Departement für Physik
Ludwig-Maximilians-Universität München

Automatische Kalibration eines Polarisationsanalysators für Qubit-Tomographie

Bachelorarbeit

Luisa Hofmann

6. Juli 2012
(überarbeitete Version vom 27. Juli 2012)

Arbeitsgruppe Experimentelle Quantenphysik
(Prof. Dr. H. Weinfurter)

Inhaltsverzeichnis

1 Einleitung

Um die Jahrhundertwende häuften sich Beobachtungen, die mit der klassischen Physik nicht mehr erklärt werden konnten. Max Planck entwickelte daraufhin als Erster die Vorstellung, die Absorption und Emission eines schwarzen Strahlers sei quantisiert. Diese revolutionäre Annahme, sowie deren weitere Folgen, wurden immer mehr akzeptiert, weil viele Phänomene dadurch erklärt werden konnten. Auch Albert Einstein entwickelte die neue Theorie weiter und konnte durch Einführen von Lichtquanten den Photoelektrischen Effekt beschreiben [9].

Photonen als Lichtquanten gewannen immer mehr an Bedeutung und sind heute, ein Jahrhundert später, die Grundlage vieler Forschungsgebiete. Zum Beispiel spielen Photonen in dem Bereich der Quanteninformationstheorie als Quanten Bits, kurz Qubits, eine zentrale Rolle. Zur Zeit wird auf diesem Gebiet noch Grundlagenforschung betrieben. Ein wichtiges Instrument für die Messung von Qubits ist die Qubit-Tomographie. Werden Qubits als die Polarisationszustände von Photonen realisiert, so ist eine genaue Analyse der Polarisation unumgänglich. Durch einen geeigneten Aufbau eines Polarisationsanalysators und Ablauf verschiedener Messungen kann die Polarisation von Licht bestimmt werden.

Ziel meiner Arbeit ist es, solch einen Polarisationsanalysator automatisch zu kalibrieren. Dadurch soll in späteren Experimenten nicht nur das umständliche manuelle Kalibrieren eingespart werden, sondern es soll auch eine exaktere Kalibration möglich sein.

Für die automatische Kalibration werden die Wellenplättchen eines Polarisationsanalysators in computergesteuerte Motoren eingebaut. Die Schrittmotoren werden so gesteuert, dass die Wellenplättchen beliebig um die Achse eines Laserstrahls gedreht werden können. Mit Photodioden kann die Intensität des Lasers gemessen werden. Gesucht ist nun ein Algorithmus, der durch Steuern der Motoren und gleichzeitigen Spannungsmessungen den Analysator kalibriert. Das Programm, welches diesen Algorithmus umsetzt, wurde in der Programmiersprache C++ geschrieben.

Um die Umsetzung und Idee des Algorithmus verstehen zu können, muss erst der theoretische Hintergrund dafür erörtert werden. Daher gliedert sich meine Arbeit in zwei Teile. Im Theorieteil werden Qubits beschrieben und deren Eigenschaften behandelt. Die Realisierung von Qubits als Polarisationszustände eines Photons wird danach besprochen. Anschließend wird erklärt, wie durch Polarisationsanalysen der Zustand eines Qubits vollständig rekonstruiert werden kann. Diese Prozedur nennt sich Qubit-Tomographie.

Im zweiten Teil wird dann die Umsetzung der automatischen Kalibration beschrieben. Die Ideen zu den einzelnen Kalibrationsalgorithmen werden erklärt und die Programmstruktur wird analysiert, sowie durch Flussdiagramme veranschaulicht. Die Resultate mehrerer Kalibrationsmessungen werden verglichen, um eine Aussage über die Funktionalität und Effizienz des entwickelten Programms treffen zu können.

2 Theorie

2.1 Quantum Bits

2.1.1 Konzept der Qubits

Quanten bits, kurz Qubits, sind das quantenmechanische Analogon zum klassischen Bit. Der Begriff Qubit wurde 1995 von Benjamin Schumacher eingeführt [14]. Ein klassisches Bit kann die Werte 0 oder 1 annehmen. Das Besondere an Qubits ist, dass sie als quantenmechanisches System nicht nur die Zustände $|0\rangle$ oder $|1\rangle$ annehmen können, sondern auch jede Superposition der beiden Zustände [11]:

$$|\psi\rangle = c_0\,|0\rangle + c_1\,|1\rangle \qquad \text{mit} \quad |c_0|^2 + |c_1|^2 = 1 \qquad \text{und} \quad c_0,\, c_1 \in \mathbb{C} \tag{2.1}$$

Die Zustände $|0\rangle$ und $|1\rangle$ sind die Eigenzustände der hermiteschen Pauli-Spin-Matrix σ_z zu den Eigenwerten $+1$ und -1. Es gilt:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1| \tag{2.2}$$

Jede hermitesche Matrix besitzt reelle Eigenwerte und die Eigenvektoren zu verschiedenen Eigenwerten sind zueinander orthogonal [13]. Die Zustände $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ und $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ bilden also eine orthonormale Basis des zweidimensionalen Hilbertraums $\mathcal{H}$, in welchem ein Qubit definiert ist [11]. Diese Basis ist die Z-Basis, die im Folgenden stets als Berechnungsbasis verwendet wird [8].
Die Pauli-Matrizen σ_x und σ_y lassen sich analog zu (2.2) ebenfalls in ihre orthogonalen Eigenzustände zerlegen:

$$\begin{aligned} \sigma_x &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = |+\rangle\langle +| - |-\rangle\langle -| \\ \sigma_y &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = |R\rangle\langle R| - |L\rangle\langle L| \end{aligned} \tag{2.3}$$

Die Eigenzustände $|+\rangle$ und $|-\rangle$ der Pauli-Matrix σ_x, sowie die Eigenzustände $|R\rangle$ und $|L\rangle$ der Pauli-Matrix σ_y werden in der Z-Basis folgendermaßen ausgedrückt[15]:

$$\begin{aligned} |+\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ |-\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} \\ |R\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + i\,|1\rangle\right) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ i \end{pmatrix} \\ |L\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle - i\,|1\rangle\right) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -i \end{pmatrix} \end{aligned} \tag{2.4}$$

2.1.2 Polarisationszustände

Die zwei Zustände $|0\rangle$ und $|1\rangle$ können auf vielfältige Weise als physikalische Zustände realisiert werden. Prinzipiell kann jedes Zwei-Niveau-System ein Qubit repräsentieren. Beispiele sind Spin-1/2 Teilchen mit den Zuständen spin-up $|\uparrow\rangle$ und spin-down $|\downarrow\rangle$, 2-Niveau Atome mit dem Grundzustand $|g\rangle$ und dem angeregten Zustand $|e\rangle$ und die orthogonalen Polarisationszustände horizontal $|H\rangle$ und vertikal $|V\rangle$ eines Photons. [3]

In meiner Arbeit werden Qubits als Polarisationszustände realisiert. Ein beliebiger Polarisationszustand $|\psi\rangle \in \mathcal{H}$ kann durch Umschreiben der Gleichung (2.1) folgendermaßen ausgedrückt werden:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|H\rangle + \sin\left(\frac{\theta}{2}\right)e^{i\varphi}|V\rangle \tag{2.5}$$

Dabei wird eine globale Phase $e^{i\gamma}$ vernachlässigt, da diese bei Betragsbildung verschwindet und daher nicht weiter relevant ist [11].

Die Winkel $\theta \in [0, \pi]$ und $\varphi \in [0, 2\pi]$ definieren einen Punkt auf einer Einheitskugel [8]. Jedes Qubit kann als Punkt auf dieser Kugel, der sogenannten Blochkugel dargestellt werden. Lineare Polarisationszustände ergeben sich für $\varphi \in \{0, \pi, 2\pi\}$, rechtselliptische Polarisation für $\varphi \in [0, \pi]$ und linkselliptische Polarisation für $\varphi \in [\pi, 2\pi]$.

Die Polarisationszustände diagonal $|+\rangle$, antidiagonal $|-\rangle$, rechtszirkular $|R\rangle$ und linkszirkular $|L\rangle$ aus (2.4) sind in Abbildung 2.1 auf der Blochkugel skizziert.

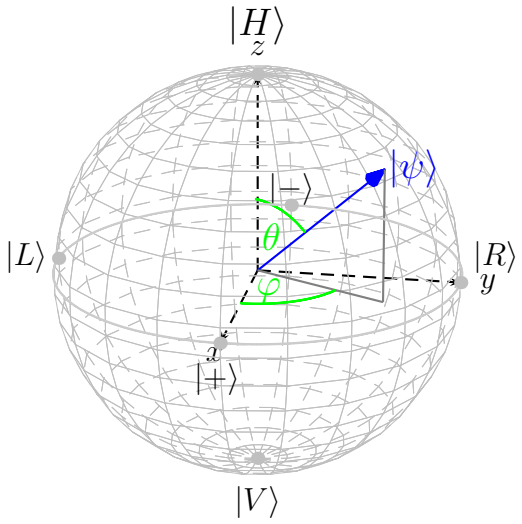

Abbildung 2.1: Polarisationszustände auf der Blochkugel. Die linearen Polarisationszustände $|H\rangle$, $|V\rangle$, $|+\rangle$ und $|-\rangle$, sowie die zirkularen Polarisationszustände $|R\rangle$ und $|L\rangle$ sind skizziert. Ein beliebiger Zustand $|\psi\rangle$ wird wie in Gleichung (2.5) durch die Winkel θ und φ charakterisiert.

Wie man sieht, liegen die Eigenzustände der Pauli-Matrizen σ_x, σ_y und σ_z entlang der x-, y- und z-Achsen [8].

2.1.3 Messungen

Eine physikalische Observable O wird durch einen hermiteschen Operator repräsentiert. Misst man den Zustand $|\psi\rangle$, so wird dieser mit einer bestimmten Wahrscheinlichkeit in

einen der Eigenzustände der Observablen O überführt. Dies wird auch als Kollaps der Wellenfunktion bezeichnet.
Die Wahrscheinlichkeit P_a, den Eigenwert a zum Eigenzustand $|a\rangle$ zu messen, ergibt sich zu:

$$P_a = |\langle a|\ \psi\rangle|^2$$

Der Erwartungswert der Observablen O lässt sich durch

$$\langle O\rangle = \langle\psi|\, O\, |\psi\rangle = \sum_a a \cdot |\langle a|\ \psi\rangle|^2$$

berechnen. Auf Grund der Wahrscheinlichkeitserhaltung muss gelten:

$$\sum_a P_a = \sum_a |\langle a|\ \psi\rangle|^2 = 1$$

Eine besondere Messung ist die Projektionsmessung, oder auch selektive Messung. Diese wird durch den Projektionsoperator $|a\rangle\langle a|$ aus einem der Eigenzustände repräsentiert und filtert bei einer Messung genau den Eigenzustand $|a\rangle$ heraus. Die Wahrscheinlichkeiten für alle Projektionsmessungen müssen sich wieder zu eins aufsummieren.
Es folgt: $\sum_a |a\rangle\langle a| = \underline{1}$. [13]

Für die Observable σ_z und den Zustand $|\psi\rangle$ aus (2.5) würde sich folgender Erwartungswert ergeben:

$$\langle\sigma_z\rangle = +1 \cdot |\langle H|\ \psi\rangle|^2 - 1 \cdot |\langle V|\ \psi\rangle|^2 = \cos\left(\frac{\theta}{2}\right)^2 - \sin\left(\frac{\theta}{2}\right)^2$$

Bei Messung dieses Qubits wird also mit der Wahrscheinlichkeit $P_{|H\rangle} = \cos\left(\frac{\theta}{2}\right)^2$ der Zustand $|\psi\rangle$ in den Eigenzustand $|H\rangle$ projiziert und mit der Wahrscheinlichkeit $P_{|V\rangle} = \sin\left(\frac{\theta}{2}\right)^2$ in den Eigenzustand $|V\rangle$.

Wurde der Zustand einmal gemessen, so erhält man aus diesem keine weitere Information, da er irreversibel in einen Eigenzustand projiziert wurde.

Um den Zustand $|\psi\rangle$ vollständig zu bestimmen, bräuchte man somit theoretisch unendlich viele identische Kopien dieses Zustands [11]. Im Experiment werden die Wahrscheinlichkeiten $P_{|H\rangle}$ und $P_{|V\rangle}$ durch mehrere Messungen genähert. Die Wahrscheinlichkeit $P_{|H\rangle}$ ergibt sich zum Beispiel aus der Anzahl $N_{|H\rangle}$ der detektierten Photonen mit dem Messergebnis $+1$ und der Gesamtzahl $N_{gesamt} = N_{|H\rangle} + N_{|V\rangle}$ detektierter Photonen:

$$P_{|H\rangle} \approx \frac{N_{|H\rangle}}{N_{gesamt}}$$

2.1.4 Reine und gemischte Ensemble

Mehrere identische physikalische Systeme werden als Ensemble bezeichnet. Können alle Systeme eines solchen Ensembles durch den gleichen Quantenzustand $|\psi\rangle$ beschrieben werden, so spricht man von einem reinen Ensemble. Befindet sich ein relativer Anteil w_1 der Systeme im Zustand $|\psi^{(1)}\rangle$ und der andere relative Anteil w_2 im Zustand $|\psi^{(2)}\rangle$, so kann dieses Ensemble nicht mehr durch Superposition der Zustände $\left|\psi^{(1)}\right\rangle$ und $\left|\psi^{(2)}\right\rangle$ ausgedrückt werden. Man spricht hier von einem gemischten Ensemble. Sind die Anteile w_1 und w_2 gleich groß, so entspricht dies einem vollständig gemischten Ensemble.

Um gemischte Ensemble beschreiben zu können, wird der Dichtematrixformalismus eingeführt. Der hermitesche Dichteoperator ρ

$$\rho = \sum_i w_i \left|\psi^{(i)}\right\rangle \left\langle\psi^{(i)}\right| \qquad \text{mit} \quad \sum_i w_i = 1 \qquad \text{und} \quad tr\left(\rho\right) = 1$$

beschreibt ein Ensemble.

Der Ensembledurchschnitt oder Mittelwert des Messergebnisses der Observablen O nach vielen Messungen ist die Spur[1] über $\rho \cdot O$. Mit den Eigenzuständen $\{|a\rangle\}$ der Observablen O ergibt sich:

$$\langle O\rangle = tr\left(O \cdot \rho\right) = \sum_a a \cdot tr\left(\left|a\right\rangle \left\langle a\right| \cdot \rho\right) \tag{2.6}$$

Würde man ein reines Ensemble im Dichtematrixformalismus ausdrücken, so wäre dies:

$$\rho_{rein} = \left|\psi\right\rangle \left\langle\psi\right|$$

Wobei $|\psi\rangle$ der Quantenzustand ist, in dem sich alle Ensembleteile befinden.

Daraus folgt sofort, dass nur für ein reines Ensemble gilt:

$$\rho_{rein} = \rho_{rein}^2 \implies tr\left(\rho_{rein}^2\right) = 1$$

[13]

2.1.5 Stokesparameter

Jede 2×2 Dichtematrix ρ für ein Ensemble aus Qubits kann durch die Stokesparameter S_i ausgedrückt werden [1]:

$$\rho = \begin{pmatrix} \rho_{11} & \rho_{12} \\ \rho_{21} & \rho_{22} \end{pmatrix} = \frac{1}{2} \sum_{i=0,x,y,z} S_i \sigma_i \tag{2.7}$$

Die Matrix $\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ ist die Einheitsmatrix und σ_x, σ_y und σ_z sind die Pauli-Matrizen.

Da die Spur der Dichtematrix stets normiert ist, gilt $S_0 = \rho_{11} + \rho_{22} = 1$. Die weiteren Parameter ergeben sich zu:

$$\begin{aligned} S_x &= 2 \cdot Re\left(\rho_{12}\right) \\ S_y &= 2 \cdot Im\left(\rho_{12}\right) \\ S_z &= \rho_{22} - \rho_{11} \end{aligned}$$

Durch den Blochvektor $\overrightarrow{S} = \begin{pmatrix} S_x & S_y & S_z \end{pmatrix}^T$, lässt sich jede Dichtematrix für ein Ensemble aus Qubits mit Hilfe der Blochkugel darstellen. [10]

Für reine Ensemble gilt $\sum\limits_{i=x,y,z} S_i^2 = 1$, für gemischte Ensemble $\sum\limits_{i=x,y,z}^{3} S_i^2 < 1$ und für vollständig gemischte Ensemble $\sum\limits_{i=x,y,z}^{3} S_i^2 = 0$ [1].

[1] $tr\left(X\right) = \sum\limits_{a'} \left\langle a'\right| X \left|a'\right\rangle = \sum\limits_{b'} \left\langle b'\right| X \left|b'\right\rangle$

Da die Länge des Blochvektors

$$\sqrt{\sum_{i=x,y,z} S_i^2} = r \tag{2.8}$$

die Bedeutung eines Radius hat, liegen reine Zustände wie in Kapitel 2.1.2 auf der Oberfläche der Blochkugel. Ein vollständig gemischtes Ensemble liegt im Ursprung und gemischte Ensemble liegen innerhalb der Blochkugel.

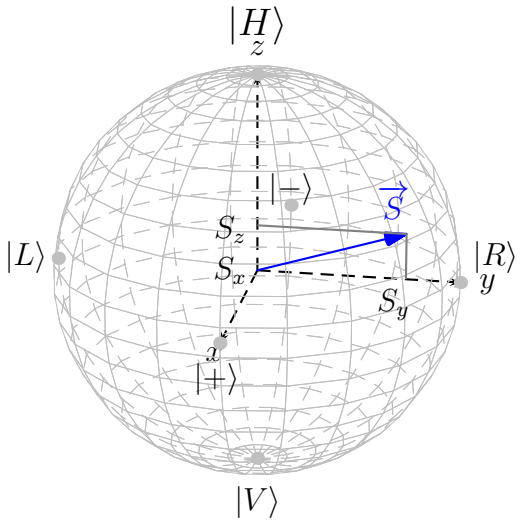

Abbildung 2.2: Beliebiger Blochvektor $\vec{S}$. Durch die Stokesparameter S_x, S_y und S_z wird ein gemischter Polarisationszustand auf der Blochkugel definiert.

2.2 Qubit-Tomographie

Qubit-Tomographie ist die Rekonstruktion der Dichtematrix eines quantenmechanischen Ensembles aus bestimmten Projektionsmessungen. Im Folgenden wird auf die Tomographie eines Single-Qubit Zustands eingegangen und die experimentelle Umsetzung erklärt. Anschließend folgt die Verallgemeinerung auf mehrere Qubits.

2.2.1 Single-Qubit Tomographie

Mit Hilfe der Stokesparameter S_i lässt sich eine Dichtematrix ρ in eine Summe aus den Pauli-Matrizen zerlegen, siehe (2.7). Das bedeutet, durch Bestimmung der Stokesparameter kann die Dichtematrix ρ vollständig rekonstruiert werden.

Es kann gezeigt werden, dass sich die Stokesparameter aus (2.7) zu

$$S_i \equiv tr(\sigma_i \rho) \tag{2.9}$$

ergeben [1]. Mit (2.6) folgt daher, dass die Stokesparameter die Erwartungswerte zu den Messungen der entsprechenden Pauli-Matrizen sind.

Das bedeutet, durch Projektionsmessungen entlang der Eigenzustände $\{|\psi\rangle\}$ der Pauli-Matrizen ergeben sich die Wahrscheinlichkeiten

$$P_{|\psi\rangle} = tr(|\psi\rangle \langle\psi| \rho)$$

und somit die Stokesparameter [1]:

$$\begin{aligned} S_0 &= P_{|H\rangle} + P_{|V\rangle} = 1 \\ S_x &= P_{|+\rangle} - P_{|-\rangle} \\ S_y &= P_{|R\rangle} - P_{|L\rangle} \\ S_z &= P_{|H\rangle} - P_{|V\rangle} \end{aligned} \tag{2.10}$$

Die Ausdrücke $P_{|\psi\rangle} - P_{\left|\psi^\perp\right\rangle}$ können umgeschrieben werden.
Mit $\langle\psi|\ \psi^\perp\rangle = 0$ und $P_{|\psi\rangle} + P_{\left|\psi^\perp\right\rangle} = 1$ folgt:

$$P_{|\psi\rangle} - P_{\left|\psi^\perp\right\rangle} = 2 \cdot P_{|\psi\rangle} - 1 \tag{2.11}$$

Das bedeutet, dass vier verschiedene Projektionsmessungen reichen, um die Dichtematrix ρ vollständig zu rekonstruieren. Die Summe N_{gesamt} der detektierten Photonen ergibt sich aus den Messungen $|H\rangle\langle H|$ und $|V\rangle\langle V|$. Die Stokesparameter S_x, S_y und S_z ergeben sich dann aus den normierten Anzahlen detektierter Photonen für die Messungen $|+\rangle\langle +|$, $|R\rangle\langle R|$ und $|H\rangle\langle H|$. Diese Zustände bilden die orthonormale Basis der Blochkugel.

Um eine Dichtematrix zu rekonstruieren kann jedoch auch eine nicht orthogonale Basis gewählt werden. Mit drei beliebigen linear unabhängigen Zuständen $|\psi_{i=1,2,3}\rangle$, analogen Operatoren $\tau_i \equiv |\psi_i\rangle\langle\psi_i| - \left|\psi_i^\perp\right\rangle\left\langle\psi_i^\perp\right|$ und den Parametern $T_i = tr\,(\tau_i \cdot \rho)$ kann ebenso die Dichtematrix ρ rekonstruiert werden. Es gilt weiterhin $T_0 = 1$ und $\tau_0 = \sigma_0$. Bei diesen nicht-orthogonalen Messungen müssen die neuen Parameter T_i in die Stokesparameter S_i überführt werden, um dann die Dichtematrix wie in (2.7) darzustellen. Es gilt der Zusammenhang [1]:

$$\begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} tr(\tau_0\sigma_0) & tr(\tau_0\sigma_1) & tr(\tau_0\sigma_2) & tr(\tau_0\sigma_3) \\ tr(\tau_1\sigma_0) & tr(\tau_1\sigma_1) & tr(\tau_1\sigma_2) & tr(\tau_1\sigma_3) \\ tr(\tau_2\sigma_0) & tr(\tau_2\sigma_1) & tr(\tau_2\sigma_2) & tr(\tau_2\sigma_3) \\ tr(\tau_3\sigma_0) & tr(\tau_3\sigma_1) & tr(\tau_3\sigma_2) & tr(\tau_3\sigma_3) \end{pmatrix} \cdot \begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix}$$

2.2.2 Wellenplättchen

Wellenplättchen werden für Projektionsmessungen benötigt. Durch Ausnutzung der Eigenschaften doppelbrechender Kristalle, können Wellenplättchen als unitäre Transformationen die Polarisationszustände von Licht ineinander überführen [8].

Optisch einachsige, doppelbrechende Kristalle besitzen auf Grund ihrer Struktur zwei verschiedene Brechungsindizes entlang zweier senkrechter Raumrichtungen. Die optische Achse des Kristalls ist definiert als Richtung entlang eines konstanten Brechungsindex $n_{ao}\,(\omega)$. Die senkrechte Ebene zur optischen Achse weist den anderen Brechungsindex $n_o\,(\omega)$ auf. Nun sei die optische Achse parallel zur Vorder- und Rückseite des Kristalls ausgerichtet. Fällt ein Lichtstrahl senkrecht auf den Kristall, so kann sein $\overrightarrow{E}$-Feld in zwei Komponenten - parallel und senkrecht zur optischen Achse - aufgeteilt werden. Daraus resultieren zwei parallel verlaufende ebene Wellen, die sich im Kristall ausbreiten. Diese Wellen unterscheiden sich in ihrer Geschwindigkeit v, da mit der Relation $n_{ao} = c/v_\parallel$ bzw. $n_o = c/v_\perp$ und $n_o > n_{ao}$ (im Fall von negativ einachsigen Kristallen) folgt: $v_\parallel > v_\perp$. Nach Durchqueren des Kristalls der Dicke d ist die resultierende Welle wieder eine Überlagerung der beiden Komponenten. Jedoch weisen die beiden Anteile auf Grund der unterschiedlichen Ausbreitungsgeschwindigkeiten einen Phasenunterschied $\triangle\varphi$ auf. Dieser Phasenunterschied ergibt sich aus dem

Weglängenunterschied $\triangle s = d \cdot (|n_o - n_{ao}|)$ und $\triangle\varphi = k_0 \cdot \triangle s$ mit dem Wellenvektor im Vakuum $k_0 = \frac{2\cdot\pi}{\lambda_0}$ zu [6, 16]:

$$\triangle\varphi = \frac{2 \cdot \pi}{\lambda_0} \cdot d \cdot (|n_o - n_{ao}|) \tag{2.12}$$

Die Dicke eines $\lambda/2$-Plättchen (HWP) ist so gewählt, dass für die Wellenlänge λ_0 ein relativer Phasenunterschied von $\triangle\varphi = \pi \widehat{=} 180°$ zwischen den senkrechten $\overrightarrow{E}$-Feldern erzeugt wird. Schließt der $\overrightarrow{E}$-Feldvektor des einfallenden Lichts einen beliebigen Winkel θ mit der optischen Achse ein, so dreht sich dieser bei Durchqueren des Plättchens um die optische Achse. Da sich die senkrechte $\overrightarrow{E}$-Feldkomponente relativ um 180° gedreht hat, folgt daher eine Rotation des $\overrightarrow{E}$-Feldvektors um $2 \cdot \theta$.

Durch ein $\lambda/4$-Plättchen (QWP) wird ein Phasenunterschied $\triangle\varphi = \pi/2 \widehat{=} 90°$ zwischen den Komponenten erzeugt. Somit wird linear polarisiertes Licht in elliptisch polarisiertes Licht und umgekehrt überführt. [6, 16]

Die Wirkungsweise von Wellenplättchen kann gut mit Hilfe der Blochkugel veranschaulicht werden. Für eine beliebige Position der optischen Achse dreht ein HWP den Polarisationszustand um 180° und ein QWP um 90° um die optische Achse. In Abbildung 2.3 ist der linearer Polarisationszustand $|H\rangle$ eingezeichnet, der mit der optischen Achse des HWP den Winkel $\theta = 45°$ einschließt. Bei Durchqueren des Plättchens wird der Zustand $|+\rangle$ um $2 \cdot \theta$ genau auf $|H\rangle$ gedreht.

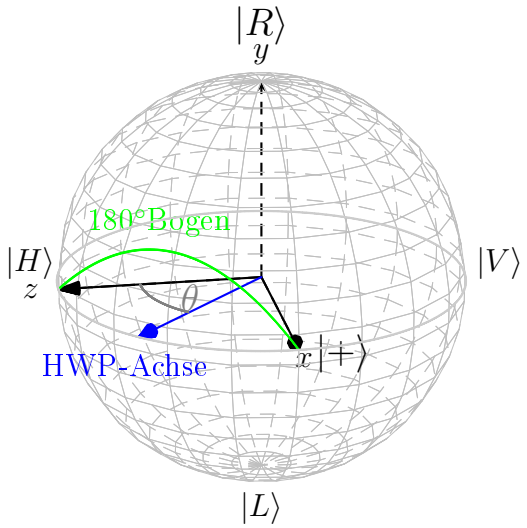

Abbildung 2.3: Wirkungsweise eines Halbwellenplättchens, dessen optische Achse mit dem Zustand $|H\rangle$ einen Winkel $\theta = 45°$ einschließt

In Abbildung 2.4 dreht ein QWP den Zustand $|R\rangle$ auf $|H\rangle$.

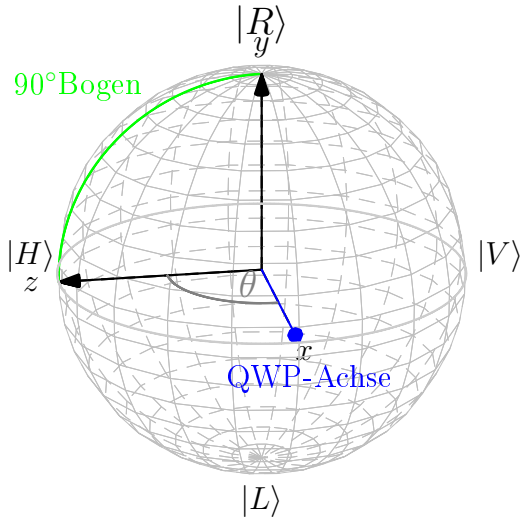

Abbildung 2.4: Wirkungsweise eines Viertelwellenplättchens, dessen optische Achse mit dem Zustand $|H\rangle$ einen Winkel $\theta = 90°$ einschließt.

Um die Wirkung von Wellenplättchen auf Polarisationszustände mathematisch zu beschreiben, werden diese Operatoren in ihrer Matrixrepräsentation dargestellt [15]:

$$\begin{aligned} HWP(\theta) &\doteq \begin{pmatrix} \cos(2\cdot\theta) & \sin(2\cdot\theta) \\ \sin(2\cdot\theta) & -\cos(2\cdot\theta) \end{pmatrix} \\ QWP(\theta) &\doteq \begin{pmatrix} \cos(\theta)^2 - i\cdot\sin(\theta)^2 & (1+i)\cdot\cos(\theta)\cdot\sin(\theta) \\ (1+i)\cdot\cos(\theta)\cdot\sin(\theta) & -i\cdot\cos(\theta)^2 + \sin(\theta)^2 \end{pmatrix} \end{aligned} \tag{2.13}$$

2.2.3 Projektionsmessungen

Um nun einen beliebigen Polarisationszustand $|\psi\rangle$ einer Projektionsmessung zu unterziehen, wird ein $\lambda/2$-Plättchen (HWP), ein $\lambda/4$-Plättchen (QWP) und ein polarisierender Strahlteiler (PBS) benötigt [7]. Der polarisierende Strahlteiler transmittiert horizontal polarisiertes Licht und reflektiert vertikal polarisiertes Licht.

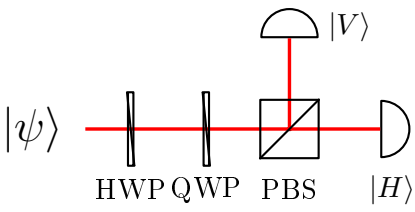

Abbildung 2.5: Aufbau für Projektionsmessung. Der Zustand $|\psi\rangle$ wird durch Einstellen der Wellenplättchen HWP und QWP und des polarisierenden Strahlteilers PBS in zwei orthogonale Zustände projiziert.

Geht man von einem beliebigen Zustand $|\psi_P\rangle = \cos\left(\frac{\theta}{2}\right)|H\rangle + \sin\left(\frac{\theta}{2}\right)\cdot e^{i\cdot\varphi}|V\rangle$ aus, in den projiziert werden soll, dann müssen die Wellenplättchen genau so justiert werden, dass der Zustand $|\psi_P\rangle$ in den Zustand $|H\rangle$ überführt wird und somit am PBS transmittiert wird. Der zu $|\psi_P\rangle$ orthogonale Zustand $\left|\psi_P^{\perp}\right\rangle$ wird dann am PBS reflektiert. Nimmt man an, die Winkel für die Plättchen seien θ_{HWP} und θ_{QWP}, so lässt sich die Wahrscheinlichkeit einen beliebigen Zustand $|\psi\rangle$ nach einer Projektionsmessung $|\psi_P\rangle\langle\psi_P|$ am transmittierten Ausgang des PBS zu messen, wie folgt ausdrücken:

$$P_{|\psi_P\rangle} = \left|\langle H|\, QWP\left(\theta_{QWP}\right)\cdot HWP\left(\theta_{HWP}\right)|\psi\rangle\right|^2 \tag{2.14}$$

Am reflektierten Ausgang erhält man die Wahrscheinlichkeit einer Projektionsmessung $\left|\psi_P^{\perp}\right\rangle\left\langle\psi_P^{\perp}\right|$:

$$P_{\left|\psi_P^{\perp}\right\rangle} = \left|\langle V|\, QWP\left(\theta_{QWP}\right)\cdot HWP\left(\theta_{HWP}\right)|\psi\rangle\right|^2 \tag{2.15}$$

Im Kapitel 2.2.1 wurden die Projektionsmessungen $|+\rangle\langle+|$, $|R\rangle\langle R|$ und $|H\rangle\langle H|$ benutzt, um eine Dichtematrix zu rekonstruieren.

Um in den Zustand $|H\rangle$ zu projizieren, müssen beide Wellenplättchen parallel zu ihrer optischen Achse ausgerichtet sein. Es muss also $\theta_{HWP} = \theta_{QWP} = 0$ gelten.

Soll in den Zustand $|R\rangle$ projiziert werden, so lässt sich leicht aus Abbildung 2.4 ablesen, dass $\theta_{HWP} = 0$ und $\theta_{QWP} = {}^{90°}/_{2} = 45°$ eingestellt werden muss.

Für den Zustand $|+\rangle$ ergeben sich die Einstellungen $\theta_{HWP} = {}^{45°}/_{2} = 22,5°$ und $\theta_{QWP} = 0$, was man an Hand der Abbildung 2.3 sehen kann.

Diese Einstellungen werden im späteren Experiment verwendet.

Die Stokesparameter ergeben sich aus den Gleichungen (2.11) und (2.10). Theoretisch reichen vier verschiedene Projektionsmessungen. Im späteren Experiment werden jedoch beide Wahrscheinlichkeiten $P_{|\psi_P\rangle}$ und $P_{\left|\psi_P^{\perp}\right\rangle}$ermittelt. Aus diesen lassen sich über (2.10) die Stokesparameter berechnen und somit die Dichtematrix rekonstruieren. [1]

2.2.4 Multiple-Qubit Tomographie

Die Erweiterung zur Single-Qubit Tomographie ist die Multiple-Qubit Tomographie. N Qubits lassen sich durch 4^N Parameter analog zu (2.7) mit Hilfe des Tensorprodukts ausdrücken:

$$\rho = \frac{1}{2^N}\sum_{i_1,i_2,...,i_N=0,x,y,z} S_{i_1,i_2,...,i_N}\sigma_{i_1}\otimes\sigma_{i_2}\otimes...\otimes\sigma_{i_N}$$

Es gilt wieder $S_{0,...,0} = 1$. $S_{i_1,i_2,...,i_N}$ sind die 4^N Stokesparameter für mehrere Qubits. Der Index $1, 2, ..., N$ gibt an, welchem der N Qubits die entsprechende Projektionsmessung gilt.

Für Projektionsmessungen $\tau_{i=1,2,3}$ in beliebigen Basen gilt die allgemeine Relation:

$$\begin{aligned} T_{i_1,i_2,...,i_N} &= tr\left\{\left(\tau_{i_1}\otimes\tau_{i_2}\otimes...\otimes\tau_{i_N}\right)\rho\right\} \\ &= \frac{1}{2^N}\sum_{j_1,j_2,...,j_N=0,x,y,z} tr\left\{\tau_{i_1}\sigma_{j_1}\right\}\cdot tr\left\{\tau_{i_2}\sigma_{j_2}\right\}\cdot...\cdot tr\left\{\tau_{i_N}\sigma_{j_N}\right\}\cdot S_{j_1,j_2,...,j_N} \end{aligned}$$

Weiterhin ist $\tau_{i_0} = \sigma_0$. Der Ausdruck für die T-Parameter kann nun wieder durch die Wahrscheinlichkeiten $P_{|\psi_{i_n}\rangle}$ und $P_{\left|\psi_{i_n}^{\perp}\right\rangle}$ für die entsprechende Projektionsmessung $\tau_{i_n} = |\psi_{i_n}\rangle\langle\psi_{i_n}| - \left|\psi_{i_n}^{\perp}\right\rangle\left\langle\psi_{i_n}^{\perp}\right|$ für das n-te Qubit ausgedrückt werden. Es ergibt sich:

$$T_{i_1,i_2,...,i_N} = \left(P_{|\psi_{i_1}\rangle} \pm P_{\left|\psi_{i_1}^{\perp}\right\rangle}\right)\otimes\left(P_{|\psi_{i_2}\rangle} \pm P_{\left|\psi_{i_2}^{\perp}\right\rangle}\right)\otimes...\otimes\left(P_{|\psi_{i_N}\rangle} \pm P_{\left|\psi_{i_N}^{\perp}\right\rangle}\right)$$

Addition der Wahrscheinlichkeiten ergibt sich für $i_n = 0$ und die Subtraktion für $i_n = x, y, z$. Wird an beiden Ausgängen des PBS $P_{|\psi_{in}\rangle}$ bzw. $P_{\left|\psi_{in}^{\perp}\right\rangle}$ gemessen, so werden $2 \cdot N$ Detektoren benötigt, die N-fache Koinzidenzen messen, um den N-Qubit Zustand zu rekonstruieren. Tatsächlich müssen nicht 4^N verschiedene Analysen, auf Grund der 4^N Stokesparameter, durchgeführt werden. Wurden die Wahrscheinlichkeiten $P_{|\psi_{in}\rangle}$ und $P_{\left|\psi_{in}^{\perp}\right\rangle}$ gemessen, so erhält man sofort die beiden Faktoren $\left(P_{|\psi_{in}\rangle} \pm P_{\left|\psi_{in}^{\perp}\right\rangle}\right)$. Daher reichen 3^N verschiedene Analysen, das bedeutet 3^N verschiedene Einstellungen aller Wellenplättchen, wobei dafür N mal der Aufbau aus Abbildung 2.5 benutzt wird. [1]

2.2.5 Fehleranalyse und Fidelity

Exakte Tomographie lässt sich nur mit perfekten Versuchsbedingungen und einem Ensemble mit unendlich vielen Photonen durchführen. Im echten Experiment kann dies natürlich nicht mehr gewährleistet werden. Daher sind eine Fehleranalyse und numerische Verfahren zur Bestimmung des Zustands notwendig.

Ein Fehler kann zum Beispiel aus einem Fehler der Basis für die Projektionsmessungen kommen. Die Projektionsmessungen werden durch die verschiedenen Einstellungen der Wellenplättchen realisiert, sind diese jedoch nicht exakt eingestellt, ergibt sich ein Fehler im rekonstruierten Zustand. Im realen Experiment können die Wellenplättchen nicht perfekt eingestellt werden, jedoch kann versucht werden den Fehler möglichst gering zu halten. Mit der automatischen Kalibration der Wellenplättchen, was dem Ziel meiner Arbeit entspricht, ergibt sich eine Möglichkeit diesen Fehler zu minimieren und somit bessere Resultate zu erzielen.

Optimalerweise kann der Zustand auf einen kleinen Ball in der Blochkugel reduziert werden. Bei der Messung reiner Zustände liegt ein Teil dieses Balls oft außerhalb der Blochkugel. Dies würde aber einem unphysikalischen Zustand entsprechen. Daher verwendet man zum Beispiel die "maximum likelihood" Technik als numerische Methode, um einen physikalisch sinnvollen Zustand zu ermitteln.[1]

Um den resultierenden Zustand mit dem zu erwartenden theoretischen Zustand zu vergleichen, benutzt man die Fidelity. Die Fidelity $\mathcal{F}$ gibt an zu welchem Anteil der Zustand ρ dem Zustand σ entspricht:

$$\mathcal{F}(\rho, \sigma) = \left(tr\sqrt{\rho^{1/2} \cdot \sigma \cdot \rho^{1/2}}\right)^2$$

Für den Vergleich mit einem reinen Zustand lässt sich dies umformen [11]:

$$\mathcal{F}(|\psi\rangle, \rho) = \langle\psi| \rho |\psi\rangle$$

3 Automatische Kalibration

Der Hauptteil meiner Arbeit war die Entwicklung eines C++ Programms zur automatischen Kalibration eines Polarisationsanalysators. Ein Polarisator, ein $\lambda/2$-Wellenplättchen und ein $\lambda/4$-Wellenplättchen werden dazu in Motoren eingebaut, welche über einen Computer gesteuert werden können. Die Spannung an Photodioden kann über ein Multimeter gemessen werden. Ziel ist es nun einen passenden Algorithmus zu finden, der durch Spannungsmessungen und Steuern der Motoren die Nullstellungen des Polarisators und der Wellenplättchen ermittelt. Das bedeutet, nur der horizontal polarisierte Anteil des eingestrahlten Lichts wird durchgelassen. Nach erfolgter Kalibration kann dann der Polarisationszustand von Photonen analysiert werden, so wie im Kapitel 2.2 Qubit Tomographie beschrieben.

3.1 Vorbereitungen

3.1.1 Einstellung des Aufbaus

Für die automatische Kalibration wird der Aufbau in Abbildung 3.1 verwendet. Dieser Aufbau ist eine Erweiterung des in Abschnitt 2.2.3 beschriebenen Aufbaus 2.5 für Projektionsmessungen.

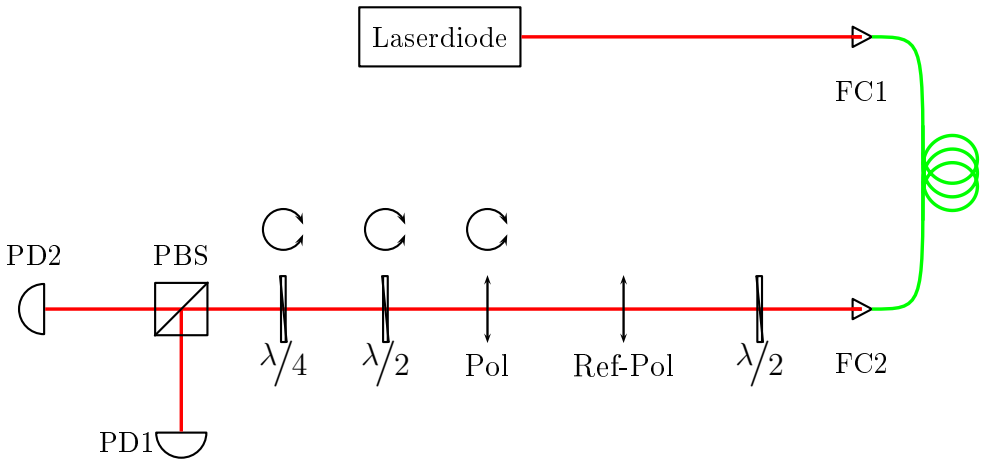

Abbildung 3.1: Grundaufbau für die automatische Kalibration: Der Strahl einer Laserdiode wird in eine Glasfaser eingekoppelt, diese führt zu den weiteren Komponenten. Nach der Auskopplung durchläuft der Laserstrahl ein Wellenplättchen, den Referenzpolarisator, einen Polarisator in einem Motor und zwei weitere Wellenplättchen in Motoren. Anschließend teilt sich der Strahl an einem polarisierenden Strahlteiler auf und beide Teilstrahlen werden durch Photodioden detektiert.

Ich habe eine Laserdiode mit einer Leistung von ca. $5\,mW$ benutzt, die rotes Licht der Wellenlänge $780\,nm$ emittiert. Das emittierte Licht wird in eine single-mode Glasfaser eingekoppelt, die als Modenfilter dient. Durch einen Kollimator ($FC2$) wird das Licht wieder ausgekoppelt und kollimiert.

Für die spätere Kalibration wird ein Referenzpolarisator ($Ref\text{-}Pol$) benötigt, der nur horizontal polarisiertes Licht durchlässt. Das $\lambda/2$-Plättchen zwischen $Ref\text{-}Pol$ und $FC2$ wird so eingestellt, dass nach dem Referenzpolarisator möglichst viel Leistung mit einem Powermeter gemessen werden kann. Da das Licht aus der Glasfaser eine beliebige Polarisationsrichtung aufweist, wird somit verhindert, dass sich dieses genau auslöscht. Besonders zu beachten ist, dass horizontal polarisiertes Licht sich nun immer auf das transmittierte Licht des Referenzpolarisators bezieht. Alle weiteren Komponenten werden auf diesen Referenzpolarisator eingestellt.

Der polarisierende Strahlteiler (PBS) wird nun so justiert, dass horizontal polarisiertes Licht transmittiert und vertikal polarisiertes Licht reflektiert wird. Dazu wird mit einem Powermeter die Leistung des reflektierten Lichts gemessen, welche durch Drehen des PBS minimiert werden muss. Es ergibt sich einen Kontrast $C = \frac{P_V}{P_H} = \frac{3{,}0\,\mu W}{2150\,\mu W} \cong \frac{1}{700}$ zwischen der reflektierten und transmittierten Leistung.

Zwischen dem justierten PBS und dem Referenzpolarisator werden die Motoren mit den Wellenplättchen und dem Polarisator wie in der Skizze 3.1 eingebaut. Dass sich diese Bauteile in Motoren befinden, wird durch die Bögen mit Pfeilenden in der Abbildung gekennzeichnet. Die Motoren können sich um die Achse parallel zum Strahlverlauf drehen. Die Nullposition entspricht den Motorstellungen, bei denen die optischen Achsen der Plättchen und des Polarisators parallel zur horizontalen Polarisationsrichtung liegen. Zur Veranschaulichung auf der Blochkugel würde dies einem Winkel $\theta = 0$ für die optischen Achsen entsprechen[1].

Die zwei Photodioden ($PD1$ und $PD2$) messen die Intensität des reflektierten und transmittierten Lichts.

3.1.2 Motoren

Die Motoren sind computergesteuerte Schrittmotoren, die zwei Mitglieder der Arbeitsgruppe Weinfurter, Sebastian Nauerth und Martin Fürst, gebaut und programmiert haben.

Eine Motorstufe besteht aus zwei Motoren ("motor 0" und "motor 1"), die unabhängig von einander gesteuert werden können. In eine Motorstufe wird der Polarisator eingesetzt und in die andere Motorstufe werden die Wellenplättchen eingesetzt. Bild 3.2 zeigt eine Motorstufe mit eingesetzten Wellenplättchen.

[1] siehe Abbildung 2.3 und 2.4

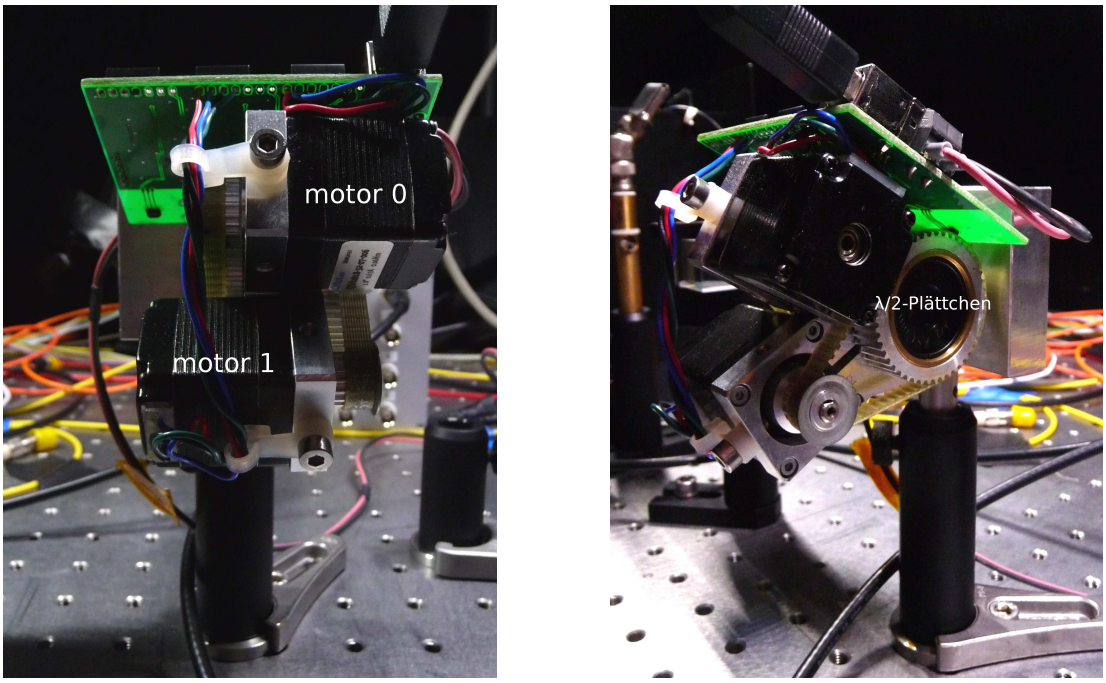

Abbildung 3.2: Zwei Fotos einer Motorstufe mit den Motoren “motor 0” und “motor 1”. In “motor 0” ist das $\lambda/2$-Plättchen eingebaut. Das $\lambda/4$-Plättchen ist in “motor 1” eingebaut.

Eine 360° Drehung eines Motors entspricht 9600 Schritten, das bedeutet der Motor kann auf $0,0375°$ genau eingestellt werden. Die Motoren sind mit einem kleinen Magneten und einem Hall-Sensor ausgestattet, somit kann durch eine Referenzfahrt eine feste Position wieder gefunden werden. Alle anderen Positionen beziehen sich auf diese Referenzposition.

Die Motoren werden über die serielle Schnittstelle (RS232) gesteuert. Diese wird durch einen USB/RS232-Umsetzer emuliert. Das bedeutet, die Motoren werden via USB (Universal Serial Bus) an den Computer angeschlossen, doch die Kommunikation erfolgt über ein serielles Protokoll. Im Dateisystem wird die Schnittstelle als Terminal-Device `/dev/ttyUSBn` eingebunden. Um die Schnittstelle zu steuern, werden Funktionen verwendet, die die Terminal-Parameter, wie zum Beispiel die Baudrate, einstellen. [4]

Das Programm *seriell.c*, welches die Steuerung der Motoren ermöglicht, wurde mir zur Verfügung gestellt. Ich möchte hier nur kurz die wichtigsten Funktionen des Programms erklären. Die Funktion $init\_ser(char * devname)$ dient zur Initialisierung der Schnittstelle. Anschließend kann durch die Funktion $send\_command(const\,char * str)$ ein Kommando an den Motor gesendet werden. Mit der Funktion $close\_ser()$ wird die Schnittstelle wieder geschlossen.

Die Tabelle 3.1 soll einen kleinen Überblick über die wesentlichen Kommandos geben, die im späteren Programm benutzt werden.

Befehl	
restart	Motor wird neu gestartet. Dabei wird eine Referenzfahrt durchgeführt: Der Motor sucht mit Hilfe eines Magneten und eines Hall-Sensors seine Referenzposition. Ist diese gefunden, fährt er zu dieser.
refall	Die Referenzfahrt wird durchgeführt und der Motor fährt anschließend zur Referenzposition.
motor 0/1	Die folgenden Befehle beziehen sich auf "motor 0" oder "motor 1".
go to x	Motor geht zu der Position x, wobei ab der Referenzposition gezählt wird.

Tabelle 3.1: Wichtige Befehle, die über die Funktion *send_command("Befehl")* dem Motor gegeben werden können.

3.1.3 Multimeter

Über das Multimeter können die Spannungen an den Photodioden ausgelesen und an den Computer übermittelt werden.

Ebenso wie die Motoren wird das Multimeter via USB an den Computer angeschlossen. Die Kommunikation erfolgt wieder über eine emulierte serielle Schnittstelle. Das Multimeter besitzt acht Eingänge, in meiner Arbeit werden jedoch nur zwei benötigt. An jedem dieser acht Eingänge kann die Spannung abgegriffen werden. Über das Terminalprogramm *adc_ad7609_reader*, welches mir zur Verfügung gestellt wurde, können diese Spannungen ausgelesen werden. Als Rückgabe erhält man im Terminal die Werte der acht Spannungen.

Bild 3.3 zeigt das Multimeter. An Eingang 1 wird $PD1$ und an Eingang 2 wird $PD2$ angeschlossen.

Abbildung 3.3: Multimeter mit acht Eingängen. An den Eingängen 1 und 2 sind über ein Koaxialkabel die Photodioden angeschlossen.

3.1.4 Strukturierung

Für die Programmierung der automatischen Kalibration werden viele Funktionen gebraucht, die kleine Zwischenschritte übernehmen. Um den Programmtext übersichtlicher zu gestalten, werden diese Funktionen in verschiedenen Dateien implementiert und erst beim Kompilieren des Programms miteinander verlinkt.

Da die genaue Programmierung einiger Funktionen nicht für das Verständnis der automatischen Kalibration relevant ist, werde ich diese nicht im Detail erklären. Diese Funktionen befinden sich in der Datei *Calib-functions.cpp*. Der Quellcode ist soweit auskommentiert, dass die Wirkung jeder Funktion verstanden werden kann.

Die Funktion zur Kalibration des Polarisators befindet sich in *PolCalib.cpp* und die verschiedenen Funktionen zur Kalibration der Wellenplättchen in *LambdaCalib.cpp*. Diese Funktionen greifen auf Funktionen aus *Calib-functions.cpp* zurück.

Der Quellcode zum Hauptprogramm ist *calibration.cpp*. Das Programm *calibration* kann im Terminal gestartet werden und führt die Kalibration aus.

Um die automatische Kalibration zu testen und eine Statistik über die Ergebnisse aufzustellen, wird das Programm *qubit_test_tom* benutzt. Dieses ist nur ein Hilfsprogramm, um mehrere Kalibrationen hintereinander auszuführen.

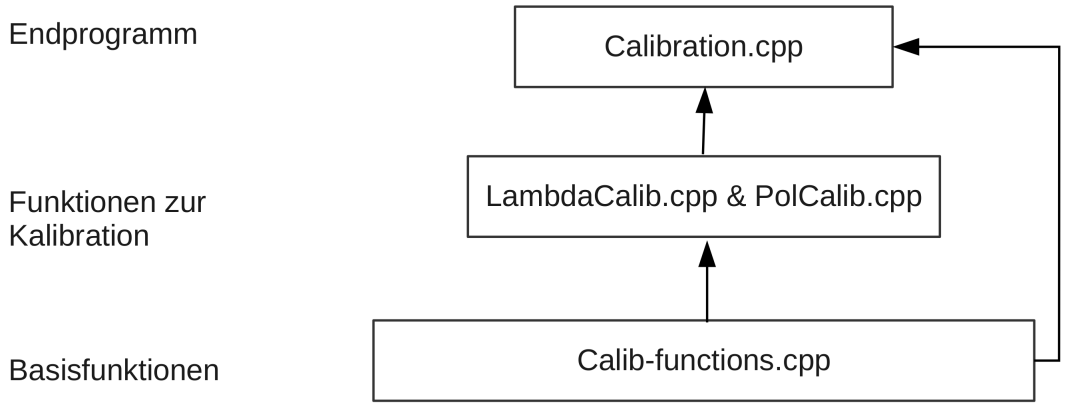

Abbildung 3.4: Abhängigkeiten der Funktionen aus den verschiedenen Programmen. Die Basis bildet *Calib-functions.cpp*. Funktionen aus diesem Quellcode werden für alle weiteren Programme verwendet. *PolCalib.cpp* und *LambdaCalib.cpp* enthalten Funktionen zur Kalibration. *calibration.cpp* ist das Hauptprogramm.

3.2 Kalibration des Polarisators

Zuerst soll der Polarisator kalibriert werden. Dazu werden die Wellenplättchen und der PBS noch nicht benötigt. Abbildung 3.5 zeigt den erforderlichen Aufbau, der durch einfaches Umstellen von $PD1$ aus Aufbau 3.1 erreicht werden kann.

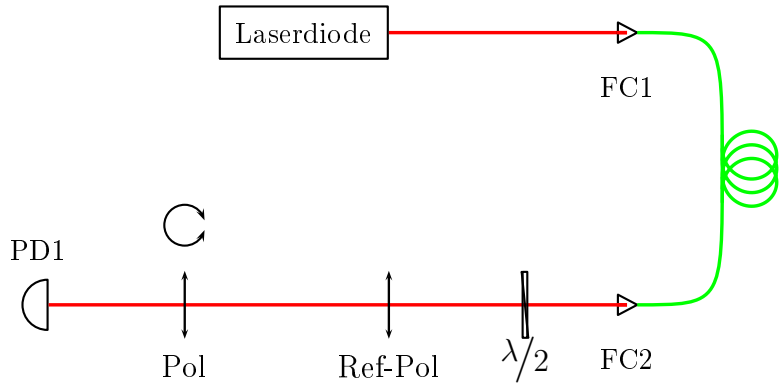

Abbildung 3.5: Aufbau zur Kalibration des Polarisator. Der Laserstrahl durchläuft nach Auskopplung aus der Glasfaser ein Wellenplättchen, den Referenzpolarisator und den zu kalibrierenden Polarisator. Die Photodiode misst die Lichtintensität.

3.2.1 Idee

Nach einer Referenzfahrt schließt die optische Achse des Polarisators einen unbekannten Winkel θ_{off} mit der horizontalen Achse ein.

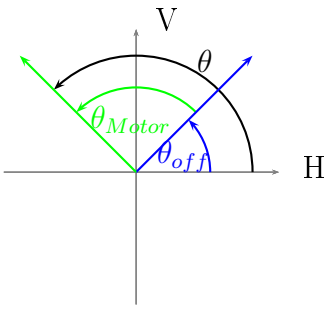

Abbildung 3.6: Die optische Achse des Polarisators schließt nach einer Referenzfahrt des Motors einen beliebigen Winkel θ_{off} mit der horizontalen Achse ein. Fährt der Motor nun um θ_{Motor} weiter, so befindet dich der Polarisator an der Position $\theta = \theta_{Motor} + \theta_{off}$ in Bezug auf die horizontale Achse.

Ziel ist es nun diesen Winkel θ_{off} zu bestimmen. Der lineare Polarisationszustand, der durch den Polarisator transmittiert wird, lässt sich für eine beliebige Position $\theta = \theta_{Motor} + \theta_{off}$ schreiben als $|\psi\rangle = \cos(\theta)|H\rangle + \sin(\theta)|V\rangle$. Der Zustand $|H\rangle$ nach dem Referenzpolarisator wird auf diesen Zustand projiziert. Als Wahrscheinlichkeitsamplitude erhalten wir $P =$

$|\langle\psi|\ H\rangle|^2 = \cos(\theta)^2$. An der Photodiode wird somit über einen Widerstand eine Spannung

$$U \propto \cos(\theta)^2 \tag{3.1}$$

abgegriffen. Durch Umformen ergibt sich [2]:

$$U \propto \cos(\theta)^2 = \frac{1}{2}\cos(2\theta) + \frac{1}{2} = \frac{1}{2} + \frac{1}{2}\sin\left(2\theta_{Motor} + 2\theta_{off} + \frac{\pi}{2}\right)$$

Das bedeutet für verschiedene Winkel $\theta_{Motor} = \theta - \theta_{off}$, können die jeweiligen Spannungen gemessen werden und die erhaltenen Datenpunkte können an eine Sinus-Funktion mit einer Periode von $180° = 4800\,Schritte$ gefittet werden. Anschließend kann leicht das Maximum der gefitteten Funktion berechnet werden. $\theta_{max} = 0 - \theta_{off}$ entspricht dann der Nullposition des Polarisators.

Es ist auch möglich ohne einen Fit das Maximum zu suchen. Dazu wird einfach die Motorposition mit der höchsten Spannung aus den Messdaten gesucht. Dies ist jedoch nicht zu empfehlen, da ein Fit alle vorhandenen Datenpunkte berücksichtigt und somit das bessere Resultat liefert.

Ziel des Programms ist es also, den Motor immer um einen bestimmten Winkel weiter zu drehen und jeweils die Spannung an $PD1$ zu messen. Der Winkel, um den weiter gedreht wird, kann in eine Schrittweite für den Motor umgerechnet werden. Anschließend werden die Daten gefittet. Daraus ermittelt sich die Position θ_{max}. Die Visibility ergibt sich zu

$$V = \frac{U_H - U_V}{U_H + U_V} \tag{3.2}$$

aus den Spannungen U_H an der Position θ_{max} und U_V an der Position $\theta_{min} = \theta_{max} \pm 90°$.

3.2.2 Programmierung

Die Funktion zur Kalibration des Polarisator befindet sich im Programm *PolCalib.cpp*. In Zeile 1-22 werden alle nötigen Bibliotheken eingebunden, in Zeile 23-26 werden die Headerfiles anderer Programme eingebunden, die die nötigen weiteren Funktionen enthalten und in den Zeilen 28 und 29 wird die benutzte Anzahl der Motoren und Photodioden definiert. Diese Programmzeilen befinden sich in ähnlicher Form in jedem der Programme.

Die Funktion "$PolCalib(double\,steps,\ FILE\ *results,\ double\,offset[\,],double\,array[\,])$" in den Zeilen 44-142 führt die Kalibration des Polarisators durch. Als Variablen werden dieser die Schrittweite *steps* in Grad, eine Datei *results*, ein Feld $offset$ mit dem Offset des Multimeters und ein Feld *array* übergeben.

Zuerst wird der Motor mit dem Polarisator neu gestartet (Zeile 46) und eine Referenzfahrt wird durchgeführt. Diese Referenzfahrt soll immer die gleiche Position liefern. In den Zeilen 50-61 wird eine Matrix initialisiert, die in der ersten Spalte die aufsummierten Schritte *steps* in Grad enthält. Das bedeutet, in jeder Zeile dieser Spalte steht die Position des Motors in Grad. Die anderen Spalten wären theoretisch für die Positionen der Wellenplättchen und werden daher gleich Null gesetzt. Diese Matrix $mat1$ wird anschließend (Zeile 63) in der Standardausgabe ausgegeben. Somit erhält der Benutzer während der Kalibration die nötige Information über den Ablauf des Programms. Mit der Funktion *convertmat* (Zeile 65) wird die Matrix nun in die für den Motor nötigen Schritte statt Grad konvertiert. Nun wird eine neue Matrix $mat2$ initialisiert. In diese sollen die Motorschritte in der ersten

Spalte, die gemessene Spannung an $PD1$ in der vierten Spalte und die gemessene Spannung an $PD2$ in der fünften Spalte gespeichert werden. Wichtig ist nur die Spannung an $PD1$, da diese Photodiode hinter den Polarisator gestellt wurde und daher die Intensität des Lichts, welches durch den Polarisator transmittiert wird, misst. In Zeile 73-93 befindet sich eine Schleife. In jeder Runde fährt der Motor an die passende Position, die in der Matrix $mat1$ gespeichert wurde (Zeile 77) und speichert die Position in Matrix $mat2$ (Zeile 79-81). Anschließend werden die Spannungen ausgelesen (Zeile 84-86) und ebenfalls in die zugehörigen Matrixelemente von $mat2$ gespeichert (Zeile 90-93). Nach Beenden der Schleife wird die Matrix $mat2$ in die Datei $results$ geschrieben (Zeile 96). Da nun alle Datenpunkte gemessen und gespeichert wurden, müssen diese an eine Sinus-Kurve gefittet werden. Dies übernimmt die Funktion $fitting()$, der als Parameter die Matrix mit den Datenpunkten übergeben wird (Zeile 100). Wie die Fit-Routine genau funktioniert wird später erklärt. Die Funktion $fitting()$ ermittelt die Nullstellung θ_{max} des Polarisators und speichert diese in die Datei $hpol\_is\_at.txt$. In den Zeilen 104-115 wird die Nullstellung aus der Datei gelesen und der Motor fährt nun an diese Nullstellung (Zeile 118). Die Spannung wird an dieser Position gemessen (Zeile 120-122). Anschließend fährt der Motor um $90° = 2400\,Schritte$ weiter (Zeile 124-125) und auch an dieser Position wird die Spannung gemessen (Zeile 127-128). Aus diesen Spannungen lässt sich mit (3.2) die Visibility errechnen (Zeile 130). Abschließend fährt der Motor wieder an seine gefundene Nullstellung (Zeile 134) und die Nullposition θ_{max}, sowie die Visibility werden in das Feld $array$ für die weitere Verwendung geschrieben (Zeile 137-138).

In dem Flussdiagramm 3.7 wird der Programmablauf nochmal veranschaulicht.

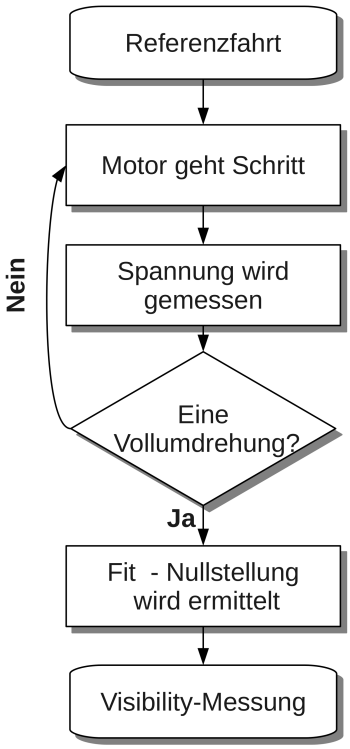

Abbildung 3.7: Flussdiagramm der Funktion$PolCalib()$ aus dem Programm $PolCalib.cpp$. Der grundlegende Ablauf ist dargestellt.

3.2.3 Durchführung und Resultate

Nach Beenden der Kalibration des Polarisators steht dieser an seiner Nullposition. Die Matrix *mat*2 mit den Schritten und den zugehörigen Spannungswerten wurde in eine Datei gespeichert. Der Ablauf der Fit-Routine, die ermittelte Nullposition und die gemessene Visibility wurden ebenfalls in diese Datei gespeichert. Diese Daten können nun ausgewertet werden.

Als erstes wurde die Kalibration für eine Schrittweite von 1° ≅ 27 Schritte getestet. Die gemessenen Daten für eine Kalibration sind in der Graphik 3.8 als Punkte geplottet.

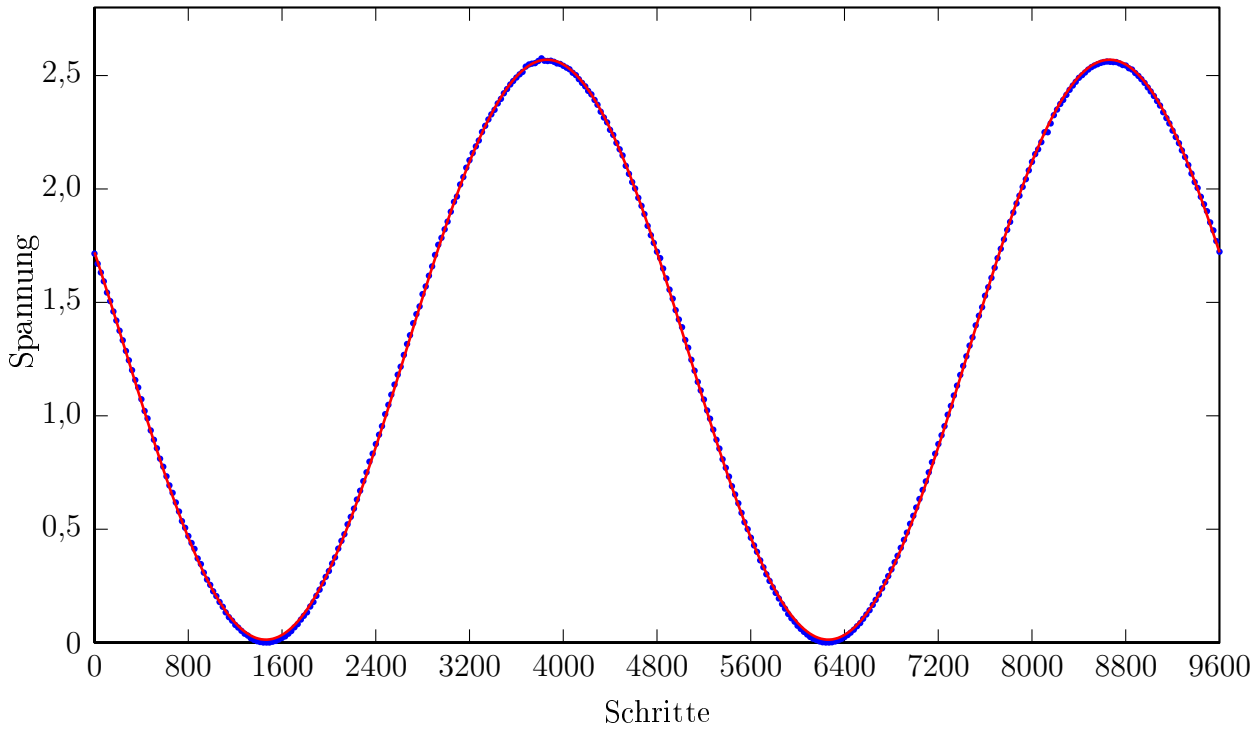

Abbildung 3.8: Polarisator Kalibrationsmessung in 1° Schritten mit einer ermittelten Nullposition bei 3861 Schritten (Punkte). Die Kurve entspricht der gefitteten Sinus-Kurve. Es ergibt sich eine gemessene Visibility V=0,999890.

Aus dieser Kurve ermittelt die Fit-Routine die Nullstellung des Polarisators bei 3861 Schritten. In der obigen Abbildung ist die gefittete Kurve rot eingezeichnet. Die gemessene Visibility betrug hier $V = 0.999890$.

Um eine Statistik aufzustellen wurde mit dem Hilfsprogramm *qubit_test_tom* für verschiedene Schrittweiten mehrere Kalibrationen durchgeführt. Mit Matlab werden die erhaltenen Daten ausgewertet. Interessant ist, wie groß die Schwankung der ermittelten Nullpositionen ist, wie gut die Kalibration für große Schrittweiten klappt und wie hoch die gemessene Visibility ist.

Insgesamt wurden 314 Kalibrationsmessungen für verschiedene Schrittweiten von 1° bis 30° durchgeführt. In Abbildung 3.9 wurde die Häufigkeit der ermittelten Nullpositionen des Polarisators geplottet. Dieses Histogramm zeigt noch nicht die Unterschiede für verschiedene Schrittweiten. Wie man sehen kann, liegt ein großer Teil der Messungen im Bereich des

Mittelwerts. Jedoch weichen einige Ergebnisse stark ab. Somit ergibt sich eine Standardabweichung von $93,6$ Schritten $\cong 3,5°$.

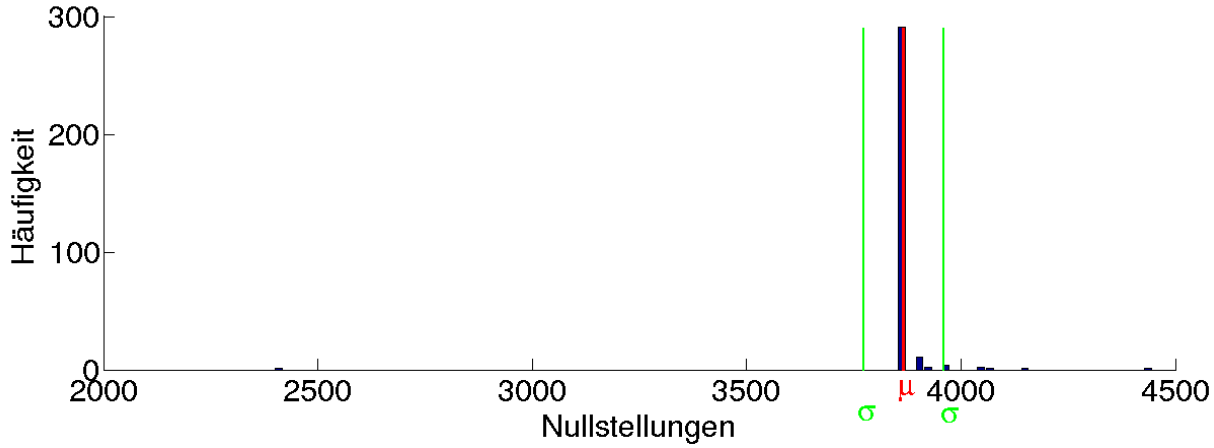

Abbildung 3.9: Histogramm für alle Kalibrationsmessungen mit verschiedenen Schrittweiten von 1° bis 30°. Mittelwert $\mu = 3865,8$ Schritte; Standardabweichung $\sigma = 93,6$ Schritte $\cong 3,5°$ in Schritten

Um einen Eindruck der Ergebnisse, die im Bereich des Mittelwerts liegen und in obiger Abbildung nicht aufgelöst werden, zu erhalten, sind in Abbildung 3.10 alle Nullstellungen, die weniger als 40 Schritte vom Mittelwert entfernt liegen geplottet. In diesem Bereich liegen 294 Kalibrationsmessungen. Der Mittelwert liegt nun bei 3862,6 Schritten und es ergibt sich eine wesentlich geringere Standardabweichung von 1,1 Schritten, dies entspricht 0,042°.

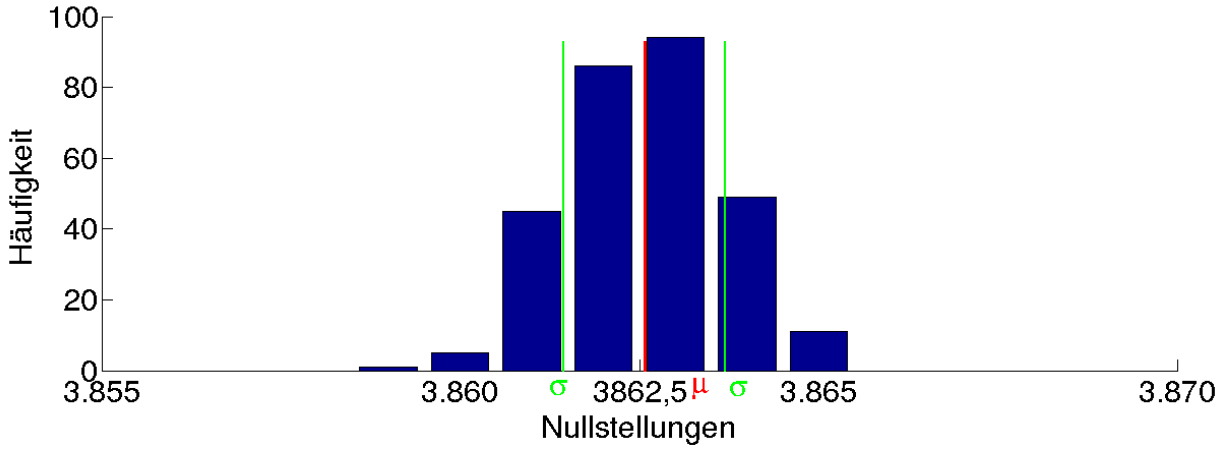

Abbildung 3.10: Histogramm für erfolgreiche Kalibrationsmessungen verschiedener Schrittweiten. Es wurden nur Ergebnisse geplottet, die nicht mehr als 40 Schritte vom Mittelwert 3865,8 abweichen. Neuer Mittelwert $\mu = 3862,6$ Schritte; neue Standardabweichung $\sigma = 1,1$ Schritte $\cong 0,042°$

Die Ergebnisse aus Abbildung 3.10 repräsentieren alle erfolgreichen Kalibrationen. Die Tatsache, dass ungefähr 6% aller Kalibrationsmessungen stark vom Mittelwert abweichen, liegt wahrscheinlich an der Referenzfahrt am Anfang der Kalibration. Diese nimmt unterschiedlich viel Zeit in Anspruch. Daher wird im Quellcode *Calib-functions.cpp* bei der Funktion

startresetclose() (Zeile 50-66) 20 Sekunden gewartet bevor das Programm weiter läuft. Meist reicht diese Zeit aus, um die Referenzfahrt erfolgreich durchzuführen. Manchmal findet der Motor über den Hall-Sensor und Magneten jedoch nicht gleich die Referenzposition und es werden daher mehr als 20 Sekunden benötigt. Läuft das Programm jedoch bereits weiter, so startet die Messung nicht an der Referenzposition und somit stimmen die Messdaten nicht. Lässt man diese Messdaten plotten, so ergibt sich eine verschobene Sinus-Kurve und somit ein falsches Ergebnis. Da die Referenzfahrt meistens nur 5 Sekunden dauert, ist es sinnvoll maximal 20 Sekunden zu warten. Eine Lösung wäre auf den Rückgabewert des Motors nach erfolgreicher Referenzfahrt zu warten. Leider haben die benutzten Motoren diese Möglichkeit nicht. In Zukunft wäre es sinnvoll eine Rückmeldung des Motors nach einer Referenzfahrt einzubauen.

Ob eine Kalibration erfolgreich ist, kann über die Visibility geprüft werden. Stimmt die gefunden Nullposition mit der tatsächlichen Nullposition gut überein, so muss sich eine hohe Visibility ergeben.

In Abbildung 3.11 ist das Histogramm für die Visibility der 294 erfolgreichen Kalibrationsmessungen geplottet.

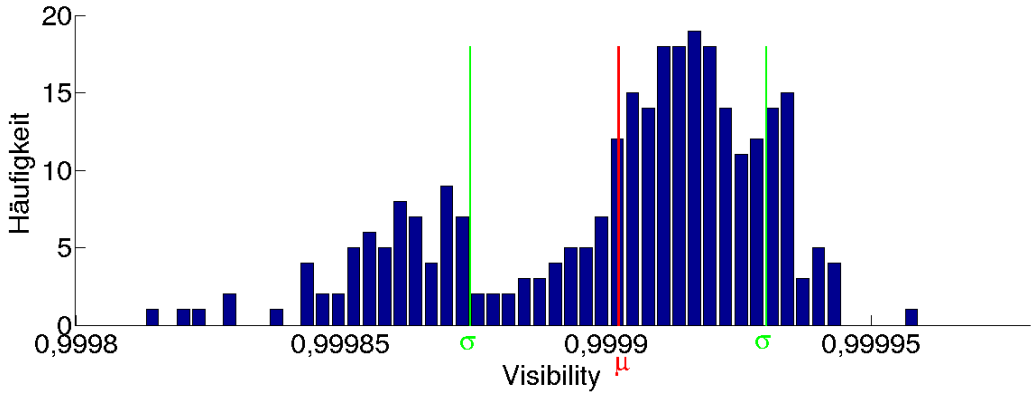

Abbildung 3.11: Histogramm der Visibility für erfolgreiche Kalibrationsmessungen. Neuer Mittelwert $\mu = 0,99990$; Neue Standardabweichung $\sigma = 0,00003$.

Für eine erfolgreiche Kalibration liegt die Visibility mindestens bei 0,99980. Der Mittelwert beträgt 0,99990. Dies wurde bei vollständig abgedunkelten Experimentierbedingungen erreicht. Für den Mittelwert der Visibility aller Messungen aus Abbildung 3.9 ergibt sich 0,9962. Dieser Mittelwert ist deutlich geringer. Der Experimentator kann also nach der Kalibration das Resultat über die Visibility einstufen.

Um die Zeit einer Kalibration möglichst gering zu halten, bei trotzdem guten Resultaten, müssen die Ergebnisse für verschiedene Schrittweiten verglichen werden. Je größer die Schrittweite gewählt wird, desto schlechter werden die Ergebnisse sein. Aber je kleiner die Schrittweite ist, desto mehr Zeit benötigt die Kalibration. Im Folgenden sind die Histogramme für einige ausgewählten Schrittweiten geplottet.

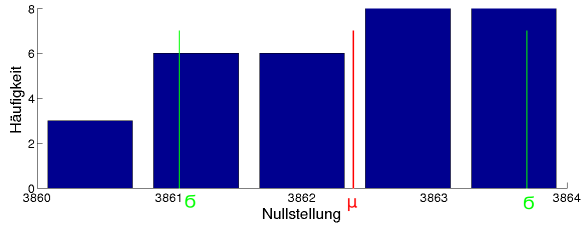

Abbildung 3.12:
Histogramm für Kalibration mit einer Schrittweite von 1°. Mittelwert $\mu = 3862,0\,Schritte$; Standardabweichung $\sigma = 1,3\,Schritte$

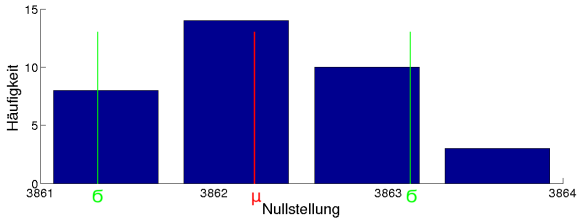

Abbildung 3.13:
Histogramm für Kalibration mit einer Schrittweite von 15°. Mittelwert $\mu = 3862,2\,Schritte$; Standardabweichung $\sigma = 0,9\,Schritte$

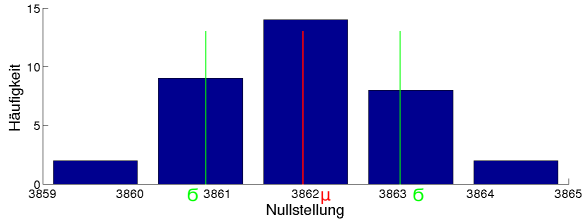

Abbildung 3.14:
Histogramm für Kalibration mit einer Schrittweite von 30°. Mittelwert $\mu = 3862,0\,Schritte$; Standardabweichung $\sigma = 1,1\,Schritte$

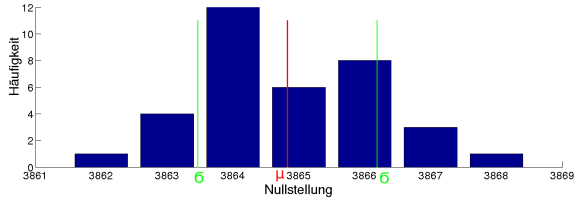

Abbildung 3.15:
Histogramm für Kalibration mit einer Schrittweite von 45°. Mittelwert $\mu = 3864,8\,Schritte$; Standardabweichung $\sigma = 1,4\,Schritte$

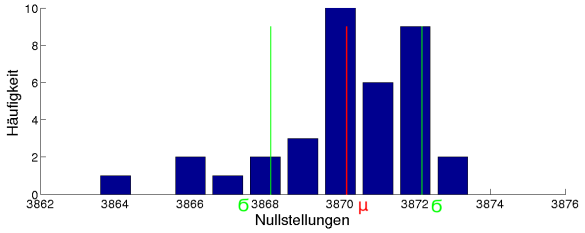

Abbildung 3.16:
Histogramm für Kalibration mit einer Schrittweite von 60°. Mittelwert $\mu = 3870,0\,Schritte$; Standardabweichung $\sigma = 2,0\,Schritte$

Die Standardabweichung steigt ab einer Schrittweite von 45° an und der Mittelwert ist verschoben. Möchte man ein gutes Resultat erzielen, sollte man eine Schrittweite wählen, die geringer als 45° ist. Problem ist eher nicht die ansteigende Standardabweichung, die mit $2\,\text{Schritten} = 0,13°$ für 60° Schrittweite immer noch relativ gering ist, sondern die verschobene Nullposition. Für 45° Schrittweite ergibt sich eine Differenz von 2 Schritten im Gegensatz zur Nullposition für 1° Schrittweite. Bei einer Schrittweite von 60° weicht der Mittelwert um $8\,\text{Schritte} \cong 0,3°$ ab.

Zum Vergleich zu Abbildung 3.8 sind in Abbildung 3.17 die Datenpunkte und gefittete Kurve für eine Kalibration mit einer Schrittweite von 60° geplottet.

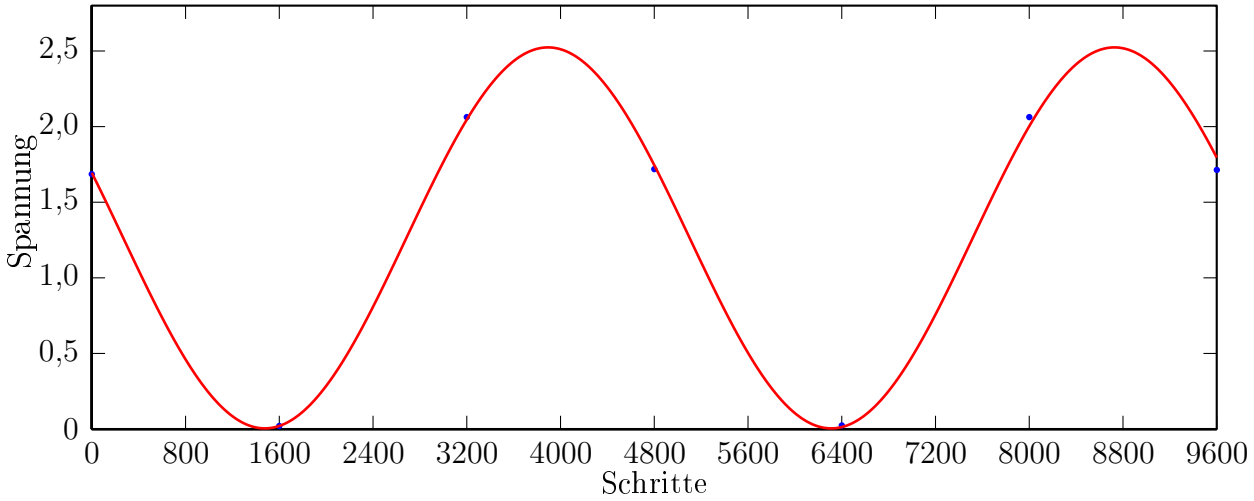

Abbildung 3.17: Polarisator Kalibrationsmessung in 60° Schritten mit einer ermittelten Nullposition bei 3872 Schritten (Punkte). Die Kurve entspricht der gefitteten Sinus-Kurve. Es ergibt sich eine gemessene Visibility V=0,9977.

Die geringere Visibility von 0,9977 ergibt sich nicht nur auf Grund der abweichenden Nullposition, sondern ist auch auf helleres Umgebungslicht beim Experimentieren zurück zuführen.

3.3 Kalibration der Wellenplättchen

Für die Kalibration der Wellenplättchen muss der Polarisator schon kalibriert sein. Daher wird der Referenzpolarisator nicht mehr benötigt. Aufbau 3.18 wird verwendet:

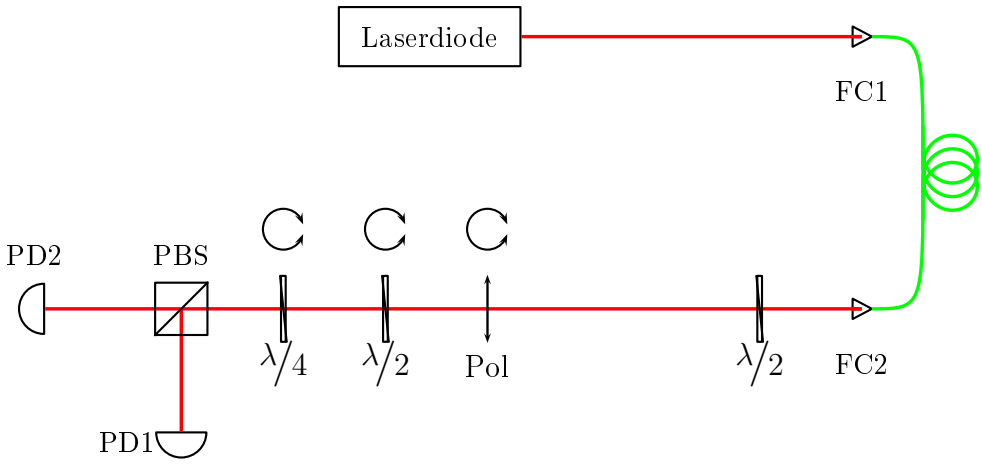

Abbildung 3.18: Aufbau für die automatische Kalibration der Wellenplättchen: Der Strahl einer Laserdiode wird in eine Glasfaser eingekoppelt, diese führt zu den weiteren optischen Komponenten. Nach der Auskopplung durchläuft der Laserstrahl ein Wellenplättchen, den kalibrierten Polarisator in einem Motor und zwei weitere Wellenplättchen in Motoren. Anschließend teilt sich der Strahl an einem Strahlteiler auf und beide Teilstrahlen werden durch Photodioden detektiert.

3.3.1 Ideen

Schließen die Wellenplättchen beliebige Winkel $\theta_{QWP} = \theta_{motor1} + \theta_{off1}$ und $\theta_{HWP} = \theta_{motor0} + \theta_{off0}$ mit der horizontalen Polarisationsachse ein, so ergibt sich durch Umformen für die Wahrscheinlichkeit der Projektionsmessung aus (2.15) bei Einstrahlen horizontal polarisierten Lichts $|\psi\rangle = |H\rangle$ am reflektierten Ausgang des PBS:

$$\begin{aligned} P &= |\langle V| QWP(\theta_{QWP}) \cdot HWP(\theta_{HWP}) |H\rangle|^2 \\ &= \frac{1}{4}\left(2 - \cos(4\theta_{HWP}) - \cos(4\theta_{QWP} - 4\theta_{HWP})\right) \\ &= f(\theta_{HWP}, \theta_{QWP}) \end{aligned} \tag{3.3}$$

Die Nullpositionen der Motoren entspricht der Position für die die Funktion f minimal wird. Der Ausdruck (3.3) ist in Abbildung 3.19 dargestellt.

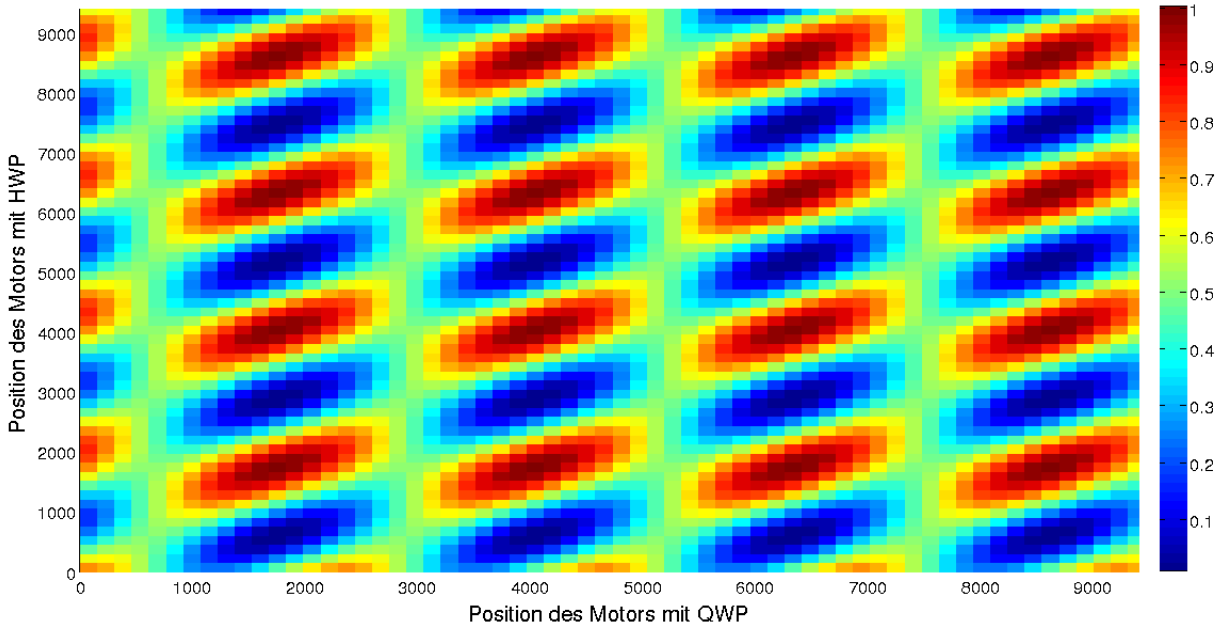

Abbildung 3.19: Gleichung (3.3). Die Funktion f ist farbig dargestellt in Abhängigkeit von den Motorpositionen mit unbekanntem Offset der Referenzpositionen zur horizontalen Polarisationsachse. Aus der Legende kann abgelesen werden, welche Farbe welchem Wert der Funktion f entspricht.

Wird in der Funktion f der Parameter θ_{QWP} konstant gehalten, so ergibt sich:

$$\begin{aligned} f_{\theta_{QWP}}(\theta_{HWP}) &= \frac{1}{2} - \frac{1}{2} \cdot \cos(2\theta_{QWP}) \cdot \sin\left(4\theta_{HWP} - 2\theta_{QWP} + \frac{\pi}{2}\right) \\ &= \frac{1}{2} - \frac{1}{2} \cdot \text{const}_1 \cdot \sin(4 \cdot \theta_{HWP} + \text{const}_2) \end{aligned} \tag{3.4}$$

Hält man den Parameter θ_{HWP} fest, so erhält man:

$$\begin{aligned} f_{\theta_{HWP}}(\theta_{QWP}) &= \frac{1}{2} \cdot \left(1 - \frac{1}{2} \cdot \cos(4\theta_{HWP})\right) - \frac{1}{4} \cdot \sin\left(4\theta_{QWP} - 4\theta_{HWP} + \frac{\pi}{2}\right) \\ &= \frac{1}{2} \cdot \text{const}_3 - \frac{1}{4} \cdot \sin(4 \cdot \theta_{QWP} + \text{const}_4) \end{aligned}$$

Dies entspricht zwei verschiedenen Sinus-Kurven mit einer Periode von $90° = 2400$ Schritte.

Erste Idee: Die erste Idee zur Kalibration der Wellenplättchen ist, sich durch mehrere Messungen einem Minimum, in der Abbildung 3.19 blau, anzunähern. Dazu wird ein Plättchen gedreht und in jedem Schritt die Spannung gemessen. Die gemessenen Daten werden an eine Sinus-Kurve gefittet. Das Minimum wird ermittelt und der Motor fährt zu dieser Position. Anschließend wird die gleiche Prozedur für das andere Plättchen durchgeführt. Dies muss abwechselnd für beide Plättchen mehrfach wiederholt werden. So sollte sich nach mehreren Messungen die ermittelten Positionen der Motoren den Nullstellungen für die Wellenplättchen annähern.

Zweite Idee: Für die zweite Idee ermitteln wir erst das Minimum und Maximum von (3.4) in Abhängigkeit von θ_{QWP}:

$$\partial_{\theta_{HWP}} f_{\theta_{QWP}} \stackrel{!}{=} 0 \quad \Longrightarrow \quad \theta_{HWP}^{(Extremum)} = \frac{1}{2}\theta_{QWP} \pm \frac{1}{4}\pi n \quad \text{mit } n \in \mathbb{N} \tag{3.5}$$

Durch einsetzen in $f_{\theta_{QWP}}$ ergibt sich mit $\theta_{HWP}^{(min)} = \frac{1}{2}\theta_{QWP}$ und $\theta_{HWP}^{(max)} = \frac{1}{2}\theta_{QWP} + \frac{1}{4}\pi$:

$$f_{\theta_{QWP}}^{(max)} = \frac{1}{2} + \frac{1}{2}\cos\left(2\theta_{QWP}\right)$$
$$f_{\theta_{QWP}}^{(min)} = \frac{1}{2} - \frac{1}{2}\cos\left(2\theta_{QWP}\right)$$

Für die Visibility folgt somit:

$$V = \frac{f_{\theta_{QWP}}^{(max)} - f_{\theta_{QWP}}^{(min)}}{f_{\theta_{QWP}}^{(max)} + f_{\theta_{QWP}}^{(min)}} = \cos\left(2\theta_{QWP}\right) \tag{3.6}$$

Für eine Drehung des $\lambda/2$-Wellenplättchens und entsprechenden Spannungsmessungen kann durch einen Sinus-Fit an die Messdaten die Visibility bestimmt werden. Daraus ergibt sich der Winkel $\pm\theta_{QWP}(V) = \frac{1}{2}\arccos(V)$. Dies ist die aktuelle Position des $\lambda/4$-Wellenplättchens. Da jedoch nicht bekannt ist, in welcher Drehrichtung von der aktuellen Position aus der Winkel $\theta_{QWP} = 0$ liegt, müssen beide Möglichkeiten in Betracht gezogen werden. Der Motor mit dem $\lambda/4$-Wellenplättchen fährt zu beiden möglichen Positionen $\theta_{Motor\,aktuell} \pm \theta_{QWP}(V)$ und anschließend wird durch Drehen des $\lambda/2$-Wellenplättchens wieder die Position für eine minimale Spannung bestimmt. Es ergeben sich somit zwei verschiedene Paare für die Nullpositionen. Für das richtige Paar muss sich eine Visibility von fast 1 ergeben.

Dritte Idee: Eine dritte Idee ergab sich aus Gleichung (3.5). Durch Drehen des $\lambda/2$-Wellenplättchens und entsprechenden Spannungsmessungen ergibt sich aus dem Fit der Messdaten ein Minimum $\theta_{Motor} = \theta_{HWP}^{(min)} - \theta_{off}$ für eine bestimmte Motorposition. Da der Offset nach einer Referenzfahrt nicht bekannt ist, ist die Position des Wellenplättchens $\theta_{HWP}^{(min)}$ nicht bekannt, sondern nur θ_{Motor}. Somit kann auch nicht über (3.5) auf die aktuelle Position des $\lambda/4$-Wellenplättchens θ_{QWP} geschlossen werden. Daher wird die dritte Idee nicht implementiert.

3.3.2 Programmierung

Im Quellcode *LambdaCalib.cpp* befinden sich die Funktionen zur Kalibration der Wellenplättchen.

Algorithmus für die erste Idee: Für die erste Idee zur Umsetzung der Kalibration werden die Funktionen *IterationLambda*() (Zeile 42-136) und *LambdaCalib*2() (Zeile 140-228) benötigt. Die Funktion *IterationLambda*() führt eine Viertel-Drehung des $\lambda/4$-Wellenplättchens mit schrittweiser Spannungsmessung durch (Zeile 45-58). Anschließend wird das Minimum aus dem Fit ermittelt (Zeile 68) und der Motor des $\lambda/4$-Wellenplättchens fährt an diese Stelle (Zeile 73). Danach erfolgt dies analog für das $\lambda/2$-Wellenplättchen (Zeile 80-107). Für diese ermittelten Nullpositionen wird die Visibility gemessen. Dazu wird einmal die Spannung an den Nullpositionen (Zeile 110-114) und einmal für ein um 45° gedrehtes $\lambda/2$-Wellenplättchen gemessen (Zeile 117-122). Aus diesen Spannungswerten wird die Visibility berechnet (Zeile 124-127). Nach Zurückfahren auf die Minimumpositionen liefert das Programm als Rückgabewert die Visibility (Zeile130-135).

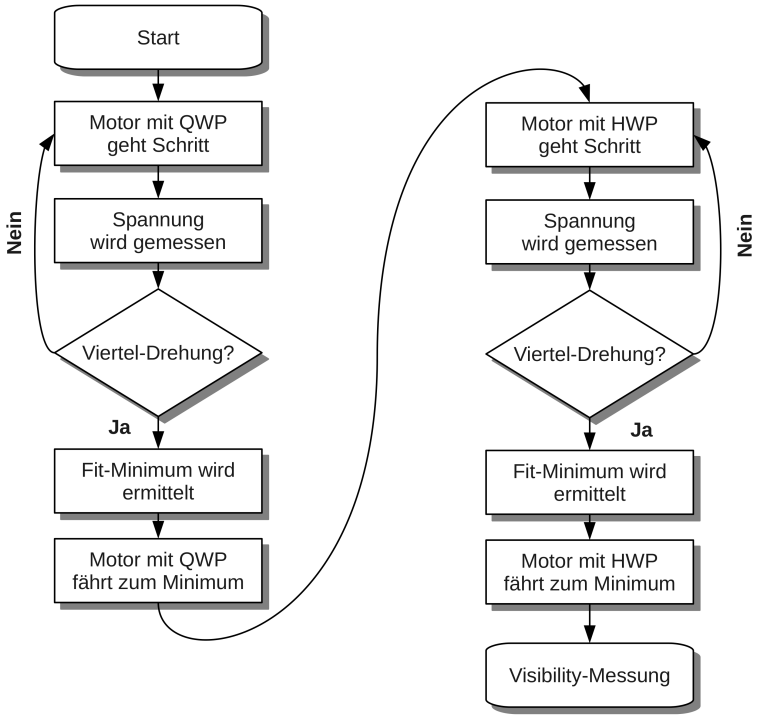

Abbildung 3.20: Flussdiagramm der Funktion *IterationLambda*(). Diese befindet sich in der Datei *LambdaCalib.cpp*

Die Funktion *LambdaCalib*2() führt die Kalibration der Wellenplättchen nach der ersten Idee aus und benötigt die Funktion *IterationLambda*(). Nach einer Referenzfahrt aller Motoren (Zeile 144-151), fährt der Motor mit dem Polarisator an die schon bekannte Nullposition (Zeile 153-167). Für die Iteration werden beliebige Startwerte für die Motoren der Wellenplättchen genommen, die der Funktion übergeben wurden (Zeile 169-173). Anschließend wir mit einer Schrittweite von 100 *Schritten* die Funktion *IterationLambda*() ausgeführt, bis die Visibility größer als 0,99 ist oder sechs Iterationen ausgeführt wurden (Zeile 183-191). Danach erfolgt die gleiche Prozedur für eine Schrittweite von 40 *Schritten* bis die Visibility größer als 0,9995 ist oder maximal 12 Iterationen ausgeführt wurden (Zeile

194-202). Die so ermittelten Nullpositionen werden in eine Datei gespeichert (Zeile 208-225) und die Visibility wird als Rückgabewert übergeben (Zeile 227).

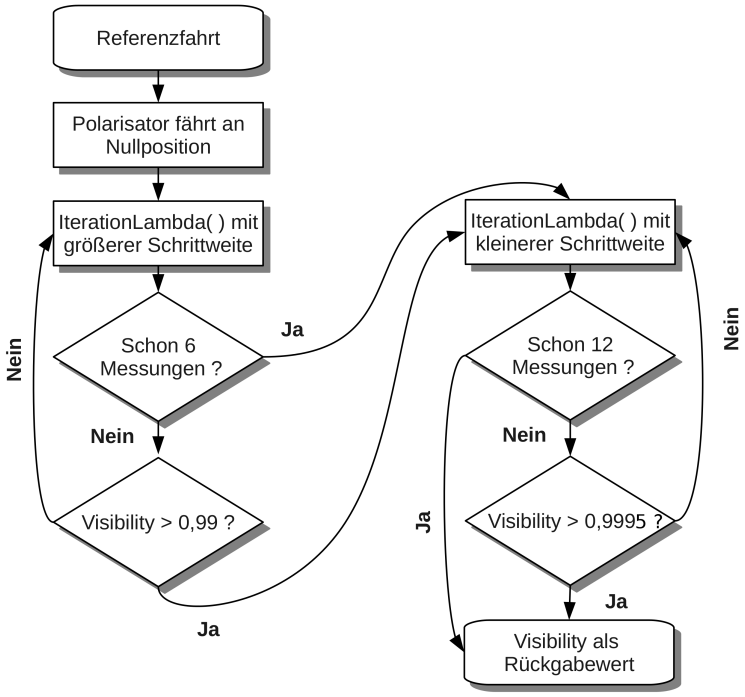

Abbildung 3.21: Flussdiagramm der Funktion *LambdaCalib*2() für iterative Ermittlung der Nullpositionen. Diese befindet sich in der Datei *LambdaCalib.cpp*

Algorithmus für die zweite Idee: Die Funktion *LambdaCalib*3() (Zeile 232-413) setzt die zweite Idee zur Kalibration der Wellenplättchen um. Nach den Referenzfahrten (Zeile 234-238) wird der Polarisator auf seine Nullposition eingestellt (Zeile 240-254). Das $\lambda/4$-Wellenplättchen fährt an eine beliebige Startposition $\theta_{Motor\,aktuell}$ (Zeile 270), welche der Funktion *LambdaCalib*3() übergeben wurde. Das $\lambda/2$-Wellenplättchen macht eine Viertel-Drehung und in jedem Schritt wird die Spannung gemessen (Zeile 272-283). Ursprünglich wurde nach einem Fit der Messpunkte die Visibility über die gefundenen Parameter der Sinus-Kurve berechnet. Bessere Ergebnisse wurden jedoch erzielt, wenn die Visibility gemessen wird. Das bedeutet die Fit-Routine berechnet das Minimum des $\lambda/2$-Wellenplättchens für die aktuelle Position des $\lambda/4$-Wellenplättchens. Und somit kann aus zwei Spannungsmessungen am Minimum und Maximum des $\lambda/2$-Wellenplättchens die Visibility nach (3.1) ermittelt werden. Aus dieser gemessenen Visibility V wird nun über (3.6) die tatsächliche Position $\theta_{QWP}(V)$ des $\lambda/4$-Wellenplättchens berechnet. Dies ist der Rückgabewert der Fit-Routine (Zeile 289). Nun wird zuerst die mögliche Nullposition $\theta_{Motor\,aktuell} - \theta_{QWP}(V)$ betrachtet (Zeile 290). In einer Schleife (Zeile 303-358) fährt das $\lambda/4$-Wellenplättchen an die mögliche Nullposition (Zeile 307) und das $\lambda/2$-Wellenplättchen rotiert immer um 40 Schritte weiter, während die Spannung gemessen wird (Zeile 311-321). Das Minimum $\theta_{HWP}^{(min)}$ aus dem Fit der Datenpunkte für das $\lambda/2$-Wellenplättchen wird ermittelt (Zeile 327) und die

Visibility für diese Nullpositionen $\theta_{Motor\,aktuell} - \theta_{QWP}(V)$ und $\theta_{HWP}^{(min)}$ wird gemessen (Zeile 330-350). Anschließend fährt das $\lambda/4$-Wellenplättchen an die andere mögliche Nullposition $\theta_{Motor\,aktuell} + \theta_{QWP}(V)$ (Zeile 353). Die zweite Iteration der Schleife läuft analog ab. Es ergibt sich jedoch ein anderes Minimum für das $\lambda/2$-Wellenplättchen. Für beide ermittelten Paare der Nullpositionen wurde die Visibility in der Schleife gemessen. Das Paar, für welches sich eine höhere Visibility ergab, entspricht den Nullpositionen (Zeile 360-369). Diese werden in Dateien gespeichert (Zeile 392-409).

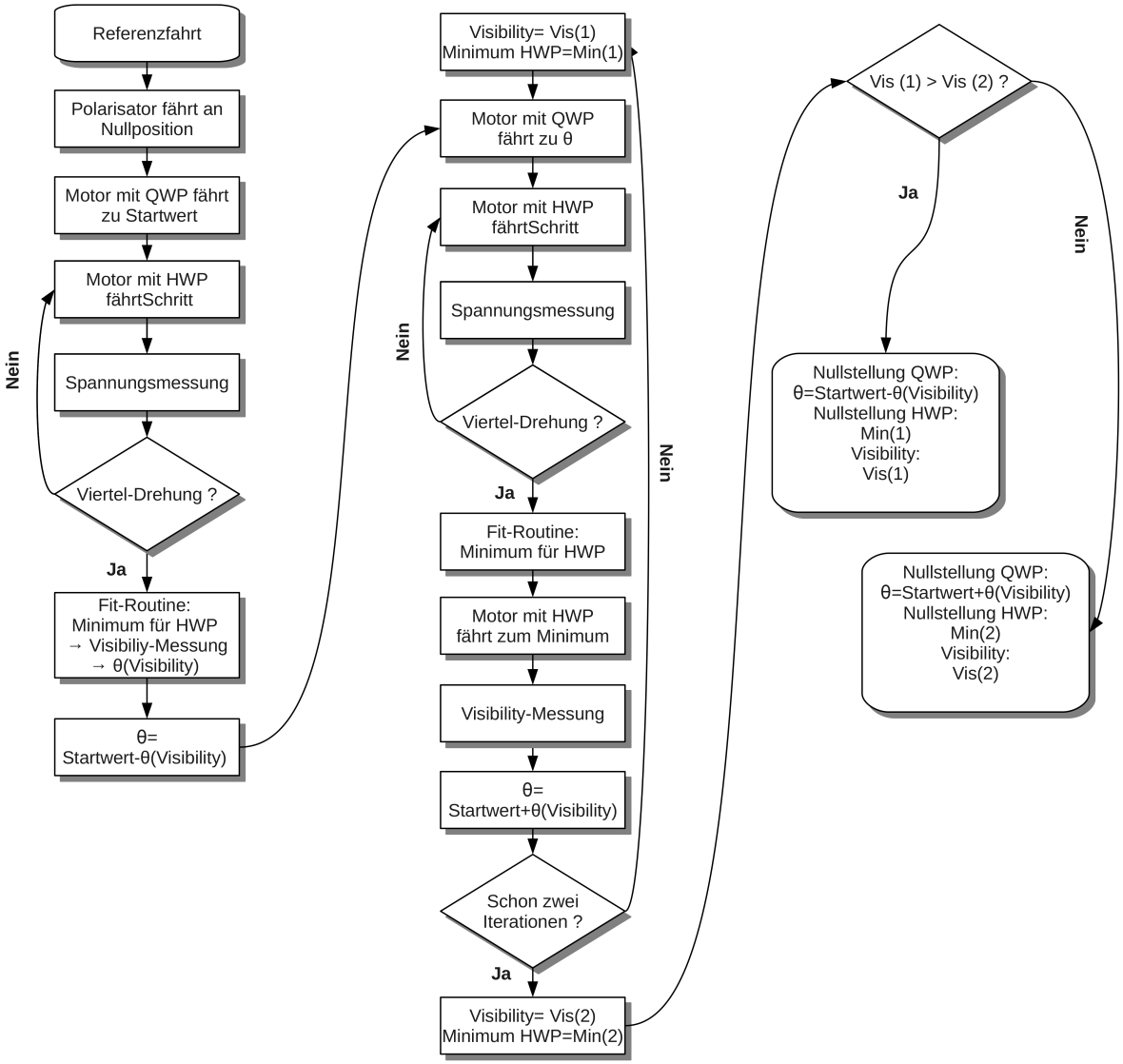

Abbildung 3.22: Flussdiagramm der Funktion *LambdaCalib*3() für Ermittlung der Nullposition des $\lambda/4$-Wellenplättchens über eine gemessene Visibility. Diese Funktion befindet sich in der Datei *LambdaCalib.cpp*

3.3.3 Durchführung und Resultate

Die Visibility ist wieder ein Kriterium für die Güte der erhaltenen Nullpositionen. Für beliebige Positionen θ_{HWP} und θ_{QWP} ergibt sich aus Gleichung (3.3) die Visibility zu:

$$V = \frac{f\left(\theta_{QWP}, \theta_{HWP} + 45°\right) - f\left(\theta_{QWP}, \theta_{HWP}\right)}{f\left(\theta_{QWP}, \theta_{HWP} + 45°\right) + f\left(\theta_{QWP}, \theta_{HWP}\right)}$$

Abbildung 3.23 ist ein Contour-Plot der Visibility in Abhängigkeit der Abweichungen von den exakten Nullpositionen.

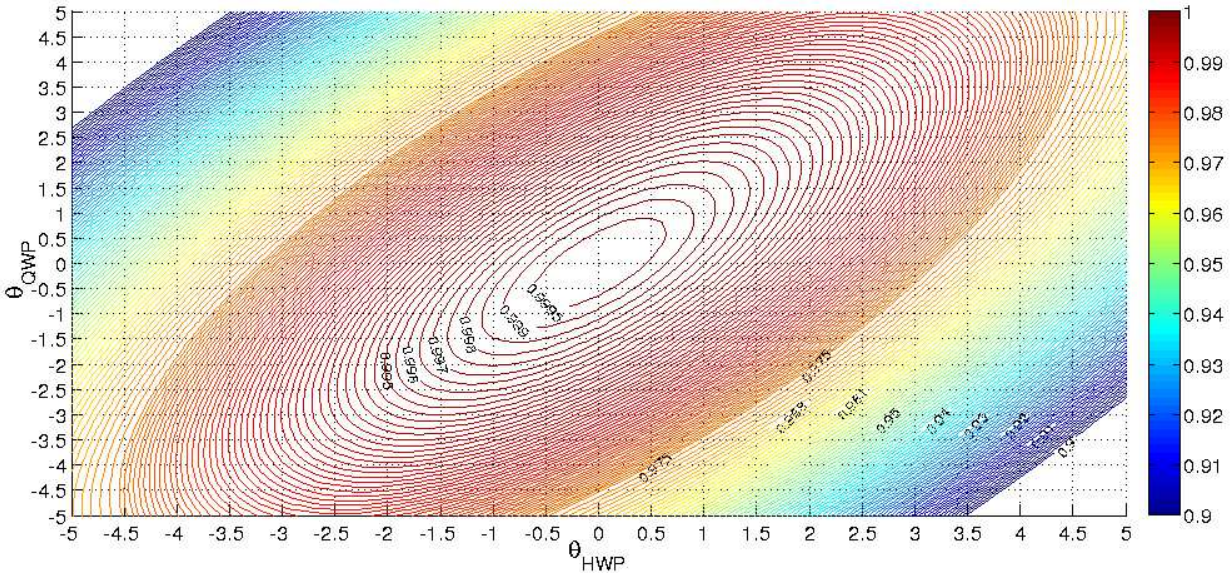

Abbildung 3.23: Visibility in Abhängigkeit von den abweichenden Nullpositionen θ_{QWP} und θ_{HWP}. Eine hohe Visibility ist rot gekennzeichnet und blau entspricht einer niedrigen Visibility. Da die Kontourlinien für große Abweichungen immer näher zusammenrücken, werden bei ca. 0,975 wieder gröbere Abstände benutzt.

Aus der Grafik lässt sich ablesen: Weichen die ermittelten Nullpositionen bereits um jeweils 0,5° ab, so ergibt sich trotzdem eine noch sehr hohe Visibility von 0,9995. Diese kann natürlich je nach Umgebungslicht im Experiment variieren.

Resultate für die erste Idee: Für die Kalibration nach der ersten Idee konnten keine guten Resultate erzielt werden. Die Auswertung von einigen Messergebnisse zeigt, dass für die ermittelten Nullpositionen eine zu geringe Visibility gemessen wurde. Die Visibility betrug etwa 0,993. Aus der Grafik 3.23 ergibt sich bei einer Visibility von 0,995 bereits ein maximaler Winkelfehler von jeweils 2°.

Tabelle 3.2 zeigt die Ergebnisse für eine Kalibrationsmessung zur Veranschaulichung.

Iterationsschritt	Minimum für HWP	Minimum für QWP	Visibility
Startwerte	7104	1564	-
0.	1926	1962	0.496593
1.	1736	1605	0.831593
2.	1641	1421	0.939493
3.	1594	1329	0.973734
4.	1571	1283	0.985478
5.	1560	1261	0.989934
6.	1555	1250	0.991603
7.	1553	1245	0.992456
8.	1552	1243	0.992841
9.	1551	1242	0.993007
10.	1551	1242	0.992996
11.	1551	1241	0.993192
12.	1551	1242	0.993006

Tabelle 3.2: Beispiel für eine Kalibrationsmessung nach der ersten Idee. Nach 12 Iterationen bricht das Programm ab. Für die gefundenen Nullpositionen ergibt sich eine zu geringe Visibility von 0,993006.

Bis zum achten Iterationsschritt wird nach jeder Iteration eine höhere Visibility gemessen, dass bedeutet die Positionen der Minima nähern sich den Nullpositionen an. Doch ab dem achten Schritt bleiben die Positionen der Minima und die Visibility fast konstant. Entweder konvergiert die Routine sehr langsam oder an diesen Positionen befindet sich tatsächlich ein lokales Minimum. Eine Erklärung für dieses Verhalten könnte sein, dass die Wellenplättchen nicht perfekt sind, so dass die Oberfläche aus Abbildung 3.19 deformiert ist und Dellen aufweist.

Der Vergleich mit den Ergebnissen aus späteren Messungen zeigt, dass die ermittelten Nullpositionen 1551 und 1242 aus Tabelle 3.2 um etwa 1,5° für das $\lambda/2$-Wellenplättchen und um 2,5° für das $\lambda/4$-Wellenplättchen von den tatsächlichen Nullpositionen abweichen.

Obwohl diese Methode in meinem Experiment keine zufrieden stellenden Ergebnisse lieferte, möchte ich sie trotzdem als Alternative zur zweiten Idee beibehalten. Vielleicht ergibt sich für andere Wellenplättchen ein besseres Resultat. Dem Benutzer steht es frei, zu entscheiden, welche Methode er anwenden möchte. Die zweite Variante, würde ich empfehlen.

Resultate für die zweite Idee: Mit dem Hilfsprogramm *qubit_test_tom* wurden 70 Messungen nach der zweiten Idee mit zufälligen Startwerten ausgeführt und daraus eine Statistik erstellt. Für die jeweils ermittelten Nullpositionen wird die Visibility gemessen und eine Qubit Tomographie durchgeführt. Durch Einstellen des kalibrierten Polarisators auf $\theta_{Pol} = 0°$ und $\theta_{Pol} = 45°$ wurden die zwei Zustände $|H\rangle$ und $|+\rangle$ realisiert. Diese werden anschließend durch die drei Einstellungen der Wellenplättchen aus Kapitel 2.2.3 in die Zustände $|+\rangle$, $|R\rangle$ und $|H\rangle$ projiziert. Es wurden die Spannungen am reflektierten sowie am transmittierten Ausgang des PBS gemessen. Daher ergibt sich ebenfalls die Projektion in die Zustände $|-\rangle$, $|L\rangle$ und $|V\rangle$. Weil die Photodioden unterschiedliche Skalierungsfaktoren aufweisen, muss ihr Verhältnis gemessen werden. Nachdem die Nullpositionen ermittelt wurden, wird für ein Rotieren des $\lambda/2$-Wellenplättchens die Spannung an beiden Photodioden gemessen. Das Amplitudenverhältnis der gefitteten Sinus-Kurven entspricht

dem Skalierungsfaktor zwischen den Photodioden. Nach den Projektionsmessungen kann dann unter Einberechnen dieses Faktors aus Gleichung (2.10) mit einem Matlab-Skript die Stokesparameter und somit die Dichtematrix rekonstruiert werden. Interessant ist nun die Fidelity zwischen gemessenen und theoretisch erwarteten Zuständen. Auch die Länge des Blochvektors (2.8) wurde mit Matlab berechnet. Diese sollte gleich Eins sein, da es sich um die reinen Zustände $|H\rangle$ und $|+\rangle$ handelt. Im Folgenden werden die so ermittelten Daten ausgewertet.

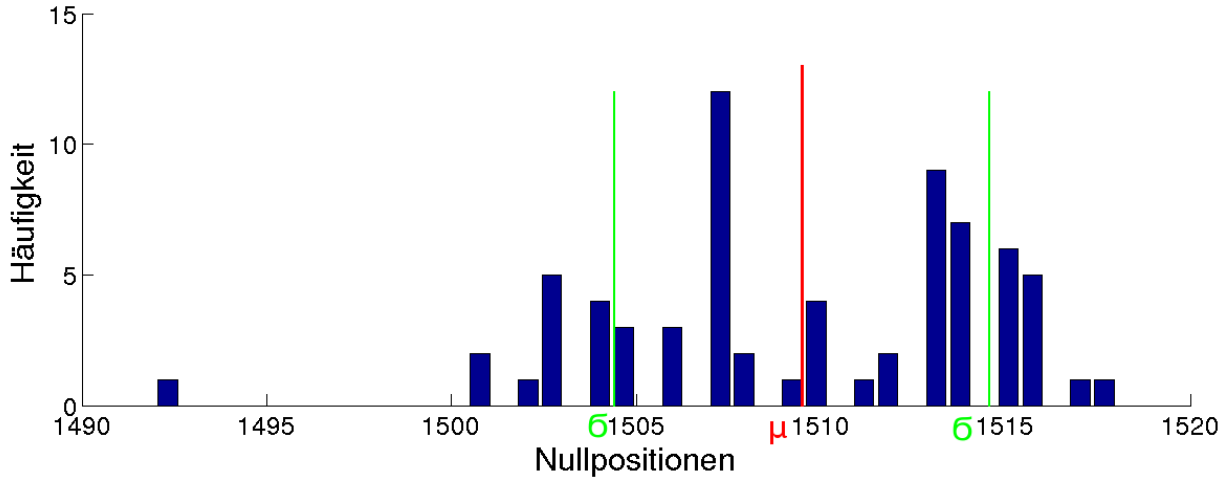

Abbildung 3.24: Histogramm für die ermittelten Nullpositionen des $\lambda/2$-Wellenplättchens. Mittelwert $\mu = 1510$ Schritte; Standardabweichung $\sigma = 5,1$ Schritte $\cong 0,2°$

Abbildung 3.24 zeigt das Histogramm der ermittelten Nullpositionen des $\lambda/2$-Wellenplättchens. Der Mittelwert liegt bei 1510 Schritten und es ergibt sich für die 70 Messungen eine Standardabweichung von $5,1$ Schritten $\cong 0,2°$.

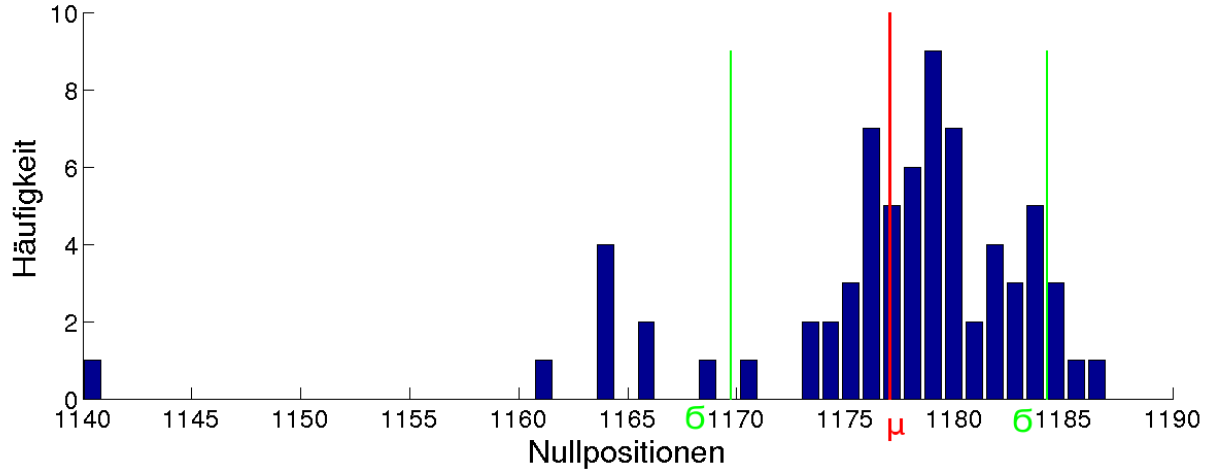

Abbildung 3.25: Histogramm für die ermittelten Nullpositionen des $\lambda/4$-Wellenplättchens. Mittelwert $\mu = 1177$ Schritte; Standardabweichung $\sigma = 7,2$ Schritte $\cong 0,3°$

Für das $\lambda/4$-Wellenplättchen ergibt sich ein Mittelwert von 1177 Schritten und eine Standardabweichung von $7,2$ Schritten $\cong 0,3°$. In Abbildung 3.25 ist das entsprechende Histogramm gezeigt.

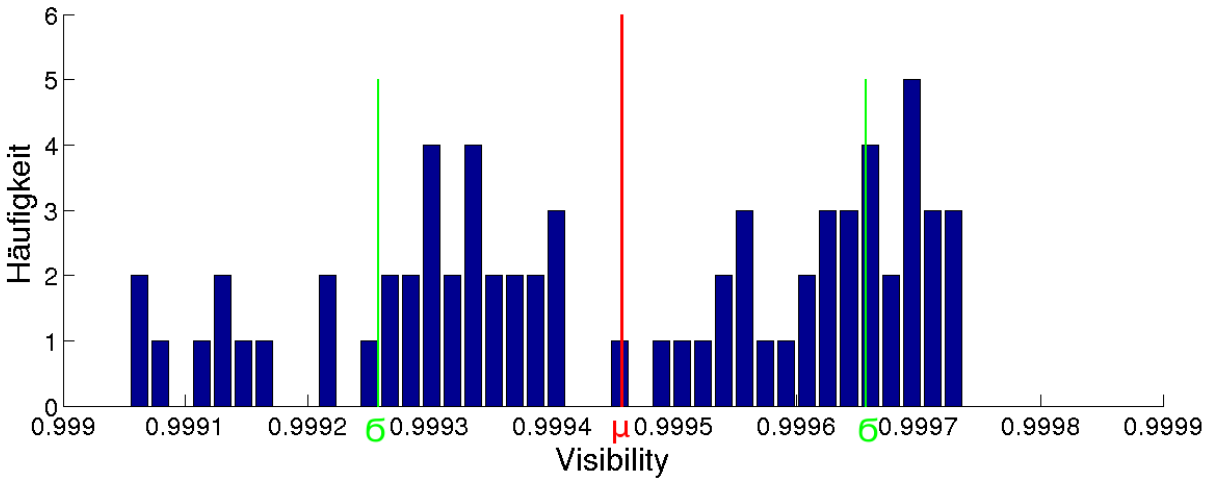

Abbildung 3.26: Histogramm für die gemessene Visibility für jedes Paar ermittelter Nullpositionen. Mittelwert $\mu = 0,99946$; Standardabweichung $\sigma = 0,0002$

Die gemessene Visibility für jedes ermittelte Paar der Nullpositionen liegt im Durchschnitt bei $0,99946$ mit einer Standardabweichung von $0,0002$. Bei perfekten Bedingungen würden also die Ergebnisse um etwa maximal $0,5°$ vom tatsächlichen Wert abweichen. Da die Visibility jedoch von den Experimentierbedingungen abhängt, ist das Resultat einer Polarisationsanalyse noch aussagekräftiger. In Kapitel 2.2.5 wurde beschrieben, dass ein Fehler in den Messbasen, und somit eine nicht perfekte Kalibration und Einstellung der Wellenplättchen, sich auf das Ergebnis der Analyse auswirkt. Die Fidelity $\mathcal{F}$ dient daher als Kriterium für die Güte einer Kalibration. In den Abbildungen 3.27 und 3.28 sind die Histogramme für die Fidelitys und in den Abbildungen 3.29 und 3.30 die Histogramme für die Länge der Blochvektoren geplottet.

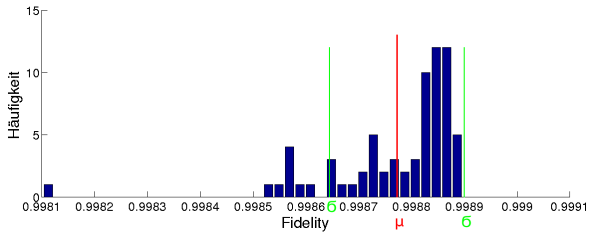

Abbildung 3.27:
Histogramm für die Fidelity des gemessenen Zustands mit dem Zustand $|H\rangle$. Mittelwert $\mu = 0,9988$; Standardabweichung $\sigma = 0,0001$

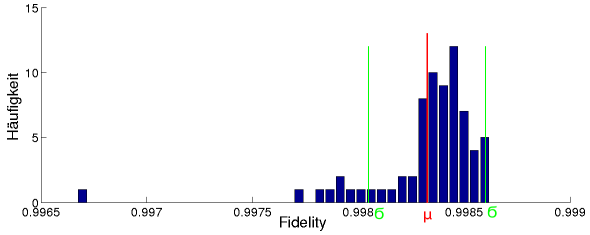

Abbildung 3.28:
Histogramm für die Fidelity des gemessenen Zustands mit dem Zustand $|+\rangle$. Mittelwert $\mu = 0,9983$; Standardabweichung $\sigma = 0,0003$

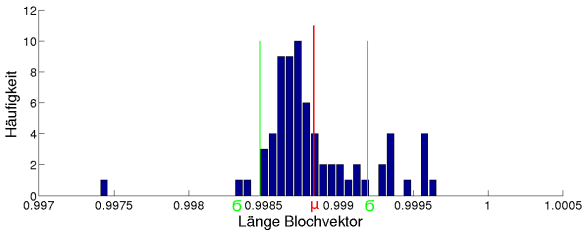

Abbildung 3.29:
Länge des Blochvektors für den rekonstruierten Zustand bei Einstellen des Polarisators auf $|H\rangle$. Mittelwert $\mu = 0,9988$; Standardabweichung $\sigma = 0,0004$

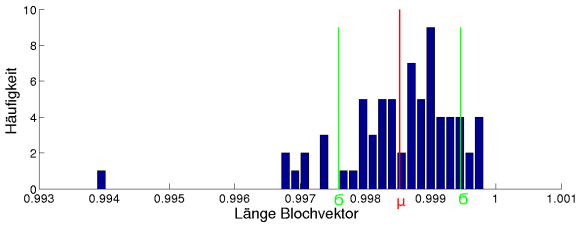

Abbildung 3.30:
Länge des Blochvektors für den rekonstruierten Zustand bei Einstellen des Polarisators auf $|+\rangle$. Mittelwert $\mu = 0,9985$; Standardabweichung $\sigma = 0,0010$

Im Mittel liegt die Fidelity für die Rekonstruktion aus den Messdaten bei $\mathcal{F}\left(|H\rangle, \rho_{|H\rangle}\right) = 0,9988$ und $\mathcal{F}\left(|+\rangle, \rho_{|+\rangle}\right) = 0,9983$. Für die Länge der Blochvektoren ergibt sich im Mittel kein unphysikalisches Ergebnis, sondern $r\left(\rho_{|H\rangle}\right) = 0,9988$ und $r\left(\rho_{|+\rangle}\right) = 0,9985$.

3.4 Fit-Routine

Um die gemessenen Datenpunkte an eine Sinus-Kurve zu fitten, wird die GNU Scientific Library [5] benutzt. Diese Bibliothek stellt die nötigen Funktionen für eine Fit-Routine zur Verfügung und muss in den Quellcode von *fitting.cpp* eingebunden werden.

3.4.1 Numerik

Der Fit an die Sinus-Kurve entspricht einem nichtlinearen Ausgleichsproblem. Es muss eine Sinusfunktion $Y\left(\vec{x}, t\right) = a \cdot \sin\left(b \cdot t + d\right) + c$ mit den zu bestimmenden Parametern $\vec{x} = (a, b, c, d)$ gefunden werden von der die gemessenen Datenpunkte am wenigsten abweichen. Dazu werden erst die Residuenfunktionen f_i

$$f_i\left(\vec{x}\right) = \frac{Y_i\left(\vec{x}, t_i\right) - y_i}{\sigma_i}$$

für jedes Datenpaar $\{t_i, y_i\}$ und die Gausschen Fehler $\{\sigma_i\}$ berechnet. In dem Programm *fitting.cpp* werden alle Residuen f_i durch $\sigma_i = 1$ gleich gewichtet, da angenommen wird alle Datenpunkte haben den gleichen Fehler. Für die N Datenpunkte muss die Länge des Residuenvektors $F\left(\vec{x}\right)$ minimiert werden:

$$\Phi\left(\vec{x}\right) = \frac{1}{2}\left\|F\left(\vec{x}\right)\right\|^2 = \frac{1}{2} \cdot \sum_{i=1}^{N} f_i\left(\vec{x}\right)^2$$

Die Funktion $\Phi\left(\vec{x}\right)$ hängt hier von den vier Parametern a, b, c und d ab. Aus der Minimierungsbedingung

$$\nabla\Phi\left(\vec{x}\right) \stackrel{!}{=} \vec{0}$$

ergeben sich vier gekoppelte Differentialgleichungen.

Iterativ muss nun die Lösung $\vec{x}$ mit den Startparametern $\overrightarrow{x^{(Start)}}$ gefunden werden. Dies funktioniert mit dem Levenberg-Marquardt Verfahren das iterativ aus den Werten $\overrightarrow{x^{(k)}}$ im k-ten Iterationsschritt die neuen Werte $\overrightarrow{x^{(k+1)}}$ berechnet. [12, 5]

3.4.2 Programmierung

Im Folgenden soll nur grob der Ablauf des Programms *fitting.cpp* erklärt werden. Die Funktionen der GSL Bibliothek werden im Handbuch [5] beschrieben.

Zuerst werden einige Funktionen benötigt (Zeile 51-186) und eine Datenstruktur wird initialisiert (Zeile 41-48). Die Funktion $sin\_f()$ berechnet die Werte $f_i\left(\overrightarrow{x^{(Start)}}\right)$ (Zeile 51-74) und die Funktion $sin\_df()$ berechnet die Elemente der Jacobimatrix $J_{ij} = \partial f_i/\partial x_j$ (Zeile 77-104). Die Funktion $sin\_fdf()$ ruft diese beiden Funktionen auf (Zeile 107-113). Anschließend folgen Funktionen, die den minimalen und maximalen Spannungswert, sowie die zugehörigen Motorstellungen aus den Messdaten ermitteln (Zeile 116-181). Aus diesen Werten werden später die Startparameter ermittelt. Die Funktion $print\_state()$ gibt den Status einer Iteration an die Standardausgabe aus (Zeile 184-185 und 499-505). Nun folgt die Hauptfunktion $fit()$ (Zeile 189-519), die für ein Ausführen der Fit-Routine aufgerufen werden muss. Dieser Funktion wird übergeben, ob sie für die Kalibration des Polarisators ($option =' H'$), zum Finden des Minimums eines Wellenplättchens ($option =' L'$), für Ermitteln der Amplitude, woraus später der Skalierungsfaktor für die Photodioden berechnet wird ($option =' S'$), oder zum Berechnen der Nullstellung des $\lambda/4$-Wellenplättchens nach der zweiten Idee ($option =' V'$), benutzt wird. Zusätzlich wird der Funktion übergeben, welche Spalten a und b der Matrix $mat3$ den Datenpunkten entsprechen. Somit können für die verschiedenen Optionen die Startparameter aus den Datenpunkten berechnet werden (Zeile 233-280). Anschließend findet die Iteration statt, bis die Fehler eine obere Schranke unterbieten (Zeile 311-322). Die ermittelten Parameter a, b, c und d für die gefittete Funktion $Y(t) = a \cdot \sin(b \cdot t + d) + c$ werden in eine Datei geschrieben (Zeile 334-338). Aus den Parametern können das Minimum $Y^{(min)} = a \cdot \sin(-\pi/2) + c$ und Maximum $Y^{(max)} = a \cdot \sin(\pi/2) + c$, sowie die entsprechenden Positionen der Motoren $t^{(min)} = \frac{-\pi/2-d}{b}$ und $t^{(max)} = \frac{\pi/2-d}{b}$ berechnet werden (Zeile 346-350). Daraus kann nun entweder die Nullstellung des Polarisators $t^{(max)}$, die Amplitude der Kurve $|a|$, das Minimum des Wellenplättchens $t^{(min)}$ oder die Nullstellung des $\lambda/4$-Wellenplättchens $\frac{1}{2}\arccos(V)$ aus der Visibility $V = \frac{U\left(HWP\,bei\,t^{(min)}\right)-U\left(HWP\,bei\,t^{(max)}\right)}{U\left(HWP\,bei\,t^{(min)}\right)+U\left(HWP\,bei\,t^{(max)}\right)}$ gemessen an $PD1$ berechnet werden (Zeile 352-489).

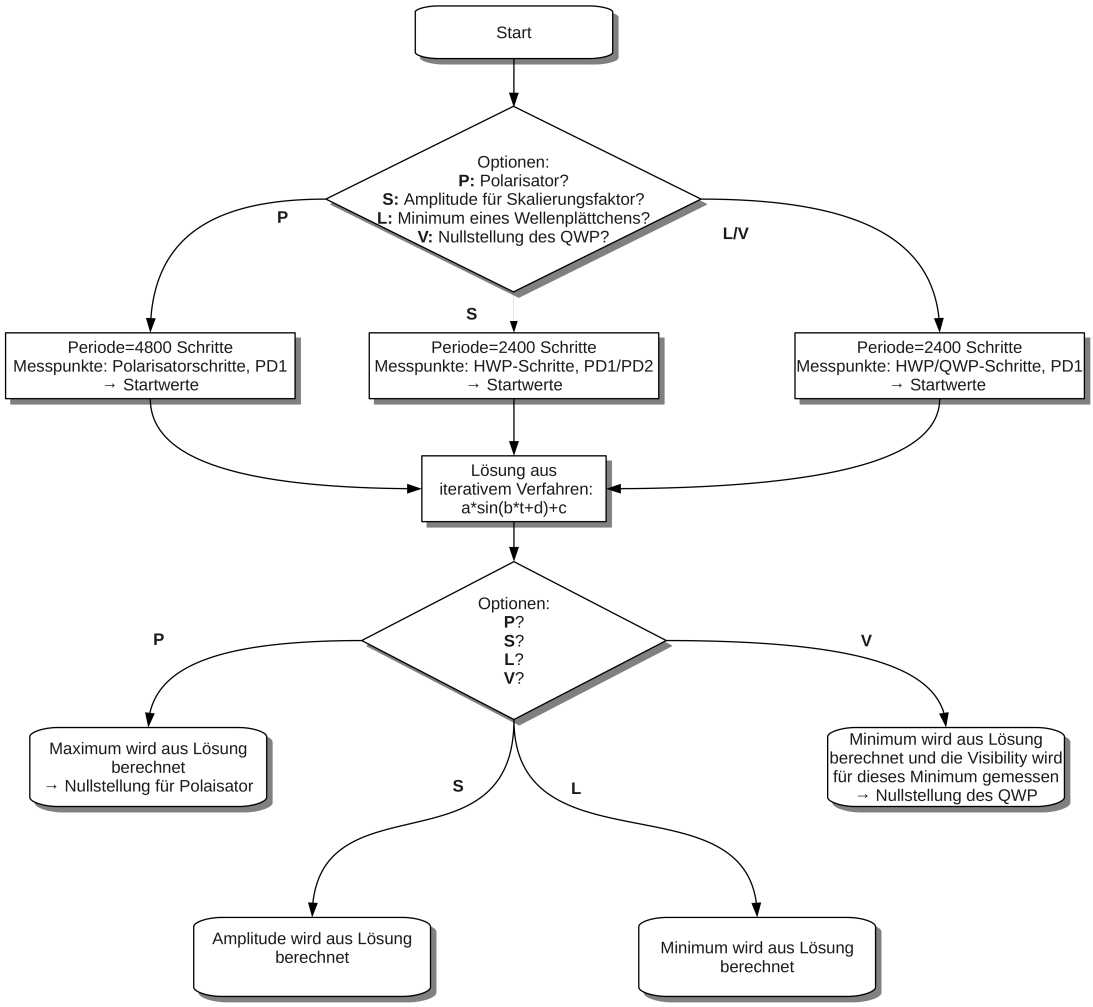

Abbildung 3.31: Flussdiagramm der Funktion *fitting*() aus *fitting.cpp*. Es ist der Fluss für die unterschiedlichen Optionen dargestellt. Option H: für Kalibration des Polarisators; Option L: für Berechnung des Minimums eines Wellenplättchens; Option S: für Ermitteln des Skalierungsfaktors muss die Amplitude bestimmt werden; Option V: für Berechnen der Nullstellung des $\lambda/4$-Wellenplättchens aus der gemessenen Visibility.

3.5 Programm für den Endnutzer

Das Hauptprogramm ist *calibration.cpp*. In diesem werden die Funktionen zur Kalibration des Polarisators oder der Wellenplättchen aufgerufen. Nach Kompilieren des Quellcodes kann das Programm in einem Terminal gestartet werden. Dazu können Optionen übergeben werden. Tabelle 3.3 zeigt alle möglichen Optionen und erklärt diese.

Optionen	Erklärung und entsprechende Zeilenangabe im Programm
-help	Aufrufen der Hilfe. Es wurde ein Manual für dieses Programm geschrieben, in dem alle Optionen und die Durchführung erläutert werden. (Zeile 57-63)
-file filename	In der Textdatei 'filename' werden alle Ergebnisse gespeichert. Wird diese Option nicht angegeben, so werden alle Ergebnisse in 'results.txt' gespeichert. (Zeile 99-112)
-offset times	Es kann der Offset des Multimeters bestimmt werden. Dazu müssen die Photodioden ausgesteckt werden. Nun wird die Anzahl 'times' eine Spannungsmessung durchgeführt und der Mittelwert gebildet (Zeile 148-185). Anschließend müssen die Photodioden wieder angeschlossen werden. Es ergeben sich sehr kleine Spannungswerte, welche nun als Offset von allen Spannungsmessungen abgezogen werden. Wird die Option nicht angegeben, so wird kein Offset abgezogen. Da die Einberechnung des Offsets in meiner Durchführung oft zu negativen Spannungswerten führte, habe ich diese bei den Kalibrationen nicht mit einberechnet.
-b offsetPD1	Der Offset von Photodiode 1 'offsetPD1' kann auch direkt im Programmaufruf übergeben werden, wenn dieser zum Beispiel früher schon gemessen wurde. Wurden beide Optionen '-offset times' und '-b offsetPD1' angegeben, so wird der Wert 'offsetPD1' vorgezogen. (Zeile 189-195)
-d offsetPD2	Der Offset von Photodiode 2 'offsetPD2' kann auch direkt im Programmaufruf übergeben werden, wenn dieser zum Beispiel früher schon gemessen wurde. Wurden beide Optionen '-offset times' und '-d offsetPD2' angegeben, so wird der Wert 'offsetPD2' vorgezogen. (Zeile 197-202)
-noise times	Um eine Statistik über Schwankungen der Spannungswerte aufzustellen, kann die Anzahl 'times' Spannungsmessungen durchgeführt werden. Es wird der Mittelwert, sowie die Standardabweichung berechnet (Zeile 360-373). Zum Beispiel ergab sich für 100 Messungen eine Standardabweichung von 0,1%.
-polarizer steps	Die Kalibration des Polarisators wird durchgeführt. Dazu rotiert der Polarisator in jedem Messvorgang um den Winkel 'steps' in Grad weiter. Der Benutzer kann also selbst die Schrittweite bestimmen. (Zeile 215-257)
-lambda 1	Die Wellenplättchen werden kalibriert. Dazu wird die Methode nach der zweiten Idee benutzt: Die Nullstellung des $\lambda/4$-Wellenplättchens wird über die Visibility berechnet (Zeile 271-308). Diese Methode stellte sich bei meiner Durchführung als die bessere heraus.
-lambda 2	Die Wellenplättchen werden nach der ersten Idee kalibriert: Iterativ wird das globale Minimum und somit die Nullstellungen gefunden (Zeile 271-308). Diese Methode kann ausprobiert werden, garantiert aber keine richtige Kalibration.
-scale	Es wird der Skalierungsfaktor $a(PD2)/a(PD1)$ der Detektoren berechnet. Dies ergibt sich aus den Amplituden a der Spannungswerte für eine Rotation des $\lambda/2$-Wellenplättchens nach erfolgter Kalibration. (Zeile 320-337)

Tabelle 3.3: Optionen für das Programm *calibration*. Durch Aufrufen der Hilfe werden alle möglichen Optionen und die Durchführung erklärt.

4 Fazit

Mit dem Programm *calibration* kann die Kalibration eines Polarisators und die Kalibration der zwei Wellenplättchen eines Polarisationsanalysators durchgeführt werden. Zustätzlich bietet das Programm noch weitere Funktionen. Ist der Nutzer mit dem Programm nicht vertraut, so bekommt er durch Aufrufen der Hilfeoption eine Erklärung zur richtigen Verwendung und zu allen Möglichkeiten des Programms.

Mit Testmessungen konnte gezeigt werden, dass die Kalibration des Polarisators in 94% aller Fälle erfolgreich funktionierte. Erfolgreich bedeutet, es konnte eine Visibility von mindestens 0,99980 erreicht werden. Dabei lag die Standardabweichung vom Mittelwert der Nullpositionen bei nur 0,042°. Um die Häufigkeit einer erfolgreichen Kalibration weiter zu erhöhen, ist es sinnvoll eine Rückmeldung der Motoren nach beendeter Referenzfahrt einzubauen.

Besonders wichtig ist eine sehr gute Kalibration der Wellenplättchen. Weisen die Messbasen für eine Polarisationanalyse Fehler auf Grund schlecht kalibrierter Wellenplättchen auf, so wirkt sich dies negativ auf die Resultate der Messung aus.

Es wurden zwei unterschiedliche Ideen für den Kalibrationsablauf entwickelt.

Die erste Idee ist eine iterative Suche nach den Nullpositionen. Auf Grund unperfekter Wellenplättchen oder einer zu langsamen Konvergenz konnten jedoch keine zufrieden stellenden Resultate erzielt werden.

Für die zweite Idee wird die Nullstellung des $\lambda/4$-Wellenplättchens aus einer Messung der Visibility berechnet. Testmessungen ergaben für die Standardabweichung der Nullpositionen für das $\lambda/2$-Wellenplättchen 0,2° und für die Standardabweichung der Nullpositionen für das $\lambda/4$-Wellenplättchen 0,3°. Die Visibility betrug im Mittel 0,99946. Theoretisch würden daher die Nullpositionen jeweils um etwa 0,5° von den tatsächlichen Werten abweichen. Die Mittelwerte der Nullpositionen liegen wahrscheinlich deutlich näher an den tatsächlichen Werten.

Ob eine Kalibration sehr gut funktioniert hat, erweist sich bei der Benutzung des kalibrierten Analysators. Daher wurde der transmittierte Zustand des zuvor kalibrierten Polarisators rekonstruiert. Die Fidelity zwischen rekonstruiertem und theoretisch erwartetem Zustand $|H\rangle$ ergab im Mittel 0,9988.

Um wie viel besser die automatische Kalibration im Gegensatz zur manuellen Kalibration funktioniert, wird sich erst bei der Verwendung in späteren Experimenten zeigen. Die Zeitersparnis einer automatischen Kalibration ist auf jeden Fall deutlich. Wählt man für die Kalibration des Polarisators eine Schrittweite von 30°, so dauert die Kalibration nur eine Minute. Die Kalibration der beiden Wellenplättchen dauert für die zweite Idee knapp zehn Minuten.

Ein weiterer großer Vorteil ist, dass die Nullpositionen für die Motoren gespeichert werden und somit nach einer Referenzfahrt immer wieder gefunden werden können.

Daher ist die automatische Kalibration, die über das Programm *calibration* gesteuert wird, ein hilfreiches Werkzeug, welches beim Aufbau späterer Experimente stets benutzt werden kann.

Anhang

Calib-functions.cpp

```
1  /*
2   *
3   * Functions needed for Calibration
4   *
5   *
6   */
7  # include <iostream>     /*Standard Input / Output Streams Library*/
8  # include <stdio.h>      /* Standard input/ouput definitions */
9  # include <stdlib.h>     /* C standard library */
10 # include <string.h>     /* string manipulation in C */
11 # include <unistd.h>     /* UNIX standard function definitions */
12 # include <fcntl.h>      /* File control definitions */
13 # include <termios.h>    /* POSIX terminal control definitions  */
14 # include <sys/wait.h>   /* for wait function*/
15 # include <math.h>              /*to compute mathematical operations and
       transformations*/
16 # include <iomanip>      /*Stream manipulators*/
17 # include <ftdi.h>              /*to control ftdi-USB-chips*/
18 # include <ctime>               /*to get and manipulate date and time
       information*/
19 # include <sched.h>             /*contains the scheduling parameters*/
20 # include <signal.h>     /*to handle signals*/
21
22 # include "fitting.h"
23
24 # include "seriell.h"
25
26 # define number_motors 3    /* Definition of number of motors, which are used
       */
27 # define number_pd 2        /*Definition of number of photodiodes, which are
       used */
28
29
30 using namespace std;
31
32
33
34 //global commands for motors
35 char voltage[20] = "/dev/ttyUSB0";       //Multimeter at USB0
36 char pol[20] = "/dev/ttyUSB1";           //Polarizer at USB1
37 char lambda[20] = "/dev/ttyUSB2";        //Waveplates at USB2
38 char cont[20] = "cont\r\n";
39 char stop[20] = "stop\r\n";
40 char goto0[20]="goto␣0\r\n";
41 char refall[20]="refall\r\n";
42 char restart[20]="restart\r\n";
```

```
43 char motor_0[20] = "motor␣0\r\n";
44 char motor_1[20] = "motor␣1\r\n";
45 char setzero[20]="setzero\r\n";
46
47
48
49 //function: restarting motor
50 int startresetclose ( char *motor)
51 {
52   init_ser(motor);
53   send_command(restart);
54   sleep(20);
55   send_command(motor_0);
56   send_command(setzero);
57   send_command(goto0);
58   sleep(4);
59   send_command(motor_1);
60   send_command(setzero);
61   send_command(goto0);
62   sleep(4);
63   close_ser();
64
65   return 0;
66 }
67
68
69 //function: rounds a double 'angle' to integer 'steps' <=9600
70 int angletosteps (double angle)
71 {
72   while (angle > 360)
73   {angle = angle - 360;}
74
75   while (angle < -360)
76   {angle = angle + 360;}
77
78   if (angle > 0 )
79   {angle = angle *9600 /360 +0.5;        }
80   else if (angle < 0)
81   {angle =angle*9600/360 - 0.5;}
82   else
83   {angle =0;}
84
85   return angle;
86 }
87
88
89 //function: 'motor' of 'seriell' goes to position 'steps'
90 int motorgotomatelement (int steps, char *seriell, char *motor)
91 {
92   init_ser(seriell);
93   send_command(motor);
94   char buffer[30];
95   char str[30] = "goto␣";
96   usleep(2000);
97   sprintf(buffer,"%i",steps);
98   strcat(str,buffer);
99   strcat(str,"\r\n");
100  send_command(str);
```

```
101   usleep(4000);
102   close_ser();
103 }
104
105
106 //function: 'motor' of 'seriell' goes to position 'steps' and sleeps
107 int motorgotomatelementlong (int steps, char *seriell, char *motor)
108 {
109   init_ser(seriell);
110   send_command(motor);
111   usleep(2000);
112   char buffer[30];
113   char str[30] = "goto ";
114   sprintf(buffer,"%i",steps);
115   strcat(str,buffer);
116   strcat(str,"\r\n");
117   send_command(str);
118   sleep(4);
119   close_ser();
120   sleep(1);
121 }
122
123
124 //function: reads voltages via pipes and stores data to array
125 int readvolt (double array[8])
126 {
127   int volt1[2], volt2[2], l;
128   char bufferp[200];
129
130   //make pipe or perror
131   if ( (pipe(volt1)!=0) || (pipe(volt2)!=0) )
132   {
133     perror("pipe() failed!");
134     return 1;
135   }
136
137   pid_t pid;
138   int status;
139
140   if ( (pid=fork()) == 0)
141   {
142     close(volt1[1]);
143     close(volt2[0]);
144     //standard in- and output to pipes
145     if ((dup2(volt1[0],STDIN_FILENO)==-1) || (dup2(volt2[1],STDOUT_FILENO)
          ==-1) )
146     {
147       perror ("pipe: dup2() failed");
148       return 1;
149     }
150     close(volt1[0]);
151     close(volt2[1]);
152     //child-process
153     execl ("./adc_ad7609_reader","./adc_ad7609_reader","-s 1","-t","-d",NULL)
          ;
154     perror("./adc_ad7609_reader: execl() failed\n");
155     return(1);
156
```

```
157   }
158
159   close(volt1[0]);
160   close(volt2[1]);
161   close(volt1[1]);
162
163   //waits until chil-process is finished, with 'status'
164   wait(&status);
165   printf("Child process exited with return code %d.\n",WEXITSTATUS(status));
166
167   //writing output to buffer or perror
168   if ( ( l=read(volt2[0], bufferp, 199)) == -1)
169   {
170     perror("pipe: read() failed");
171   }
172   else
173   {
174     write (STDOUT_FILENO, bufferp,l);
175   }
176
177
178   close(volt2[0]);
179
180
181   usleep(100000);
182
183   // storing data of voltages into array
184   char * p1 ,*p2 , *p3 , *p4 , *p5 , *p6 , *p7 , *p8;
185   double d1, d2 , d3 , d4 , d5 ,d6 , d7 , d8;
186   strtod (bufferp,&p1);
187   d1 = strtod (p1,&p2);
188   d2 = strtod (p2,&p3);
189   d3 = strtod (p3,&p4);
190   d4= strtod (p4,&p5);
191   d5 = strtod (p5,&p6);
192   d6= strtod (p6,&p7);
193   d7 = strtod (p7,&p8);
194   d8 = strtod (p8,NULL);
195   array[0]=d1;
196   array[1]=d2;
197   array[2]=d3;
198   array[3]=d4;
199   array[4]=d5;
200   array[5]=d6;
201   array[6]=d7;
202   array[7]=d8;
203
204   return 0;
205 }
206
207
208 //function: prints 'mat' to standard output
209 int printmat2(int lines , double mat[][5])
210 {
211   for (int a=0; a <lines; a++)
212   {
213     for (int b=0; b<5; b++)
214     { cout << setw(9) << mat [a][b] <<"   ";}
```

```
215     cout <<endl;
216   }
217
218   return 0;
219 }
220
221
222
223 //function: prints 'mat' with angles for motors to standard output
224 int printmat1(int lines , double mat[][number_motors])
225 {
226         for (int a=0; a <lines; a++)
227         {
228                 for (int b=0; b<number_motors; b++)
229                 { cout << setw(7) << mat [a][b] <<"   ";}
230                 cout <<endl;
231         }
232         return 0;
233 }
234
235
236 //function:converts 'mat' to steps and stores the fastest way from one line
      to the next
237 int convertmat (int lines , double mat[][number_motors])
238 {
239   //through all lines of mat
240   for (int a=0; a <lines; a++)
241   {
242     //through all columns
243     for (int b=0; b<number_motors; b++)
244     {
245       mat[a][b]=angletosteps(mat[a][b]);
246       //searching fastest way to next angle
247       if (a!=0)
248       {
249         int c=abs(mat[a-1][b]-abs(mat[a][b]));
250         int d=abs(mat[a-1][b]-abs(mat[a][b]+9600));
251         //storing fastest way in mat
252         if ( c > d)
253         { mat[a][b]=mat[a][b]+9600 ;}
254       }
255     }
256   }
257   return 0;
258 }
259
260
261 //function: stores 'mat' to FILE 'result'
262 int mat2tofile (double mat[][number_motors+number_pd] , int lines, FILE *
      result)
263 {
264   //storing time and date to file
265   time_t t;
266   time(&t);
267   char buffert[200];
268   sprintf(buffert,"%s",ctime(&t));
269   fputs(buffert, result);
270
```

```
271   //storing each line of mat to file
272   for (int i=0 ; i<lines ; i++)
273   {
274     char buffer[400];
275     sprintf(buffer,"%.6f", mat[i][0]);
276     for (int j=1 ; j<5; j++)
277     {
278       strcat(buffer,"␣");
279       char buffer2[20];
280       sprintf(buffer2 , "%.6f" , mat[i][j]);
281       strcat(buffer,buffer2);
282     }
283     strcat(buffer , "\n");
284     fputs(buffer , result);
285   }
286   fprintf(result,"\n");
287   return 0;
288 }
289
290
291
292
293 //function:caluclates the average voltages and the standard deviation of '
      times' measurements
294 int noisemes(FILE *result, int times, double offset[])
295 {
296   double mat[2][times];
297   double average0=0;
298   double average1=0;
299   double average2=0;
300   double average3=0;
301   for (int i=0; i<times; i++)
302   {
303     double array[8];
304     readvolt(array);
305     fprintf(result,"%.6f␣%.6f\n",array[0],array[1]);
306     mat[0][i]=array[0]-offset[0];
307     mat[1][i]=array[1]-offset[1];
308     average0=average0+mat[0][i];
309     average1=average1+mat[1][i];
310     average2=average2+mat[0][i]*mat[0][i];
311     average3=average3+mat[1][i]*mat[1][i];
312     usleep(50000);
313   }
314   //averages of voltage
315   average0=average0/times;
316   average1=average1/times;
317   fprintf(result,"Averages:␣␣%.6f␣␣␣%.6f\n",average0,average1);
318   //averages of voltage^2
319   average2=average2/times;
320   average3=average3/times;
321
322   //standard deviations
323   double dev1=sqrt(average2-average0*average0);
324   double dev2=sqrt(average3-average1*average1);
325   fprintf(result,"Standard␣deviations:␣%.6f␣␣%.6f\n",dev1,dev2);
326   fprintf(result,"Standard␣deviations␣in␣percentage:␣%.6f␣␣%.6f\n",dev1/
        average0,dev2/average1);
```

```
327   offset[0]=average0;
328   offset[1]=average1;
329 
330   return 0;
331 
332 }
333 
334 
335 //function: measures scalefactor for photodiodes
336 double scalefactor(double offset[], FILE *result)
337 {
338   fprintf(result,"getting␣scalefactor\n");
339   startresetclose(pol);
340   sleep(2);
341   startresetclose(lambda);
342   sleep(4);
343 
344   //set H polarizer to zero
345   FILE *stream0;
346   char filename0[100]="hpol_is_at.txt";
347   if ((stream0=fopen(filename0,"r"))==NULL)
348   {
349     fprintf (stderr, "Can't␣open␣file␣'%s'␣for␣input:␣", filename0);
350     perror("");
351     return 1;
352   }
353   int H_Pol;
354   fscanf(stream0,"%d", &H_Pol);
355   fclose(stream0);
356   fprintf(result,"H-Pol␣is␣read␣from␣file:␣%d\n",H_Pol);
357   motorgotomatelementlong ( H_Pol , pol , motor_1);
358   sleep(2);
359 
360 
361   // set lambda_2 to zero
362   FILE *stream1;
363   char filename1[100]="lambda_2_is_at.txt";
364   if ((stream1=fopen(filename1,"r"))==NULL)
365   {
366     fprintf (stderr, "Can't␣open␣file␣'%s'␣for␣input:␣", filename1);
367     perror("");
368     return 1;
369   }
370   int lambda_2;
371   fscanf(stream1,"%d", &lambda_2);
372   fclose(stream1);
373   fprintf(result,"Zero␣position␣for␣lambda_2␣is␣read␣from␣file:␣%d\n",
        lambda_2);
374   motorgotomatelementlong ( lambda_2 , lambda , motor_0);
375   sleep(2);
376 
377   //set lambda_4 to zero
378   FILE *stream2;
379   char filename2[100]="lambda_4_is_at.txt";
380   if ((stream2=fopen(filename2,"r"))==NULL)
381   {
382     fprintf (stderr, "Can't␣open␣file␣'%s'␣for␣input:␣", filename2);
383     perror("");
```

```
384     return 1;
385   }
386   int lambda_4;
387   fscanf(stream2,"%d", &lambda_4);
388   fclose(stream2);
389   fprintf(result,"Zero position for lambda_4 is read from file: %d\n",
          lambda_4);
390   motorgotomatelementlong ( lambda_4 , lambda , motor_1);
391   sleep(4);
392
393   //rotating lambda_2 from 0 to 4800 and measuring voltages
394   double mat[41][5];
395   for (int i=0; i<41; i++)
396   {
397     motorgotomatelement(i*60,lambda,motor_0);
398     mat[i][0]=mat[i][2]=0;
399     mat[i][1]=i*40;
400     usleep(40000);
401     double array[8];
402     readvolt(array);
403     mat[i][3]=array[0]-offset[0];
404     mat[i][4]=array[1]-offset[1];
405     usleep(30000);
406   }
407
408   mat2tofile(mat,41,result);
409
410   //fitting x=mat[i][1] y=mat[i][3]=V-output
411   double scale1;
412   scale1=fit(41,mat,result,'S',1,3);
413
414   //fitting x=mat[i][1] y=mat[i][4]=H-output
415   double scale2;
416   scale2=fit(41,mat,result,'S',1,4);
417
418   //scalefactor=amplitude(H)/amplitude(V)
419   double scale_res=scale2/scale1;
420   fprintf(result,"Scalefactor: %f\n",scale_res);
421   return scale_res;
422 }
```

PolCalib.cpp

```
1  /*
2   *
3   * Function for calibrating the H-Polarizer.
4   *
5   *
6   */
7
8  # include <iostream>     /*Standard Input / Output Streams Library*/
9  # include <stdio.h>      /* Standard input/ouput definitions */
10 # include <stdlib.h>     /* C standard library */
11 # include <string.h>     /* string manipulation in C */
12 # include <unistd.h>     /* UNIX standard function definitions */
13 # include <fcntl.h>      /* File control definitions */
14 # include <termios.h>    /* POSIX terminal control definitions  */
```

```
15 # include <sys/wait.h>  /* for wait function*/
16 # include <math.h>              /*to compute mathematical operations and
     transformations*/
17 # include <iomanip>     /*Stream manipulators*/
18 # include <ftdi.h>              /*to control ftdi-USB-chips*/
19 # include <ctime>               /*to get and manipulate date and time
     information*/
20 # include <sched.h>             /*contains the scheduling parameters*/
21 # include <signal.h>    /*to handle signals*/
22
23 # include "fitting.h"
24 # include "reading_data.h"
25 # include "seriell.h"
26 # include "Calib-functions.h"
27
28 # define number_motors 3   /* Definition of number of motors, which are used
     */
29 # define number_pd 2       /*Definition of number of photodiodes, which are
     used */
30
31
32 using namespace std;
33
34
35 //function: for signal
36 void my_handler2(int signum)
37 {
38   printf("Fit routine: timeout\n");
39   exit(1);
40 }
41
42
43 //function: calibrates the polarizer
44 int PolCalib(double steps, FILE *results, double offset[], double array[])
45 {
46   startresetclose(pol);
47   sleep(2);
48
49   //creating matrix with steps
50   double lines2=360./steps+1.;
51   int lines=lines2;
52   double mat1[lines][number_motors];
53   mat1[0][0]=0;
54   mat1[0][1]=0;
55   mat1[0][2]=0;
56   for (int i=1; i<lines; i++)
57   {
58     mat1[i][0]=mat1[i-1][0]+steps;
59     mat1[i][1]=0;
60     mat1[i][2]=0;
61   }
62
63   printmat1(lines,mat1);
64   //converts angles in mat1 to steps
65   convertmat(lines, mat1);
66
67   printmat1(lines , mat1);
68
```

```
69  //creating matrix for storing angles of motors and voltage
70  double mat2 [lines][number_motors+number_pd];
71
72  //each loop one measurement of voltages
73  for(int i=0; i<lines; i++)
74  {
75    // moving motors to angles stored in mat1/stream
76    usleep(50000);
77    motorgotomatelement ( (mat1[i][0]) , pol , motor_1);
78
79    mat2[i][0]=mat1[i][0];
80    mat2[i][1]=mat1[i][1];
81    mat2[i][2]=mat1[i][2];
82    usleep(70000);
83
84    // reading voltage 1 and 2
85    double array [8];
86    readvolt(array);
87    usleep(50000);
88
89    // stores voltage to mat2
90    for (int d =0 ; d < number_pd ; d++)
91    {    mat2[i][number_motors+d]=(array[d]-offset[d]);}
92    usleep(5000);
93  }
94
95  //printing mat2 with results and angles to file
96  mat2tofile( mat2 , lines , results );
97  sleep(1);
98
99  //fitting x=mat2[i][0] y=mat2[i][z]
100 fit(lines , mat2,results,'H',0,3);
101
102
103 //reading zero position of polarizer from file
104 FILE *hpol;
105 char filename0[100]="hpol_is_at.txt";
106 if ((hpol=fopen(filename0,"r"))==NULL)
107 {
108   fprintf (stderr, "Can't␣open␣file␣'%s'␣for␣input:␣", filename0);
109   perror("");
110   return 1;
111 }
112 int angle_H;
113 fscanf(hpol,"%d", &angle_H);
114 fclose(hpol);
115 printf("H-Pol␣is␣read␣from␣file:␣%d\n\n",angle_H);
116
117 //measuring visibility
118 motorgotomatelementlong( angle_H , pol , motor_1);
119 sleep(3);
120 double array1[8];
121 readvolt(array1);
122 float visa=array1[0]-offset[0];
123 sleep(1);
124 int angle_V=angle_H+2400;
125 motorgotomatelementlong( angle_V , pol , motor_1);
126 sleep(3);
```

```
127   readvolt(array1);
128   float visb=array1[0]-offset[0];
129   sleep(1);
130   double vis=((visa-visb)/(visa+visb));
131   fprintf(results,"Visibility measured : %f\n",vis);
132
133
134   motorgotomatelementlong ( angle_H , pol , motor_1);
135   sleep(1);
136
137   array[0]=angle_H;
138   array[1]=vis;
139
140   return 0;
141
142 }
```

LambdaCalib.cpp

```
1  /*
2   *
3   * Functions for calibrating the waveplates
4   *
5   *
6   */
7
8  # include <iostream>     /*Standard Input / Output Streams Library*/
9  # include <stdio.h>      /* Standard input/ouput definitions */
10 # include <stdlib.h>     /* C standard library */
11 # include <string.h>     /* string manipulation in C */
12 # include <unistd.h>     /* UNIX standard function definitions */
13 # include <fcntl.h>      /* File control definitions */
14 # include <termios.h>    /* POSIX terminal control definitions  */
15 # include <sys/wait.h>   /* for wait function*/
16 # include <math.h>              /*to compute mathematical operations and
       transformations*/
17 # include <iomanip>      /*Stream manipulators*/
18 # include <ftdi.h>              /*to control ftdi-USB-chips*/
19 # include <ctime>               /*to get and manipulate date and time
       information*/
20 # include <sched.h>             /*contains the scheduling parameters*/
21 # include <signal.h>     /*to handle signals*/
22
23 # include "fitting.h"
24
25 # include "seriell.h"
26 # include "Calib-functions.h"
27
28 # define PI 3.14159265
29
30 # define number_motors 3    /* Definition of number of motors, which are used
       */
31 # define number_pd 2        /*Definition of number of photodiodes, which are
       used */
32
33
34 using namespace std;
```

```
35
36
37
38
39
40
41 // function: one iteration procedure for finding zero position of waveplates
42 float IterationLambda(FILE *result,int i, double mat[][5],int lines, int
       steps,double offset[],int mins4[],int mins2[],int H_Pol)
43 {
44   //lambda_4: measuring voltage while rotating from 0 to 2400
45   for(int j=0; j<lines; j++)
46   {
47     motorgotomatelement(j*steps,lambda, motor_1);
48     mat[j][2]=j*steps;
49     //read voltage V and store voltage in matrix
50     double array3[8];
51     usleep(40000);
52     readvolt(array3);
53     mat[j][4]=(array3[1]-offset[1]);
54     mat[j][3]=(array3[0]-offset[0]);
55     mat[j][1]=mins2[i];
56     mat[j][0]=H_Pol;
57     usleep(50000);
58   }
59
60   fprintf(result,"\nData␣for␣%i␣iteration␣of␣lambda/4\n",i+1);
61
62   //storing matrix with voltages and steps to file
63   mat2tofile(mat,lines,result);
64   sleep(1);
65
66   //fit routine for finding minimum: x=mat[i][2] y=mat[i][3]
67   fprintf(result,"%i␣iteration␣for␣finding␣minimum␣of␣lambda/4\n",i+1);
68   mins4[i]=fit(lines,mat,result,'L',2,3);
69   sleep(2);
70   fprintf(result,"Minimum␣of␣lambda/4␣:␣iteration␣%i␣␣minimum␣␣%i\n",i+1,
       mins4[i]);
71
72   //measuring voltage of minimum position
73   motorgotomatelementlong(mins4[i],lambda, motor_1);
74   double array4[8];
75   sleep(1);
76   readvolt(array4);
77   fprintf(result,"Voltage␣of␣minmum␣position:␣%f\n",array4[0]);
78   sleep(2);
79
80   //lambda_2: measuring voltage while rotating from 0 to 2400
81   for(int j=0; j<lines; j++)
82   {
83     motorgotomatelement(j*steps,lambda, motor_0);
84     mat[j][1]=j*steps;
85     // reading voltage V and store voltage in matrix
86     double array5[8];
87     usleep(40000);
88     readvolt(array5);
89     mat[j][4]=(array5[1]-offset[1]);
90     mat[j][3]=(array5[0]-offset[0]);
```

```
91      mat[j][2]=mins4[i];
92      mat[j][0]=H_Pol;
93      usleep(50000);
94    }
95
96    //storing matrix with voltages and steps to file
97    fprintf(result,"\nData for %i iteration of lambda/2\n",i+1);
98    mat2tofile(mat,lines,result);
99    sleep(1);
100
101   //fit routine for finding minimum: x=mat[i][1] y=mat[i][3]
102   fprintf(result,"%i iteration for finding minimum of lambda/2\n",i+1);
103   mins2[i]=fit(lines,mat,result,'L',1,3);
104   mins2[i+1]=mins2[i];
105   sleep(2);
106   fprintf(result,"Minimum of lambda/2 : iteration %i  minimum %i\n",i+1,mins2
          [i]);
107   motorgotomatelementlong(mins2[i],lambda, motor_0);
108
109   //measuring voltage of minimum position
110   double array6[8];
111   sleep(4);
112   readvolt(array6);
113   fprintf(result,"Voltage of minmum position: %f\n",array6[0]);
114   sleep(1);
115
116   // measuring voltage of maximum position= min2+1200
117   double vis[8];
118   double visa=array6[1];
119   motorgotomatelementlong(mins2[i]+1200,lambda, motor_0);//motor goes to
          maximum
120   sleep(4);
121   readvolt(vis);
122   usleep(50000);
123
124   //calculating visibility
125   double visb=vis[1];
126   float visibility=((visa-offset[1])-(visb-offset[1]))/((visa-offset[1])+(
          visb-offset[1]));
127   fprintf(result,"\nVisibility : %.6f\n",visibility);
128
129   //motors going to minima
130   motorgotomatelementlong(mins2[i],lambda, motor_0);
131   sleep(1);
132   motorgotomatelementlong(mins4[i],lambda, motor_1);
133   sleep(2);
134
135   return visibility;
136 }
137
138
139 //function: whole procedure for calibrating the waveplates
140 float LambdaCalib2(FILE *result,double noise[],int start0, int start1)
141 {
142
143   //restarting all motors and go to their start positions
144   startresetclose(lambda);
145   sleep(5);
```

```
146   motorgotomatelementlong(start0, lambda,motor_0);
147   sleep(1);
148   motorgotomatelementlong(start1, lambda,motor_1);
149   sleep(1);
150   startresetclose(pol);
151   sleep(8);
152
153   // set polarizer to zero
154   FILE *stream0;
155   char filename0[100]="hpol_is_at.txt";
156   if ((stream0=fopen(filename0,"r"))==NULL)
157   {
158     fprintf (stderr, "Can't open file '%s' for input: ", filename0);
159     perror("");
160     return 1;
161   }
162   int H_Pol;
163   fscanf(stream0,"%d", &H_Pol);
164   fclose(stream0);
165   fprintf(result,"H-Pol is read from file: %d\n",H_Pol);
166   motorgotomatelementlong ( H_Pol , pol , motor_1);
167   sleep(2);
168
169   //storing start points for calibration
170   int mins2[12];
171   mins2[0]=start0;
172   int mins4[12];
173   mins4[0]=start1;
174
175   //visibility
176   float visibility=0;
177
178   //counts iteration
179   int i=0;
180
181
182   //Now doing the iteration procedure with the function IterationLambda for
        different steps while visibility is good enough
183   while (visibility <0.99)
184   {
185     double mat[25][5];
186     visibility=IterationLambda(result,i,mat,25,100,noise,mins4,mins2,H_Pol);
187     if (i==6)
188     { fprintf(result,"Too many iterations: 6\n");
189     break;}
190     i=i+1;
191   }
192
193
194   while (visibility <0.9995)
195   {
196     double matz[61][5];
197     visibility=IterationLambda(result,i,matz,61,40,noise,mins4,mins2,H_Pol);
198     if (i==12)
199     { fprintf(result,"Too many iterations: 12\n");
200     break;}
201     i=i+1;
202   }
```

```
203
204
205
206   fprintf(result,"Zero␣positions␣for␣lambda_4␣and␣lambda_2:␣␣%d␣␣␣%d\n",mins4
          [i-1],mins2[i-1]);
207
208   //writing the zero positions to a file
209   FILE* l_2;
210   if ( (l_2= fopen("lambda_2_is_at.txt","w"))==NULL)
211   {
212     fprintf(stderr, "Can't␣open/make␣file␣'lamda_2_is_at.txt'\n");
213     return 1;
214   }
215   fprintf(l_2,"%d",mins2[i-1]);
216   fclose(l_2);
217
218   FILE* l_4;
219   if ( (l_4= fopen("lambda_4_is_at.txt","w"))==NULL)
220   {
221     fprintf(stderr, "Can't␣open/make␣file␣'lambda_4_is_at.txt'\n");
222     return 1;
223   }
224   fprintf(l_4,"%d",mins4[i-1]);
225   fclose(l_4);
226
227   return visibility;
228 }
229
230
231 //function: for finding zero positions by calculating it from measured
       visibility
232 double LambdaCalib3(FILE *result,double offset[],int start0, int start1)
233 {
234   //restarting all motors and go to their start positions
235   startresetclose(lambda);
236   sleep(3);
237   startresetclose(pol);
238   sleep(3);
239
240   // set polarizer to zero
241   FILE *stream0;
242   char filename0[100]="hpol_is_at.txt";
243   if ((stream0=fopen(filename0,"r"))==NULL)
244   {
245     fprintf (stderr, "Can't␣open␣file␣'%s'␣for␣input:␣", filename0);
246     perror("");
247     return 1;
248   }
249   int H_Pol;
250   fscanf(stream0,"%d", &H_Pol);
251   fclose(stream0);
252   printf("H-Pol␣is␣read␣from␣file:␣%d\n",H_Pol);
253   motorgotomatelementlong ( H_Pol , pol , motor_1);
254   sleep(2);
255
256
257   //matrix for storing results
258   double mat[61][5];
```

```
259
260
261
262   double visibility;
263   int min4;
264   int min2;
265
266   double vis[2];
267   double min[2];
268
269   //lambda_4 go to random start point
270   motorgotomatelementlong(start1,lambda,motor_1);
271   sleep(2);
272   //lambda_2: measuring voltage while rotating from 0 to 2400
273   for (int i=0; i<61; i++)
274   {
275     motorgotomatelement(i*40,lambda,motor_0);
276     mat[i][1]=i*40;
277     mat[i][0]=H_Pol;
278     mat[i][2]=start1;
279     double array[8];
280     readvolt(array);
281     mat[i][3]=array[0]-offset[0];
282     mat[i][4]=array[1]-offset[1];
283   }
284
285   //print results to file
286   mat2tofile(mat,61,result);
287
288   //fit routine for calculating zero position from visibility: x=mat[i][1] y=
        mat[i][3]
289   double min4_fit=fit(61,mat,result,'V',1,3);
290   min4=start1-min4_fit;
291   while(min4>=2400)
292   {min4=min4-2400;}
293   while(min4<0)
294   {min4=min4+2400;}
295   sleep(3);
296
297
298   //two directions +phi and -phi
299   fprintf(result,"Two direction: first -phi, then +phi from startpoint\n");
300   fprintf(result,"phi_0= %d - %f = %f\n",start1,min4_fit,start1-min4_fit);
301   fprintf(result,"phi_0= %d + %f = %f\nphi_0 gets transformed --> phi_0
        element"
302                  "of [0,2400] / one period\n\n",start1,min4_fit,start1+
                       min4_fit);
303   for (int j=0; j<2; j++)
304   {
305     fprintf(result,"phi_0=%d\n",min4);
306     //lambda_4: go to zero position
307     motorgotomatelementlong(min4,lambda,motor_1);
308     sleep(3);
309
310     //lambda_2: measuring voltage while rotating from 0 to 4800
311     for (int i=0; i<61; i++)
312     {
313       motorgotomatelement(i*40,lambda,motor_0);
```

```
314         mat[i][1]=i*40;
315         mat[i][0]=H_Pol;
316         mat[i][2]=min4;
317         double array[8];
318         readvolt(array);
319         mat[i][3]=array[0]-offset[0];
320         mat[i][4]=array[1]-offset[1];
321       }
322
323       //print results to file
324       mat2tofile(mat,61,result);
325
326       //fit routine for finding minimum: x=mat[i][1] y=mat[i][3]
327       min[j]=fit(61,mat,result,'L',1,3);
328
329       //lambda_2:go to minimum
330       motorgotomatelementlong(min[j],lambda,motor_0);
331       sleep(3);
332
333       fprintf(result,"Zero positions for lambda_4 and lambda_2:  %d   %f\n",
              min4,min[j]);
334
335       //measuring visibility at these zero positions
336       double array6[8];
337       sleep(4);
338       readvolt(array6);
339       double vol1=array6[1]-offset[1];
340       sleep(1);
341       motorgotomatelementlong(min[j]+1200,lambda, motor_0);
342       sleep(4);
343       readvolt(array6);
344       usleep(50000);
345
346       //calculating visibility
347       double vol2=array6[1]-offset[1];
348       double visibility=(vol1-vol2)/(vol1+vol2);
349       vis[j]=visibility;
350       fprintf(result,"Visibility : %.6f\n\n",visibility);
351
352       //other direction
353       min4=start1+min4_fit;
354       while(min4>=2400)
355       {min4=min4-2400;}
356
357
358   }
359
360   //testing visibility
361   if(vis[0]>vis[1])
362   {min4=start1-min4_fit;
363   min2=min[0];
364   visibility=vis[0];
365   }
366   else
367   {min2=min[1];
368   visibility=vis[1];
369   }
370
```

```
371   while(min4>2400)
372   {min4=min4-2400;}
373   while(min4<0)
374   {min4=min4+2400;}
375
376   //motors going to minima
377   motorgotomatelementlong(min2,lambda, motor_0);
378   sleep(1);
379   motorgotomatelementlong(min4,lambda, motor_1);
380   sleep(2);
381
382
383   fprintf(result,"Found zero positions: lambda/2   %d, lambda/4   %d\n\n",min2,
          min4);
384
385
386
387
388
389
390
391
392   //writing the zero positions to a file
393   FILE* l_2;
394   if ( (l_2= fopen("lambda_2_is_at.txt","w"))==NULL)
395   {
396     fprintf(stderr, "Can't open/make file 'lamda_2_is_at.txt'\n");
397     return 1;
398   }
399   fprintf(l_2,"%d",min2);
400   fclose(l_2);
401
402   FILE* l_4;
403   if ( (l_4= fopen("lambda_4_is_at.txt","w"))==NULL)
404   {
405     fprintf(stderr, "Can't open/make file 'lambda_4_is_at.txt'\n");
406     return 1;
407   }
408   fprintf(l_4,"%d",min4);
409   fclose(l_4);
410
411   return visibility;
412
413 }
```

fitting.cpp

```
1 /*
2  *
3  *
4  * This Programm contains functions for fitting data to a sine
5  * with help of the gsl library:
6  * http://www.gnu.org/software/gsl/manual/html_node/
7  *
8  *
9  */
10
```

```
11 /* GNU SCIENTIFIC LIBRARY */
12 #include <gsl/gsl_rng.h>
13 #include <gsl/gsl_randist.h>
14 #include <gsl/gsl_vector.h>
15 #include <gsl/gsl_blas.h>
16 #include <gsl/gsl_multifit_nlin.h>
17
18 # include <iostream>     /*Standard Input / Output Streams Library*/
19 # include <stdio.h>      /* Standard input/ouput definitions */
20 # include <stdlib.h>     /* C standard library */
21 # include <string.h>     /* string manipulation in C */
22 # include <unistd.h>     /* UNIX standard function definitions */
23 # include <fcntl.h>      /* File control definitions */
24 # include <termios.h>    /* POSIX terminal control definitions  */
25 # include <sys/wait.h>   /* for wait function*/
26 # include <math.h>              /*to compute mathematical operations and
     transformations*/
27 # include <iomanip>      /*Stream manipulators*/
28 # include <ftdi.h>              /*to control ftdi-USB-chips*/
29 # include <ctime>               /*to get and manipulate date and time
     information*/
30 # include <sched.h>             /*contains the scheduling parameters*/
31 # include <signal.h>     /*to handle signals*/
32 # include "Calib-functions.h"     /*for calibrating*/
33
34 # define PI 3.14159265
35 # define number_motors 3    /* Definition of number of motors, which are used
     */
36 # define number_pd 2        /*Definition of number of photodiodes, which are
     used */
37
38 using namespace std;
39
40
41 // data structure contains: Number of measurements , steps , voltage
42 struct data
43 {
44         size_t n;
45         double *x1;
46         double * y;
47         double * sigma;
48 };
49
50
51 //function: stores in vector f the difference of the measured voltage and the
     calculated voltage
52 int sin_f (const gsl_vector * x, void *data, gsl_vector * f)
53 {
54         size_t n = ((struct data *)data)->n;
55         double *y = ((struct data *)data)->y;
56         double *x1 = ((struct data *)data)->x1;
57         double *sigma = ((struct data *) data)->sigma;
58         double a = gsl_vector_get (x, 0);
59         double b= gsl_vector_get (x, 1);
60         double c = gsl_vector_get (x, 2);
61         double d = gsl_vector_get (x, 3);
62
63         size_t i;
```

```
64
65          for (i = 0; i < n; i++)
66          {
67                  // Model Yi = a*sin(b*x+d)+c
68                  double t = x1[i];
69                  double Yi = a*sin(b*t+d)+c;
70                  gsl_vector_set (f, i, (Yi - y[i])/sigma[i]);
71          }
72
73          return GSL_SUCCESS;
74  }
75
76
77  // function: calculates the Jacobian matrix
78  int sin_df (const gsl_vector * x, void *data, gsl_matrix * J)
79  {
80          size_t n = ((struct data *)data)->n;
81          double *sigma = ((struct data *) data)->sigma;
82          double *x1 = ((struct data *) data)->x1;
83          double a = gsl_vector_get (x, 0);
84          double b= gsl_vector_get (x, 1);
85          double c = gsl_vector_get (x, 2);
86          double d = gsl_vector_get (x, 3);
87
88          size_t i;
89
90          for (i = 0; i < n; i++)
91          {
92                  /* Jacobian matrix J(i,j) = dfi / dxj,
93                  where fi = (Yi - yi)/sigma[i],
94          Yi =a*sin(b*x+d)+c
95                  and the xj are the parameters (a,b,c,d) */
96                  double t = x1[i];
97                  double s = sigma[i];
98                  gsl_matrix_set (J, i, 0, sin(b*t+d)/s);
99                  gsl_matrix_set (J, i, 1, (a*cos(b*t+d)*t)/s);
100                 gsl_matrix_set (J, i, 2, 1/s);
101                 gsl_matrix_set (J, i, 3, (a*cos(b*t+d))/s);
102         }
103         return GSL_SUCCESS;
104 }
105
106
107 //function:executes the function sin_f ans sin_df
108 int sin_fdf (const gsl_vector * x, void *data, gsl_vector * f, gsl_matrix * J
        )
109 {
110         sin_f (x, data, f);
111         sin_df (x, data, J);
112         return GSL_SUCCESS;
113 }
114
115
116 //function: searches step in column x for max voltage in column y
117 int searchmax( int N,double mat[][number_motors+number_pd],int x, int y)
118 {
119         int a;
120         double b=0;
```

```
121         for (int i=0; i<N ; i++)
122         {
123                 if (mat[i][y]>b)
124                 {
125                         b=mat[i][y];
126                         a=mat[i][x];
127                 }
128         }
129     if ( a >= 4800 )
130         { a=a-4800 ; }
131         printf("Angle of max Voltage: %d\n",a);
132 
133         return a;
134 }
135 
136 
137 //function: seaches step for min voltage
138 int searchmin(int N,double mat[][number_motors+number_pd],int x, int y)
139 {
140         int a;
141         double b=100;
142         for (int i=0 ; i<N ; i++)
143         {
144                 if (mat[i][y]<b)
145                 {
146                         b=mat[i][y];
147                         a=mat[i][x];}
148                 }
149         if ( a >= 4800 )
150         { a=a-4800 ; }
151         printf("Angle of Min Voltage: %d\n",a);
152         return a;
153 }
154 
155 //function: searches min voltage
156 double searchminvolt(int N, double mat[][number_motors+number_pd],int x, int
      y)
157 {
158         double a=100;
159         for (int i=0; i< N ; i++)
160         {
161                 if (mat[i][y]<a)
162                 {a=mat[i][y];}
163         }
164 
165         printf("Min Voltage: %.6f\n",a);
166         return a;
167 }
168 
169 
170 //function searches max voltage
171 double searchmaxvolt(int N, double mat[][number_motors+number_pd],int x, int
      y)
172 {
173         double a=0;
174         for (int i=0; i<N ; i++)
175         {
176                 if (mat[i][y]>a)
```

```
177                 {a=mat[i][y];}
178         }
179         printf("Max Voltage: %.6f\n",a);
180         return a;
181 }
182
183
184 //function: prints state and parameters of an iteration
185 void print_state (size_t iter, gsl_multifit_fdfsolver * s,FILE *result);
186
187
188 //function: fits the data to a sine with different options
189 double fit ( int N , double mat3[][number_motors+number_pd],FILE *result, int
        option, int a, int b)
190 {
191         //initializing the Solver
192         const gsl_multifit_fdfsolver_type *T;
193         gsl_multifit_fdfsolver *s;
194
195         int status;
196         unsigned int i, iter = 0;
197         const size_t n = N;
198         const size_t p = 4;
199
200         //covariance matrix
201         gsl_matrix *covar = gsl_matrix_alloc (p, p);
202
203         //data structure
204         double y[N], sigma[N], x1[N] ;
205         struct data d = { n, x1, y, sigma};
206
207         //providing the Function to be Minimized
208         gsl_multifit_function_fdf f;
209
210         //storing time and date
211         time_t t;
212         time(&t);
213         char buffert[200];
214         sprintf(buffert,"%s",ctime(&t));
215         fputs(buffert, result);
216
217         //calculating start parameters for fit
218         double a_start;
219         double b_start;
220         double c_start;
221         double d_start;
222
223         //option for polarizer; period is 4800 steps
224         if (option=='H')
225         {       printf("option H\n");
226                 double c_start1=searchmaxvolt(N,mat3,0,3);
227                 double a_start1=searchminvolt(N,mat3,0,3);
228                 double b_start1=searchmax(N,mat3,0,3);
229                 double d_start1=searchmin(N,mat3,0,3);
230                 a_start=(c_start1-a_start1)/2.0;
231                 d_start=(d_start1+labs(b_start1-d_start1)/2.);
232                 c_start=a_start1+(c_start1-a_start1)/2.;
233                 while (d_start>4800)
```

```
234                 {d_start=d_start-4800.;}
235
236                 b_start=2*PI/4800;
237                 d_start=-d_start*2*PI/4800;
238         }
239
240         //option for scaling; period is 2400
241         if (option=='S')
242         {       printf("option␣S\n");
243                 double c_start1=searchmaxvolt(N,mat3,1,b);
244                 double a_start1=searchminvolt(N,mat3,1,b);
245                 double b_start1=searchmax(N,mat3,1,b);
246                 if (b_start1>=2400)
247                 {b_start1=b_start1-2400;}
248                 while(b_start1<0)
249                 {b_start1=b_start1+2400;}
250                 double d_start1=searchmin(N,mat3,1,b);
251                 if (d_start1>=2400)
252                 {d_start1=d_start1-2400;}
253                 while(d_start1<0)
254                 {d_start1=d_start1+2400;}
255                 a_start=(c_start1-a_start1)/2.0;
256                 d_start=-1*(d_start1+labs(b_start1-d_start1)/2.)*2*PI/2400;
257                 c_start=a_start1+(c_start1-a_start1)/2.;
258                 b_start=2*PI/2400;
259         }
260
261         //option for lambda or for visibility
262         if ( (option=='L') || (option=='V') )
263         {       printf("option␣L/V\n");
264                 double c_start1=searchmaxvolt(N,mat3,a,3);
265                 double a_start1=searchminvolt(N,mat3,a,3);
266                 double b_start1=searchmax(N,mat3,a,3);
267                 if (b_start1>=2400)
268                 {b_start1=b_start1-2400;}
269                 while(b_start1<0)
270                 {b_start1=b_start1+2400;}
271                 double d_start1=searchmin(N,mat3,a,3);
272                 if (d_start1>=2400)
273                 {d_start1=d_start1-2400;}
274                 while(d_start1<0)
275                 {d_start1=d_start1+2400;}
276                 a_start=(c_start1-a_start1)/2.0;
277                 d_start=(-1*(d_start1+labs(b_start1-d_start1)/2.)*2*PI/2400);
278                 c_start=a_start1+(c_start1-a_start1)/2.;
279                 b_start=2*PI/2400;
280         }
281
282         //start parameters for fit in vector
283         double x_init[4] = { a_start , b_start , c_start , d_start};
284         gsl_vector_view x = gsl_vector_view_array (x_init, p);
285         fprintf(result,"start␣parameters␣of␣fit:␣a=%.5f␣␣b=%.5f␣␣c=%5f␣␣d=%5f
              ␣\n␣",a_start,b_start,c_start,d_start);
286
287     //parameters of multifit_function
288         f.f = &sin_f;
289         f.df = &sin_df;
290         f.fdf = &sin_fdf;
```

```
291         f.n = n;
292         f.p = p;
293         f.params = &d;
294 
295         //storing data to be fitted in data structure
296         for (i = 0; i < n; i++)
297         {
298                 x1[i]=mat3[i][a];
299                 y[i]=mat3[i][b];
300                 sigma[i] = 1;
301                 printf ("data: %g %g %g\n", x1[i], y[i], sigma[i]);
302         }
303 
304      //initializing solver as Levenberg-Marquardt algorithm
305         T = gsl_multifit_fdfsolver_lmsder;
306         s = gsl_multifit_fdfsolver_alloc (T, n, p);
307         gsl_multifit_fdfsolver_set (s, &f, &x.vector);
308 
309         print_state (iter, s,result);
310 
311     //do iteration for solving
312         do
313         {
314                 iter++;
315                 status = gsl_multifit_fdfsolver_iterate (s);
316                 print_state (iter, s,result);
317                 if (status)
318                 break;
319                 //break if x,dx < 1e-5
320                 status = gsl_multifit_test_delta (s->dx, s->x,1e-5, 1e-5);
321         }
322         while (status == GSL_CONTINUE && iter < 500);
323     fprintf (result,"status = %s\n", gsl_strerror (status));
324 
325     //calculating covariance matrix
326         gsl_multifit_covar (s->J, 0.0, covar);
327 
328         //results
329         #define FIT(i) gsl_vector_get(s->x, i)
330         #define ERR(i) sqrt(gsl_matrix_get(covar,i,i))
331     double chi = gsl_blas_dnrm2(s->f);
332         double dof = n - p;
333         double c = GSL_MAX_DBL(1, chi / sqrt(dof));
334     fprintf (result,"a       = %.7f +/- %.7f\n",FIT(0), c*ERR(0) );
335         fprintf (result,"b       = %.7f +/- %.7f\n", FIT(1), c*ERR(1));
336         fprintf (result,"c       = %.7f +/- %.7f\n",FIT(2), c*ERR(2));
337         fprintf (result,"d       = (%.7f * PI) +/- %.7f\n", FIT(3)/PI, c*ERR
                (3));
338         fprintf (result,"fitted function: a*sin(b*x+d)+c\n");
339 
340         //a2*sin(b2*x+d2)+c2
341         double d2=FIT(3);
342         double b2=FIT(1);
343         double a2=FIT(0);
344         double c2=FIT(2);
345 
346         double x_max=((PI/2.-d2)/b2);
347         double x_min=((-PI/2.-d2)/b2);
```

```
348         double f_max=(a2*sin(PI/2.)+c2);
349         double f_min=(a2*sin(-PI/2.)+c2);
350         double Vis=((f_max-f_min)/(f_max+f_min));
351 
352         if (option=='H')
353         {
354                 fprintf(result,"Visibility from Fit = %f\n",Vis);
355                 //Minimum --> Position for H Polarizer
356                 double erg=(PI/2.-d2)/b2;
357                 fprintf(result,"H Polarisation at %f\n",erg);
358                 // 0 < Minimum < 4800 and rounded
359                 while (erg >=4800)
360                 {erg=erg-4800;}
361                 if (erg > 0 )
362                 {erg = erg +0.5;}
363                 else if (erg<0)
364                 {erg=erg-0.5;}
365                 else
366                 {erg =0;}
367 
368                 //save position for H-Polarisation to file
369                 int h=erg;
370                 FILE* hpol;
371                 if ( (hpol= fopen("hpol_is_at.txt","w"))==NULL) //writes
                        angle of H Polarisation in file, which gets overwritten
372                 {
373                         fprintf(stderr, "Can't open/make file 'hpol_is_at.txt
                                '\n");
374                         return 1;
375                 }
376                 fprintf(hpol,"%i",h);
377                 fclose(hpol);
378 
379                 fprintf(result," --> H Polarisation at %i\n\n",h);
380                 return erg;
381         }
382 
383         if (option=='S')
384         {
385                 if (a2<0)
386                 {a2=(-1*a2);}
387                 fprintf(result,"Amplitude: %f\n\n",a2);
388 
389                 return a2;
390         }
391 
392         if (option=='L')
393         {
394                 double erg=(-PI/2.-d2)/b2;
395 
396                 while (erg >=2400)
397                 {erg=erg-2400;}
398                 while (erg <0)
399                 {erg=erg+2400;}
400 
401 
402                 double erg2;
403                 if (erg > 0 )
```

```
404                 {erg2 = erg +0.5;}
405                 else if (erg<0)
406                 {erg2=erg-0.5;}
407                 else
408                 {erg2 =0;}
409                 int erg_round=erg2;
410                 fprintf(result,"Minimum: %i\n\n",erg_round);
411
412                 return erg;
413
414
415
416         }
417
418         if (option=='V')
419         {
420
421                 //measuring visibility
422                 double erg=(-PI/2.-d2)/b2;
423
424                 while (erg >=2400)
425                 {erg=erg-2400;}
426                 while (erg <0)
427                 {erg=erg+2400;}
428
429
430                 if (erg > 0 )
431                 {erg = erg +0.5;}
432                 else if (erg<0)
433                 {erg=erg-0.5;}
434                 else
435                 {erg =0;}
436                 int erg_round=erg;
437
438                 motorgotomatelement(erg_round,lambda,motor_0);
439                 sleep(2);
440                 double array[8];
441                 readvolt(array);
442                 double vol1=array[1];
443                 motorgotomatelement(erg_round+1200,lambda,motor_0);
444                 sleep(2);
445                 readvolt(array);
446                 double vol2=array[1];
447                 double vis=(vol1-vol2)/(vol1+vol2);
448                 double phi_0=acos(vis)/2.;
449                 fprintf(result,"Visibility from measurement: %f\n",vis);
450                 double erg_1=phi_0/2./PI*9600;
451                 while(erg_1>=2400)
452                 {erg_1=erg_1-2400;}
453                 while(erg_1<0)
454                 {erg_1=erg_1+2400;}
455
456                 int erg_round2;
457                 if(erg_1>=0)
458                 {erg_1=erg_1+0.5;}
459                 else
460                 {erg_1=erg_1-0.5;}
461                 erg_round2=erg_1;
```

```
462                 fprintf(result,"Zero postition of lambda/4 --> acos(
                        visibility)/(4*pi)*9600= %i\n",erg_round2);
463                 int res=erg_1;
464
465
466                 vis=(f_max-f_min)/(f_max+f_min);
467                 phi_0=acos(vis)/2.;
468                 erg_1=phi_0/2./PI*9600;
469                 while(erg_1>=2400)
470                 {erg_1=erg_1-2400;}
471                 while(erg_1<0)
472                 {erg_1=erg_1+2400;}
473
474
475
476                 fprintf(result,"Visibility from fit: %f\n",vis);
477                 erg_round;
478                 if(erg_1>=0)
479                 {erg_1=erg_1+0.5;}
480                 else
481                 {erg_1=erg_1-0.5;}
482                 erg_round=erg_1;
483                 fprintf(result,"Zero postition of lambda/4 from fit --> acos(
                        visibility)/(4*pi)*9600= %i NOT USED\n\n",erg_round);
484                 return res;
485
486
487
488
489         }
490
491
492         //free all memory
493         gsl_multifit_fdfsolver_free (s);
494         gsl_matrix_free (covar);
495
496 }
497
498
499 //function: prints state of iteration
500 void print_state (size_t iter, gsl_multifit_fdfsolver * s,FILE *result)
501 {
502         fprintf (result, "iter: %3u x = % 15.8f % 15.8f % 15.8f % 15.8f  |f(x
                )| = %f\n",
503     iter, gsl_vector_get (s->x, 0), gsl_vector_get (s->x, 1),
504     gsl_vector_get (s->x, 2), gsl_vector_get (s->x, 3), gsl_blas_dnrm2 (s->f)
            );
505 }
```

calibration.cpp

```
1 /*
2  *
3  *
4  * This programm is for calbrating the polarizer and the waveplates.
5  *
6  *
```

```
 *
 */

#include <iostream>     /*Standard Input / Output Streams Library*/
#include <stdio.h>      /* Standard input/ouput definitions */
#include <stdlib.h>     /* C standard library */
#include <string.h>     /* string manipulation in C */
#include <unistd.h>     /* UNIX standard function definitions */
#include <fcntl.h>      /* File control definitions */
#include <termios.h>    /* POSIX terminal control definitions  */
#include <sys/wait.h>   /* for wait function*/
#include <math.h>               /*to compute mathematical operations and transformations*/
#include <iomanip>      /*Stream manipulators*/
#include <ftdi.h>               /*to control ftdi-USB-chips*/
#include <ctime>                /*to get and manipulate date and time information*/

#include "seriell.h"    /*for communicating with the motor */

#include "fitting.h"     /*for fitting data to a sine*/
#include "Calib-functions.h"      /*for calibrating*/
#include "PolCalib.h" /*for calibrating H polarizer*/
#include "LambdaCalib.h" /*for calibrating waveplates*/


using namespace std;


int main(int argc, char *argv[])
{

  int option,i;
  int opt_pol=0;
  int opt_scale=0;
  int opt_offset=0;
  int opt_Vis=0;
  int opt_lambda=0;
  int opt_noise=0;
  char *arg_offset0=NULL;
  char *arg_offset1=NULL;
  char *arg_number=NULL;
  char *arg_filename=NULL;
  char *arg_steps=NULL;
  char *arg_times=NULL;
  char *arg_times2=NULL;
  char *arg_lambda=NULL;


  while ( (option=getopt(argc, argv , "hsp:l:b:f:d:n:o:")) >= 0)  //getting options
    switch (option)
    {
      case 'h' :  FILE *manpage;
      if ( (manpage = fopen("calibration.1","r"))==NULL)
      {
                // text from manpage
      }
```

```
62        else
63        {system("man -l calibration.1");}
64
65        return 0;
66        case 's' :  opt_scale=1;
67                break;
68        case 'p' :  opt_pol=1;
69                arg_steps=optarg;
70                break;
71
72
73        case 'l' :  opt_lambda=1;
74                arg_lambda=optarg;
75                break;
76        case 'b' :  arg_offset0=optarg;
77                break;
78        case 'd' :  arg_offset1=optarg;
79                break;
80        case 'n' :  opt_noise=1;
81                arg_times=optarg;
82                break;
83        case 'o':   opt_offset=1;
84                arg_times2=optarg;
85                break;
86        case 'f' :  arg_filename=optarg;
87                break;
88        case '?' :  printf("Usage : %s [-help][-offset times] "
89          "[-polarizer steps][-lambda 1/2] [-file filename] [-b offsetPD1]"
90          "[-d offsetPD2] [-noise times] [-scale]\n",argv[0]);
91        return 1;
92        }
93
94
95        //there are two photodiodes
96        int numberpd=2;
97        printf("Number PDs : %d\n",numberpd);
98
99        // if filename isnt given -> results are getting stored in results.txt
100       char filename[100];
101       char buffer[40];
102       if (arg_filename)
103       {
104         sprintf(buffer,"%s",arg_filename);
105         strcpy(filename,buffer);
106       }
107       else
108       {
109         sprintf(buffer,"%s","results.txt");
110         strcpy(filename,buffer);
111       }
112       printf("Results are getting stored in '%s'. \n",filename);
113
114         //opening file for storing all results
115         FILE *result;
116         if ( (result = fopen(filename,"a+"))==NULL)
117         {
118           fprintf(stderr, "Can't open/make file '%s'\n",filename);
119           return 1;
```

```
120         }
121
122         fprintf(result,"\n\n\n\nMeasurment:␣");
123
124         //storing time and date to file
125         time_t t;
126         time(&t);
127         char buffert[200];
128         sprintf(buffert,"%s",ctime(&t));
129         fputs(buffert, result);
130
131         //array for storing offset of multimeter
132         double offset[numberpd];
133
134         //option Lambda 1/2
135         int optionL;
136         if(arg_lambda==NULL)
137         {optionL=1;}
138         else
139         {
140           optionL=atoi(arg_lambda);
141           if( (optionL==1) || (optionL==2))
142           {}
143           else
144           { fprintf(stderr,"non␣valid␣option␣for␣lambda.\nUsage␣:␣%s␣[-lambda
                ␣option1/2]\n",argv[0]);
145           return 1;}
146         }
147
148         // if you want to measure the offset of the multimeter
149         if (opt_offset==1)
150         {
151
152           //waiting untill ready
153           if((opt_pol==1) || (opt_lambda==1) || (opt_scale==1) )
154           {
155             int test=0;
156             while (test==0)
157             {printf("Ready␣for␣measuring␣offset?␣Put␣off␣Photodiodes␣from␣
                    Multimeter␣[YES/NO]␣");
158       char res[10];
159       scanf("%s",&res);
160           int res_i=toupper(res[0]);
161           if(res_i=='Y')
162           {test=1;}
163           else
164           {test=0;}
165               }
166           }
167
168
169           fprintf(result,"\n[-offset]:\n");
170           sleep(1);
171
172           // times2 is number of iterations
173           int times2=20;
174           if (arg_times2==NULL)
175           {}
```

```
176           else
177           {times2=atoi(arg_times2);}
178           printf("doing measurment %d times\n",times2);
179
180         // measuring the offset of the multimeter
181         noisemes(result,times2,offset);
182         fprintf(result,"Offsets: PD1 %f, PD2 %f\n",offset[0],offset[1]);
183         printf("Put Photodiodes again to Multimeter!\n");
184         sleep(2);
185         }
186
187         fclose(result);
188
189         // storing offset in array offset
190         if ((arg_offset0==NULL) && (opt_offset==0) )
191         {offset[0]=0; }
192         else if ( (arg_offset0==NULL) && (opt_offset==1) )
193         {}
194         else
195         {offset[0]=atof(arg_offset0);}
196
197         if ((arg_offset1==NULL) && (opt_offset==0) )
198         {offset[1]=0; }
199         else if ( (arg_offset1==NULL) && (opt_offset==1) )
200         {}
201         else
202         {offset[1]=atof(arg_offset1);}
203
204
205         //open file for storing results again
206         FILE* result2;
207         if ( (result2 = fopen(filename,"a+"))==NULL)
208         {
209           fprintf(stderr, "Can't open/make file '%s'\n",filename);
210           return 1;
211         }
212
213
214
215         if (opt_pol==1)  // if you want to find H Polarisation
216 {
217   //waiting untill ready
218   if((opt_offset==1) || (opt_lambda==1) || (opt_scale==1) )
219   {
220     int test=0;
221     while (test==0)
222     {printf("Ready for calibrating polarizer? [YES/NO] ");
223     char res[10];
224     scanf("%s",&res);
225     int res_i=toupper(res[0]);
226     if(res_i=='Y')
227     {test=1;}
228     else
229     {test=0;}
230     }
231   }
232   fprintf(result2,"\n[-polarizer]:\n");
233
```

```
234   for (int i=0 ; i<numberpd ; i++)
235   {
236     fprintf(result2,"PD%i␣␣␣offset:%.6f␣\n",i+1,offset[i]);
237   }
238
239   //converting given steps to integer
240   int steps=3;
241   if (arg_steps==NULL)
242   {}
243   else if ( atoi(arg_steps)>=0 )
244   {steps=atoi(arg_steps); }
245   else
246   {steps=-atoi(arg_steps);}
247
248   if (steps>45)
249   {fprintf(result2,"Attention:␣steps=%i␣are␣very␣great\n",steps);}
250
251   fprintf(result2,"Steps␣to␣take␣for␣motor␣in␣degree:␣%i\n",steps);
252
253   // calibrating H polarizer
254   double array[2];
255   PolCalib(steps,result2,offset,array);
256
257 }
258
259 fclose(result2);
260
261
262 //open file for storing results again
263 FILE* result4;
264 if ( (result4 = fopen(filename,"a+"))==NULL) //open file
265 {
266   fprintf(stderr, "Can't␣open/make␣file␣'%s'\n",filename);
267   return 1;
268 }
269
270
271 // if you want to find phi_0 and theta_0
272 if( opt_lambda==1  )
273 {
274   //waiting untill ready
275   if((opt_offset==1) || (opt_pol==1) || (opt_scale==1))
276   {
277     int test=0;
278     while (test==0)
279     {printf("Ready␣for␣calibrating␣waveplates?␣[YES/NO]␣");
280     char res[10];
281     scanf("%s",&res);
282     int res_i=toupper(res[0]);
283     if(res_i=='Y')
284     {test=1;}
285     else
286     {test=0;}
287     }
288   }
289   fprintf(result4,"\n[-lambda]:\n");
290
291   for (int i=0 ; i<numberpd ; i++)
```

```
292   {
293     fprintf(result4,"PD%i  offset:%.6f \n",i+1,offset[i]);
294   }
295
296   //visibility method
297   if (optionL==1)
298   {
299     fprintf(result4,"Option 1\n");
300     LambdaCalib3(result4,offset,0,0);
301   }
302
303   //iteration method
304   else
305   {
306     fprintf(result4,"Option 2\n");
307     LambdaCalib2(result4,offset,0,0);}
308 }
309 fclose(result4);
310
311
312 //open file for storing results again
313 FILE* result3;
314 if ( (result3 = fopen(filename,"a+"))==NULL) //open file
315 {
316   fprintf(stderr, "Can't open/make file '%s'\n",filename);
317   return 1;
318 }
319
320 // if you want to measure the scalefactor
321 if (opt_scale==1)
322 {
323   //waiting untill ready
324   if((opt_pol==1)|| (opt_lambda==1) || (opt_offset==1) )
325   {
326     int test=0;
327     while (test==0)
328     {printf("Ready for measuring scalefactor? [YES/NO] ");
329     char res[10];
330     scanf("%s",&res);
331     int res_i=toupper(res[0]);
332     if(res_i=='Y')
333     {test=1;}
334     else
335     {test=0;}
336     }
337   }
338
339   fprintf(result3,"\n[-scale]:\n");
340   for (int i=0 ; i<numberpd ; i++)
341   {
342     fprintf(result3,"PD%i     offset:%.6f \n",i+1,offset[i]);
343   }
344
345   scalefactor(offset,result3);
346
347 }
348 fclose(result3);
349
```

```
350
351 //open file for storing results again
352 FILE* result5;
353 if ( (result5 = fopen(filename,"a+"))==NULL) //open file
354 {
355   fprintf(stderr, "Can't open/make file '%s'\n",filename);
356   return 1;
357 }
358
359
360 //if you want to measure the noise
361 if (opt_noise==1)
362 {
363   fprintf(result5,"\n[- noise]:\n");
364   int times=atoi(arg_times);
365   double averages[numberpd];
366   noisemes(result5,times,averages);
367 }
368 fclose(result5);
369
370
371 return 0;
372
373     }
```

Literaturverzeichnis

[1] Josep B. Altepeter, Daniel F.V. James, and Paul G. Kwiat, *Qubit Quantum State Tomography, Lect. Notes Phys. 649, 113-145* (Springer-Verlag Berlin Heidelberg 2004)

[2] Bronstein, Semendjajew, Musiol, Mühling, *Taschenbuch der Mathematik* (Wissenschaftlicher Verlag Harri Deutsch GmbH, 2008)

[3] Mark Fox, *Quantum Optics: an introduction* (Oxford Universitiy Press 2006)

[4] Martin Gräfe, *C und Linux* (2010 Carl Hanser Verlag München Wien)

[5] *GNU Scientific Libary - Reference Manual, Edition 1.15, April 2011* (http://www.gnu.org/software/gsl/manual/gsl-ref.ps.gz)

[6] E. Hecht, *Optik* (2005 Oldenbourg Wissenschaftsverlag GmbH)

[7] Daniel F. V. James, Paul G. Kwiat, William J. Munro, *Measurement of qubits* (Physical Review A, Volume 64, 052312, 2001)

[8] Nikolai Kiesel, *Experiments on Multiphoton Entanglement* (Dissertation an der Ludwig-Maximilians-Universität München, 2007)

[9] Manjit Kumar, *Quanten* (2009 Berlin Verlag GmbH)

[10] Leonard Mandel, Emil Wolf, *Optical coherence and quantum optics* (Cambridge University Press 1995)

[11] M. Nielsen, I.Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press 2010)

[12] J. Puls, S. Stintzing, überarbeitet durch M.Kerscher, *Numerik fur Physiker* (Vorlesungsskript an der Ludwig-Maximilians-Universität, http://www.mathematik.uni-muenchen.de/~kerscher/vorlesungen/numeriksose11/docscript.pdf)

[13] J. J. Sakurai, *Modern Quantum Mechanics* (1994 Addison-Wesley Publishing Company, Inc.)

[14] Benjamin Schumacher, *Quantum coding* (Physical review A, Volume 51 Number 4, 1995)

[15] Witlef Wieczorek, *Multi-Photon Entanglement* (Dissertation an der Ludwig-Maximilians-Universität München, 2009)

[16] W. Zinth, U. Zinth, *Optik* (2008 Oldenbourg Wissenschaftsverlag GmbH)

Erklärung

Mit der Abgabe dieser Bachelorarbeit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 6. Juli 2012

Luisa Hofmann