneinanderreihung von OFDM-Symbolen mit einem *CP* dargestellt. Es gilt der Zusammenhang $T_{s+CP} = T_s + T_{CP}$. Jeder der Kanalimpulse hat eine Dauer von

[16] Symbol Time Offset

τ, bis dieser auf ein Maß abgefallen ist, dass ein OFDM-Symbol nicht mehr beeinträchtigt werden kann. Im unteren Teil ist zu sehen, wie der Kanalimpuls $h_1(t)$ in das OFDM-Symbol hineinragt und damit die *Orthogonalität* der einzelnen *Träger* zerstört (*ISI*). Im oberen Teil ist jedem OFDM-Symbol ein *CP* vorangestellt, sodass der Kanalimpuls $h_2(t)$ zwar in das OFDM-Symbol hineinragt, jedoch nur den *CP* beeinträchtigt, nicht jedoch den Daten tragenden Teil. Da eine Kopie des *CP* am Ende des OFDM-Symbols liegt, hat diese Störung keine Auswirkungen auf die Daten selbst.

Die Kunst liegt nun darin, das richtige Verhältnis zwischen T_{CP} und T_s zu finden. Je größer dieses ist, um so mehr sinkt die Bandbreiteneffizienz, da zwar redundante Daten, aber damit keine zusätzlichen Informationen, hinzugefügt werden. Je kleiner das Verhältnis ist, umso mehr steigt das Risiko, dass das Signal durch die Mehrwegausbreitung gestört wird. Im Allgemeinen wird in der Literatur oftmals vorgeschlagen, ein Verhältnis von $\frac{T_{CP}}{T_s} = \frac{1}{4}$ einzuhalten. Das ist zwar gut zum orientieren, allerdings muss T_{CP} mindestens so lang sein wie der *Delay-Spread*, um *ISI*-Freiheit zu garantieren. Das Verhältnis ist also abhängig vom Kanal zu wählen.

Wie bereits erwähnt, kommt ein *Schmidl-Cox-Korrelator* zum Einsatz, um die eingehenden OFDM-Frames zu detektieren. Dabei kann es bei stärkerem Rauschen vorkommen, dass das OFDM-Signal zum falschen Zeitpunkt erkannt wird. Hierfür gibt es vier Fälle, die unterschieden werden müssen[17]:

- $0 \leq t \leq \tau$: Der Kanalimpuls bewirkt eine Störung des Signals und damit eine Zerstörung der *Orthogonalität*. Dabei handelt es sich um *ISI*, da andere OFDM-Symbole als Echo des Originalsignals stören.

- $\tau < t < T_{CP}$: Der Zeitpunkt der Abtastung beginnt im hinteren Teil des *CP*, in dem keine Störung durch Echos auftreten, da der Kanalimpuls bereit abgeklungen ist. Dennoch wird das Signal zu früh erkannt. Da der *CP* jedoch redundante Daten beinhaltet, die am Ende des OFDM-Symbols wiederzufinden sind und dennoch mit einer Zeitdauer von T_s abgetastet wird, bewirkt dies eine zyklische Verschiebung des Signals $\{x_n\}_{n=0}^{N-1}$ und ergibt die Reihe $\{x_{(n+\delta)_N}\}_{n=0}^{N-1}$. Aufgrund einer Eigenschaft der *DFT* resultiert dies einzig und allein in eine Phasenverschiebung der einzelnen Träger. Ansonsten kommt es zu keinerlei Störung des Signals und die *Orthogonalität* bleibt erhalten.

- $t = T_{CP}$: Das OFDM-Frame wird zum genau richtigen Zeitpunkt abgetastet. Es kommt zu keinerlei Störungen und die *Orthogonalität* bleibt erhalten.

- $t > t_{CP}$: Das Fenster der Dauer T_s ragt mit seinem Ende bis in das nächste OFDM-Symbol hinein und führt deswegen zu einer Zerstörung der *Orthogonalität*, verursacht durch eine *ICI*.

Wie bereits erwähnt führt der zweite Fall dazu, dass eine Phasenverschiebung der einzelnen *Träger* stattfindet. Diese ist abhängig vom *Träger* und der Zeitdifferenz, um die zu früh abgetastet wird.

[17] Siehe [1, S. 154-156]

Um zu verstehen, wodurch diese Eigenschaft verursacht wird, soll folgende Gleichung betrachtet werden[18]:

$$Y[k] = \sum_{n=0}^{N-1} y[n+\delta] e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.17a}$$

$$= \sum_{n=0}^{N-1} \left\{ \frac{1}{N} \sum_{i=0}^{N-1} X[i] e^{\mathrm{j}2\pi i \frac{n+\delta}{N}} \right\} e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.17b}$$

$$= \frac{1}{N} \sum_{n=0}^{N-1} \sum_{i=0}^{N-1} X[i] e^{\mathrm{j}2\pi i \frac{n+\delta}{N}} e^{-\mathrm{j}2\pi k \frac{n}{N}} \tag{1.17c}$$

$$= \frac{1}{N} \sum_{n=0}^{N-1} \sum_{i=0}^{N-1} X[i] e^{\mathrm{j}2\pi n \frac{i-k}{N}} e^{\mathrm{j}2\pi \delta \frac{i}{N}} \tag{1.17d}$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} \sum_{n=0}^{N-1} X[i] e^{\mathrm{j}2\pi n \frac{i-k}{N}} e^{\mathrm{j}2\pi \delta \frac{i}{N}} \tag{1.17e}$$

$$= \sum_{i=0}^{N-1} X[i] e^{\mathrm{j}2\pi \delta \frac{i}{N}} \frac{1}{N} \sum_{n=0}^{N-1} e^{\mathrm{j}2\pi n \frac{i-k}{N}} \tag{1.17f}$$

$$= X[k] e^{\mathrm{j}2\pi \delta \frac{k}{N}} \text{ für } i = k \tag{1.17g}$$

Damit ist ersichtlich, dass das Verschieben des Abtastbereichs um δ Werte in den *CP* für $t > \tau$ hinein eine Phasenverschiebung bei jedem *Träger* abhängig von dessen Index k und der Verschiebung δ selbst entsteht. Damit lässt sich die Phasenverschiebung ohne Probleme auch wieder heraus rechnen ohne Informationsverlust.

Das bedeutet, dass sich der *Schmidl-Cox-Korrelator* um einen gewissen Offset δ irren darf, ohne dass es zu einem Verlust von Informationen kommt. Das erstellte Übertragungsprogramm macht von dieser Eigenschaft Gebrauch.

[18] Siehe [1, S. 154]

2 Präzisierung und Interpretation der Aufgabenstellung

Ziel dieser Arbeit ist es, einen Demonstrator zu schaffen, welcher mittels *OFDM* ein 2x2 MIMO-Übertragungssystem verwirklicht. Dafür muss die MIMO-Fähigkeit der verwendeten Geräte untersucht werden. Bei dem Gerät, das hierbei eingesetzt werden soll, handelt es sich um das USRP2 N210. Dieses besitzt zwei Antennen, deren unabhängige Nutzung jeweils als Sender oder Empfänger untersucht werden soll. Weiterhin soll das *OFDM*-Übertragungssystem die variable Nutzung eines jeden *Träger* als Daten- oder *Pilotträger* ermöglichen. Die Anzahl der *Träger* und das eingesetzte Synchronisationsverfahren sollen frei wählbar bzw. austauschbar sein. Es reicht aus, wenn das Übertragungssystem allein die Modulationsverfahren *BPSK*[1], *QPSK*[2] und *8-PSK* unterstützt.

Um dies zu ermöglichen, ist die bereits vorhandene Software mit ihrem Quellcode auf die Tauglichkeit zur Nutzung als Basis hin zu überprüfen. Dazu zählt bspw. der Aufbau und die Organisation des Quellcodes mit dem Augenmerk auf Modularität und Wiederverwendbarkeit, als auch auf die Aktualität der genutzten von *GNU Radio* angebotenen Funktionen.

Das resultierende Übertragungssystem sollte möglichst unkompliziert und einfach gestaltet sein, um keine unnötigen Fehlerquellen zu generieren.

Sobald eine funktionierende Übertragung mit einer akzeptablen Fehlerrate (unter 5%) zu Stande kommt, soll das System durch Nutzung zweier miteinander synchronisierter UHD-Devices auf Sender- und Empfängerseite auf dessen MIMO-Fähigkeit hin untersucht werden. Dabei müssen sich beide Kanäle ausreichend voneinander unterscheiden.

In Abhängigkeit dieses Ergebnisses sollen verschiedene Nutzungsszenarien dargestellt werden, die durch die Synchronisation zweier UHD-Devices ermöglicht werden.

[1] Binary Phase Shift Keying
[2] Quadrature Phase-shift keying

3 Entwurf und Implementierung

3.1 Quelltexte anderer Arbeiten als Basis

3.1.1 `gr-jwdiplom`

`gr-jwdiplom` ist ein Projekt einer früheren studentischen Arbeit. Dieses Projekt sollte ursprünglich als Basis für diese Arbeit fungieren. Eine Analyse des Quelltexts ergab allerdings, dass dieses Projekt als Grundlage für diese Arbeit nicht das Geeignetste ist. Dabei spielen folgende Punkte eine große Rolle:

- Die empfangenen Signale werden nicht in Echtzeit durch den arbeitenden Signalflussgraphen analysiert, sodass dieser variabel auf Veränderungen reagieren könnte (z.B. Abschalten bestimmter Träger). Die Signale werden stattdessen zwischengespeichert und erst nach Beendigung des Programms analysiert.
- Der Aufbau des Programms gestaltet die Integration des Quellcodes in andere Projekte schwierig, da der Quellcode nicht genügend modularisiert worden ist. Es fehlt an Schnittstellen, die eine lose Kopplung der einzelnen Module ermöglicht.
- Das Programm wurde für das UHD-Device USRP N200 entwickelt. Mit einem USRP N210 funktionierte die Datenübertragung nicht und konnte auch trotz hohem Aufwand nicht ermöglicht werden.

3.1.2 OFDM-Beispielquelltext aus dem Modul `gr-digital`

Im Verzeichnis `gr-digital/examples/ofdm` des *GNU Radio*-Repositorys befinden sich die beiden Programme `benchmnark_tx.py` und `benchmnark_rx.py`.

Diese greifen auf mehrere Module und Blöcke im selben Verzeichnis zu und realisieren ein OFDM-Übertragungssystem. Zuerst schien der Quelltext vielversprechend, zeigte aber bei genauerer Betrachtung mehrere Probleme:

- Der Quelltext besaß stark verschachtelte Strukturen, wodurch er schwer zu analysieren war.
- Die Module waren stark miteinander gekoppelt, was es sehr schwer machte einzelne Teile des Quelltextes wiederzuverwenden.
- Der veraltete *GNU Radio*-Block `ofdm_mapper_bvc` wird zur Belegung der einzelnen *Träger* eines OFDM-Symbols genutzt und ist wenig flexibel. Er erlaubt nicht die freie Wahl der Belegung einzelner *Tons* als *Datenton* oder *Pilotton*. Zudem wird eine extra Signalleitung genutzt um Metainformationen an die kommenden Blöcke weiterzugeben anstatt in Streams eingebettete Tags zu nutzen.
- Nach mehreren Testläufen zeigte sich eine hohe Fehlerrate bei der Datenübertragung mittels *8-PSK* von 10-12%.

Da die hohe Fehlerrate nicht hinnehmbar war, die Möglichkeit fehlte die *Träger* frei nach Funktion zu belegen und die Art und Weise der Parameterweitergabe an andere Blöcke veraltet war, wurde nach einer weiteren Alternative gesucht.

3.1.3 Der Block `ofdm_txrx.py`

Der in Python geschrieben Block war eine Aneinanderreihung von anderen C++-Blöcken aus *GNU Radio*, die aus einer neueren Generation von OFDM-Blöcken stammen. Der Quelltext und die verwendeten Blöcke boten folgende Vorteile gegenüber den vorher untersuchten Programmen:

- Die verwendeten Blöcke gaben Parameter mittels Tagged Streams an andere Blöcke weiter, was den Signalflussgraphen deutlich einfacher und flexibler machte.
- Zur Belegung der einzelnen *Träger* mittels *Datensymbole* oder *Pilotsymbole* wird der Block `ofdm_carrier_allocator_cvc` genutzt. Er erlaubt das flexible Konfigurieren der Trägerbelegung und das Voranstellen von bis zu zwei unabhängig voneinander wählbaren Synchronisationssymbolen.
- Da hier auf einfache Weise mehrere Blöcke miteinander verschaltet wurden, ist der Aufbau hochgradig modular, was die Wiederverwendung einzelner Quellcode-Teile vereinfacht.

Nachdem der Block `ofdm_txrx.py` in ein lauffähiges Programm eingebettet worden ist, ergaben sich einige Probleme beim Übertragen von Daten:

- Es gingen mehrere OFDM-Frames verloren, was teilweise eine hohe Fehlerrate zur Folge hatte.
- Der Block verwendet zur Datenübermittlung einen Header, der die Bit-Anzahl der Modulationsart übertrug. Dies machte das Programm kompliziert und fehleranfällig in der Übertragung.

- Beim Empfang der Daten wird die Bit-Anzahl der Modulationsart aus dem empfangenen Header ausgelesen und über eine Messaging-Funktion an einen Block übergeben, der die OFDM-Symbole aus den Samples extrahieren soll. Bis die Message bei diesem Block eintrifft, muss dieser warten und kann keine Aufgaben abarbeiten, was beim Empfang der Daten zu Problemen führen kann.
- Der Bock ermöglicht nur das Versenden und Empfangen von *PSK*[1]-modulierten Trägern, was angesichts der Aufgabenstellung aber akzeptabel war.

Obwohl das Programm einige Probleme bereitete, schien es für diese Arbeit am geeignetsten zu sein, woraufhin die These aufgestellt wurde, dass durch Vereinfachung des Signalflussgraphen die Probleme größtenteils gelöst oder zumindest auf ein Maß begrenzt werden können, welches nicht mehr stark ins Gewicht fällt.

Das Ergebnis dazu ist ein neues *GNU Radio*-Modul namens `gr-mimoots`, welches auf diesem Quellcode basiert.

3.2 Das GNU Radio-Modul `gr-mimoots`

`gr-mimoots` steht für MIMO-OFDM-Transmission-System und basiert auf dem *GNU Radio*-Block `ofdm_txrx.py`. Dazu wurde dieser stark vereinfacht. Der gesonderte Pfad für das Erstellen und Auslesen des Headers wurde entfernt, was den Signalflusspfad und den Quellcode stark vereinfachte. Schlussendlich konnte mit `gr-mimoots` eine funktionierende Übertragung hergestellt werden, deren Fehlerrate nicht über 2% hinausging.

Im Folgenden soll dazu das Modul und dessen einzelne Bestandsteile und deren Funktionsweise erklärt werden.

In Grafik 3.1 sind die einzelnen Komponenten von `gr-mimoots` und deren Zusammenwirken zu erkennen. Der gesamte Signallaufpfad ist wiederum in mehrere Blöcke unterteilt, die allesamt Teil von `gr-mimoots` sind. Ihr Zusammenspiel ergibt ein funktionierendes Programm, dass ein OFDM-Basisband-Signal bestehend aus komplexen Werten erstellt und dieses wieder empfangen und verarbeiten kann, um am Ende die übertragenen Daten zu rekonstruieren.

[1] Phase Shift Keying

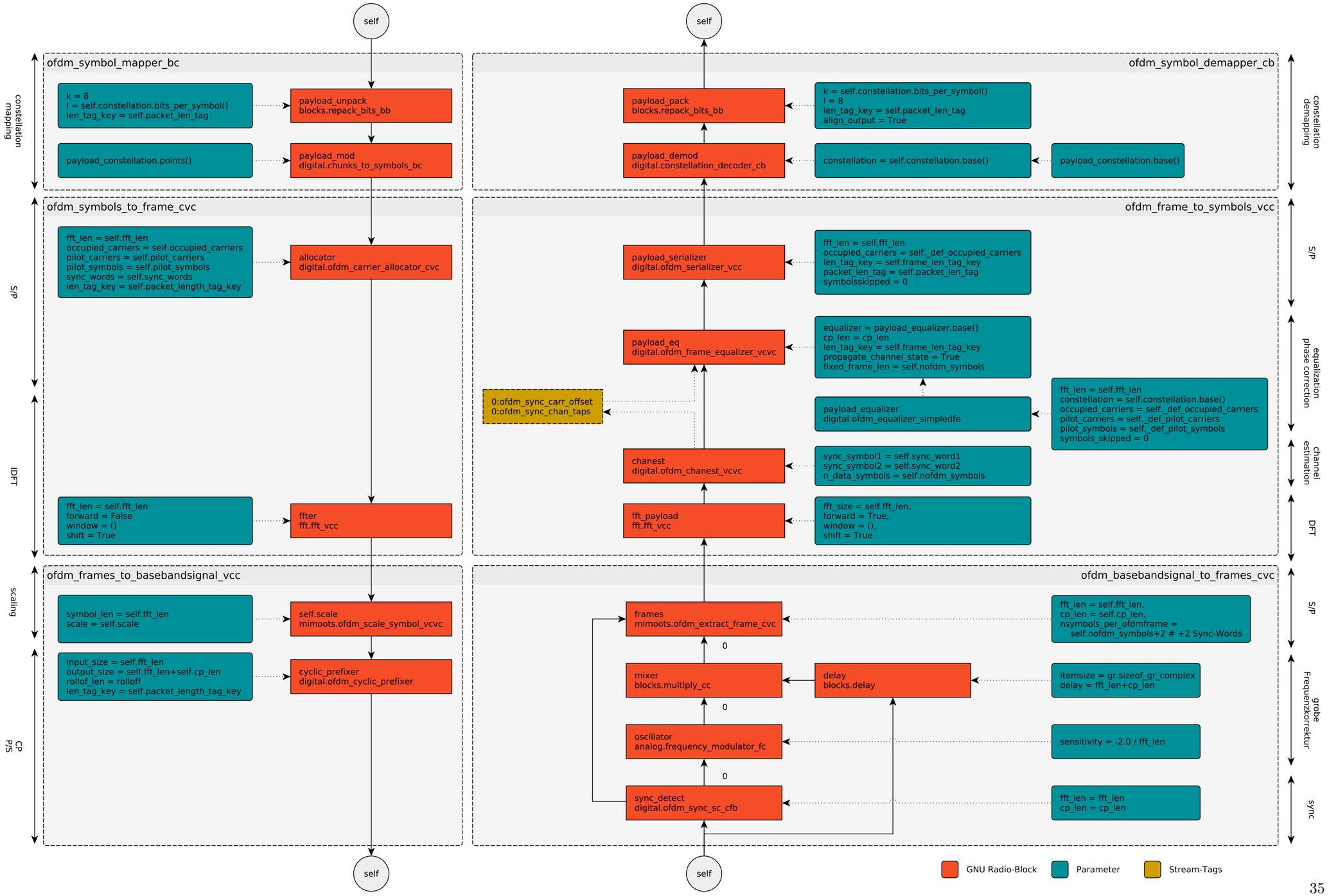

Abbildung 3.1: Das *GNU Radio*-Modul `gr-mimoots` und deren Bestandteile und Funktionsweise

3.2.1 Blöcke

3.2.1.1 ofdm_symbol_mapper_bc

Dieser Block bildet die empfangenen Bits auf Symbole aus dem ihm übergebenen Symbolvorrat ab (`payload_constellation.points()`). Dazu werden ihm die Anzahl an Bits, die für die gewählte Modulation nötig sind (z.B. 3 Bit für *8-PSK*) als Argument übergeben (`self.constellation.bits_per_symbol()`). *GNU Radio* hat als kleinste Einheit nur den Typ `char`. Der Bit-Strom muss deshalb auf einzelne Einheiten vom Typ `char` aufgeteilt werden. `char` besteht aus 8 Bit (bspw. wird für eine *8-PSK* der Bit-Strom $...101111_2$ zu 00000101_2 und 00000111_2, da *8-PSK* $ld(8) = 3Bits$ auf ein komplexes Symbol abbildet).

Dieser und andere nachfolgende Blöcke werten das Tag `len_tag_key` aus um zu ermitteln, an welchen Stellen im Eingangsdatenstrom ein neues OFDM-Frame beginnen soll.

Dazu muss dem Eingangsdatenstrom das Tag `len_tag_key` hinzugefügt werden. Eine Möglichkeit wäre es, die Datenpakete in eine zweidimensionale Liste zu packen, d.h. eine Liste von Listen zu generieren. Jede Liste in der Liste entspricht dann einem OFDM-Symbol. Diese Liste könnte folgender Funktion, bereitgestellt durch *GNU Radio*, übergeben werden:

```
(data_tosend, tags) = packet_utils.packets_to_vectors(
    self.data,
    self.packet_len_tag
)
```

`packets_to_vectors` generiert eine große 1-dimensionale Liste für die Daten und eine Liste mit den an den richtigen Stellen platzierten Tags vom Typ `packet_len_tag`, die dann bspw. dem Block `vector_source_b` übergeben werden können, der die Liste als Datenstrom an den nächsten Block weiterleitet.

Bezogen auf Abbildung 1.1 kann dieser Block dem Bereich *constellation mapping* zugeordnet werden.

3.2.1.2 ofdm_symbols_to_frame_cvc

Die komplexen Symbole werden nun auf die einzelnen *Träger* eines OFDM-Symbols aufgeteilt.

Damit der Block `ofdm_carrier_allocator_cvc` auch weiß, wie groß die zu verarbeitenden Daten-Pakete (OFDM-Symbole) sind, wird ihm der Parameter `fft_len` übergeben. Dieser gibt an, wie viele *Träger* zur Verfügung stehen (da nicht alle *Träger* genutzt werden müssen). Gleichfalls gibt der Parameter auch an, aus wie vielen Samples das *OFDM-Symbol* im Zeitbereich besteht.

Es folgt der Parameter `occupied_carriers`, der eine Liste von Indizes derjenigen *Träger* enthält, die Datensymbole aufnehmen sollen. Dabei entspricht der Wertebereich der Indizes zum adressieren der *Träger* dem Intervall $(-(\frac{N}{2}), \frac{N}{2} - 1)$, wobei N dem Parameter `fft_len` entspricht. Hierbei wird der *Träger* mit dem Index 0 normalerweise nicht

belegt um Gleichanteilsfreiheit im zu generierenden Signal sicherzustellen. Die eingehenden komplexen Werte aus dem Signalflussgraphen sind dementsprechend die komplexen Symbole, die auf die einzelnen Datenträger fortlaufend verteilt werden.

Der Parameter `pilot_carriers` gibt eine Liste mit den Indizes von Trägern an, die als *Pilotträger* genutzt werden sollen. Die Liste dieser komplexen Pilotsymbole wird mit dem Parameter `pilot_symbols` übergeben. Der Inhalt diese Liste wird fortlaufend und zyklisch den als *Pilotträger* bestimmten Trägern zugeordnet. Ist das Ende dieser Liste erreicht, wird wieder an deren Anfang gesprungen. Damit kann die Liste auch länger oder kürzer sein als die Anzahl an Pilotträgern in einem OFDM-Symbol.

Träger, die nicht in den vorhergehenden Listen als Indizes enthalten waren, werden einfach nicht belegt.

Außerdem erhält der Block über den Parameter `sync_words` eine Liste von zwei Listen, die die zwei *Synchronisationssymbole* enthalten. Um diese zu generieren werden die beiden Funktionen `ofdm_make_sync_word1` und `ofdm_make_sync_word2` aus dem Modul `utils` genutzt.

Die entstandenen Daten werden parallelisiert und als Vektor weitergegeben, was bezogen auf Abbildung 1.1 dem Bereich *parallelizer* entspricht und anders als in 1.1 abgebildet nach dem *constellation mapping* geschieht.

Als nächstes folgt die Berechnung des OFDM-Basisband-Signals mittels der IDFT, was gleichzusetzen ist mit dem Bereich *IDFT* aus Abbildung 1.1.

Zum Schluss wird das Symbol noch Vektor-weise skaliert, d.h. die Amplituden werden pro Vektor so normiert, dass höchstens Amplituden mit dem Wert 1 im Vektor vorkommen. Dies geschieht immer bezogen auf die Werte im selben Vektor unabhängig von den Werten in vorhergehenden oder nachfolgenden Vektoren. Danach wird jedem Vektor noch ein *CP* vorangestellt sowie der Vektor selbst serialisiert und als Folge komplexer Werte ausgegeben, was bezogen auf Abbildung 1.1 dem Abschnitt *serializer* entspricht.

3.2.1.3 `ofdm_basebandsignal_to_frames_cvc`

Der empfangene Datenstrom bestehend aus komplexen Werten wird mit einem *Schmidl-Cox-Korrelator* analysiert. Dieser ermittelt bei eintreffenden OFDM-Signalen den groben Frequenzoffset und gibt diesen Wert weiter, damit das Signal um diesen Frequenzoffset korrigiert werden kann. Der Block `ofdm_extract_frames_cvc` nimmt dieses korrigierte Signal und extrahiert an der richtigen Stelle, die ihm über eine Signalleitung mitgeteilt werden, das empfangene OFDM-Frame heraus, entfernt aus jedem Synchronisations- und *OFDM-Symbol* den *CP* und übergibt das OFDM-Frame dem nächsten Block als Vektor. Bezogen auf Abbildung 1.1 entspricht dies dem Bereich *serializer*.

3.2.1.4 `ofdm_frame_to_symbols_vcc`

Die vektorisierten OFDM-Frames werden mittels einer *DFT* wieder in den Frequenzbereich umgerechnet, um die Symbole extrahieren zu können. Durch Störungen, verursacht durch den Übertragungskanal, können diese Symbole verfälscht sein. Um die Kanalparameter zu schätzen, untersucht der Block `ofdm_chanest_vcvc` das OFDM-Frame und die

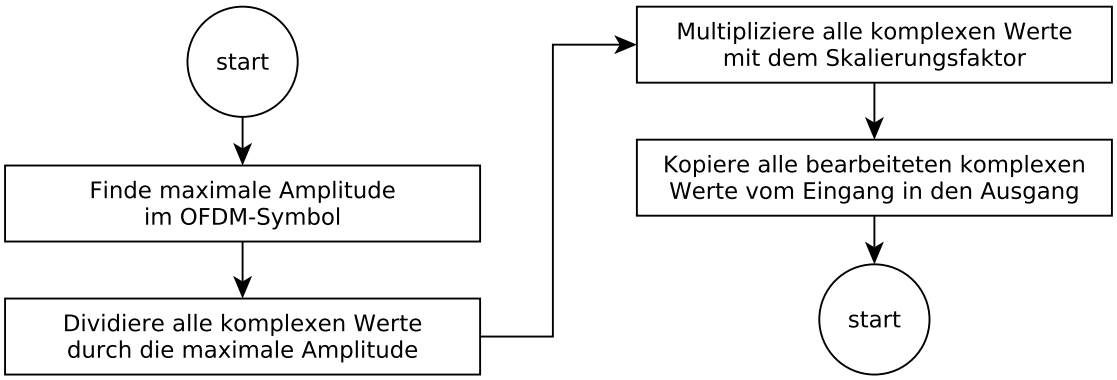

Abbildung 3.2: *PAP*[2] für die Funktion `work()` vom Block `ofdm_scale_frames_vcvc`

zwei ihm vorangestellten *Synchronisationssymbole*. Dazu werden ihm die unverfälschten *Synchronisationssymbole* als Parameter übergeben und aus der Abweichung die Kanal-Parameter geschätzt. Weiterhin wird noch eine feine Frequenzabweichung berechnet. Beides, Frequenzabweichung und Kanalparameter, werden dem Datenstrom als Tags mit übergeben (`ofdm_sync_carr_offset` und `ofdm_carr_chan_taps`).

Durch den Block `ofdm_frame_equalizer_vcvc` werden diese Tags ausgewertet und zur Datenrekonstruktion eingesetzt. Da davon auszugehen ist, dass die Synchronisation um einige Samples verschoben ist und daher eine Phasenverschiebung der Daten verursacht wird, kann diese mittels der *Pilottöne* heraus gerechnet werden.

Zuletzt werden die Träger, die die Datensymbole beinhalten, heraus gefiltert und als Datenstrom weitergegeben.

Ausgehend von Abbildung 1.1 deckt dieser Block die Bereiche *inverse Fourier transformation* und *serializer* ab.

3.2.1.5 `ofdm_symbol_demapper_cb`

Die komplexen Symbole werden nun durch einen Entscheider (*estimator*) und einen Demapper (*constellation demapping*) auf die ursprünglichen Bits abgebildet. Das Ergebnis ist eine Folge von `char`-Werten, die die Bits enthalten. Diese müssen nun noch zu einem zusammenhängenden Bit-Strom vereint werden, der schlussendlich an den nächsten Block weitergegeben wird.

3.2.1.6 `ofdm_scale_symbol_vcvc`

Als Eingabe erhält der C++-Block einen Vektor komplexer Wert. Die Länge dieses Vektors wird durch den Parameter `symbol_len` festgelegt und entspricht normalerweise dem Wert der Variable `fft_len`, die wiederum dem Wert N entspricht. Die Ausgabe ist der Eingabe-Vektor, dessen Amplituden-Werte auf den größten im Vektor vorkommenden

[2] Programm-Ablauf-Plan

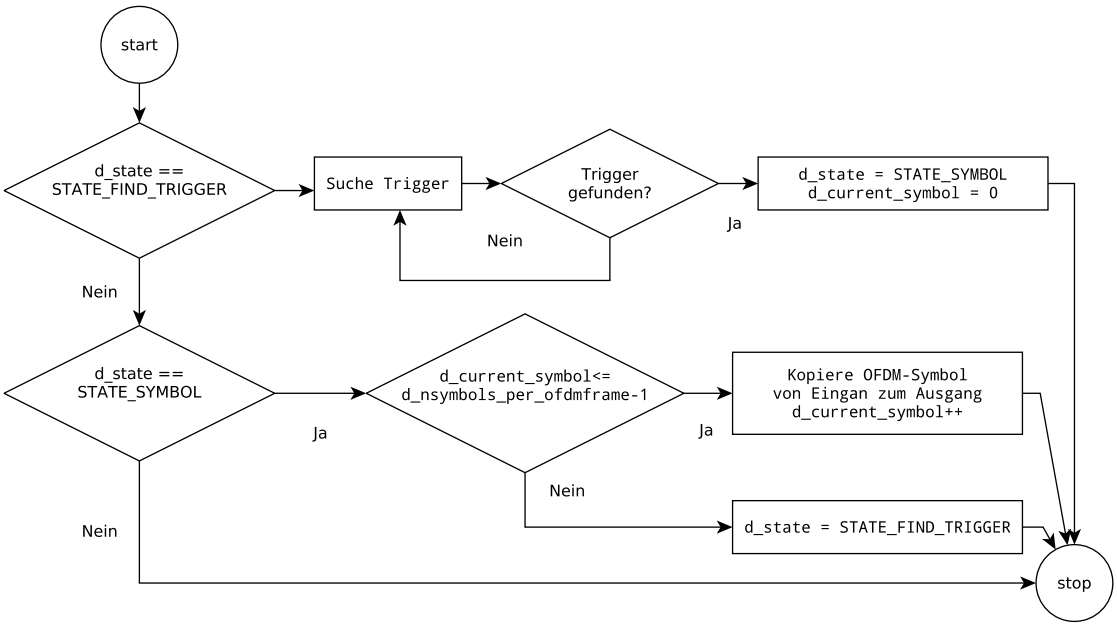

Abbildung 3.3: *PAP* für die Funktion `general_work()` vom Block `ofdm_extract_frames_cvc`

Amplituden-Wert normiert worden sind. Dadurch kommt im Vektor kein Amplituden-Wert vor, der größer als 1 ist. Zusätzlich können noch über den Parameter `scale = 0.0 ... 1.0` die Amplituden-Werte gemeinsam skaliert werden. Der Ablauf der Funktion `work()` ist im *PAP* dargestellt durch die Abbildung 3.2 veranschaulicht.

3.2.1.7 `ofdm_extract_frame_vcvc`

Dieser C++-Block bekommt den vom *Schmidl-Cox-Korrelator* übergebenen Strom komplexer Werte am Eingang 0 übergeben und extrahiert die detektierten OFDM-Frames heraus. Dazu wertet er die Signalleitung am Eingang 1 heraus, die ein Flag setzt, wenn sich an der aktuellen Position in dem ihm übergebenen Datenstrom der Anfang eines OFDM-Frames befindet. Zum einen werden ihm die Parameter `fft_len` und `cp_len` übergeben, die der Länge des OFDM-Symbols ohne *CP* und der Länge des *CP* selbst entsprechen, sowie der Parameter `nsymbols_per_ofdmframe`, welcher der Anzahl aller *OFDM-Symbole* und der *Synchronisationssymbole* zusammen entspricht.

Der Block wechselt in seiner Funktion `general_work()` zwischen den beiden Zuständen (states) `STATE_FIND_TRIGGER` und `STATE_SYMBOL` hin und her.

Im Zustand `STATE_FIND_TRIGGER` sucht er nach dem gesetzten Flag im Eingang 1, das vom *Schmidl-Cox-Korrelator* an der Stelle gesetzt wird, an der sich schlussendlich der Anfang des OFDM-Frames befindet. Ist dieses Flag gesetzt, wechselt der Block in den Zustand `STATE_SYMBOL` und kopiert das Synchronisations- oder *OFDM-Symbol* der Länge `fft_len` in die Ausgabe, wobei automatisch der *CP* entfernt oder besser gesagt nicht mit kopiert wird. Die Anzahl an kopierten OFDM-Symbolen wird um den Wert 1

erhöht. Nach dem Kopiervorgang für ein *OFDM-Symbol* wird die Funktion beendet und die Kontrolle wieder an *GNU Radio* zurückgegeben. Dabei bleiben die Zustandsvariablen des Objekts und der aktuelle Zustand erhalten. Daraufhin ruft *GNU Radio* die Funktion `general_work` wieder auf. Der Block (realisiert als Objekt) merkt sich, wie viele Symbole bereits kopiert worden sind und kopiert das nächste OFDM-Symbol solange, bis das letzte *OFDM-Symbol* kopiert worden ist. Dann wechselt der Block wieder in den Zustand `STATE_FIND_TRIGGER` und sucht aufs Neue nach dem Flag. Der Ablauf der Funktion `general_work()` ist in Abbildung 3.3 veranschaulicht.

3.2.1.8 `uhd_source`

Zum Ansprechen eines UHD-Devices als Empfänger stellt das Modul `gr-mimoots` den Block `uhd_source`. Dieser kann mit folgenden Parametern konfiguriert werden:

`freq` Angabe der Frequenz, auf der empfangen werden soll.

`gain` Angabe der Verstärkung des Signals beim Empfangen. Die Verstärkung wird in dB angegeben und kann in den vom UHD-Device zur Verfügung gestellten Werten variieren. Je nach UHD-Device wählt der Block den Mittelwert aller vom Gerät angebotenen Werte als Standardwert.

`samp_rate` Angabe der abzutastenden Samples pro Sekunde. Der Standardwert beträgt 500000Hz.

`antenna` Angabe der Antenne, die zum Empfangen genutzt werden soll. Der Standardwert lautet `"RX2"`

Bis auf den Parameter `freq` besitzen alle Parameter einen sinnvollen Defaultwert, der die Benutzung dieses Blocks vereinfachen soll. Es muss kein Parameter für die Adresse angegeben werden, da der Block das UHD-Device automatisch findet.

3.2.1.9 `uhd_source2`

Dieser Block funktioniert grundsätzlich wie der Block `uhd_source`. Er dient zum Ansteuern zweier miteinander synchronisierter UHD-Devices[3]. Diese müssen mit einem speziellen Synchronisationskabel verbunden sein.

Die einzelnen Parameter stimmen mit denen des Blocks `uhd_source` in ihrer Funktion überein. Um jedes UHD-Device separat zu konfigurieren, müssen die Werte für einen Parameter als Liste übergeben werden. Wird nur ein einzelner Wert übergeben, so wird er für beide Geräte gesetzt.

Zum Betrieb der Geräte gibt es zwei verschiedene Modi:

Betrieb mit einem Netzwerkkabel In diesem Fall ist nur eines der beiden Geräte über ein Netzwerkkabel mit dem Computer verbunden. Dieses ist das Master-Device. Das andere Gerät, welches Slave-Device genannt wird, überträgt seine Daten über

[3] Siehe [4]

das Synchronisationskabel an das Master-Device und dieses leitet die Daten an den Computer weiter. Hierzu müssen die Adressen der Geräte im selben Subnetzwerk liegen. Als Standardwerte für die Adressen beider Geräte wurde `"addr0=192.168.10.2"` für das Master- und `"addr1=192.168.10.3"` für das Slave-Device festgelegt.

Betrieb mit zwei Netzwerkkabeln Es ist auch möglich, dass beide Geräte jeweils separat über ein eigenes Netzwerkkabel mit dem Computer verbunden sind. Dazu müssen die Adressen jedoch in unterschiedlichen Subnetzen liegen. Hierzu muss für das eine UHD-Device die Adresse `"addr0=192.168.10.2"` und für das Andere die Adresse `"addr1=192.168.20.2"` gesetzt werden. Da beide Geräte separat über jeweils ein Netzwerkkabel angesteuert werden, stellt sich die Frage nach der zeitlichen Synchronität beider Geräte, die im Fall eines gemeinsamen Netzwerkkabels schon durch die Art der Verbindung sichergestellt ist. Diese Frage muss bei der Nutzung zweier Netzwerkkabel extra beantwortet werden.

3.2.1.10 `uhd_sink`

Zum Ansprechen eines UHD-Devices als Sender stellt das Modul `gr-mimoots` den Block `uhd_sink`. Dieser kann mit folgenden Parametern konfiguriert werden:

`freq` Angabe der Frequenz, auf der gesendet werden soll.

`gain` Angabe der Verstärkung des Signals beim Senden. Die Verstärkung wird in dB angegeben und kann in den vom UHD-Device zur Verfügung gestellten Werten variieren. Je nach UHD-Device wählt der Block den Mittelwert aller vom Gerät angebotenen Werte als Standardwert.

`samp_rate` Angabe der abzutastenden Samples pro Sekunde. Der Standardwert beträgt 500000Hz.

`antenna` Angabe der Antenne, die zum Senden genutzt werden soll. Der Standardwert lautet `"TX/RX"`.

Bis auf den Parameter `freq` besitzen alle Parameter einen sinnvollen Defaultwert, der die Benutzung dieses Blocks vereinfachen soll. Es muss kein Parameter für die Adresse angegeben werden, da der Block das UHD-Device automatisch findet.

3.2.1.11 `uhd_sink2`

Dieser Block funktioniert grundsätzlich wie der Block `uhd_sink`. Er dient zum Ansteuern zweier miteinander synchronisierter UHD-Devices[4]. Diese müssen mit einem speziellen Synchronisationskabel verbunden sein.

Die einzelnen Parameter stimmen mit denen des Blocks `uhd_sink` in ihrer Funktion überein. Um jedes UHD-Device separat zu konfigurieren, müssen die Werte für einen

[4] Siehe [4]

Parameter als Liste übergeben werden. Wird nur ein einzelnen Wert übergeben, so wird er für beide Geräte gesetzt.

Zum Betrieb der Geräte gibt es zwei verschiedene Modi:

Betrieb mit einem Netzwerkkabel In diesem Fall ist nur eines der beiden Geräte über ein Netzwerkkabel mit dem Computer verbunden. Dieses ist das Master-Device. Das andere Gerät, welches Slave-Device genannt wird, erhält seine zu übertragenden Daten über das Synchronisationskabel vom Master-Device. Hierzu müssen die Adressen der Geräte im selben Subnetzwerk liegen. Als Standardwerte für die Adressen beider Geräte wurde `"addr0=192.168.10.2"` für das Master- und `"addr1=192.168.10.3"` für das Slave-Device festgelegt.

Betrieb mit zwei Netzwerkkabeln Es ist auch möglich, dass beide Geräte jeweils separat über ein eigenes Netzwerkkabel mit dem Computer verbunden sind. Dazu müssen die Adressen jedoch in unterschiedlichen Subnetzen liegen. Hierzu muss bspw.für das eine UHD-Device die Adresse `"addr0=192.168.10.2"` und für das andere die Adresse `"addr1=192.168.20.2"` gesetzt werden. Auch hier ergibt sich wieder das Problem der Sicherstellung der zeitlichen Synchronität beider Geräte, da zwei verschiedene Netzwerkkabel genutzt werden.

3.2.1.12 `file_source2`

Im Gegensatz zum Block `file_source` kann dieser Block zwei Datenstreams aufnehmen und jeweils in eine Datei leiten. Dies ist zwar auch mit zwei Blöcken vom Typ `file_source` möglich, macht die Verschaltung dieser Blöcke aber aufwendiger, wenn als zweite Möglichkeit an deren Stelle ein Block vom Typ `uhd_source2` treten soll.

3.2.1.13 `file_sink2`

Im Gegensatz zum Block `file_source` kann dieser Block Daten aus zwei verschiedenen Dateien jeweils mit einem Datenstream weiterleiten. Dies ist zwar auch mit zwei Blöcken vom Typ `file_sink` möglich, macht die Verschaltung dieser Blöcke aber aufwendiger, wenn als zweite Möglichkeit an deren Stelle ein Block vom Typ `uhd_sink2` treten soll.

3.2.2 Das Modul `utils`

Bei diesem Block handelt es sich um keinen funktionsfähigen Block zur Signalverarbeitung, wie die vorhergehenden Blöcke. `utils` ist ein Modul, welches folgende Funktionen unabhängig vom Einsatzort im Quellcode zur Verfügung stellt:

- `ofdm_get_active_carriers`: gibt die Anzahl der *Träger* zurück, die *Datensymbole* oder *Pilotsymbole* tragen
- `ofdm_make_sync_word1`: erstellt das erste *Synchronisationssymbol* für den *Schmidl-Cox-Korrelator*

- `ofdm_make_sync_word2`: erstellt das zweite *Synchronisationssymbol* für den *Schmidl-Cox-Korrelator*
- `ofdm_get_data_len`: gibt die Länge einer Liste von `char`-Elementen in Python zurück, die benötigt wird um ein *OFDM-Symbol* zu erstellen

Mittels der Anweisung

```
from mimoots import utils
```

kann über den Befehl

```
utils.function()
```

auf eine der Funktionen zugegriffen werden.

3.2.3 Programme

In diesem Abschnitt sollen kurz die in dieser Arbeit erstellten Anwendungsprogramme und ihre Funktionsweise erklärt werden.

3.2.3.1 `mimoots_ofdm_tx.py`

Um ein OFDM-Basisband zu generieren, müssen die bereits diskutierten Blöcke miteinander verschaltet werden und in einen *GNU Radio*-Graphen eingebettet werden. Genau dazu dient dieses Kommandozeilen-Programm. Es greift auf die in Python geschriebenen Blöcke von `gr-mimoots` zurück und kann ein OFDM-Basisbandsignal wahlweise in eine Datei oder an ein UHD-Device zum Senden leiten.

Um das Programm nutzen zu können, werden folgende Parameter zur Verfügung gestellt:

`--to-file TO_FILE` speichert die erstellten Werte in der Datei `TO_FILE` und kann nicht mit `-f` verwendet werden

`-f FREQ, --freq FREQ` nutzt die Frequenz `FREQ` als Übertragungsfrequenz und kann nicht verwendet werden mit `--to-file TO_FILE`

`--nframes NFRAMES` gibt die Anzahl an OFDM-Frames an, die erzeugt werden sollen

`--nsymbols NSYMBOLS` gibt die Anzahl an OFDM-Symbolen an, die in einem OFDM-Frame erzeugt werden sollen

`--bits BITS` gibt die Anzahl an Bits an, die mit einem Datensymbol kodiert werden sollen und damit auch die Modulationsart (*BPSK*, *QPSK* oder *8-PSK*)

`-v, --verbose` gibt zusätzliche Informationen aus

`--dummy-frame-start` stellt den OFDM-Frames ein einzelnes OFDM-Frame als Dummy voran

`--dummy-frame-end` hängt den OFDM-Frames zum Schluss ein einzelnes OFDM-Frame als Dummy an

`--gain GAIN` stellt den zu verwendeten Gain für das UHD-Device ein und ist bei der Nutzung von `--to-file TO_FILE` ohne Bedeutung

`--scale SCALE` Skaliert die ausgegebenen Samples noch einmal mit einem Wert zwischen 0.0 und 1.0

3.2.3.2 `mimoots_ofdm_rx.py`

Dieses Programm ist das Gegenstück zu `mimoots_ofdm_tx.py` und wandelt das OFDM-Basisbandsignal wieder in eine Liste aus Werten um. Am Ende werden diese Werte mit den zu erwartenden Werten verglichen und damit die Bit-Fehlerrate berechnet und ausgegeben.

`--from-file FROM_FILE` liest die zu empfangenen Werte aus der Datei `FROM_FILE` und kann nicht mit `-f` verwendet werden

`-f FREQ, --freq FREQ` nutzt die Frequenz `FREQ` als Übertragungsfrequenz und kann nicht verwendet werden mit `--from-file FROM_FILE`

`--nframes NFRAMES` gibt die Anzahl an OFDM-Frames an, die empfangen werden sollen

`--nsymbols NSYMBOLS` gibt die Anzahl an OFDM-Symbolen an, die sich beim Empfang in einem OFDM-Frame befinden

`--bits BITS` gibt die Anzahl an Bits an, die mit einem Datensymbol kodiert worden sind und damit auch die Modulationsart (*BPSK*, *QPSK* oder *8-PSK*), mit der die empfangenen Daten interpretiert werden sollen

`-v, --verbose` gibt zusätzliche Informationen aus

`-d, --dummy-frame` interpretiert das erste OFDM-Frame als Dummy-Frame

`--gain GAIN` stellt den zu verwendeten Gain für das UHD-Device ein und ist bei der Nutzung von `--from-file FROM_FILE` ohne Bedeutung

`--skiphead SKIPHEAD` Überspringt die Anzahl von `SKIPHEAD` Werten zu Beginn der zu verarbeitenden Daten

3.2.3.3 `mimoots_ofdm2_tx.py`

Anstatt ein OFDM-Basisbandsignal werden durch dieses Programm zwei Basisbandsignale generiert und über zwei miteinander synchronisierte UHD-Devices geschickt, wobei jedes Signal über eine Antenne abgestrahlt wird. Genauso kann jedes Signal in eine Datei gespeichert werden.

Die Parameter stimmen mit denen aus dem Programm `mimoots_ofdm_tx.py` überein.

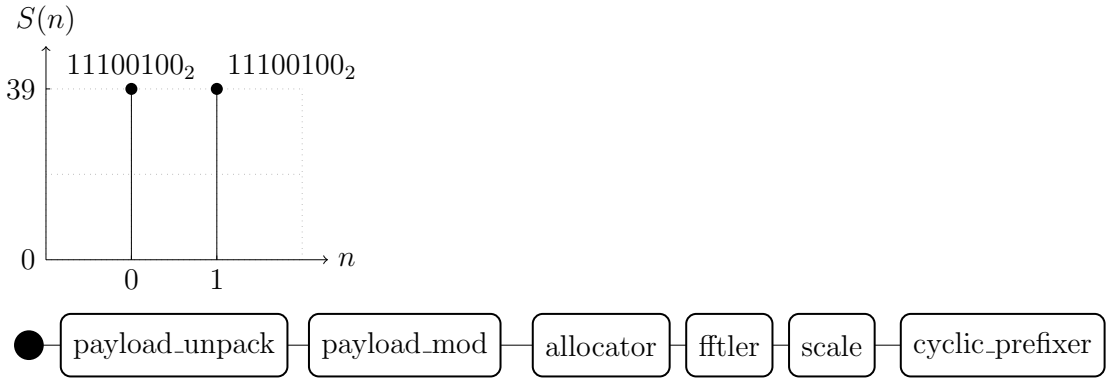

Abbildung 3.4: Die Zahlen 39_{10} in binärer Form dargestellt, wie sie an den nächsten Block weitergeleitet werden

3.2.3.4 `mimoots_ofdm2_rx.py`

Dies ist das Gegenstück zu `mimoots_ofdm2_tx.py` und kann zwei Signale über zwei miteinander Synchronisierte UHD-Devices empfangen. Wahlweise können die Signale auch aus zwei verschiedene Dateien gelesen werden.

Die Parameter stimmen mit denen aus dem Programm `mimoots_ofdm_rx.py` überein.

3.2.3.5 `mimoots_ofdm2_interleaved_tx.py`

Damit es möglich ist, in einer 2x2-MIMO-Übertragung die unterschiedliche Qualität beider Kanäle miteinander zu vergleichen, sendet dieses Programm einen versetzten Datenstrom ab. Dabei wird ein OFDM-Frame über die eine Antenne abgesendet, während die andere Antenne kein Signal sendet. Nach diesem OFDM-Frame wird von der anderen Antenne ein OFDM-Frame abgesendet und die eine Antenne sendet für dieses OFDM-Frame keine Daten. Es können so keine Störungen durch die jeweils andere Antenne entstehen, da die OFDM-Symbole abwechselnd zeitversetzt gesendet werden

Die Parameter haben die selbe Wirkung wie in den vorangegangenen Programmen, außer dass an den übergebenen Dateinamen für die unterschiedlichen Antennen jeweils eine 1 oder eine 2 angehangen wird.

3.2.3.6 `mimoots_ofdm2_interleaved_rx.py`

Dies ist das Gegenstück zu `mimoots_ofdm2_interleaved_tx.py` und setzt die Zeit versetzten OFDM-Frames wieder zusammen und speichert die rohen Samples in die Datei `s1r1.gr` für die Daten von Antenne 1 und `s2r2.gr` für Antenne 2. Damit können die komplexen Werte mittels Matlab oder Octave analysiert und miteinander verglichen werden.

3.2.4 Generierung eines OFDM-Frames

Da es durchaus schwierig ist, die Funktionsweise der einzelnen Blöcke zu verstehen, soll dies an einem kleinen Beispiel erklärt werden. Mit dem Programm `mimoots_ofdm_tx.py` soll ein einfaches OFDM-Frame erstellt werden. In dem Prozess soll durch Darstellung der einzelnen Samples (welche zwischen den Blöcken weiter geleitet werden) der Aufbau und die Funktionsweise verständlich gemacht werden.

Das Programm wird mit folgenden Befehl aufgerufen:

```
mimoots_ofdm_tx.py --bits 2 --nframes 1 --nsymbols 1 --verbose
```

Der Parameter `--bits 2` wählt die Modulationsart *QPSK* aus. Die Parameter `--nframes 1` und `--nsymbols 1` legen fest, dass ein OFDM-Frame mit einem *OFDM-Symbol* darin generiert werden soll. Zusätzlich müssen noch folgende Parameter im Quelltext geändert werden:

```
fft_len = 16
cp_len = 4

occupied_carriers = ((-5, -4, -2, -1, 1, 2, 4, 5),)
pilot_carriers = ((-3, 3),)
pilot_symbols = tuple([(1, -1),])
```

Damit werden 16 *Träger* bereitgestellt, sowie die Länge eines OFDM-Symbols auf 16 Samples festgelegt. Die Länge des *CP* wird auf 4 Samples gesetzt, also $\frac{1}{4}$ des eigentlichen OFDM-Symbols. Von den 16 Trägern sollen durch die Liste `occupied_carriers` nur die darin aufgezählten *Träger* als Datenträger genutzt werden, also 8 an der Zahl. `pilot_carriers` gibt die Indizes der Pilotträger an, in diesem Fall 2 an der Zahl. `pilot_symbols` gibt die Pilotsymbole an, die auf die Pilotträger aufeinanderfolgend verteilt werden sollen. Damit erhält der Pilotträger -3 das Pilotsymbol 1 und der Pilotträger 3 das *Pilotsymbole* -1. Die *Pilotsymbole* 1 und -1 sind *BPSK*-Modulierte Werte und entsprechen den Phasen 0 und π mit einer Amplitude 1. Es können aber auch andere Modulationsarten gewählt werden. Dazu müssen einfach Symbole aus deren Wertevorrat in der Liste vergeben werden. Beispielsweise würde die Liste

```
# QPSK
pilot_symbols = tuple([(0.7071+0.7071j, 0.7071-0.7071j),])
```

der Modulationsart *QPSK* entsprechen, da deren Symbole genutzt werden.

Eine Liste mit den Werten

```
# 8-PSK
pilot_symbols = tuple([(-0.7071+0.7071j, 0.7071j),])
```

würde der Modulation *8-PSK* entsprechen, da die darin verwendeten Werte aus deren Symbolvorrat sind. Es gibt mehrere Arbeiten in denen bewiesen worden ist, dass die

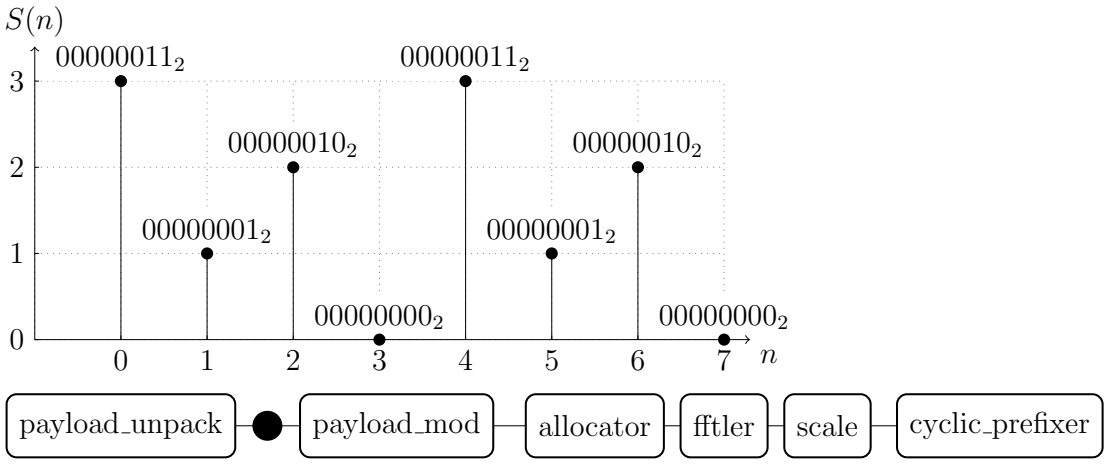

Abbildung 3.5: Die zwei Zahlen 39_{10} zerlegt in einzelne Chunks der Größe 2 Bit

Wahl bestimmter *Pilotsymbole* die Effektivität der Übertragung eines MIMO-*OFDM*-Systems steigern kann [5].

Das Programm generiert durch die Quellcodezeilen

```
data = args.nframes*[[39 for x in xrange(data_len)],]
```

und

```
(data_tosend, tags) = packet_utils.packets_to_vectors(
        data,
        packet_len_tag
)
```

eine Liste mit den Werten $[39_{10}, 39_{10}]$ oder binär ausgedrückt $[00100111_2, 00100111_2]$. Die dadurch generierten Samples sind in Abbildung 3.4 dargestellt. Der schwarze Punkt im unteren Bereich der Abbildung gibt die aktuelle Position in der Kette von Blöcken an.

Die beiden Bytes werden nun so zerlegt, dass jeweils 2 Bit (da es sich hier um eine *QPSK*-Modulation handelt, können 2 Bits mit einem komplexen Symbol codiert werden) in einem `char`-Typ gespeichert werden. Das Ergebnis zeigt Abbildung 3.5.

Als nächstes werden diesen Werten jeweils ein komplexes Symbol (über die Modulation *QPSK*) zugeordnet. Das Resultat wird in Abbildung 3.6 dargestellt.

Es ist deutlich zu erkennen, dass jedes komplexe Symbol eine Amplitude von 1 besitzt, während die Phase Werte von $\frac{3}{4}\pi, \frac{1}{4}\pi, -\frac{1}{4}\pi$ oder $-\frac{3}{4}\pi$ annimmt. Damit die Datenträger in den einzelnen Abbildungen besser von anderen Trägern unterschieden werden können, sind diese in dieser und den nachfolgenden Abbildungen blau dargestellt.

[5] Siehe [5], [6] und [7, S. 67-75] bzgl. Frank-Zadoff-Chu (FZC)-Folgen

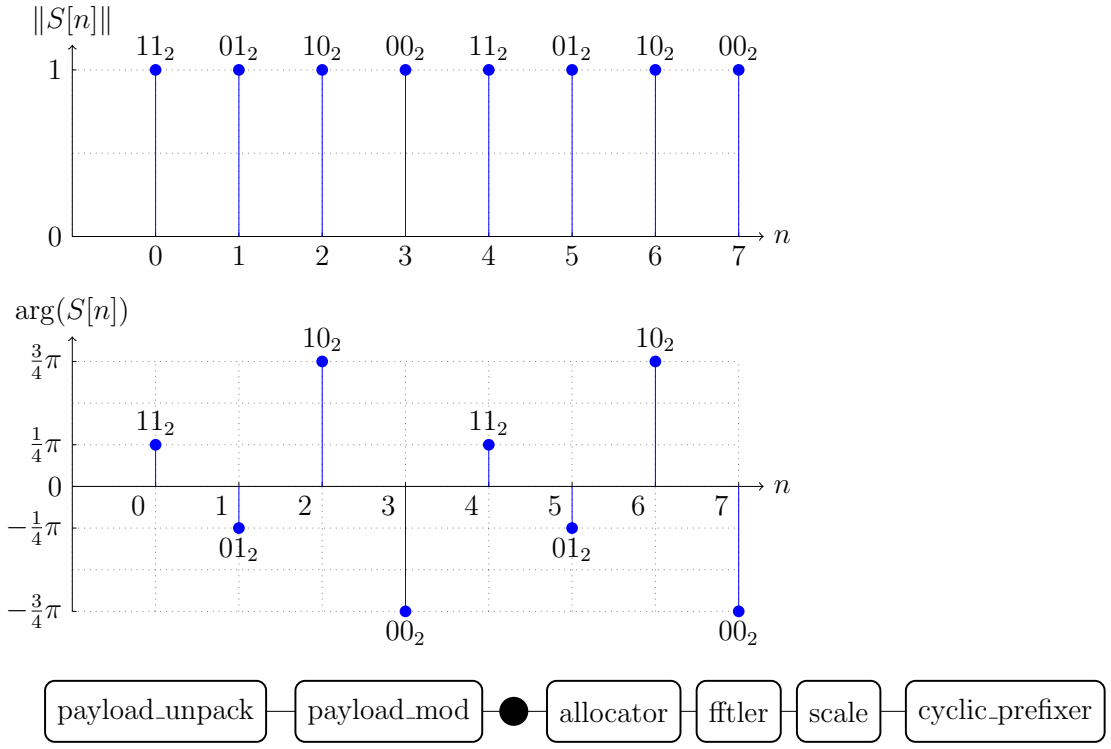

Abbildung 3.6: Die komplexen Symbole, auf die die einzelnen Chunks gemappt worden sind

Als nächstes werden die Samples (also die Datensymbole), sowie die drei anfangs beschriebenen Listen dem Block `ofdm_carrier_allocater_cvc` übergeben. Dieser platziert die komplexen Datensymbole und die Pilotsymbole auf die laut Listen (`occupied_carriers` und `pilot_carriers`) übergebenen Träger.

In Abbildung 3.7 wird ein ganzes OFDM-Frame bestehend aus drei verschiedenen Symbolen dargestellt.

Zu Beginn wurde ein *Synchronisationssymbol* für den *Schmidl-Cox-Korrelator* eingefügt, dessen *Träger* nur moduliert werden, wenn deren Indizes gerade und im *OFDM-Symbol* als Daten- oder Pilotträger verwendet werden (in der Abbildung mit der Farbe Braun dargestellt). Eben diese Bedingungen sorgen dafür, dass im Zeitbereich ein spiegelsymmetrisches Signal entsteht, dessen Struktur durch den *Schmidl-Cox-Korrelator* erkannt werden kann. Die Werte für die einzelnen *Träger* werden von der Funktion `ofdm_make_sync_word1` geliefert. Mithilfe dieses Synchronisationssymbols kann der *Schmidl-Cox-Korrelator* ein OFDM-Frame erkennen und eine grobe Frequenzkorrektur durchführen. Auffallend ist, dass die Amplitude eines jeden modulierten Trägers den Betrag $\frac{\sqrt{2}}{2} \approx 0,707$ aufweist. Dies stellt sicher, dass das *Synchronisationssymbol* einen bestimmten Energiegehalt aufweist, der für dessen Erkennung eine wichtige Rolle spielt. Die einzelnen *Träger* sind mittels BPSK moduliert.

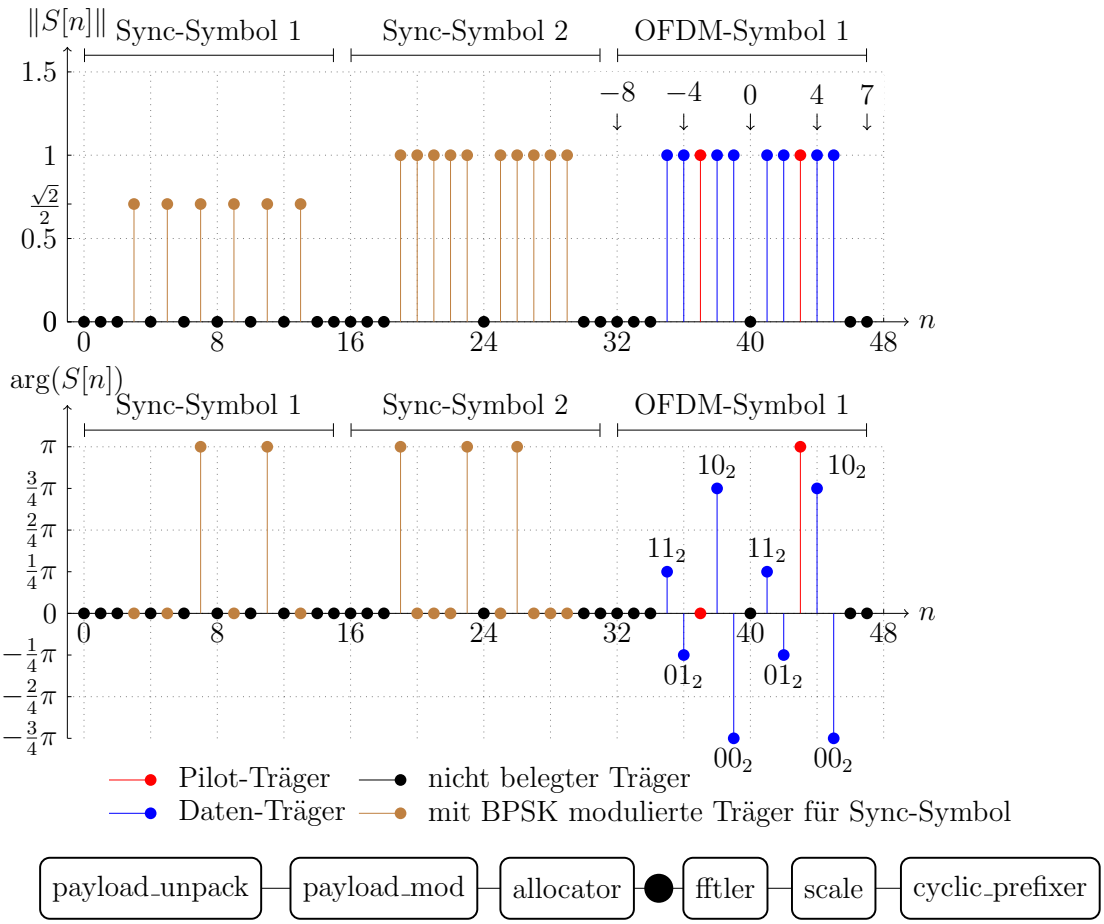

Abbildung 3.7: Ein *OFDM-Frame* bestehend aus zwei *Synchronisationssymbolen* und am Ende einem *OFDM*-Symbol, auf dessen *Träger* die einzelnen komplexen Symbole verteilt worden sind

Daraufhin folgt ein zweites *Synchronisationssymbol* für die feinere Frequenzkorrektur und das Ermitteln der Kanal-Parameter aller Träger-Frequenzen. In der Abbildung ist zu erkennen, dass jeder *Träger* im zweiten *Synchronisationssymbol* (mit der Farbe Braun markiert) nur modelliert wird, wenn dessen Index mit dem eines Daten- oder Pilotträger aus dem *OFDM-Symbol* übereinstimmt. Dieser wird dann mittels BPSK moduliert, wobei die Amplituden den Betrag 1 haben, da hier auf keinen konstanten Energiegehalt geachtet werden muss.

Zum Schluss folgt das eigentliche OFDM-Frame mit den einzelnen komplexen Daten- (in der Farbe Blau) und Pilotsymbolen (in der Farbe Rot). Die einzelnen *Träger* besitzen Indizes aus den Wertebereich $(-8, 7)$, welche im oberen Bereich zur Darstellung der Amplituden abgebildet ist. Auch hier betragen sämtliche Amplituden der modulierten *Träger* den Wert 1. Im unteren Diagramm sind die einzelnen Phasen der *Träger* dargestellt. Die Pilotträger verwenden dabei die Werte $(0, \pi)$ entsprechend der *BPSK*, während

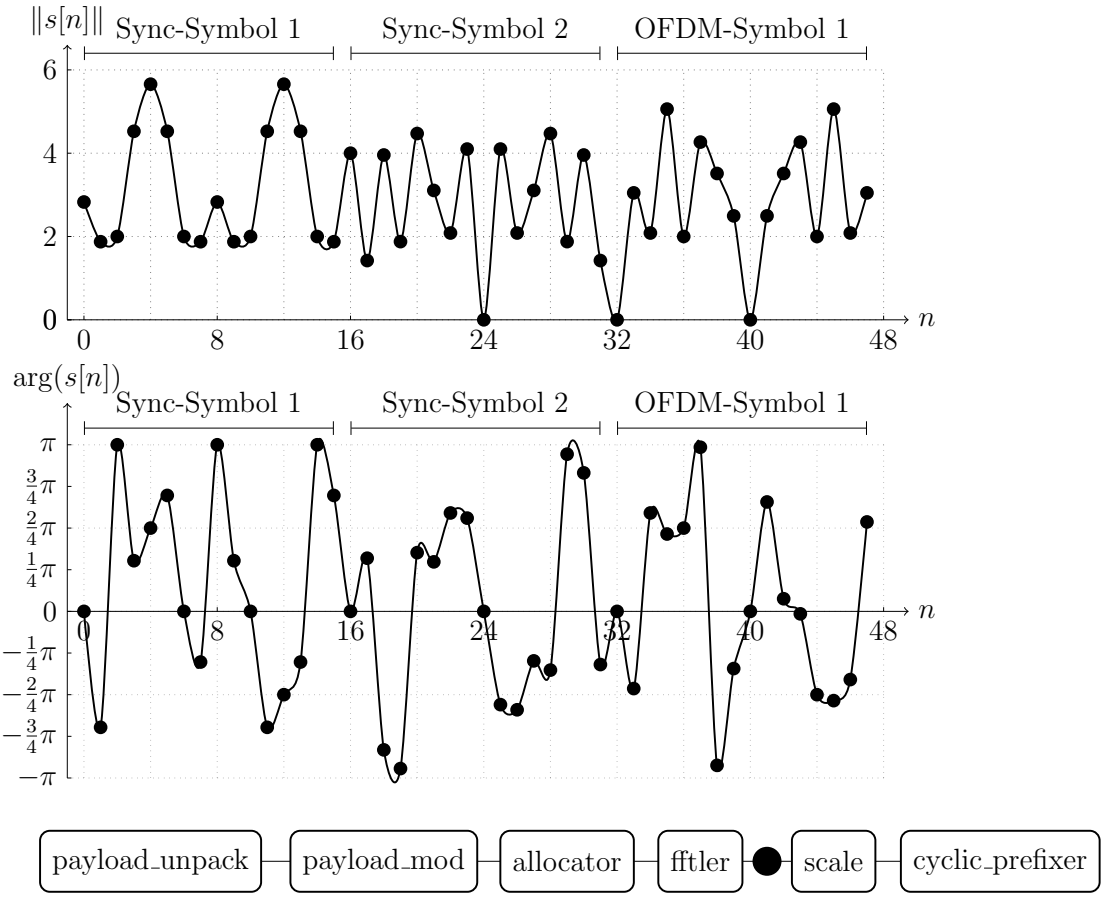

Abbildung 3.8: Das Basisband-Signal im Zeitbereich erstellt durch die

die Datenträger aufgrund der *QPSK* jeweils einen der Werte $(-\frac{3}{4}\pi, -\frac{1}{4}\pi, \frac{1}{4}\pi, \frac{3}{4}\pi)$ annehmen. Die einzelnen komplexen Symbole sind ihren Datenträgern binär zugeordnet. Es ist zu erkennen, dass im *OFDM-Symbol* die einzelnen *Träger* mit den Indizes moduliert worden sind, welche mit denen aus der Liste `occupied_carriers` übereinstimmen.

Die einzelnen Werte werden nun der *IDFT* übergeben und damit in den Zeitbereich transformiert. Als Ergebnis entsteht ein Basisband-Signal bestehend aus einzelnen Samples, wie in Abbildung 3.8 zu erkennen ist. Allerdings übersteigt der Betrag des Signals mehrfach den Wert 1 und ist damit nicht dafür geeignet dem UHD-Device direkt übergeben zu werden, da dieses größere Werte einfach abschneidet und damit das Signal verfälscht.

Um dies zu vermeiden, wird das Signal normiert. Es wird immer eine Gruppe von Samples, die durch die *IDFT* zu einem Synchronisations- oder *OFDM-Symbol* gehören, auf deren größten Wert normiert. In Abbildung 3.9 ist das Ergebnis zu erkennen. Keine Amplitude übersteigt mehr den Wert 1.

Zuletzt muss noch der *CP* hinzugefügt werden. Dies ist in Abbildung 3.10 abgebildet.

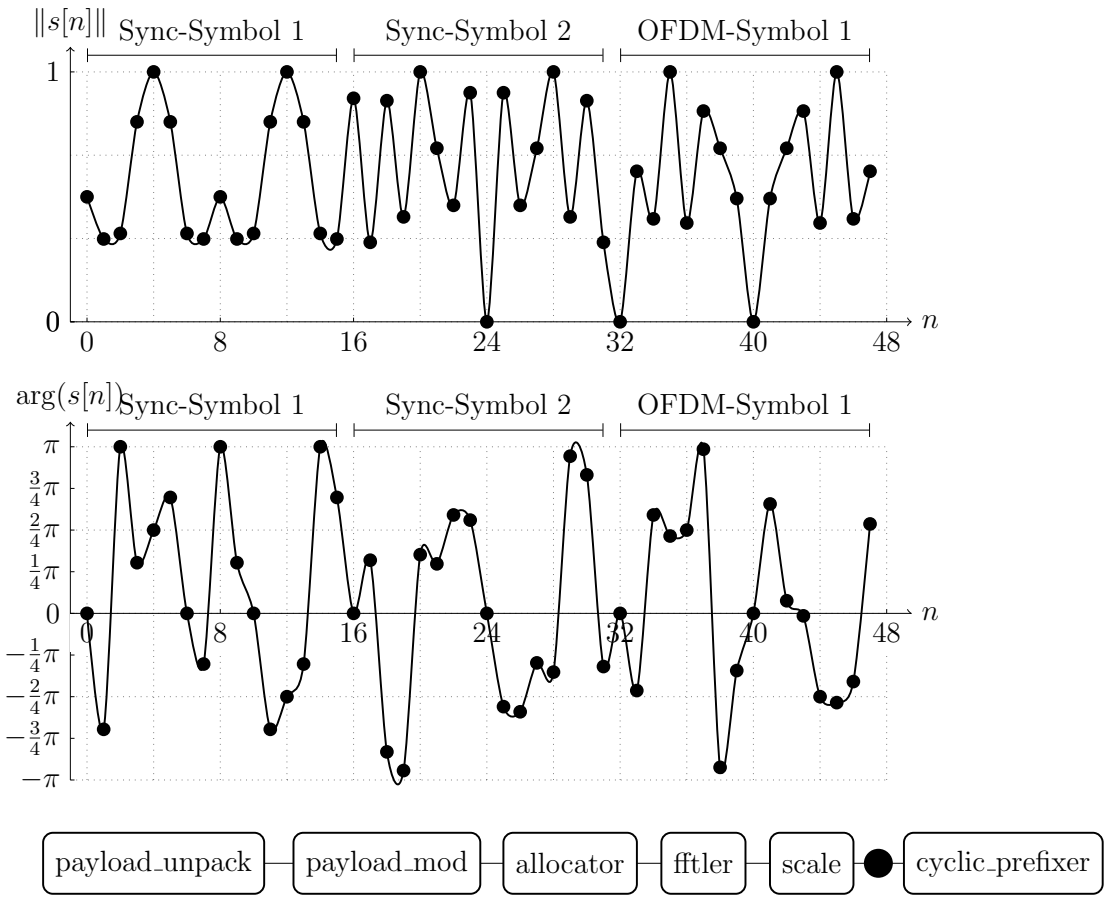

Abbildung 3.9: Basisbandsignal mit normierten *OFDM*-Symbolen

Es ist zu erkennen, wie das gesamte Signal um $\frac{1}{4}$ der ursprünglichen Sampleanzahl länger wird. Dies entspricht genau dem gewählten Verhältnis zwischen dem *CP* und dem OFDM-Symbol.

In dieser Form ist das Basisband-Signal nun in der Lage, über ein UHD-Device in die Umgebung abgestrahlt zu werden.

3.3 Implementierung, Probleme und ihr Einfluss auf Design-Entscheidungen

3.3.1 Generelle Prinzipien

GNU Radio bietet die Möglichkeit Signalblöcke mit dem Prinzip der *Testgetriebenen Entwicklung* mittels Unit Tests zu erstellen. Dazu werden Bedingungen definiert, die der Block erfüllen muss (bspw. muss dieser bei bestimmten Eingangsdaten wiederum festgelegte Ausgangsdaten erzeugen). Dazu gibt es für jeden Block eine Datei, die den

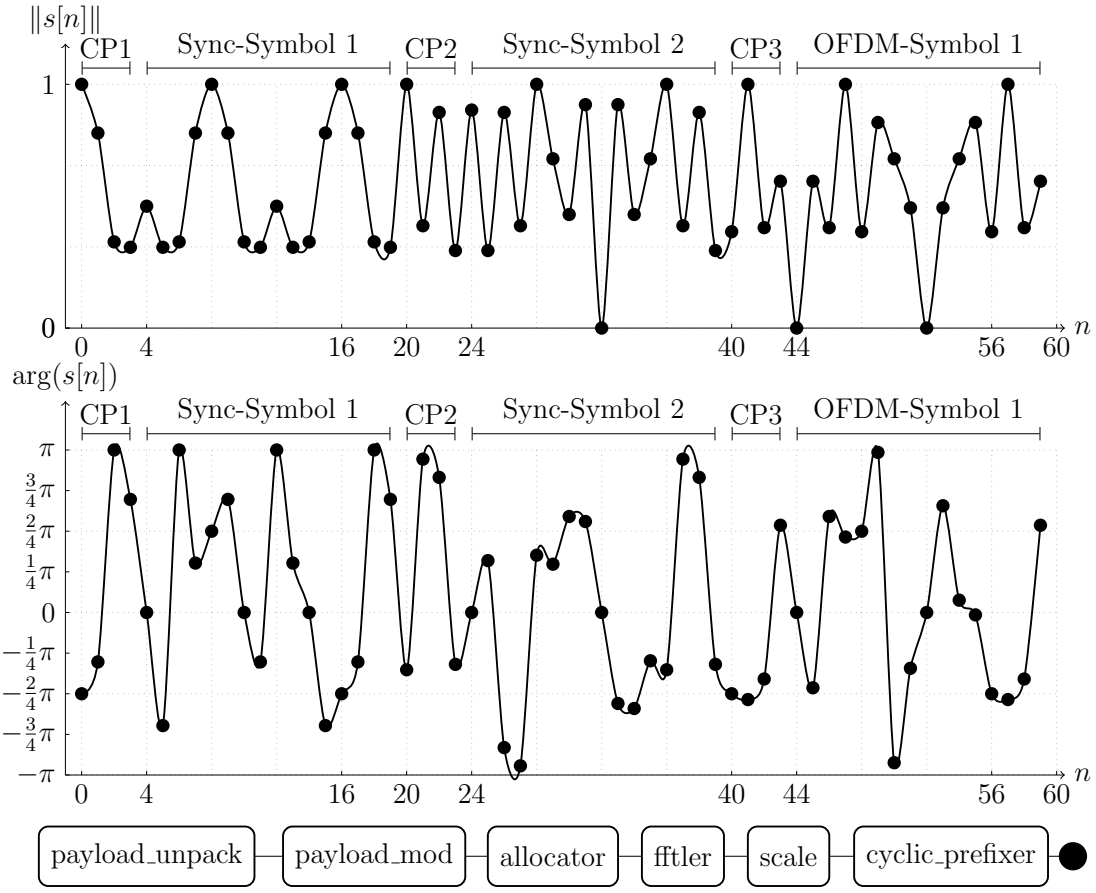

Abbildung 3.10: Normiertes Basisbandsignal mit einem *CP* in jedem *OFDM*-Symbol

Unit Test definiert und ausgeführt werden muss.

Über die definierten Bedingungen wird die Funktionsweise definiert. Entwickelt wird der Block solange, bis alle definierten Tests erfolgreich sind. Dann ist die Entwicklung beendet. Dieses Prinzip wurde in dieser Arbeit übernommen, weil es einige wichtige Vorteile bietet:

- falscher Programmquelltext kann schnell identifiziert werden, da Fehler sofort durch die Tests auffallen
- andere Programmierer können sich schnell in den Quelltext einarbeiten, da die Tests zugleich die Funktionsweise des Blocks definieren (wie eine Art Dokumentation)

Zudem wurde in dieser Arbeit nach dem Programmierstil programmiert, der von *GNU Radio* vorgeschlagen wird, was den Quelltext wiederum mit dem Rest des Frameworks vereinheitlicht und damit besser lesbar macht.

3.3.2 Skalierung der Amplituden für das Device USRP2 N210

Beim Testen des Blocks `ofdm_txrx.py` ergab sich das Problem, dass die Signalwerte oder besser gesagt die komplexen Werte am Ausgang in ihren Amplituden weit über den Wert 1 hinausgingen. Das UHD-Device USRP2 N210 verarbeitet jedoch komplexe Signalwerte nur bis zu einer Amplitude von 1. Amplituden größer als 1 wurden automatisch auf den Wert 1 beschnitten, was zu einem Verfälschen des Amplitudenverhältnisses im *OFDM-Symbol* untereinander und damit zu anderen Phasen führte. Zusätzlich wurde das Signal so verfälscht, dass sich das Spektrum in der Art änderte, dass eine *Orthogonalität* zwischen den einzelnen Trägern nicht mehr gewährleistet werden konnte. Um dieses Problem zu lösen, musste der Block `ofdm_scale_symbol_vcvc` geschrieben werden, der die Amplitudenwerte so skaliert, dass keine der Amplituden mehr über den Wert 1 kam.

Dies war jedoch mit einigen Schwierigkeiten verbunden. Über das gesamte *OFDM-Frame* hinweg musste eine absolute Bezugsamplitude zur Verfügung stehen, auf die sich sämtliche im *OFDM-Frame* befindlichen Werte beziehen können, um ihnen einen Wert zuweisen zu können. Nur so können alle Amplituden im gleichen Maße skaliert werden, ohne dass sie ihr Verhältnis in einem *OFDM-Frame* untereinander verlieren. Der Block müsste also über ein einzelnes *OFDM-Symbol* hinweg alle Werte kennen, um den größten Amplitudenwert als absoluten Wert verwenden zu können. Dazu bräuchte es einen Zwischen-Speicher, ein Gedächtnis, was mit einem hohen Verwaltungsaufwand einhergehen würde.

Einfacher wäre es nur die Verhältnisse der Amplituden in einem *OFDM-Symbol* untereinander zu erhalten, also für jedes *OFDM-Symbol* den darin größten befindlichen Wert als Absolutwert für eben dieses *OFDM-Symbol* zu verwenden und darauf alle Werte zu normieren.

Dies verfälscht zwar die Amplitudenverhältnisse der Vektoren untereinander, stellt allerdings kein Problem dar, da laut Aufgabenstellung nur die Modulationsart *PSK* unterstützt werden soll, die allein die Phase als Informationsträger moduliert, nicht jedoch die Amplitude. Bleibt das Verhältnis aller Amplitudenwerte innerhalb eines Vektor konstant, so wird auch die Phase nicht verfälscht.

Es ist jedoch wichtig zu beachten, dass je nach Wahl des Skalierungsverfahren der *SNR*[6] unterschiedlich berechnet werden muss. Sollten die Amplituden bezogen auf einen ganzen *OFDM-Frame* normiert werden, gilt auch der *SNR* für den gesamten *OFDM-Frame*. Wird zur Normierung nur ein *OFDM-Symbol* verwendet, so gilt auch der berechnete *SNR* nur für dieses *OFDM-Symbol* und kann nicht ohne weiteres mit einem *SNR* eines anderen *OFDM-Symbol* verglichen werden.

3.3.3 Samples aus dem Buffer des USRP2 N210 beim Aktivieren als Empfänger

Wird das UHD-Device USRP2 N210 durch den Code

[6] Signal-to-Noise Ratio

```
self.u = uhd.usrp_source(
    ",".join(("", "")),
    uhd.stream_args(
        cpu_format="fc32",
        channels=range(1),
    ),
)
```

geladen, in einen Signalflussgraphen als Quelle eingebunden und dann ausgelesen, werden als erstes um die 5000 Samples aus dem Speicher des Geräts geliefert. Dies ist problematisch, da der *Schmidl-Cox-Korrelator* in diesem Fall ein OFDM-Frame zu detektieren glaubt und dementsprechend den nachfolgenden Blöcken Daten weiterleitet, die nicht in diesem Moment gesendet worden sind.

Das Problem wurde durch Verwendung des Blocks `skiphead` in den Griff bekommen. Dieser Block erhält den Parameter `nitems_to_skip`, mit dem angegeben werden soll, wie viele Werte zu Beginn verworfen werden sollen. Die dahinter liegenden Werte werden ganz einfach weitergeleitet.

Aus Programmierer-Sicht ist es sinnvoll, diesen Block nicht mit in den Python-Block `ofdm_symbol_mapper_bc` aufzunehmen. In diesem Fall handelt es sich um eine spezielle technische Eigenheit in Kombination mit dem UHD-Device USRP2 N210 und gehört damit nicht zu der allgemeinen Aufgabe, die dieser Python-Block erfüllen soll. Eine Aufnahme hätte zur Folge, dass der Python-Block unnötig komplizierter und umfangreicher werden würde.

3.3.4 Häufiges Ignorieren des ersten OFDM-Frames durch den *Schmidl-Cox-Korrelator*

In den meisten Fällen kommt es vor, dass der *Schmidl-Cox-Korrelator* das erste OFDM-Frame einfach ignoriert. Nur hin und wieder erkennt er das erste Frame, ohne dass darin ein Muster zu erkennen wäre. So kann auch schlecht mit Mitteln der Programmierung darauf reagiert werden. Der *Schmidl-Cox-Korrelator* detektiert ein OFDM-Frame mithilfe einer Metrik, die den Energiegehalt mehrerer aufeinander folgender Bereiche des empfangenen Signals vergleicht und so bestimmen kann, wann ein OFDM-Frame eintrifft und wann nicht.

Sobald ein OFDM-Frame erkannt worden ist, werden auch alle darauf folgenden Frames, zwischen denen keine Pause liegt, erkannt. Die Ursache dieses Phänomens liegt vermutlich im gleichbleibenden Energiespiegel des Signals, der vor dem ersten OFDM-Frame deutlich geringer ist.

Die eigentliche Ursache für dieses Problem wurde in dieser Arbeit nicht gefunden. Dies brachte allerdings keine großen Probleme mit sich. Den Daten wurde einfach eine Art Dummy-OFDM-Frame ohne Inhalt vorangestellt, welches der Korrelator in fast allen Fällen ignorierte. Sollte das Dummy-Frame einmal doch erkannt worden sein, wurde dieser Fall einfach ignoriert. Dies geschah jedoch so selten, dass es kaum Auswirkungen auf den Arbeitsablauf hatte.

3.3.5 Erhöhung der Fehlerrate bei der Nutzung von 8-PSK

Es hat sich herausgestellt, dass die Nutzung der Modulationstechnik *8-PSK* mit einer erhöhten Fehlerrate von etwa 2% einhergeht. Selbst die Datenübertragung über ein Kabel ließ eine zwar geringere Fehlerrate ermitteln, allerdings sollte bei der Übertragung per Kabel keine Fehlerrate auftreten. Die hohe Fehlerrate kann also nicht einfach durch die Störung im Kanal erklärt werden.

Das Zwischenspeichern des Basisband-Signals in eine Datei und darauf folgendes Auslesen dieser Daten durch das Empfängerprogramm ergab jedoch keinerlei Übertragungsfehler.

Somit kommen nur zwei verschiedene Gründe in Betracht:

Zum einen kann es sich um ein Mapping-Problem handeln von den Bit-Daten zum komplexen Datensymbol durch das *constellation mapping*. Bei einer *BPSK* und *QPSK* treten keine Übertragungsfehler beim Senden der Daten über Kabel auf. Bei diesen beiden Modulationsarten ist die Anzahl der zu verarbeitenden Bits einmal $M_{BPSK} = ld(2) = 1$ und $M_{QPSK} = ld(4) = 2$. Die Länge eines `char`-Typs in *GNU Radio* entspricht immer einem Vielfachen dieser beiden Größen. Bei *8-PSK* jedoch beträgt die Anzahl an zu verarbeitenden Bits $M_{8\text{-}PSK} = ld(8) = 3$. Die Länge eines `char`-Typs ist damit nicht mehr ein Vielfaches dieser Größe. Erst 3 `char`-Typs ergeben ein Vielfaches von $M_{8\text{-}PSK} = 3$. Dies kann dazu führen, dass einige Werte verfälscht werden.

Zum anderen kann aber auch der *PAPR*[7] der Grund für eine erhöhte Fehlerrate sein[8]. So kann es passieren, dass eine Spitze im *OFDM-Symbol* entsteht, die deutlich größer ist als die Amplituden der restliche Werte. Dies führt dazu, dass der gesamte Wertebereich zur Darstellung der Spitze genutzt werden muss, woraufhin die kleineren Werte nicht mehr gut genug aufgelöst werden können. Beim Empfänger kommt nur noch ein unklares Signal an, dass nicht mehr ohne Probleme dekodiert werden kann.

Da die Fehlerrate auch in diesem Fall bei etwa 2% blieb, hatte das Problem keine nennenswerten Auswirkungen. Da andere Probleme und Aufgaben Vorrang hatten, erschien es sinnvoll es dabei zu belassen. Trotzdem sollte es hier nochmal angesprochen werden, damit hierauf aufbauende Arbeiten sich dieser Problemquelle bewusst sein können.

3.3.6 Das Design vom Block `ofdm_extract_frame_vcvc`

Anders als bei dem Block `ofdm_scale_symbols_vcvc`, dessen Länge der Eingangsdaten der der Ausgangsdaten entspricht (`1:1`) und somit ein *Synchronous Block*[9] ist, ist das Verhältnis der Länge der Eingangs- und Ausgangsdaten bei diesem Block variabel. In einem Fall werden gar keine Daten weitergegeben im anderen Fall (wenn ein Flag signalisiert, dass ein OFDM-Frame eingetroffen ist) wird ein OFDM-Symbole der Länge `fft_len` mit jedem Schritt ausgegeben. Das Verhältnis der Anzahl an Items am Eingang

[7] Peak-to-Average Power Ratio
[8] Siehe [1, S. 209]
[9] Siehe `http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide#Synchronous-Block`

und am Ausgang steht nicht fest und ist variabel. Damit handelt es sich um einen *Basic Block*[10].

Damit müssen viele Funktionen, die normalerweise bei anderen Block-Typen von *GNU Radio* übernommen werden, vom Entwickler selbst implementiert werden. Das ist fehleranfälliger, da der Code komplizierter und umfangreicher ist. Allerdings war dies leider nicht zu vermeiden, da die nötige Funktionsweise anders nicht umgesetzt werden konnte.

Außerdem war es nicht möglich, einen bereits vorhandenen Block aus *GNU Radio* zu finden, der diese Aufgabe übernehmen könnte. Zur Auswahl stand der Block `ofdm_sampler`, welcher eine ähnliche Aufgabe im OFDM-Beispielprogramm im Block `gr-digital` hatte. Dieser Block entfernte die *Synchronisationssymbole* eines jeden ihm übergebenen OFDM-Symbols und gab dieses dann weiter an den nächsten Block. Leider konnte nicht konfiguriert werden, wie viele OFDM-Symbole hintereinander extrahiert werden sollen, was in dieser Arbeit, wenn möglich, variabel konfigurierbar sein sollte. Auch die Dokumentation lieferte keine weitere Aussagen darüber, wie dieser Block intern arbeitet und was seine Aufgabe ist. Stattdessen lieferte sie folgenden Satz:

> does the rest of the OFDM stuff

Dadurch war es unvermeidlich, den Quellcode des Blocks genau zu analysieren.

Die Implementierung des Blocks nahm einen großen Zeitraum in Anspruch, da die komplizierte Struktur fehleranfällig war. Das Programmier-Modell *Testgetriebene Programmierung* hat die Entwicklung jedoch enorm vereinfacht und effektiver gemacht. Jede Quellcode-Änderung konnte getestet werden. So konnten sich bei jeder kleineren Anpassung nicht gleich ein Fehler einschleichen.

Um *GNU Radio* mitzuteilen, wie viele Items der Block bei einem Aufruf aufnimmt und wieder abgibt, wird im Konstruktor folgender Code ausgeführt:

```
ofdm_extract_frame_cvc_impl::ofdm_extract_frame_cvc_impl(
  size_t fft_len,
  size_t cp_len,
  size_t nsymbols_per_ofdmframe,
  bool info
)
  : gr::block("ofdm_extract_frame_cvc",
      gr::io_signature::make2(2, 2, sizeof(gr_complex), sizeof(char)),
      gr::io_signature::make(1, 1, fft_len*sizeof(gr_complex))
  ),
  d_fft_len(fft_len),
  d_cp_len(cp_len),
  d_nsymbols_per_ofdmframe(nsymbols_per_ofdmframe),
  d_current_symbol(0),
  d_state(STATE_FIND_TRIGGER),
  d_info(info)
```

[10] Siehe `http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide#Basic-Block`

```
{
}
```

Mithilfe der Funktion `gr::io_signature::make` und `gr::io_signature::make2` werden die Anzahl an zu verarbeitenden Items festgelegt. Damit werden zum einen ein Stream bestehend aus dem Typ `gr_complex` aufgenommen, zum anderen Vektoren bestehend aus `gr_complex`-Werten der Länge `fft_len` wieder ausgegeben. Damit übernimmt dieser Block gleich die Aufgabe eines *parallelizer*. Nur so kann sich dieser Block in den übernommen Signalflussgraphen aus `ofdm_txrx.py` eingliedern.

Tagged Streams finden an dieser Stelle noch keine Anwendung. Erst der Block `ofdm_chanest_vcvc` führt Tags in die zu verarbeitenden Daten ein.

Der zentrale Kern des Blocks besteht in der Funktion `general_work()`:

```
int
ofdm_extract_frame_cvc_impl::general_work(
  int noutput_items,
  gr_vector_int &ninput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items
)
{
  const gr_complex *data = (const gr_complex *) input_items[0];
  const char *flags = (const char *) input_items[1];
  gr_complex *out = (gr_complex *) output_items[0];

  size_t nitem_consumed = 0;
  size_t nitem_ret = 0;

  switch(d_state) {
  case STATE_FIND_TRIGGER: {
    size_t nitems = std::min(
      std::min(ninput_items[0], ninput_items[1]),
      noutput_items
    );

    nitem_consumed = nitems;
    nitem_ret=0;

    for (size_t item_index = 0; item_index < nitems; item_index++) {
      if (flags[item_index] == 1) {
        nitem_consumed=item_index+1;

        d_state=STATE_SYMBOL;
        break;
```

```
      }
    }

    break;
  }
  case STATE_SYMBOL: {
    std::memcpy(out, &data[d_cp_len], d_fft_len*sizeof(gr_complex));

    if (d_current_symbol < d_nsymbols_per_ofdmframe-1) {
      d_current_symbol++;

      nitem_consumed = (d_fft_len+d_cp_len);
      nitem_ret = 1;
    } else {
      d_current_symbol = 0;
      d_state = STATE_FIND_TRIGGER;

      nitem_consumed = 0;
      nitem_ret = 1;
    }

    break;
  }
  }

  consume_each(nitem_consumed);
  return nitem_ret;
}
```

Durch die `switch`-Anweisung wird der Automat realisiert und entschieden, welcher Code abhängig vom derzeitigen Zustand abgearbeitet wird.

Für `STATE_FIND_TRIGGER` werden aus Eingang 1 alle übergebenen Werte (deren Anzahl angegeben wird durch das Minimum der Parameter `ninput_items[0]` und `ninput_items[1]`) durchgegangen und nach einem gesetzten Flag gesucht. Wird ein Flag gefunden, werden für beide Eingänge alle vorhergehenden Items als gelesen markiert (`consume_each(nitem_consumed)`), um beide Eingänge synchron zu halten. Der Index des Flags aus Eingang 1 stimmt mit dem Index aus Eingang 0, an dem sich das erste OFDM-Frame befindet, überein. Zusätzlich wird der Zustand auf `STATE_SYMBOL` gesetzt, damit beim nächsten Aufruf dieser Funktion die OFDM-Symbole extrahiert werden können. Andernfalls werden alle gelesenen Items als gelesen markiert und die Funktion beendet ohne den Zustand zu ändern, woraufhin das Suchen des Flags im nächsten übergebenen Datenstrom beginnt.

Für `STATE_SYMBOL` wird aus Eingang 0 ein OFDM-Symbol der Länge `fft_len` in den Ausgangsspeicher kopiert. Der *CP* wird dabei übersprungen und nicht mit kopiert:

```
std::memcpy(out, &data[d_cp_len], d_fft_len*sizeof(gr_complex));
```

Der Zähler kopierter OFDM-Symbole wird um 1 erhöht und die bearbeiteten Items als gelesen markiert, damit *GNU Radio* sie aus dem Eingang löscht und beim nächsten Aufruf der Funktion das nächste *OFDM-Symbol* am Anfang des Eingangs 0 steht:

```
d_current_symbol++;
nitem_consumed = (d_fft_len+d_cp_len);
```

Sobald die Anzahl der kopierten OFDM-Symbole die Anzahl der zu kopierenden OFDM-Symbole erreicht hat, wird wieder zum Zustand `STATE_FIND_TRIGGER` gewechselt.

Die Dokumentation von *GNU Radio* hat bei den ersten Schritten gute Dienste geleistet. Als es jedoch notwendig war, tiefer in die Funktionsweise des Frameworks einzutauchen, fehlten die Details, sodass die korrekte Funktionsweise des Blocks über *trial and error* zustande kam, was zeitraubend war.

Ein Beispiel dafür ist dieser Quelltext:

```
void
ofdm_extract_frame_cvc_impl::forecast(
  int noutput_items,
  gr_vector_int &ninput_items_required
)
{
  // Is there a good init-value to avoid hidden bugs?
  size_t items_required = 1;
  size_t ofdmsymbol_len = d_fft_len + d_cp_len;

  switch(d_state) {
  case STATE_FIND_TRIGGER: {
    items_required = ofdmsymbol_len;
    break;
  }
  case STATE_SYMBOL: {
    items_required = ofdmsymbol_len;
    break;
  }
  default: {
    throw std::runtime_error("invalid state");
    break;
  }
  }

  size_t ninputs = ninput_items_required.size();
  for (size_t inputs_index = 0; inputs_index < ninputs; inputs_index++) {
```

```
    ninput_items_required[inputs_index] = items_required;
  }

  return;
}
```

GNU Radio ruft die Funktion `forecast()` regelmäßig auf um zu erfahren, wie das Verhältnis der Eingangs- und Ausgangsdatenlänge ist. Je nach Zustand gibt der Block an *GNU Radio* das richtige Verhältnis zurück. Allerdings wurde in der Dokumentation nicht detailliert erklärt, wie genau diese Funktion mit dem Rest des Frameworks interagiert. Auch bei anderen oder falschen Werten lief der Block bei kleineren Testdatenmengen. Da scheinbar kein Fehler auftrat, fiel diese Funktion bei der Entwicklung in den Hintergrund.

Oftmals lieferte *GNU Radio* nur unbestimmte und unscharfe Fehlermeldungen, die mit den eigentlichen Fehler nicht viel gemeinsam hatten, was die Fehlersuche erschwerte. Häufig gab es *Buffer-Overflows*, da sich der Programmierer selber um die Speicherverwaltung kümmern musste. Dies hat das Framework sicher beschleunigt, ließ die Zeit für die Entwicklung aber länger werden.

Um die zwei Zustände möglichst effektiv, aber auch einfach implementieren zu können, wurde das Prinzip der *state mashine* gewählt. Je nach Bedingung wechselt der Block in verschiedene Zustände, die seine Arbeitsweise bestimmen.

3.3.7 `ofdm_extract_frame_vcvc` und der Thread-Per-Block Scheduler

Das größte und schwierigste Problem trat auf, als in der Testdatei die Menge an vorgegebenen Eingangsdaten deutlich erhöht wurde. Nach kurzer Zeit meldete *GNU Radio*, dass der Speicherbereich, der als Ausgabespeicher zur Verfügung stand, zu klein wäre, um die generierten Daten weiterzugeben. Das Problem ließ sich jedoch durch keinen Mechanismus (bspw. Vergrößern des angeforderten Speicherbereichs) lösen.

Die Fehlermeldung lieferte der Thread-Manager von *GNU Radio*. Da Threads oftmals eine Fehlerquelle darstellen können, vor allem bei nicht nachvollziehbaren Fehlern, die in keinem Zusammenhang mit der eigentlichen Situation stehen, war der nächste Schritt, den *GNU Radio*-Scheduler zu konfigurieren.

Es gibt zwei Arten von Schedulern bei *GNU Radio*:

- TBD - Thread-Per-Block scheduler
- STS - Single-Threaded-Scheduler

Ersterer weist jedem Block im Signalflussgraphen einen eigenen eigenständigen Thread zu. Letzterer lässt den gesamten Signalflussgraphen durch nur einen einzelnen Thread abarbeiten.

Durch das Ausführen des Befehls

```
$ GR_SCHEDULER=STS ./my-gnuradio-application.py
```

konnte der Test reibungslos durchlaufen.

Der Befehl

```
$ GR_SCHEDULER=TBD ./my-gnuradio-application.py
```

jedoch führte wieder zur ursprünglichen Fehlermeldung.

Die Lösung lag schlussendlich darin, jedes *OFDM-Symbol* eines OFDM-Frames einzeln pro Aufruf der Funkion `general_work()` durch den Block auszugeben. In einer früheren Version wurde der gesamte OFDM-Frame mit einem Aufruf der Funktion `general_work()` zum Ausgang weitergereicht. Nach der Änderung musste `general_work()` jedes *OFDM-Symbol* separat ausgeben und damit mehrere Male aufgerufen werden, bevor das OFDM-Frame abgearbeitet worden ist. Die Funktion `forecast()` musste ebenfalls angepasst werden. Danach funktionierte der Test einwandfrei.

Das Schwierige an diesem Problem war, den kausalen Zusammenhang herzustellen. Die Dokumentation lieferte dazu keine Details. Es scheint so zu sein, dass der *GNU Radio*-Scheduler TBD anhand der Werte, die `forecast()` liefert, die Abarbeitung der Blöcke skaliert und als grobe Richtwerte versteht. Sind die Werte zu groß oder falsch, dann funktioniert die Skalierung nicht mehr einwandfrei und es wird eben dieser schwer verständliche Fehler produziert.

3.3.8 Notwendigkeit einer Pause, damit der Signalflussgraph abgearbeitet werden kann

Das Problem liegt darin, dass der Signalflussgraph noch nicht fertig ist, wenn das eigentliche Python-Programm bereits beendet wird. Dieser gibt dem Programm sozusagen keine Rückmeldung darüber, ob die Daten fertig verarbeitet und gesendet worden sind. Beim Speichern der generierten Daten in eine Datei tritt dieses Problem nicht auf. Werden die Daten jedoch an ein UHD-Device zum Senden geleitet, dann ist das UHD-Device immer noch damit beschäftigt, die Signale über die Antenne abzustrahlen, obwohl das Python-Programm bereits sein Ende erreicht hat.

Vermutlich nutzt die Library zum Ansteuern des UHD-Device Thread-basierte Routinen, die nebenläufig zum eigentlichen Python-Programm sind.

Folgender Befehl nach der Anweisung

```
tb.begin()
```

löst das Problem:

```
time.sleep(5)
```

wobei diese Anweisung ein Warten von fünf Sekunden veranlasst. Die Zeit zum Warten wurde durch Ausprobieren soweit verkleinert, bis die empfangenen Daten beim Empfänger unvollständig waren. Daraufhin wurde sie wieder erhöht und noch ein Sicherheitsabstand von einigen Sekunden aufaddiert, um eine vollständige Übertragung sicherzustellen. Dies gewährleistet, dass das Python-Programm erst sein Ende erreicht hat, wenn sämtliche Daten über das UHD-Device bereits abgestrahlt worden sind. Es geht hier

also nicht um eine Synchronisation des Signalflussgraphen mit dem Python-Programm, sondern nur darum das Ende des Python-Programms soweit hinauszuzögern, bis auch das UHD-Device fertig mit Senden ist.

Für größere zu übertragene Datenmengen muss die Wartezeit dementsprechend erhöht werden. Diese ist abhängig von der vom UHD-Device genutzten Samplingrate und der zu übertragenen Datenmenge und kann durch Ausprobieren ermittelt werden.

4 Nutzungs-Szenarien

4.1 Messaufbau und Umgebung

Die Messungen wurden allesamt in der Weise durchgeführt, dass Sender und Empfänger nie weiter als 5 Meter voneinander entfernt standen. Zum einen gab es eine direkte Sichtlinie zwischen Sender und Empfänger (Verbindung mit Line-of-sight), zum anderen gab es den Fall, dass zwischen Sender und Empfänger ein oder mehrere Gegenstände standen (Verbindung mit None-line-of-sight), die die Übertragung behindert haben.

Abbildung 4.1 zeigt eine Skizze des Laborraums. Darin befinden sich mehrere Möbel wie Tische, Stühle und Schränke. Alles, was die Übertragung behindern könnte, ist durch eine durchgezogene Linie gezeichnet. Alles andere, wie bspw. Tische, was die Übertragung nicht behindert, wird durch gestrichelte Linien dargestellt.

Der Sender ist durch den Punkt S gekennzeichnet, während verschiedene Positionen für den Empfänger durch R_1 und R_2 kenntlich gemacht sind.

Die Strecke $S - R_1$ z.B. ergibt eine Verbindung mit Line-of-Sight, da nichts zwischen beiden Antennen steht. $S - R_2$ hingegen bildet eine Verbindung mit None-line-of-Sight, da hier zwei Flachbildschirme und ein Schrank die Sicht beider Antennen aufeinander stören.

Dieses Szenario beschreibt die einfache Nutzung jeweils einer Antenne für den Sender und den Empfänger. Es dient zur Überprüfung der Grundfunktionalität, um Daten von einer Antenne zu einer anderen Antenne zu übertragen. Dabei wurde die Übertragungsstrecke $S - R_1$ gewählt. Mit allen Modulationsarten (*BPSK*, *QPSK* und *8-PSK*) wurde hierbei die Fehlerrate von 2% nicht überschritten. Bei *BPSK* und *QPSK* traten sogar keinerlei Übertragungsfehler auf.

4.2 SISO-Datenübertragung

Es wird jeweils mit einer Antenne auf Sender- und Empfängerseite gearbeitet.

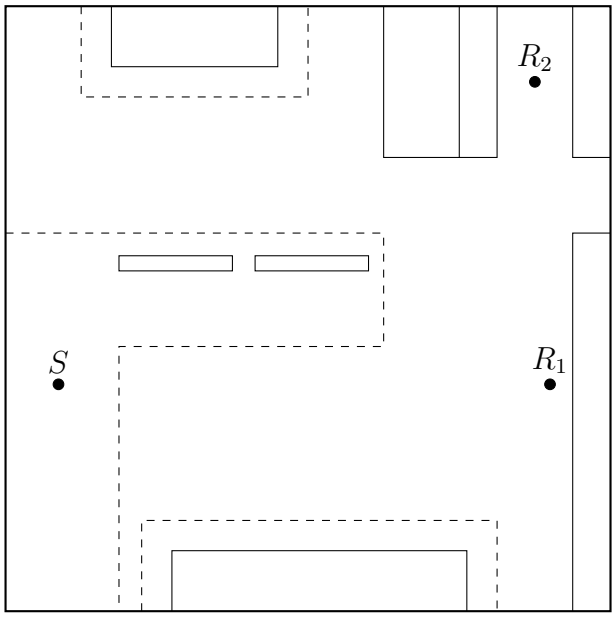

Abbildung 4.1: Der grobe Aufbau des Versuchraums

4.3 Gleichzeitige SIMO-Datenübetragung zur Messung der Diversität

In diesem Szenario existiert eine Antenne zum Senden und zwei miteinander synchronisierte Antennen zum Empfangen. Es gibt also ein vom Empfänger abgestrahltes Original-Signal und zwei durch den Kanal veränderte Kopien, die jeweils eine der beiden Antennen beim Empfänger erreichen. Dabei stehen beide Antennen zum Empfangen etwa 0.5 Meter auseinander.

Die Messung dient der Ermittlung der Unterschiedlichkeit beider Kopien, um entscheiden zu können, ob mit jeweils zwei gekoppelten Antennen auf Sender- und Empfängerseite ein MIMO-System mit unabhängigen Kanälen zustande kommt.

Die verwendeten Strecken zum Messen sind $S - R_1$ und $S - R_2$.

Abbildung 4.2 zeigt eine Antenne (A_S) auf der Sender-Seite (S) und zwei Antennen

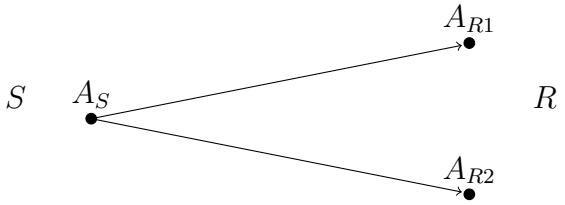

Abbildung 4.2: Ein *OFDM*-Signal wird von einer Sendeantenne zu zwei verschiedenen Empfangsantennen gesendet

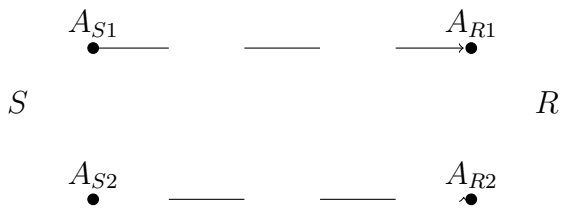

Abbildung 4.3: Zwei identische *OFDM*-Signale werden abwechselnd jeweils durch eine von zwei Sendeantennen an zwei Empfangsantennen übertragen

(A_{R1} und A_{R2}) auf der Empfänger-Seite (R). Beide Signale werden jeweils durch einen Pfeil kenntlich gemacht.

Zur Generierung des Signals wurde das Programm `mimoots_ofdm_tx.py` während zum Empfang das Programm `mimoots_ofdm2_rx_compare.py` genutzt. Letzteres speichert beide Signale jeweils separat in einer Datei (`signal1.dat` und `signal2.dat`), sodass sie mittels Octave eingelesen und analysiert werden konnten.

In sämtlichen Messungen wurde eine Samplingrate von 500000Hz, eine Datenträgerzahl von 48, eine Pilotträgerzahl von 4, eine Fensterlänge von 64 Samples und eine *CP*-Länge von 16 Samples gewählt.

4.4 Zeitversetzte MIMO-Datenübertragung zur Messung der Diversität

Auch dieses Szenario dient dazu bestimmen zu können, ob mittels der UHD-Devices ein wirkliches MIMO-System mit unabhängigen Kanälen realisiert werden kann. Anders als im Szenario zuvor werden hier, wie Abbildung 4.3 zeigt, von zwei miteinander gekoppelten Antennen (A_{S1} und A_{S2}) beim Sender (S) jeweils ein Datensignal abgesendet. Diese werden so zeitversetzt gesendet (erkennbar durch die gestrichelten Pfeile), dass ein *OFDM-Frame* wechselweise erst von der einen und dann von der anderen Antenne abgesendet wird, sodass sich die Signale beider Antennen nicht gegenseitig stören. Sendet die eine Antenne, ist die andere Antenne still und umgekehrt. Beide Signale sind ansonsten identisch.

Auf diese Weise wird beim Empfänger (R) ebenfalls mit zwei miteinander gekoppelten Antennen (A_{R1} und A_{R2}) das Signal empfangen, allerdings abwechselnd erst von der einen Sendeantenne und dann von der anderen. So können die Eigenschaften aller Kanäle, die sich aus den Zweierkombinationen aller vier Antennen ergeben, gemessen und beurteilt werden. Ausschlaggebend zur Beurteilung der MIMO-Fähigkeiten des Übertragungssystems in dieser Arbeit sind jedoch nur die Kanaleigenschaften zwischen den Antennen, die sich auf kürzesten Weg gegenüber stehen. Die Kanäle der sich kreuzenden Übertragungsstrecken werden beim Empfänger ignoriert und aussortiert.

Die verwendeten Strecken zum Messen sind $S - R_1$ und $S - R_2$.

In sämtlichen Messungen wurde eine Samplingrate von 500000Hz, eine Datenträgerzahl von 48, eine Pilotträgerzahl von 4, eine Fensterlänge von 64 Samples und eine

CP-Länge von 16 Samples gewählt.

5 Bewertung und Interpretation der Messergebnisse

5.1 Definition Korrelationskoeffizient

Der Korrelationskoeffizient ist eine dimensionslose skalare Größe, die ein Maß für einen *linearen Zusammenhang* zwischen zwei Zufallsgrößen darstellt. Er ist folgendermaßen definiert:

$$r_{XY} = \frac{\mathrm{Cov}(X,Y)}{\sigma(X)\sigma(Y)} \tag{5.1}$$

X und Y sind Zufallsvariablen, wobei σ_X bzw. σ_Y deren jeweilige Standardabweichung und $\mathrm{Cov}(X,Y)$ deren gemeinsame Kovarianz ist.

r_{XY} kann folgende Werte annehmen:

$$-1 \leq r_{XY} \leq 1 \tag{5.2}$$

Für den Fall $|r_{XY}| = 1$ wird ein vollständiger und für den Fall $r_{XY} = 0$ wird kein linearer Zusammenhang angegeben.

Die Verwendung des Korrelationskoeffizienten als statistisches Abhängigkeitsmaß hat enge Grenzen in seiner Aussagekraft[1]:

- $|r_{XY}| = 1 \Leftrightarrow Y = aX + b$ Sollte $|r_{XY}| = 1$ gelten, so besteht zwischen den beiden Zufallsgrößen X und Y ein perfekter linearer Zusammenhang. Im Allgemeinen gilt auch der Umkehrschluss.

[1] Siehe [8, S. 152-164]

- $r_{XY} = 0 \Leftrightarrow Y \neq aX + b$ Sollte $r_{XY} = 0$ gelten, so besteht zwischen den beiden Zufallsgrößen X und Y kein linearer Zusammenhang. Im Allgemeinen gilt auch der Umkehrschluss.
- Sind X und Y statistisch unabhängig voneinander, so gilt $r_{XY} = 0$. Der Umkehrschluss gilt im Allgemeinen jedoch nicht. Falls $r_{XY} = 0$ gilt, so können aufgrund anderer nichtlinearer Abhängigkeitsstrukturen eine stochastische Abhängigkeit zwischen X und Y bestehen, da r_{XY} diese nicht erfasst.
- Die Berechnung von r_{XY} zur Ermittlung eines Zusammenhangs zwischen den Zufallsgrößen X und Y ist nur sinnvoll, wenn dieser durch einen linearen Zusammenhang erklärt werden kann.
- Im Allgemeinen lässt sich aufgrund von $|r_{XY}|$ nicht auf einen kausalen Zusammenhang zwischen X und Y schließen. Dieser muss gesondert betrachtet und als durch r_{XY} hergeleitete Beziehung erklärt werden.
- Die Größe r_{XY} sagt nichts über die Form des linearen Zusammenhangs aus. D.h. die Parameter a und b werden durch r_{XY} nicht charakterisiert. Es können für den selben Wert r_{XY} unterschiedliche Regressionsgeraden existieren.

Für die Messungen in der Versuchsanordnung muss auf begrenzte empirische Stichproben zurückgegriffen werden. Hierzu ist die gemeinsame Kovarianz $\mathrm{Cov}(X, Y)$ durch die gemeinsame Stichprobenvarianz s_{xy} und die einzelnen Standardabweichungen σ_X und σ_Y durch die Stichprobenvarianzen s_x und s_y zu ersetzen.

Daraus ergibt sich die Formel:

$$r_{XY} = \frac{\sum_{i=0}^{n} (x_i - \overline{x})\,(y_i - \overline{y})}{\sqrt{\sum_{i=0}^{n} (x_i - \overline{x})^2 \cdot \sum_{i=0}^{n} (y_i - \overline{y})^2}} \tag{5.3}$$

$\overline{x}$ und $\overline{y}$ sind hierbei die empirischen Mittelwerte der Zeitreihen:

$$\overline{x} = \frac{1}{n} \sum_{i=0}^{n} x_i \qquad\qquad \overline{y} = \frac{1}{n} \sum_{i=0}^{n} y_i \tag{5.4}$$

Der empirische Korrelationskoeffizient lässt sich in Matlab oder Octave über die Funktion `corrcoef` berechnen.

5.2 Aussagekraft

Für die Messung der Unterschiedlichkeit beider Signale sind also die beiden Extremfälle $|r_{XY}| = 1$ und $r_{XY} = 0$ auf ihre Aussagekraft hin zu untersuchen, um auf die Unterschiedlichkeit beider Kanäle zu schließen.

Da beide Signale, die durch die beiden Antennen auf der Empfängerseite aufgefangen werden, ursprünglich von einen Originalsignal abstammen, kann von einem ursprünglichen linearen Zusammenhang in der Form ausgegangen werden, dass sie von den Werten her identisch waren. Bevor die Signale also die Kanäle durchlaufen, würde die Berechnung eines Korrelationskoeffizienten den Wert $r_{XY} = 1$ ergeben. Nun durchlaufen beide Signale jedes für sich einen Kanal und treten als Beobachtungsgrößen X und Y auf Empfängerseite wieder in Erscheinung.

Es lassen sich nun grob zwei Extremfälle unterscheiden:

- Sollten sich beide Kanäle voneinander unterscheiden, müsste die Linearität zwischen den beiden Signalen im hohen Maße beseitigt worden sein. Idealerweise würde in diesem Fall ein Korrelationskoeffizient mit dem Wert $r_{XY} = 0$ resultieren.

- Sollten beide Kanäle jedoch identisch sein, so müsste die Linearität zwischen beiden Signalen weiterhin im vollen Maße bestehen. Dies würde sich durch einen linearen Zusammenhang $Y = aX + b$ zwischen beiden Zufallsgrößen ausdrücken. Die Neigung dieser Gerade wäre unwichtig, da allein das Maß an Linearität ausschlaggebend ist, welches das Verhältnis der einzelnen Kanalparameter zu einander angibt. Um so mehr Messgrößen nicht auf einer Geraden liegen, sondern um sie herum streuen, um so unterschiedlicher sind beide Kanäle zueinander.

Um eine möglichst hohe Signifikanz des Korrelationskoeffizienten sicherzustellen, ist eine hohe Anzahl an zu übertragenen Bits zu wählen, um eine große Anzahl an Samples und damit Messwerten zu erreichen.

Da beide Signale ursprünglich identisch waren, wäre eine volle Linearität zwischen beiden Messergebnissen und damit $r_{XY} = 1$ sinnvoll. In diesem Fall könnte von $r_{XY} = 1$ auf einen kausalen Zusammenhang geschlossen werden, verursacht durch zwei identische Kanäle.

Allerdings ergibt sich ein großes Problem: Um die Gleichheit oder Unterschiedlichkeit zweier Kanäle mittels des Korrelationskoeffizienten ausdrücken zu können, muss auf der Skala von $[-1, 1]$ ein Punkt definiert werden, an dem die Kanäle sich vollständig unterscheiden. Dies ist jedoch nicht ohne weiteres zu bewerkstelligen. Es müsste mit den genutzten UHD-Devices eine Übertragung durchgeführt werden, in welcher die maximale Unterschiedlichkeit beider Kanäle festgelegt und als Bezugsgröße sichergestellt wäre (bspw. durch einen größeren Abstand beider Sende- oder Empfangsantennen, was durch die begrenzte Länge der Synchronisationskabel nicht möglich war). Dies war im Rahmen dieser Arbeit jedoch nicht möglich. Es fehlt also ein Bereich in diesem Intervall, der eine ausreichende Unterschiedlichkeit beider Kanäle ausdrückt.

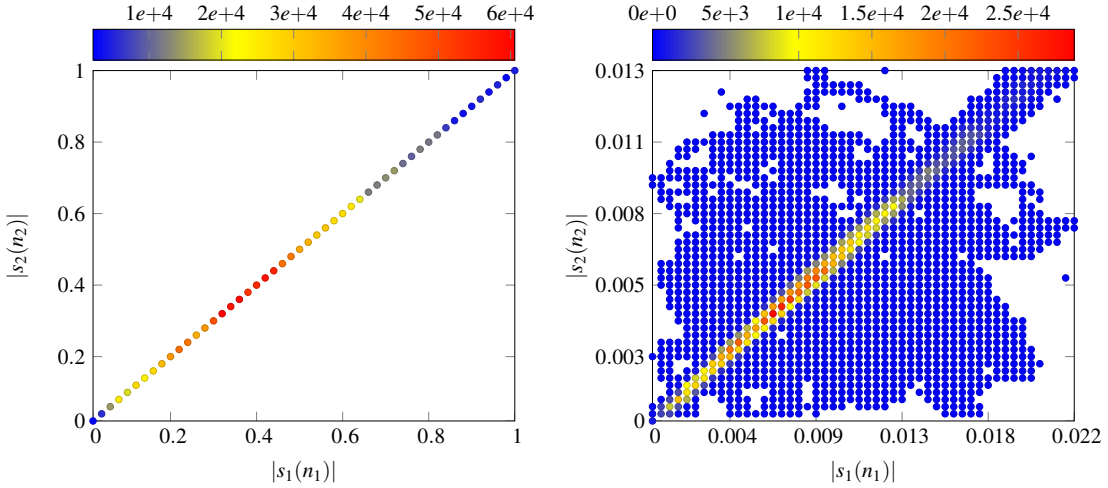

Abbildung 5.1: MISO-Datenübertragung *Links:* Verteilung der Häufigkeit der Amplitudenwerte vor der Übertragung *Rechts:* Verteilung der Häufigkeit der Amplitudenwerte beim Empfang

5.3 Messergebnisse

5.3.1 Gleichzeitige MISO-Datenübetragung zur Messung der Diversität

In Abbildung 5.1 ist im linken Diagramm zu erkennen, wie sämtliche Häufigkeiten der Amplitudenwerte beider Signale auf einer Geraden liegen. Die weißen Bereiche im Diagramm geben einen Amplitudenwert von 0 an. Die Zuordnung zwischen Amplitudenwert und Farbe ist über dem Diagramm als Farbverlauf mit einer Skala dargestellt. Es ist deutlich zu erkennen, dass die Häufigkeit nicht gleichverteilt über der Geraden ist, sondern vom Punkt $(0, 0)$ aus stark ansteigt, um die Punkte $(0.2, 0.2)$ und $(0.4, 0.4)$ ein lokales Maximum erreicht und daraufhin wieder bis zum Punkt (1,1) monoton sinkt.

Dies liegt an der Eigenschaft, dass ein *OFDM-Symbol* im Zeitbereich aus mehreren überlagerten Schwingungen besteht, die jede für sich als unabhängige Zufallsgröße aufgefasst werden kann. Deren Summe nähert sich bei steigender Anzahl an Trägern einer Normalverteilung an, was auch der Grund für die *PAPR*-Problematik in *OFDM* ist. Indem die durch *OFDM* kodierten Daten durch das Programm gleichverteilt gewählt worden sind, sollte dieser Problematik entgegengewirkt werden, konnte sie jedoch nicht ganz vermeiden.

Im rechten Diagramm wird die Häufigkeit der Amplitudenwerte nach dem Empfang dargestellt. Dabei wird klar, dass sich ein Großteil der Werte immer noch auf einer geraden befindet. Ein kleiner Teil der Werte streut recht gleichmäßig um diese Geraden herum. Dabei sind die lokalen Maxima der Häufigkeit der Amplitudenwerte auf der Geraden erhalten geblieben.

Das Intervall der Amplitudenwerte $(0, 0.013)$ des Signals s_2 ist deutlich kleiner als

von Signal s_1 $(0, 0.022)$. Die Steigung der Geraden wird also etwas kleiner, was an der Linearität der Messwerte jedoch nichts ändert.

Die Berechnung des Korrelationskoeffizienten ergab den Wert

$$r_{XY} = 0.93552 \tag{5.5}$$

und drückt damit eine deutliche und starke Linearität der Messwerte aus.

Wie bereits angesprochen worden ist, muss die Deutung des Korrelationskoeffizienten in gewissen Grenzen erfolgen, da das Ergebnis sonst falsch interpretiert werden kann. Durch eine hohe Anzahl an erzeugten Samples ($n = 1279744$) ist davon auszugehen, dass eine ausreichende Signifikanz besteht. Da $r_{XY} = 0.93552$ dem Wert $r_{XY} = 1$ besonders nahe ist, kann auch von einem sehr starken linearen Zusammenhang der Messergebnisse ausgegangen werden. Dieser lineare Zusammenhang lässt sich dadurch sinnvoll erklären, dass sich beide Kanäle kaum voneinander unterscheiden und die bereits bestehende Linearität der beiden Signale dadurch größtenteils erhalten blieb.

Damit bleibt festzuhalten, dass sich in dieser Versuchsdurchführung beide Kanäle nicht signifikant voneinander unterscheiden.

Grund dafür dürfte sein, dass die Geräte zum Empfangen eine zu geringe Samplingrate für die Abtastung zur Verfügung stellen. Damit können die Signale nicht hoch genug aufgelöst werden. Da sich die Antennen auf der Sende- als auch der Empfangsseite nur einen halben Meter auseinander befinden, könnte durch Erhöhen dieses Abstands auf Empfängerseite der Unterschied beider Kanäle vergrößert werden. Dies ist jedoch aus praktischen Gründen, die den Versuchsaufbau betreffen, problematisch.

Eine höhere Samplingrate hingegen könnte das Problem lösen, erlaubt aber aufgrund der komplexen Signalverarbeitung beim Empfänger nur eine geringere Anzahl an Trägern, was für zukünftige Szenarien ein Problem darstellen könnte. Zur Erhaltung der Trägeranzahl müsste rechenstärkere Hardware zur Signalverarbeitung eingesetzt werden.

Auch hier kann wegen der fehlenden Vergleichsmessung nicht angegeben werden, wie sehr sich beide Kanäle voneinander unterscheiden und ab welchem Wert von r_{XY} eine ausreichende Unterschiedlichkeit beider Kanäle erreicht wird.

5.3.2 Zeitversetzte MIMO-Datenübertragung zur Messung der Diversität

Wie bereits in Abbildung 5.1 zu sehen war, befinden sich auch in Abbildung 5.2 im linken Diagramm eine Gerade, auf der sich alle Amplitudenwerte sammeln. Ein linearer Zusammenhang ist damit klar zu erkennen. Auch ist der Effekt der *PAPR*-Problematik zu erkennen, weil sich zwei lokale Maxima auf der Geraden befinden, einmal bei Punkt $(0.2, 0.2)$ sowie einmal beim Punkt $(0.4, 0.4)$.

Im rechten Diagramm hingegen streuen die Häufigkeiten ein wenig um die ursprüngliche Gerade. Trotzdem ist immer noch eine deutliche Gerade zu erkennen, auf der die

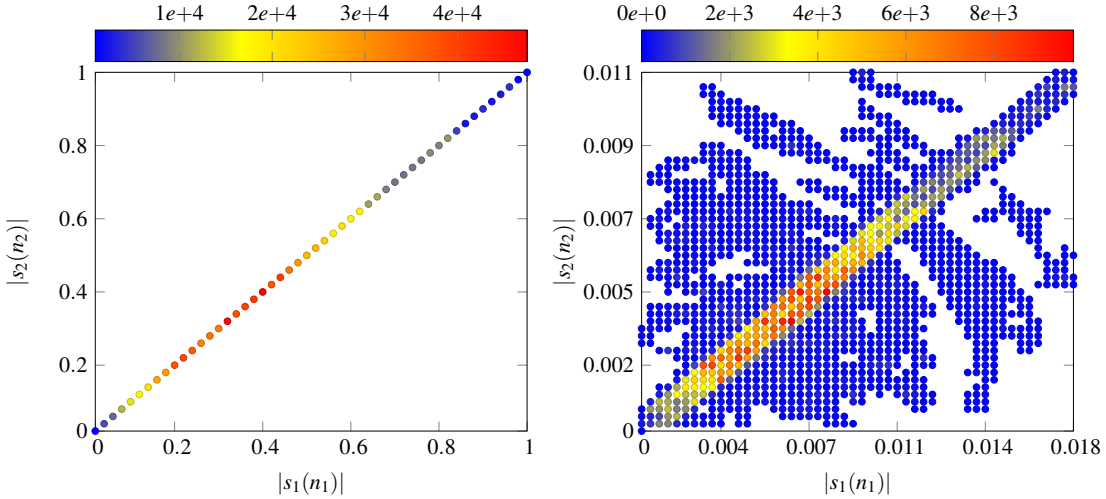

Abbildung 5.2: Zeitversetzte MIMO-Datenübertragung *Links:* Verteilung der Häufigkeit der Amplitudenwerte vor der Übertragung *Rechts:* Verteilung der Häufigkeit der Amplitudenwerte beim Empfang

meisten Punkte liegen. Diese ist jedoch ein wenig verbreitert im Vergleich zum Diagramm in Abbildung 5.1. Die Werte scheinen also in diesem Fall stärker zu streuen als in der vorhergehenden Messung. Das liegt daran, dass die Signale hier nicht durch eine, sondern durch zwei verschiedene Antennen abgestrahlt werden. Dies vergrößert die Unterschiede zwischen den beiden Kanälen, wodurch der lineare Zusammenhang der Signale gemindert wird.

Die Berechnung des Korrelationskoeffizienten ergibt allerdings den Wert

$$r_{XY} = 0.84287 \tag{5.6}$$

Er besagt deutlich, dass immer noch ein starker linearer Zusammenhang zwischen den Messwerten besteht.

Da die Anzahl der Messwerte $n = 959808$ beträgt, kann hier von einem signifikanten Ergebnis gesprochen werden.

Schlussendlich lässt sich sagen, dass auch durch die Verwendung einer weiteren Antenne zum Senden der Signale kein nennenswert höherer Unterschied zwischen beiden Kanälen zu beobachten ist. Dies wird ebenfalls durch eine zu geringe Samplingrate verursacht.

Auch hier kann wegen einer fehlenden Vergleichsmessung nicht angegeben werden, wie sehr sich beide Kanäle voneinander unterscheiden und ab welchem Wert von r_{XY} eine ausreichende Unterschiedlichkeit beider Kanäle angezeigt wird.

6 Zusammenfassung und Ausblick

In dieser Arbeit konnte ein funktionsfähiges OFDM-Übertragungssystem implementiert werden. Dazu wurde der Quellcode mehrerer Programme untersucht und auf ihre Tauglichkeit als Grundlage hin überprüft. Eines dieser Programme konnte in einer vereinfachten und überarbeiteten Form soweit weiterentwickelt werden, dass die in dieser Arbeit gestellten Anforderungen an ein Übertragungssystem erfüllt werden konnten. Das Ziel war die Entwicklungszeit so gering wie möglich zu halten, was durch die Wiederverwendung fremden Quellcodes ermöglicht werden soll.

Voraussetzung für die Funktionsfähigkeit war das Implementieren und Integrieren zweier neuer Blöcke in den vereinfachten Signalflussgraphen. Hierzu wurde ein neues *GNU Radio*-Modul erstellt, das die benötigten Blöcke kapselt und zur Verfügung stellt. Aufgrund der Aufgabenstellung konnten bei deren Implementierung einige Vereinfachungen getroffen werden, die die Entwicklung beschleunigten. Dabei wurden jedoch — um Zeit zu sparen — einige Probleme nicht gelöst, da deren Auswirkung auf die Funktionalität des Übertragungssystems begrenzt und das Erreichen in dieser Arbeit anvisierten Ziele nicht verhinderte.

Anschließend wurden in einer Versuchsanordnung mehrere Testmessungen durchgeführt um zu ermitteln, ob bei einem Zweiantennen-Betrieb sowohl auf der Sender- als auch der Empfängerseite unabhängige Kanäle zustande kommen. Mithilfe des Korrelationskoeffizienten und der grafischen Analyse der Häufigkeitsdiagramme, die die Amplitudenwerte beider Signale darstellen, konnte gezeigt werden, dass beide Signale immer noch einen starken linearen Zusammenhang besaßen. Aufgrund fehlender Vergleichsmessung zur Ermittlung eines Korrelationskoeffizienten, der eine ausreichende Unterschiedlichekeit beider Kanäle anzeigt, konnte nicht bewertet werden, in wie weit sich beide Kanäle unterscheiden und ob der Unterschied für echte MIMO-Fähigkeit ausreichend ist.

Trotzdem kann das System entsprechend der in dieser Arbeit vorgestellten Nutzungsszenarien verwendet werden, was einen Vorteil im Vergleich zu einem reinen SISO-Übertragungssystem darstellt.

Es wird sich zeigen inwieweit die technischen Möglichkeiten der nächsten Gerätegenerationen voranschreiten werden und ob sich das Problem der mangelnden zeitlichen Auflösung der Signale buchstäblich in Luft auflösen wird. Es ist jedenfalls realistisch davon auszugehen, dass sich die Signalauflösung vergrößern und damit in Zukunft nicht mehr der limitierende Faktor für die MIMO-Fähigkeiten sein wird. Spätestens ab diesem Punkt stellt sich die Frage der Weiterentwicklung des im Rahmen dieser Arbeit erstellten Quellcodes.

GNU Radio ist ein sich immer weiter entwickelndes Framework, dessen Opensourcecharakter das Umsetzen und Implementieren neuer Funktionen und Strukturen im Quellcode selbst beschleunigt. Ein Teil der Arbeit an diesem Quellcode wird also darin bestehen, diesen an die Änderungen von *GNU Radio* laufend anzupassen und damit funktionsfähig zu halten.

Beispielsweise ist jetzt schon zu erkennen, dass die Wiederverwendbarkeit von Programmteilen in *GNU Radio* erhöht werden soll. So stellt der Block `digital.ofdm_frame_equalizer_vcvc` mehr eine Art Adapter oder Rahmen dar denn einen eigenständigen Signalverarbeitungsblock, dem ein beliebiges Objekt der Klasse `ofdm_equalizer_base` übergeben werden kann. Die Instanz ist in der Lage zu entscheiden, wie die Kanalkorrektur durchgeführt werden soll. Damit wäre es möglich einen Equalizer zu implementieren, der mehrere Datenströme synchron bearbeiten kann. Damit wird die Implementierung der Kanalkorrektur unabhängig von der Art und Weise, wie sie in den *GNU Radio*-Signalflussgraphen eingefügt wird.

Abbildungsverzeichnis

1.1 Generierung eines Signals mittels *OFDM* 17
1.2 Betragsspektrum $\|S(f)\|$ der Fourier-Transformierten des Signals $x(t)$ . . 18
1.3 Extraktion der Daten aus einem *OFDM*-Signal im Basisbandbereich . . . 22
1.4 Auswirkung eines *CFO* 23
1.5 Darstellung der Metrik zum Erkennen des *OFDM*-Symbols ohne Rauschen (oben) und mit Rauschen (unten) 24
1.6 Darstellung mehrere *OFDM*-Symbole ohne *CP* (*unten*) und mit *CP* (*oben*) 25

3.1 Das *GNU Radio*-Modul `gr-mimoots` und deren Bestandteile und Funktionsweise 35
3.2 *PAP* der Funktion `work()` aus `ofdm_scale_frames_vcvc` 39
3.3 *PAP* der Funktion `general_work()` aus `ofdm_extract_frames_cvc` . . . 40
3.4 Die Zahlen 39_{10} in binärer Form dargestellt, wie sie an den nächsten Block weitergeleitet werden 46
3.5 Die zwei Zahlen 39_{10} zerlegt in einzelne Chunks der Größe 2 Bit 48
3.6 Die komplexen Symbole, auf die die einzelnen Chunks gemappt worden sind 49
3.7 Ein *OFDM-Frame* bestehend aus zwei *Synchronisationssymbolen* und am Ende einem *OFDM*-Symbol, auf dessen *Träger* die einzelnen komplexen Symbole verteilt worden sind 50
3.8 Das Basisband-Signal im Zeitbereich erstellt durch die 51
3.9 Basisbandsignal mit normierten *OFDM*-Symbolen 52
3.10 Normiertes Basisbandsignal mit einem *CP* in jedem *OFDM*-Symbol 53

4.1 Der grobe Aufbau des Versuchraums 66
4.2 Ein *OFDM*-Signal wird von einer Sendeantenne zu zwei verschiedenen Empfangsantennen gesendet 66
4.3 Zwei identische *OFDM*-Signale werden abwechselnd jeweils durch eine von zwei Sendeantennen an zwei Empfangsantennen übertragen 67

5.1 MISO-Datenübertragung *Links:* Verteilung der Häufigkeit der Amplitudenwerte vor der Übertragung *Rechts:* Verteilung der Häufigkeit der Amplitudenwerte beim Empfang 72
5.2 Zeitversetzte MIMO-Datenübertragung *Links:* Verteilung der Häufigkeit der Amplitudenwerte vor der Übertragung *Rechts:* Verteilung der Häufigkeit der Amplitudenwerte beim Empfang 74

Index

C

CFO 9, 22, 73
Constellation-Mapping 16
CP 9, 21, 24–26, 35, 37, 44, 48–50, 56, 63, 64, 73

D

Datensymbol 7
Delay-Spread 7, 25
DFT 9, 14, 20, 25, 35

E

Estimator 20

G

GNU Radio 7, 15–17, 27–30, 32, 34, 38, 41, 49, 50, 53, 54, 57–59, 71–73

I

ICI 9, 14, 18, 22, 25
IDFT 7, 9, 14–16, 48
IFFT 9, 16
ISI 9, 23, 25

O

OFDM 9, 14–17, 21, 23, 24, 27, 45, 47, 49, 50, 62, 63, 68, 73
OFDM-Frame 7, 47, 51, 63, 73
OFDM-Symbol 7, 21, 34, 35, 37, 38, 41, 44, 46–48, 51, 53, 57, 59, 68
Orthogonalität 7, 14, 19, 20, 22, 23, 25, 51

P

parallelizer 16
Peak-to-Average Power Ratio 9, 53, 68, 69
Pilotsymbol 8, 44
Programm-Ablauf-Plan 9, 36, 37, 73
PSK 10, 30, 51
  8-PSK 9, 14, 27, 29, 34, 41, 42, 44, 53, 61
  BPSK 9, 27, 41, 42, 44, 47, 53, 61
  QPSK 10, 27, 41, 42, 44, 45, 48, 53, 61

Q

QAM 10, 14

S

Schmid-Cox-Korrelator 8, 13, 23, 25, 26, 35, 37, 40, 41, 46, 52
serializer 16
Seriell-Parallel-Wandler 16, 20
signal-to-noise ratio 10, 51
source 16, 20
symbol time offset 10, 24
Synchronisationssymbol 8, 24, 40, 41, 46, 47

T

Ton 8, 29
    Datenton 7, 29
    Pilotton 8, 29
Träger 8, 14, 15, 18, 25–27, 29, 34, 40, 44, 46–48, 73
    Datenträger 7
    Pilotträger 8, 27, 35

Literaturverzeichnis

[1] Y. S. Cho, J. Kim, W. Y. Yang, and C. G. Kang, *MIMO-OFDM Wireless Communications with MATLAB*. Wiley, 2010.

[2] K.-D. Kammeyer and K. Kroschel, "Digitale signalverarbeitung / filterung und spektralanalyse ; mit matlab-Übungen ; mit 33 tabellen," 2009.

[3] T. M. Schmidl and D. C. Cox, "Robust frequency and timing synchronization for ofdm," 2010. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&tp=&arnumber=650240.

[4] E. Research, "Synchronization and mimo capability with usrp devices," Aug. 2014. http://www.ettus.com/content/files/kb/mimo_and_sync_with_usrp.pdf.

[5] O. Weikert and U. Zolzer, "Efficient mimo channel estimation with optimal training sequences." http://www.hsu-hh.de/download-1.4.1.php?brick_id=GMaN37IYM6bgtJD8.

[6] H. Minn and N. Al-Dhahir, "Optimal training signals for mimo ofdm channel estimation," 2004. http://www.utdallas.edu/~aldhahir/training_OFDM.pdf.

[7] P. Beinschob, "Untersuchung von entscheidungsbasierten mimo-ofdm-kanalschätzungsalgorithmen," 2008. http://www.hsu-hh.de/download-1.5.1.php?brick_id=AimjtWtJJqV1FWnb.

[8] J. Schwarze, "Schwarze, jochen: Grundlagen der statistik - 1 : Beschreibende verfahren," 1992.

[9] H. Nuszkowski, "Digitale signalübertragung / grundlagen der digitalen nachrichtenübertragungssysteme," 2013.

[10] P. A. Höher, "Grundlagen der digitalen informationsübertragung / von der theorie zu mobilfunkanwendungen," 2011.

[11] K.-D. Kammeyer, "Nachrichtenübertragung / mit ... 35 tabellen," 2008.

[12] J. Hoffmann and F. Quint, "Signalverarbeitung mit matlab und simulink / anwendungsorientierte simulationen," 2012.

[13] T. Rondeau, "Homepage of gnu radio," Aug. 2014. `http://www.gnuradio.org/`.

[14] S. G. Rakash, "Synchronization in mimo-ofdm systems," 2009. `http://www.hsu-hh.de/download-1.5.1.php?brick_id=w4rR5anXfKvEcwye`.