

# Flug- und Stabilisierungsalgorithmen eines Rotationsflüglers, sowie deren Umsetzung in Hard- und Software

Informatik Q1 2016

Fachlehrer: Herr Konnemann

Frederik Dunschen

8. Mai 2016

# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>3</b>
<b>2 Theoretisches Konzept</b>	<b>4</b>
2.1 Aufbau . . . . .	4
2.2 Theoretische Funktionsweise . . . . .	5
2.3 Warum Software für den Flug unabdingbar ist . . . . .	5
<b>3 Projektbeschreibung</b>	<b>6</b>
3.1 Dokumentation des Projektes . . . . .	6
3.2 Herausforderungen . . . . .	7
3.3 Verwendete Komponenten . . . . .	7
<b>4 Algorithmen</b>	<b>8</b>
4.1 Rotationsbestimmung im Raum . . . . .	8
4.2 Lagebestimmung im Raum . . . . .	10
4.3 Berechnungen zur Lageänderung . . . . .	11
4.3.1 Abgleich von IST und SOLL Werten . . . . .	11
4.3.2 Vektoren als Grundlage der Steuerung . . . . .	11
4.3.3 Ermittlung von Zeiten zur genauen Aussteuerung der Klappen . . . . .	12
4.4 Fazit . . . . .	13
<b>A</b>	<b>14</b>
A.1 Quellenverzeichnis . . . . .	14
A.2 Eigenständigkeits- und Plagiatsklärung . . . . .	14
A.3 Protokollbogen . . . . .	14
A.4 Fotos des ROFL-Copters . . . . .	14
<b>B Quellcode</b>	<b>16</b>

# Kapitel 1

## Einführung

In dieser Facharbeit möchte ich mich mit einem sog. "Rotationsflügler" beschäftigen. Der ROtationsFLügler (im folgenden "ROFL-Copter" genannt) ist ein von mir selbst entworfenes Fluggerät, das auf den Konzepten des "WowWee Bladestar" (Abb.: 1.1) und des "Lockheed Martin Samari UAV" (Abb.: 1.2) basiert. Diese Konzepte habe ich zusammengeführt, da sie sich gegenseitig in ihren Vor- und Nachteilen ergänzen. Der "WowWee Bladestar" soll sich zwar steuern lassen, die technische Umsetzung des Produktes lässt das allerdings nicht zu. Die Rotorblätter sind auf einen Winkel fest angestellt und nur die Motoren lassen sich in ihrer Umdrehungsgeschwindigkeit steuern. So lässt sich die Lage des Copters natürlich nicht verändern, sondern man kann nur hoch oder runter fliegen.

Der "Lockheed Martin Samari UAV" hat, anders als der Bladestar, eine aktive Lageregelung, ist aber asymmetrisch aufgebaut. Das kommt daher, dass sich die Entwickler von einem Ahornsamen haben inspirieren lassen. Diesen Aufbau haben sie dann auf ihren Copter übertragen. Doch durch den asymmetrischen Aufbau und die daraus entstandene Unwucht fliegt der Copter nicht ruhig.

Ich versuche nun die Vorteile beider Konzepte zu verbinden: Den symmetrischen Aufbau des "WowWee Bladestar" und die aktive Steuerung des "Lockheed Martin Samari UAV". Mit diesem Projekt möchte ich untersuchen, wie aufwändig es ist, eine eigene, neuartige Flugsteuerung zu entwickeln und umzusetzen. Die Rotationsflügler sind noch ziemlich unerforscht, es gibt nur einige wenige Forschungsarbeiten von Universitäten wie der ETH Zürich [3] oder Firmen wie Lockheed Martin [2]. Auch das hat mich besonders an dem Bau eines solchen Fluggerätes geizt, da nahezu alles selbst entwickelt werden muss. Eine weitere Herausforderung ist natürlich, die Software so zu implementieren, dass sie gleichzeitig Steuerbefehle und Sensoren einlesen, die nötigen Berechnungen vornehmen und die jew. Steueraktionen ausführen kann.



Abb. 1.1: WowWee Bladestar [1]



Abb. 1.2: Samari UAV [2]

# Kapitel 2

## Theoretisches Konzept

### 2.1 Aufbau

Der grundlegene Aufbau des ROFL-Copters zeigt sich schon an seineitm Namen. Es ist ein “Roatationsflügler”, das tragende bzw. auftriebsgebende Element ist also ein rotierender Flügel. Dieser sieht auf den ersten Blick zwar wie ein typischer Hubschrauberotor aus, er hat eine normale Rechteckform [4], grenzt sich aber durch seine Dimensionsverhältnisse von einem Hubschrauberrotor ab.

Ein Rotorblatt des ROFL Copters (Abb.: 2.1 [A]) ist Ø48mm tief, bei einer Länge von 190mm. Ein Rotorblatt eines Modellhubschraubers mit der Tiefe von 48mm hat schon eine Länge von 520mm [5]. Das Verhältnis beträgt in diesem Fall also  $\frac{190mm}{48mm} \approx \frac{19}{5}$  zu  $\frac{520mm}{48mm} \approx \frac{52}{5}$ . Diese Dimensionen wurden gewählt, um schon bei einer vergleichsweise niedrigen Drehzahl (1500 U/min.) ausreichend Auftrieb und Flugstabilität zu erreichen.

Um den Flügel in Rotation zu versetzen, hat der ROFL-Copter zwei kleine Motoren mit jew. einem 3-Blatt Propeller (Abb.: 2.1 [B]). Diese erzeugen unter Vollast einen Schub von ca. 2N. Die Motoren sind symetrisch an einem 21 cm langen Motorausleger (Abb.: 2.1 [C]) befestigt. Der Antriebsmoment, der auf den Hauptrotor wirkt, beträgt theoretisch also  $M = r * F * 2 = \frac{0.21}{2} * 4 = 0.42\text{Nm}$

Der ROFL-Copter erzeugt seinen Auftrieb Pitchgesteuert. Das bedeutet, dass die Motordrehzahl während des gesammten Fluges konstant bleibt und nur über die Änderung des Anstellwinkels beider Rotorblätter ein Auf- oder Abtrieb erzeugt wird. Zum Ändern des Anstellwinkels verfügt der ROFL-Copter über zwei Servomotoren (kurz: Servos) (Abb.: 2.1 [D]). Über diese sind die Rotorblätter mit der Steuerelektronik und dem Motorausleger verbunden. Die Servos bekommen vom Prozessor eine Gradzahl  $x$  zwischen  $0^\circ$  und  $180^\circ$  übermittelt, die sie dann einstellen und halten. Genauere Angaben darüber wie der Prozessor die Servos ansteuert, finden sich in Anlage X.

Das wohl wichtigste Element für die Steuerung sind die Steuerklappen an den Enden der Rotorblätter (Abb.: 2.1 [E]). Diese sind über einen dünnen Draht mit kleinen Elektromagneten, sog. Solenoids, verbunden. Damit können die Klappen innerhalb von sehr kurzer Zeit (ca. 3ms) angestellt werden und so dem jew. Rotorblatt für kurze Zeit einen höheren Auftrieb geben. Das dient dazu, den Copter kontrolliert ein kleines Stück in eine gewünschte Richtung kippen zu

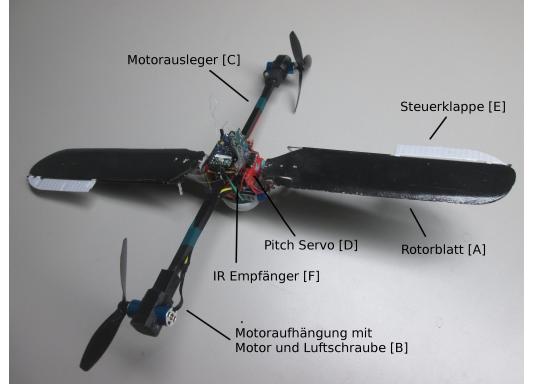


Abb. 2.1: grundlegender Aufbau

lassen und ihn so stabilisieren oder lenken zu können

Während des Fluges muss der Prozessor feststellen können, wie der ROFL-Copter relativ zu der steuernden Person ausgerichtet ist, um sich z.B. bei Steuerbefehlen wie “nach vorne fliegen” von der Person weg, also nach vorne, zu bewegen. Das geschieht über einen Infrarot Empfänger, der an einer Seite des ROFL-Copters angebracht ist und über eine Abschirmung nur ein sehr eingeschränktes Sichtfeld hat (Abb.: 2.1 [F]; Die Abschirmung ist im Bild noch nicht angebracht). Dieser Empfänger versucht nun das Signal des Infrarotsenders zu orten, welcher an der Fernsteuerung angebracht ist. Hat der Empfänger das Signal gefunden muss der Copter gerade so ausgerichtet sein, dass der IR-Empfänger in Richtung Fernsteuerung zeigt. Von dieser Positionsreferenz aus kann man nun alle weiteren Berechnungen ausführen.

Das Herzstück des Copters ist die Sensor- und Steuerplatine in der Mitte. Auf ihr befindet sich ein Microcontroller, der alle notwendigen Berechnungen vornimmt und die Steuerbefehle ausführt. Des Weiteren misst ein Initialsystem, also eine Kombination aus Gyroskop und Beschleunigungssensor, die Lage des Copters. Ein handelsüblicher RC Empfänger übermittelt die Steuerbefehle der Fernsteuerung an den Microcontroller und dieser kann seinerseits Daten über eine 2.4GHz serielle Schnittstelle an einen PC senden.

## 2.2 Theoretische Funktionsweise

Während des Fluges rotiert der Copter, angetrieben von den beiden Motoren, um sich selbst. Die Drehzahl der Motoren bleibt dabei immer konstant. Die Rotorblätter werden über die Servos kontinuierlich auf den von dem Microcontroller vorgegebenen Winkel eingestellt.

Um einen Referenzpunkt im Raum zu haben, an dem der Copter sich ausrichten und orientieren kann, habe ich an der Fernsteuerung eine Infrarot LED befestigt. Wenn der Infrarot-Empfänger am Copter nun das Signal des Infrarot-Senders empfängt, liest der Microcontroller das Initialsystem aus. Dadurch ist dem ROFL-Copter bekannt, wie dieser im Raum ausgerichtet ist (mit dem IR Empfänger in Richtung des Piloten) und somit auch, wie das Koordinatensystem des Initialsystems liegt. Die aktuellen Lagewerte werden nun mit den Steuerbefehlen des Piloten abgeglichen. Daraus werden die Ansteuerzeiten der Steuerklappen berechnet. Die Ansteuerzeiten definieren, wann, also wo im Raum, welche Klappe wie lange angestellt werden muss. Der Microkontroller wartet nun also  $x$  Millisekunden, um die Klappe dann  $y$  Millisekunden lang anzustellen. So hat ein Rotorblatt für kurze Zeit kontrolliert mehr Auftrieb als das andere und der Copter kippt leicht in die gewünschte Richtung.

## 2.3 Warum Software für den Flug unabdingbar ist

Auf Grund der hohen Drehgeschwindigkeiten und der dadurch erforderlichen, kurzen Reaktionszeiten ist eine lageregulierende Software für einen stabilen Flug unabdingbar. Dabei wird die Flugweise vor allem durch zwei Faktoren erschwert. Einerseits fliegt der Copter nicht eigenständig, das bedeutet, dass er nicht ohne eine aktive Steuerung geradeaus fliegt. Man kann das gut mit dem Balancieren eines Stiftes auf der Hand vergleichen. Es müssen andauernd kleine Korrekturen der Lage vorgenommen werden, um den Copter nicht aus der gewünschten Flugbahn ausbrechen zu lassen. Andererseits müssen alle Berechnungen und Steueraktionen bei einer Umdrehungszeit von  $\varnothing 40\text{ms}$  schnell ausgeführt werden. Das ganze System ist damit für einen menschlichen Piloten zu zeitkritisch. Eine Software übersetzt deshalb die Steuerbefehle in Steueraktionen. Noch dazu kann ein Pilot die Ausrichtung des Copters während des Fluges nicht einschätzen. Auch hierfür ist eine Software, die die Lage des Copters im Raum schnell genug auswerten und die Ausrichtung feststellen kann, unabdingbar.

# Kapitel 3

## Projektbeschreibung

### 3.1 Dokumentation des Projektes

Die Idee für den ROFL-Copter hatte ich bereits im September 2015. Ich habe die ersten Entwürfe gemacht und angefangen, Bauteile zu fertigen oder zu bestellen.

Im Oktober 2015 habe ich die Rotorblätter entworfen, und hergestellt. Diese habe ich dann an ein Grundgerüst, bestehend aus dem Motorausleger mit Motoren, Motorreglern und Pitch-Servos, angebracht. So war der Grundaufbau bereits fertig. Gegen Ende Oktober habe ich mit der Steuerelektronik angefangen und die Software zur Kommunikation mit einem Computer und die Ansteuerung der Servos und Motoren geschrieben. Zu diesem Zeitpunkt hatte ich noch keine Steuerklappen vorgesehen, sondern wollte die Servos zur Steuerung nehmen. Es hat sich jedoch herausgestellt, dass diese zu träge waren. Deshalb habe ich nach einer anderen Lösung gesucht und bin so auf die Elektromagneten gekommen.

Im November habe ich mich dann mit der Lageerkennung über ein Initialsystem beschäftigt. Neben diesem habe ich noch einen Kompasssensor auf die Hauptplatine gebaut, um die Ausrichtung des Copters bestimmen zu können. Ich musste jedoch feststellen, dass dieser Sensor zu langsam für eine schnelle und genaue Messung während der Rotation war und ihn wieder abbauen.

Die Algorithmen für die Lageerkennung habe ich im Dezember mit eigenen Filtern verbessert und weniger anfällig für kleine Erschütterungen gemacht. Ich habe auch das Einlesen des RC Empfängers so implementiert, dass ich nicht mehr auf die dafür zusätzliche Programmbibliothek angewiesen war, sondern die Steuerbefehle mit Interrupts einlesen konnte.

Im Januar 2016 habe ich die Steuerklappen und Elektromagneten an den Rotorblättern installiert und die Schaltung zur Ansteuerung der Magneten entworfen. Diese Schaltung habe ich dann im Februar getestet, verbessert und auf dem Copter installiert.

Im März habe ich die grundlegenden mathematischen Algorithmen zu Stabilisierung und La-geänderung entworfen und mit dem Bau des Infrarot Sendemodules begonnen.

Die Infrarot-Kommunikation konnte ich im April fertigstellen und anfangen, alle einzelnen Module zu testen und zu verbessern. Die Steuerung der Elektromagneten habe ich komplett neu gemacht und konnte so die Reaktionszeit der Steuerklappen erheblich beschleunigen. Mit der neuen Schaltung sind nun 100 Klappenausschläge pro Sekunde möglich. Gegen Ende April habe ich mit den ersten Tests des gesamten Systems angefangen und konnte dabei Umdrehungsgeschwindigkeiten und Lagemessungen prüfen. Mir ist dabei aufgefallen, dass zwar alles funktioniert, ich die Algorithmen und Filter für das Initialsystem aber noch verbessern muss. Da mir bei einem Test Anfang Mai eine Rotorbefestigung gebrochen und der Microcontroller durchgebrannt ist, konnte ich bis Mittwoch, den 04.05.'16, keine weiteren Tests durchführen. Zwar haben nach dem Ersetzen beider Teile wieder alle Funktionen funktioniert, ich habe aber

noch nicht weiter an der Flugsteuerung arbeiten können. Trotzdem bin ich zuversichtlich, dass ich den Copter innerhalb der nächsten Wochen zum Fliegen bringen kann.

## 3.2 Herausforderungen

Bei diesem Projekt gab es mehrere, teils sehr große Herausforderungen. Die gesammte Steuerung muss sehr geschwindigkeitsoptimiert, aber dennoch flexibel sein. Ich musste auch einige Software Bibliotheken umschreiben oder komplett umgehen, da sie zu langsam waren. So habe ich für das Einlesen der Steuerbefehle z.B. nicht die fertige pulseIn(pCHANNEL); Funktion nutzen können, da diese Funktion die Ausführung anderer Befehle für bis 20ms verhindert. Da die Funktion für alle fünf Kanäle nacheinander aufgerufen werden müsste, hat sie so schon die Rechenzeit für ca. zwei Umdrehungen blockiert. Ich habe letztendlich eine eigene Funktion geschrieben, die mit Interrupts arbeitet und für die Ausführung nur wenige Mikrosekunden braucht.

Auch hardwaretechnisch brachte das andauernde Rotieren einige Schwierigkeiten mit sich. Für die Steuerklappen musste ich eine Lösung finden, wie sie möglichst schnell mit möglichst großer Kraft gesteuert werden können. Noch dazu muss die Konstruktion dafür sehr leicht sein. Servomotoren, wie sie beim "Lockheed Martin Samari UAV" verwendet wurden [2], bieten sich für mich nicht an, da sie in der benötigten Größe nicht genügend Kraft haben und sich zu langsam bewegen (übertragungsbedingt brauchen sie mindestens 20ms [6]). Bei dem Samari UAV war das anscheinend kein Problem, da sich dieser durch den großen Flügel recht langsam dreht und dort Platz für einen großen, ausreichend starken Servo ist. In meinem Konzept habe ich das Problem mit elektromagnetischen Aktuatoren, sog. Solenoids, gelöst. Diese kann ich über eine kleine Schaltung mit dem Microcontroller kontrolliert auf die Akkuspannung schalten und die Klappen so sehr schnell (innerhalb von 3ms) voll anstellen.

Mitunter die größte technische Herausforderung war es wohl, alle benötigten Bauteile sowohl zentral, als auch sehr komprimiert anzubringen. Dabei sollte der ROFL-Copter aber weiterhin einfach zu warten sein. Ich habe deshalb die Sensor- und Steuerelektronik auf ein eigenes Modul zum Aufstecken auf den Copter gelötet. So kann ich schnell an der Elektronik etwas ändern, ohne die großen Teile des Copters, wie z.B. die Rotoren oder Motorausleger im Weg zu haben und andersherum sind auch diese Teile einfacher zu erreichen.

## 3.3 Verwendete Komponenten

Hier möchte ich kurz die Komponenten und Bauteile aufführen, die ich im ROFL-Copter verwendet habe und erläutern, welches Bauteil welche Aufgabe hat und wie diese Zusammenwirken. Die zentrale Steuerung übernimmt ein ARM Cortex-M4 Microcontroller auf einem Teensy 3.1 Board. Dieser ist an einen MPU6050 angeschlossen, um die Lage des Copters messen zu können. Als Infrarotempfänger habe ich einen HS0038b IC verwendet, der den Microcontroller durch eingebaute Filter entlastet. Als Infrarot-Sendemodul habe ich ein sog. JR Modul gebaut, dass sich direkt an der Fernsteuerung befestigen lässt. Die IR LED zum Senden hat eine Leistung von 2W. Zur Kommunikation mit einem Computer während des Fluges, z.B. um Debug Werte zu senden, verfügt der Copter über ein TB387 wireless Modul. Die Elektromagneten für die Steuerklappen sind über eine eigene, kleine Schaltung mit dem Microcontroller verbunden und können auf die Akkuspannung geschaltet werden. Die beiden bürstenlosen Motoren, die den Copter in Rotation versetzen, sind an jew. einem 10A Motorregler angeschlossen. Für die Stromversorgung verwende ich einen zweizelligen Lipo Akku mit 500mAh.

# Kapitel 4

## Algorithmen

Alle in diesem Kapitel verwendeten Programmcodeabschnitte finden sich im Zusammenhang des gesammten Programmes unter Anlage B. Programmteile, die nicht von mir stammen, sind explizit als solche aufgeführt.

### 4.1 Rotationsbestimmung im Raum

Für die Steuerung des ROFL-Copters ist es unerlässlich, dass seine Ausrichtung zum Piloten bekannt ist. Gibt dieser nämlich den Steuerbefehl "nach vorne fliegen", so muss sich der Copter vom Piloten weg bewegen. Außerdem muss die Umdrehungszeit und -geschwindigkeit bekannt sein, um mit diesen Werten die Ansteuerzeiten für die Klappen berechnen zu können.

Ich habe diese Probleme mit einem Infrarotempfänger gelöst, der nach dem Infrarotsender sucht. Der Sender ist, wie bereits in 2.2 beschrieben, an der Fernsteuerung befestigt und der Empfänger an dem Microcontroller auf dem Copter angeschlossen. Um Übertragungsfehler zu vermeiden, nimmt der Empfänger generell nur Infrarotsignale an, die mit 38kHz gepulst wurden (so stört u.a. die Sonneneinstrahlung nicht). Wenn er jetzt also ein solches Infrarotsignal empfängt, gibt der Empfänger an den Microcontroller eine logische 1 zurück, ansonsten eine logische 0.

Da der Empfänger, der für Datenübertragung über IR gedacht ist, keine anhaltenden Signale bzw. Zustände weitergeben kann, sendet der Sender nicht einfach den Dauerzustand 1, sondern wechselt alle 400us zwischen 0 und 1.

Die Software muss die eingehenden Signale nun also so interpretieren, dass der andauernde Wechsel zwischen 0 und 1 innerhalb von 400us einer logischen 1 und das Anliegen einer 0 für mindestens 800us einer logischen 0 entspricht. Ich möchte das an folgender Grafik verdeutlichen, in der die jew. Ausgaben der einzelnen Teile dargestellt sind:

Um aus den Pulsen des IR-Empfängers eine logische 1 oder 0 zu lesen, habe ich am Anfang des Algorithmus die Variable "recieveTollerance" deklariert. Diese definiert, wie viele Mikrosekunden lang das eingehende Signal mindestens 0 sein muss, um als logische 0 gewertet zu werden (das X in Abb. 4.1)

```
1 int recieveTollerance = 5000; //Tolleranzzeit in us
2 volatile unsigned int irTimer = 0;
3 volatile boolean on = false;
```

Jedesmal, wenn das Eingangssignal von 0 auf 1 wechselt (rote Pfeile in Abb. 4.1), wird diese Funktion über einen Interrupt aufgerufen:

```
1 void irFALLING () { //active-Low --> fallende Flanke = Wechsel von 0 auf 1
2     irTimer = micros() + recieveTollerance; //den Timer um x erhöhen
3     on=true;
4 }
```

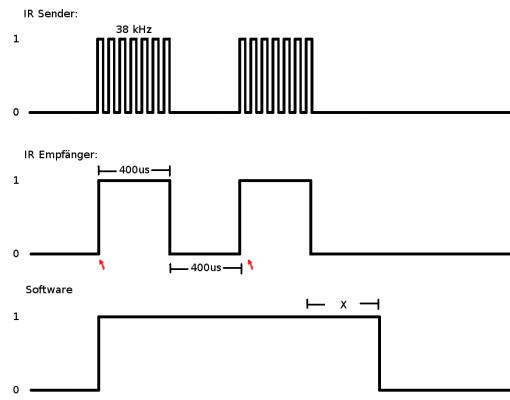


Abb. 4.1: IR Signalablauf

In der Hauptschleife wird ersichtlich, wozu die beiden Variablen gut sind.

```

1 void loop() {
2     //Wenn die Zeit größer als der gewünschte Zeitstempel ist...
3     if((micros() >= irTimer) && (irTimer != 0)) {
4         on=false; //...werden keine Pulse mehr empfangen
5         irTimer = 0;
6     }
7
8     if(on) { //Wenn eine Sichtverbindung besteht...
9         //tu irgend etwas
10    }
11    else {
12        //sonst tu etwas anderes
13    }
14 }
```

Die Variable `irTimer` wird von der Interupt-Funktion, sofern Signale empfangen werden, schneller wieder erhöht, als sie in der Hauptschleife ablaufen kann. So bleibt die Variable “on” solange auf “true”, bis keine Signale mehr empfangen werden, die Variable “`irTimer`” nicht mehr erhöht wird und schließlich kleiner ist, als die aktuelle Zeit. So ist klar, dass sich der Copter aus der Sichtverbindung zwischen IR-Sender und IR-Empfänger herausgedreht hat.

Ändert man die Funktion `irFALLING()` nun etwas ab, kann man über die Differenz der aktuellen Zeit und der Zeit, als das letzte Mal etwas empfangen wurde, die Umdrehungsgeschwindigkeit in U/min errechnen:

```

1 void irFALLING () { //active-Low --> fallende Flanke = Wechsel von 0 auf 1
2     if(!on) { //wenn on noch nicht true --> erster Wechsel von 0 auf 1
3         diffTime = micros()-lastTurnTimestamp; //Umdrehungszeit
4         lastTurnTimestamp = micros();
5         rotSpeed=(1000000/diffTime)*60;
6     }
7     irTimer = micros()+recieveTollerance; //den Timer um x erhöhen
8     on=true;
9 }
```

Die Rechnung in Z. 5 setzt sich aus den Umdrehungen pro Sekunde (1000000us geteilt durch die Zeit für eine Umdrehung in us) mal 60, zusammen.

## 4.2 Lagebestimmung im Raum

Die Lage des ROFL-Copters im Raum, also die Drehung um die X und Y Achse, wird mithilfe eines sog. Initialsystems bestimmt. Initialsysteme nennt man eine Kombination aus Gyroskop und Beschleunigungssensor, die zusammen ausgewertet werden. Beide Sensoren können zwar schon eigenständig die Drehung um die Achsen bestimmen, haben dabei jedoch einige Nachteile. Ein Gyroskopischer Sensor ist ziemlich genau, driftet nach einiger Zeit aber ab. Das bedeutet, dass er im Betrieb Werte zurückgibt, die immer stärker neben den eigentlichen Lagewerten liegen. Beschleunigungssensoren zeigen diesen Effekt nicht, sind aber sehr empfindlich und schwingen oft über oder unter dem eigentlichen Lagewert. [7]

Rechnet man die Rückgabewerte beider Sensoren mithilfe eines sog. "Kalman Filters" zusammen, bekommt man einen sehr genauen Wert für die aktuelle Lage zurück. Der Kalman Filter arbeitet grundsätzlich in drei Schritten: Zuerst wird der Wert des Beschleunigungssensors eingelesen und dann sowohl mit dem Wert des Gyroskops, als auch mit dem zuletzt berechneten Winkel verglichen und verrechnet. Dieser Wert wird dann zurückgegeben. [7]

Ich habe für die Lagebestimmung auf die Arbeit von Kristian Lauszus zurückgegriffen, der den Kalman Filter und die Ansteuerung des Sensors bereits fertig implementiert hat [8]. Ein Nachteil der Echtzeitmessungen des Sensors ist jedoch, dass die Werte für mich zu genau sind. Die Steuerung der Klappen würde sich so nur an einem einzigen Wert einer einzigen Umdrehung orientieren. Dieser Wert wäre an sich zwar sehr genau, doch im Zusammenhang nicht geeignet, um die Klappen nach einer kontinuierlich messbaren Bewegung zu stellen. Für den Steueralgorithmus brauche ich deshalb immer die durchschnittliche Lage der letzten  $x$  Umdrehungen. Diese berechne ich über ein sog. Mittlaufendes Mittel

Zuerst schreibe ich den aktuellen Lagewert in einen Array an die Stelle, die durch die Variable "kalBufferCounter" definiert wird:

```
1 kalBufferX[kalBufferCounter] = round(kalAngleX);
```

Nun habe ich einen Array, in dem alle Lagewerte der letzten  $x$  Umdrehungen stehen. Diese werden jetzt alle miteinander addiert:

```
1 kalAngleXBuffered = 0;
2
3 for(i=0; i<kalAngleBufferSize; i++) {
4     kalAngleXBuffered = kalAngleXBuffered+kalBufferX[i];
5 }
```

Um den Durchschnitt der Werte zu bekommen, teile ich die Variable "kalAngleXBuffered" durch die Puffergröße, also die Anzahl der Werte, die im Array gespeichert sind:

```
1 kalAngleXBuffered = kalAngleXBuffered/kalAngleBufferSize;
```

Dasselbe wiederhole ich mit allen Y-Werten. Zum Schluss wird die Variable "kalBufferCounter" um 1 erhöht, damit beim nächsten Durchlauf der dann aktuelle Lagewert den ältesten Eintrag überschreibt. Sollte die Variable größer als die Größe des Arrays geworden sein, wird sie zurückgesetzt:

```
1 kalBufferCounter++;
2 if(kalBufferCounter > kalAngleBufferSize-1) kalBufferCounter=0;
```

In der Praxis hat sich eine Puffergröße von sechs bewährt.

## 4.3 Berrechnungen zur Lageänderung

### 4.3.1 Abgleich von IST und SOLL Werten

Durch die Auswertung des Initialsystems ist nun bekannt, wie der Copter im Raum liegt. Es gibt also einen aktuellen IST Wert. Der Pilot des ROFL-Copters sendet über die Fernsteuerung die Steuerbefehle. Sie definieren den SOLL Wert. Die Aufgaben der Flugsteuerung ist nun, einen Weg von IST zu SOLL zu finden und den Copter dementsprechend zu steuern.

Im Fall des ROFL-Copters beinhaltet sowohl der IST, als auch der SOLL Wert die Information über die Drehung bzw. Rotation auf der X als auch auf der Y Achse. Die Werte kann man als Punkte in ein Koordinatensystem eintragen. Einen IST-Punkt und einen SOLL-Punkt mit den dazugehörigen X|Y Koordinaten. Im Beispiel sind die Werte  $IST = (2 | 1)$  &  $SOLL = (-1 | 2)$  (Abb.: 4.2).

Jetzt zieht man die SOLL Werte von den IST Werten ab, man verschiebt die Punkte auf dem Koordinatensystem also so, dass der SOLL-Punkt im Ursprung liegt (Abb.: 4.3).

Im Programm sehen diese Schritte so aus:

```

1 //IST Zustand einlesen
2 getCopterAngle();
3
4 //SOLL Zustand einlesen
5 SOLL[0] = ch1Data;
6 SOLL[1] = ch2Data;
7
8 diffVektor[0] = IST[0]-SOLL[0]; //X-Wert
9 diffVektor[1] = IST[1]-SOLL[1]; //Y-Wert

```

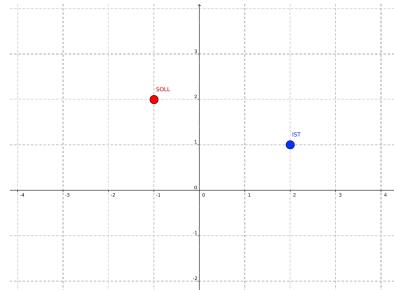


Abb. 4.2: IST und SOLL Punkt

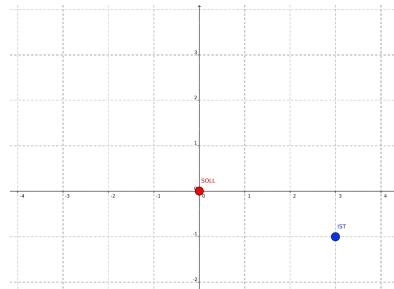


Abb. 4.3: SOLL in Ursprung verschoben

### 4.3.2 Vektoren als Grundlage der Steuerung

Im Programmcode unter 4.3.1 ist schon ersichtlich, auf welcher mathematischen Grundlage der Algorithmus basiert: Vektoren. Aber warum muss der SOLL Punkt in den Ursprung verschoben werden? Wenn man den SOLL-Punkt als Kreismittelpunkt ansieht, der Kreis außen durch den IST-Punkt verläuft und man diesen Kreis "Rotorkreisfläche" nennt, wird alles schon etwas klarer. Die Rotorkreisfläche ist der Bereich, in dem sich der Rotor des ROFL-Copters dreht.

Zeichnet man nun einen Vektor vom IST-Punkt zum SOLL-Punkt, erkennt man bereits einen Teil der Steuerung. Der IST-Punkt ist der Punkt, an dem eine Steuerklappe angestellt sein muss, damit der Copter in Richtung des Vektors kippt.

Um Herauszufinden, wo genau die Klappe in der Rotorkreisfläche angestellt werden muss, braucht man den Winkel zwischen der aktuellen Ausrichtung des Copters (mit dem IR Empfänger in Richtung des Piloten) und dem Richtungsvektor. Die aktuelle Ausrichtung des Copters ist ein weiterer Vektor. Er liegt auf der X-Achse, genauso wie die Rotorblätter, wenn der IR Empfänger eine Sichtverbindung zum IR Sender hat.

Bevor man den Winkel zwischen den Beiden Vektoren errechnen kann, muss man feststellen, in welchem Quadranten des Koordinatensystems der IST-Punkt liegt.

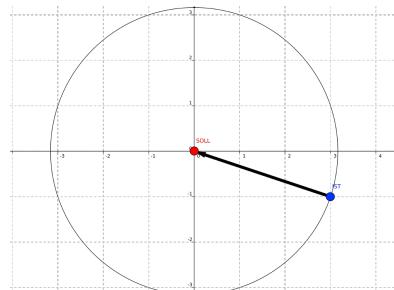


Abb. 4.4: Rotorkreisfläche und Richtungsvektor

So kann man beurteilen, ob es sich um einen stumpfen oder spitzen Winkel zur nächsten Steuerklappe handelt und dementsprechend rechnen. In Abb. 4.5 liegt der Punkt im 4. Quadranten und damit in einem spitzen Winkel zu Rotor 1. Jetzt kann man den Betrag des Skalarproduktes beider Vektoren durch die Länge des Richtungsvektors teilen. Das Ergebnis setzt man in den  $\cos^{-1}$  ein und bekommt als Ergebnis den Winkel  $\alpha$  von der aktuellen Position bis zum Richtungsvektor.

Als nächstes berechnet man den Winkel  $\beta$ , in dem die Klappe angestellt sein soll. Dieser Winkel definiert, wie stark der Copter kippt (größerer Anstellwinkel  $\rightarrow$  längere Anstellzeit der Klappe  $\rightarrow$  mehr Auftrieb für das jew. Rotorblatt  $\rightarrow$  stärkeres Kippen).

Da der Copter um so stärker aussteuern soll, wenn die Differenz zwischen IST und SOLL Wert größer ist, kann man den Winkel einfach proportional zu dieser Differenz berechnen. Die Differenz und damit der Winkel  $\beta$  ist die Länge des Richtungsvektors  $x$ .

Außerdem braucht man noch den Winkel, der zwischen der aktuellen Position und dem Anstellen der Klappe liegt. Dazu zieht man die Hälfte des Anstellwinkels  $\beta$  von dem Winkel zum Richtungsvektor  $\alpha$  ab. Man erhält den Winkel  $\gamma$ .

Letztendlich braucht man nur den Winkel  $\gamma$  zwischen der aktuellen Position und dem Anstellen der Klappen und den Anstellwinkel der Klappe  $\beta$ .

Im Programmcode sind diese Schritte so gelöst:

```

1 skalarprodukt = diffVektor[0];
2 diffVektorLaenge = sqrt(diffVektor[0]*diffVektor[0]+diffVektor[1]*diffVektor[1]);
3
4 //Berrechnungen für die einzelnen Quadranten
5 // [...]
6 else if(diffVektor[0] > 0) {
7 //4. Quadrant --> spitzer Winkel für R1
8 cosP = cosPSpitz(skalarprodukt, diffVektorLaenge);
9 }
10 // [...]
11
12 //Winkel berechnen
13 winkelBisZumVektor = round(acos(cosP));
14 anstellWinkel = diffVektorLaenge*bx; //bx ist variabler Faktor
15 winkelBisZumAnstellen = winkelBisZumVektor-(1/2*(anstellWinkel));
```

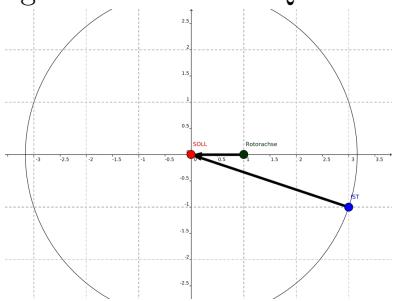


Abb. 4.5: Vektor der Rotorachse

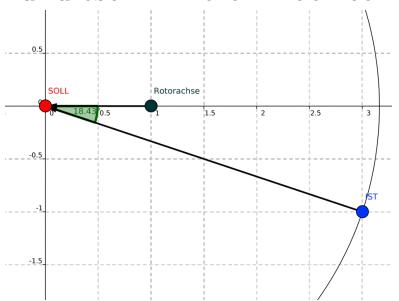


Abb. 4.6: Schnittwinkel  $\alpha$

### 4.3.3 Ermittlung von Zeiten zur genauen Aussteuerung der Klappen

Mit den errechneten Winkeln  $\beta$  &  $\gamma$  aus 4.3.2 und dem IR Empfänger kann man jetzt sehr schnell die Ansteuer- und Wartezeiten für die Klappensteuerung herausfinden. Die erste Zeit, die für die Steuerung benötigt wird, ist die Zeit, die der Microcontroller warten muss, bis er eine Steuerklappe anstellen soll. Diese Zeit setzt sich aus der letzten Umdrehungszeit (Die Variable "diffTime") geteilt durch den Anteil des Winkels  $\gamma$  am Vollkreis, zusammen. Ein Beispiel: Die letzte Umdrehung hat  $40\text{ms} = 40000\text{us}$  gedauert. Der errechnete Winkel  $\gamma$  beträgt  $60^\circ$ . Dann beträgt die Zeit, die der Microcontroller warten muss, bis er die Steuerklappe anstellt  $\frac{40000}{360/60} \approx 6666\text{us}$ .

Genauso kann man nun mit dem Anstellwinkel  $\beta$  verfahren. Jetzt kann man die Timer stellen und so im richtigen Moment die Steuerklappen präzise ansteuern und den gewünschten Kipp-effekt erhalten. Im Programm sieht dieser letzte Schritt so aus:

```

1 zeitBisZumAnstellen = diffTime/(360*winkelBisZumAnstellen);
2 anstellZeit = diffTime/(360*anstellWinkel);
3
4 klappenTimer = zeitBisZumAnstellen;

```

## 4.4 Fazit

Was lässt sich nun rückblickend über dieses Projekt sagen? Einerseits gab es Abschnitte bzw. Teile des Projektes deren Umsetzung ich anfangs schwieriger eingeschätzt habe, als sie waren. Andererseits gab es auch Abschnitte, die deutlich komplizierter waren, als ich ursprünglich gedacht habe.

Erstaunt hat mich, wie einfach der Algorithmus zur Berechnung der Steuerzeiten für die Klappen letztendlich war. Diesen Algorithmus jedoch so in das Programm einzubetten, dass er schnell genug lief, war eine Herausforderung. Da neben diesem Kernalgorithmus noch die Steuerungen für die Servos und die Motoren laufen und andauernd die Steuerbefehle der Fernsteuerung eingelesen werden mussten, ist es wichtig, dass die Flugsteuerung Modular, also in vielen kleinen, einzelnd aufrufbaren Programmfunctionen, implementiert ist. Auch diesen Aspekt habe ich zunächst etwas unterschätzt, konnte ihn letztendlich aber gut umsetzen.

Einen Rotationsflügler habe ich anfangs für dieses Projekt gewählt, da dieser Bereich ziemlich unerforscht ist. Während des Projektes ist mir klar geworden, warum das so ist: Die Idee, mit einem Propeller einen Rotor anzutreiben, klingt zwar einleuchtend, ist aber alles andere als effizient. Schon ein einzelner Motor des ROFL-Copters könnte diesen, bei einer Kraft von 2N, problemlos hochheben und in der Luft halten. Da mir das Entwerfen und Bauen des Modells aber viel Spaß gemacht hat, ist mir das nicht weiter wichtig.

Zusammenfassend möchte ich festhalten, dass die Entwicklung einer eigenen, neuartigen Flugsteuerung zwar sehr aufwändig, aber nicht unbedingt kompliziert ist. Es müssen viele verschiedene Aspekte berücksichtigt und die Software genau auf die Hardware abgestimmt werden, doch letztendlich ist auch die Flugsteuerung nur der Übersetzer zwischen Pilot und Fluggerät.

# Anhang A

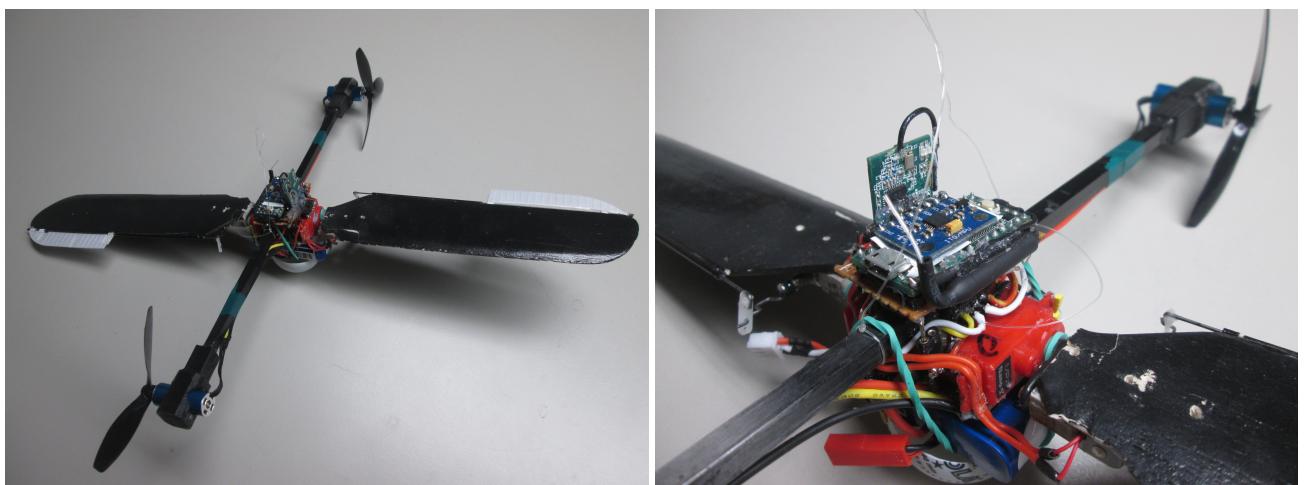
## A.1 Quellenverzeichnis

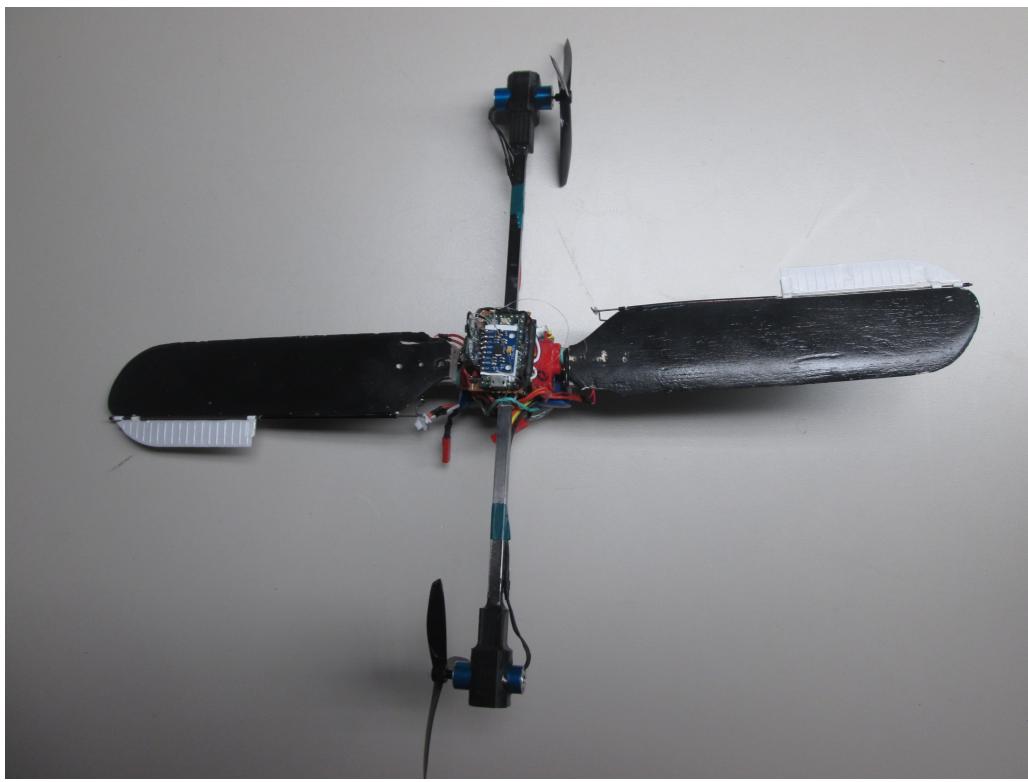
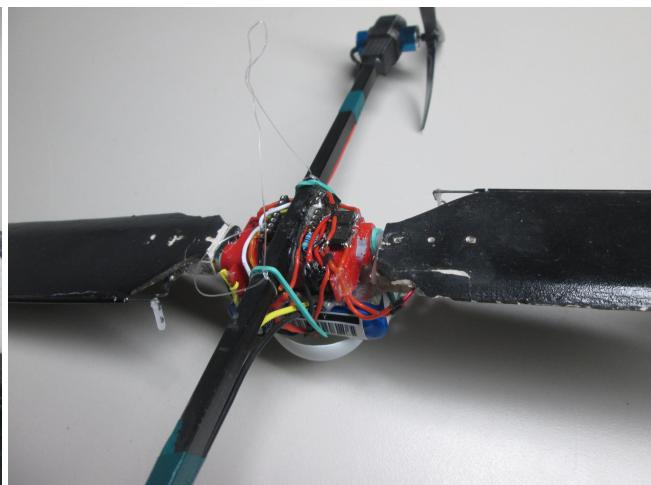
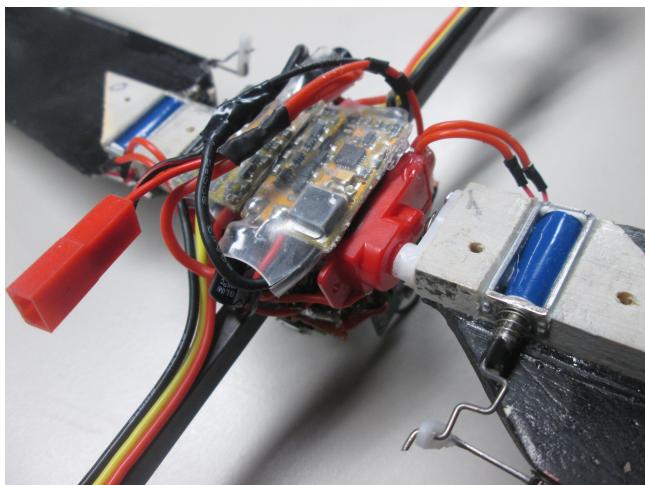
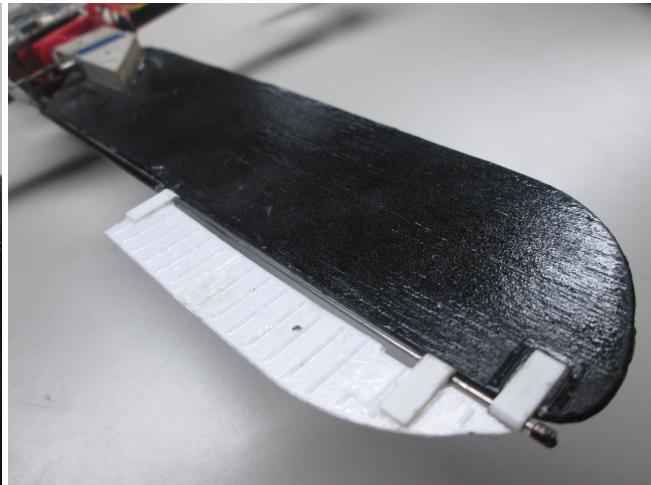
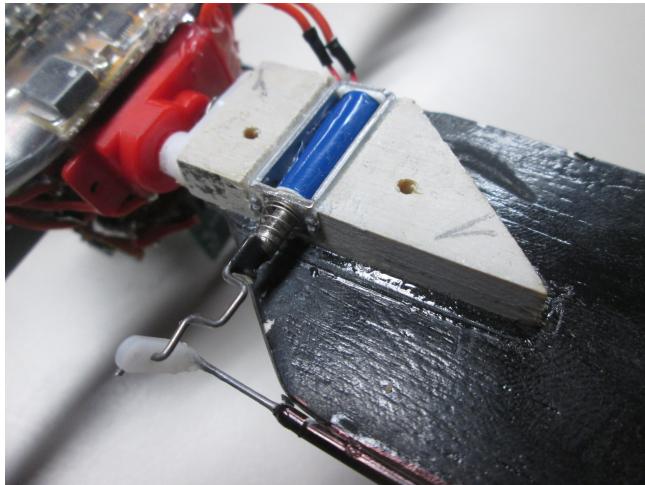
- [1] Bildquelle: <http://www.engadget.com/2008/09/15/wowwees-flytech-bladestar-can-govern-your-home-autonomously/> (eingesehen am 07.05.2016)
- [2] Bild extrahiert aus: <https://www.youtube.com/watch?v=jtU1ZsjAqsE> (eingesehen am 07.05.2016)
- [3] <http://robohub.org/the-monospinner-worlds-mechanically-simplest-controllable-flying-machine/> (eingesehen am 06.05.2016)
- [4] <https://de.wikipedia.org/wiki/Hauptrotor> (eingesehen am 06.05.2016)
- [5] [http://www.hobbyking.com/hobbyking/store/\\_/63040\\_520mm\\_RotorStar\\_Premium\\_3K\\_Carbon\\_Fiber](http://www.hobbyking.com/hobbyking/store/_/63040_520mm_RotorStar_Premium_3K_Carbon_Fiber) (eingesehen am 07.05.2016)
- [6] [http://www.rchelikiwiki.com/Servo\\_protocol](http://www.rchelikiwiki.com/Servo_protocol) (eingesehen am 24.04.2016)
- [7] <http://blog.tkjelectronics.dk/2011/06/guide-gyro-and-accelerometer-kalman-filtering-with-the-arduino/> (eingesehen am 07.05.2016)
- [8] <https://github.com/TKJElectronics/KalmanFilter/tree/master/examples/MPU6050> (eingesehen am 19.12.2015)

## A.2 Eigenständigkeits- und Plagiatserklärung

## A.3 Protokollbogen

## A.4 Fotos des ROFL-Copters





# Anhang B

## Quellcode

Dies ist die komplette Firmware für den ROFL-Copter; Stand 8.05.2016

```
1 #include <i2c_t3.h>
2 #include <math.h>
3 #include <Servo.h>
4 #include "Kalman.h" // Source: github.com/TKJElectronics/KalmanFilter
5
6 #define RESTRICT_PITCH // Comment out to restrict roll to 90deg instead
7
8
9 Kalman kalmanX, kalmanY; // Create the Kalman instances
10
11 //Variablen für die Lageerkennung
12 const uint8_t MPU6050 = 0x68; // If ADO is logic low on the PCB --> 0x68
13
14 //Variablen für die Lageerkennung
15 int accX, accY, accZ;
16 int gyroX, gyroY;
17 int16_t tempRaw;
18
19 int roll, pitch;
20
21 double gyroXangle, gyroYangle; // Angle calculate using the gyro only
22 double kalAngleX, kalAngleY; // Calculated angle using a Kalman filter
23
24 uint32_t IMUTimer;
25 uint8_t i2cData[14]; // Buffer for I2C data
26
27 int IST[2];
28
29 //Pinbelegungen
30 int IR = 22;
31 int LED = 13;
32
33 //Definieren der Empfängerpins
34 int ch1 = 14;
35 int ch2 = 15;
```

```
1 int ch3 = 16;
2 int ch4 = 17;
3 //int ch5 = 20;
4
5
6 //Klappen
7 const int klappe0 = 23;
8 const int klappe1 = 21;
9
10 //rotating time
11 long lastTurnTimestamp = 0.00;
12 double diffTime;
13 double rotSpeed;
14
15 //klappen Zeiten
16 boolean rotor0down = false;
17 boolean rotor1down = false;
18
19 unsigned long klappenTimer = 0;
20 boolean rotor0first = false;
21
22 long zeitBisZumAnstellen;
23 long anstellZeit;
24
25 //shared IRtimer
26 volatile uint32_t IRtimer = 0;
27
28 //shared misc. vars
29 volatile boolean on = false;
30 volatile boolean onPre = true;
31
32 //Initialisieren der Empfängerdaten Variablen
33 volatile uint32_t ch1Data=121;
34 volatile uint32_t ch2Data=0;
35 volatile uint32_t ch3Data=0;
36 volatile uint32_t ch4Data=0;
37 //volatile uint32_t ch5Data=0;
38
39 //Initialisieren der Zeitvariablen für die Empfänger Interrupts
40 volatile uint32_t ch1Pre=0;
41 volatile uint32_t ch2Pre=0;
42 volatile uint32_t ch3Pre=0;
43 volatile uint32_t ch4Pre=0;
44 //volatile uint32_t ch5Pre=0;
45
46 //Interrupt Funktion
47 void irFALLING () {
48     IRtimer = micros() + 810; //den IMUtimer nach hinten schieben
49     if(!on) {
50         //calc rotSpeed
```

```

1          diffTime = micros()-lastTurnTimestamp;
2          lastTurnTimestamp = micros();
3          rotSpeed=60*(1000000/diffTime);
4          on=true;
5      }
6  }
7
8 //CH1 = Lage auf X
9 void ch1RISING () {
10     ch1Pre = micros();
11     detachInterrupt(digitalPinToInterruption(ch1));
12     attachInterrupt(digitalPinToInterruption(ch1), ch1FALLING, FALLING);
13 }
14
15 void ch1FALLING () {
16     ch1Data = map(micros()-ch1Pre,987,2010,-45,45);
17     detachInterrupt(digitalPinToInterruption(ch1));
18     attachInterrupt(digitalPinToInterruption(ch1), ch1RISING, RISING);
19 }
20
21 //CH2 Lage auf Y
22 void ch2RISING () {
23     ch2Pre = micros();
24     detachInterrupt(digitalPinToInterruption(ch2));
25     attachInterrupt(digitalPinToInterruption(ch2), ch2FALLING, FALLING);
26 }
27
28 void ch2FALLING () {
29     ch2Data = map(micros()-ch2Pre,987,2010,-45,45);
30     detachInterrupt(digitalPinToInterruption(ch2));
31     attachInterrupt(digitalPinToInterruption(ch2), ch2RISING, RISING);
32 }
33
34 //CH3 collective pitch
35 void ch3RISING () {
36     ch3Pre = micros();
37     detachInterrupt(digitalPinToInterruption(ch3));
38     attachInterrupt(digitalPinToInterruption(ch3), ch3FALLING, FALLING);
39 }
40
41 void ch3FALLING () {
42     ch3Data = map(micros()-ch3Pre,987,2010,0,25);
43     detachInterrupt(digitalPinToInterruption(ch3));
44     attachInterrupt(digitalPinToInterruption(ch3), ch3RISING, RISING);
45 }
46
47 //CH4 Motordrehzahl
48 void ch4RISING () {
49     ch4Pre = micros();
50     detachInterrupt(digitalPinToInterruption(ch4));

```

```
1         attachInterrupt(digitalPinToInterrupt(ch4), ch4FALLING, FALLING);
2     }
3
4 void ch4FALLING () {
5     ch4Data = map(micros()-ch4Pre,987,2010,0,180);
6     detachInterrupt(digitalPinToInterrupt(ch4));
7     attachInterrupt(digitalPinToInterrupt(ch4), ch4RISING, RISING);
8 }
9
10 //Bei Bedarf den letzten Kanal hinzunehmen
11 /*
12 void ch5RISING () {
13     ch5Pre = micros();
14     detachInterrupt(digitalPinToInterrupt(ch5));
15     attachInterrupt(digitalPinToInterrupt(ch5), ch5FALLING, FALLING);
16 }
17
18 void ch5FALLING () {
19     ch5Data = micros()-ch5Pre;
20     detachInterrupt(digitalPinToInterrupt(ch5));
21     attachInterrupt(digitalPinToInterrupt(ch5), ch5RISING, RISING);
22 }
23 */
24
25 //Funktionsdefinition
26 //Funktionen für die Winkelwahl
27 double cosPSpitz(double pSkalarprodukt, double pDiffVektorLaenge) {
28     double result = sqrt(pSkalarprodukt*pSkalarprodukt)/pDiffVektorLaenge;
29     return result;
30 }
31
32 double cosPStumpf(double pSkalarprodukt, double pDiffVektorLaenge) {
33     double result = pSkalarprodukt/pDiffVektorLaenge;
34     return result;
35 }
36
37
38 void setup() {
39     delay(500); //wait for everything
40
41     //I2C initialise
42     Wire.begin(I2C_MASTER, 0x00, I2C_PINS_18_19, I2C_PULLUP_EXT,
43                 I2C_RATE_600);
44     Wire.setDefaultTimeout(1000);
45
46     i2cData[0] = 7; // Set the sample rate to 1000Hz
47     i2cData[1] = 0x00; // Disable FSYNC and set 260 Hz Acc filtering
48     i2cData[2] = 0x03; // Set Gyro Full Scale Range to 250deg/s
49     i2cData[3] = 0x03; // Set Accelerometer Full Scale Range to 2g
50     while (i2cWrite(MPU6050, 0x19, i2cData, 4, false));
```

```

1      while (i2cWrite(MPU6050, 0x6B, 0x01, true));
2      while (i2cRead(MPU6050, 0x75, i2cData, 1));
3      if (i2cData[0] != 0x68) { // Read "WHO_AM_I" register
4          Serial.print(F("Error reading sensor"));
5          while (1);
6      }
7
8      delay(100); // Wait for sensors to stabilize
9
10     /* Set Kalman and gyro starting angle */
11     updateMPU6050();
12     updatePitchRoll();
13
14     kalmanX.setAngle(roll); // First set roll starting angle
15     gyroXangle = roll;
16
17     kalmanY.setAngle(pitch); // Then pitch
18     gyroYangle = pitch;
19
20     IMUTimer = micros(); // Initialize the IMUTimer
21
22     //IR initialise
23     pinMode(IR, INPUT);
24     attachInterrupt(digitalPinToInterruption(IR), irFALLING, FALLING);
25
26     //Empfängerpins auf input
27     pinMode(ch1, INPUT);
28     pinMode(ch2, INPUT);
29     pinMode(ch3, INPUT);
30     pinMode(ch4, INPUT);
31     //pinMode(ch5, INPUT);
32
33     //Klappenpins auf output
34     pinMode(klappe0, OUTPUT);
35     pinMode(klappe1, OUTPUT);
36
37     //Regler Attach
38     regler0.attach(4);
39     regler1.attach(5);
40     regler0.write(0);
41     regler1.write(0);
42
43     //ServoAttach
44     servo0.attach(6);
45     servo1.attach(3);
46     servo0.write(s0n);
47     servo1.write(s1n);
48
49     //Interrupts für die Fernsteuerung
50     attachInterrupt(digitalPinToInterruption(ch1), ch1RISING, RISING);

```

```
1     attachInterrupt(digitalPinToInterruption(ch2), ch2RISING, RISING);
2     attachInterrupt(digitalPinToInterruption(ch3), ch3RISING, RISING);
3     attachInterrupt(digitalPinToInterruption(ch4), ch4RISING, RISING);
4 //attachInterrupt(digitalPinToInterruption(ch5), ch5RISING, RISING);
5 }
6
7
8 void loop() { //Hauptschleife
9     //IR Timer übergelaufen
10    if((micros() >= IRTimer)&& (IRTtimer != 0)) { //IRtimer = 0 = disabled
11        on=false;
12        onPre=true;
13        IRTimer = 0;
14    }
15
16    //Wenn Infrarotsignal empfangen
17    if(on && onPre) {
18
19        //onPre auf false setzen --> nur einmal bei Sichtkontakt ausgeführt
20        onPre=false;
21
22        //Variablen für die Steuerberrechnung
23        int SOLL[2] = {0,0};
24        // boolean skipThisRound = false;
25        int bx=1;
26        //Vektoren
27            //Differenz Vektor
28            int diffVektor[2];
29            double skalarprodukt = 0.00;
30            double diffVektorLaenge = 0.00;
31        //arccos2
32        double cosP = 0.00;
33        //Winkel
34        int winkelBisZumVektor = 0;
35        int anstellWinkel = 0;
36        int winkelBisZumAnstellen = 0;
37
38
39        //IST Zustand einlesen
40        getGyroAngle();
41
42        //SOLL speichern, damit nicht von den Interrupts überschieben [X, Y]
43        SOLL[0] = ch1Data;
44        SOLL[1] = ch2Data;
45
46        //Vektorrechnung
47        diffVektor[0] = IST[0]-SOLL[0]; //X-Wert des Differenz Vektors
48        diffVektor[1] = IST[1]-SOLL[1]; //Y-Wert des Differenz Vektors
49
50        skalarprodukt = diffVektor[0];
```

```

1      diffVektorLaenge =
2          sqrt(diffVektor[0]*diffVektor[0]+diffVektor[1]*diffVektor[1]);
3
4      //Berrechnungen für die einzelnen Quadranten
5      //oberhalb der X-Achse bei Y>0
6      if(diffVektor[1] > 0) {
7          rotor0first = true; //oberhalb von X --> R0 ist erster Rotor
8          if(diffVektor[0] > 0) {
9              //1. Quadrant --> stumpfer Winkel für R0
10             cosP = cosPStumpf(skalarprodukt, diffVektorLaenge);
11         }
12         else if(diffVektor[0] < 0) {
13             //2. Quadrant --> spitzer Winkel für R0
14             cosP = cosPSpitz(skalarprodukt, diffVektorLaenge);
15         }
16         else {
17             //X=0 --> Winkel von 90 für R0
18             cosP=0;
19         }
20     }
21
22     //unterhalb der X-Achse bei Y < 0
23     else if(diffVektor[1] < 0) {
24         rotor0first = false;
25         if(diffVektor[0] < 0) {
26             //3. Quadrant --> stumpfer Winkel für R1
27             cosP = cosPStumpf(skalarprodukt, diffVektorLaenge);
28         }
29         else if(diffVektor[0] > 0) {
30             //4. Quadrant --> spitzer Winkel für R1
31             cosP = cosPSpitz(skalarprodukt, diffVektorLaenge);
32         }
33         else {
34             //X=0 --> Winkel von 90 für R1
35             cosP=0;
36         }
37     }
38
39     //Auf der X-Achse bei Y = 0
40     else {
41         if(diffVektor[0] > 0) {
42             //Achse zwischen 1 und 4 Quadranten
43             cosP = -1; //180
44             rotor0first = true;
45         }
46         else if(diffVektor[0] < 0) {
47             //Achse zwischen 2 und 3 Quadranten
48             cosP = -1; //180
49             rotor0first = false;
50         }

```

```
1     else {
2         //Vektor zeigt auf (0|0) --> nichts tun
3         //skipThisRound = true;
4     }
5 }
6
7 //Winkel berechnen
8 winkelBisZumVektor = round(acos(cosP));
9 anstellWinkel = diffVektorLaenge*bx; //bx ist variabler Faktor
10 winkelBisZumAnstellen = winkelBisZumVektor-(1/2*(anstellWinkel));
11
12 //Zeiten berechnen
13 zeitBisZumAnstellen = diffTime/(360*winkelBisZumAnstellen);
14 anstellZeit = diffTime/(360*anstellWinkel);
15 klappenTimer = zeitBisZumAnstellen;
16
17 }//Ende der Funktion
18
19
20 //Wenn Klappen Timer übergelaufen
21 if((micros() >= klappenTimer) && klappenTimer != 0) {
22     if(!rotor0down && !rotor1down) {//keine Klappe angestellt
23         if(rotor0first) {//entweder rotor0 anstellen
24             rotor0first = false;
25             rotor0down = true;
26             rotor1down = false;
27             klappenTimer = micros()+anstellZeit;
28             digitalWrite(klappe0, HIGH);
29         }
30         else {//oder rotor1 anstellen
31             rotor0first = true;
32             rotor1down = true;
33             rotor0down = false;
34             klappenTimer = micros()+anstellZeit;
35             digitalWrite(klappe1, HIGH);
36         }
37     }
38     else if(rotor0down) {//wenn rotor0 angestellt ist
39         digitalWrite(klappe0, LOW); //lösen
40         rotor0down = false;
41         rotor1down = false;
42         klappenTimer = micros()+(diffTime/2)-anstellZeit;
43     }
44     else if(rotor1down) {//wenn rotor0 angestellt ist
45         digitalWrite(klappe1, LOW); //lösen
46         rotor0down = false;
47         rotor1down = false;
48         klappenTimer = micros()+(diffTime/2)-anstellZeit;
49     }
50 }
```

```

1      //Regler und Servos neu stellen
2      servo0.write(sOn+ch3Data);
3      servo1.write(s1nch3Data);
4
5      regler0.write(ch4Data);
6      regler1.write(ch4Data);
7 } //Ende der Hauptschleife
8
9
10 void getCopterAngle() {
11     //kal
12     int kalAngleBufferSize = 6;
13     double kalBufferX[6];
14     double kalBufferY[6];
15     int kalBufferCounter = 0;
16     double kalAngleXBuffered, kalAngleYBuffered;
17     int i;
18
19     // Update all the IMU values
20     updateMPU6050();
21
22     double dt = (double)(micros() - IMUTimer) / 1000000; //delta time
23     IMUTimer = micros();
24     // Roll and pitch estimation
25     updatePitchRoll();
26
27     double gyroXrate = gyroX / 131.0; // Convert to deg/s
28     double gyroYrate = gyroY / 131.0; // Convert to deg/s
29
30     if ((roll < -90 && kalAngleX > 90) || (roll > 90 &&
31         kalAngleX < -90)) {
32         kalmanX.setAngle(roll);
33         kalAngleX = roll;
34         gyroXangle = roll;
35     }
36     else kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt);
37
38     if (abs(kalAngleX) > 90) {
39         gyroYrate = -gyroYrate; // Invert rate
40         kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt);
41     }
42
43     //Mittlaufendes Mittel
44     //KalX
45     kalBufferX[kalBufferCounter] = round(kalAngleX);
46     kalAngleXBuffered = 0;
47     for(i=0; i<kalAngleBufferSize; i++) {
48         kalAngleXBuffered = kalAngleXBuffered+kalBufferX[i];
49     }
50
51     kalAngleXBuffered = kalAngleXBuffered/kalAngleBufferSize;

```

```
1 //KalY
2 kalBufferY[kalBufferCounter] = round(kalAngleY);
3 kalAngleYBuffered = 0;
4     for(i=0; i<kalAngleBufferSize; i++) {
5         kalAngleYBuffered = kalAngleYBuffered+kalBufferY[i];
6     }
7
8     kalAngleYBuffered = kalAngleYBuffered/kalAngleBufferSize;
9
10    kalBufferCounter++;
11    if(kalBufferCounter > kalAngleBufferSize-1) kalBufferCounter=0;
12 }
13
14
15 void updateMPU6050() {
16     while (i2cRead(MPU6050, 0x3B, i2cData, 14)); // Get values
17     accX = (int16_t)((i2cData[0] << 8) | i2cData[1]);
18     accY = -(int16_t)((i2cData[2] << 8) | i2cData[3]);
19     accZ = (int16_t)((i2cData[4] << 8) | i2cData[5]);
20     tempRaw = (i2cData[6] << 8) | i2cData[7];
21     gyroX = -(int16_t)(i2cData[8] << 8) | i2cData[9];
22     gyroY = (int16_t)(i2cData[10] << 8) | i2cData[11];
23 }
24
25
26 void updatePitchRoll() {
27 // Source: http://www.freescale.com/files/sensors/doc/app\_note/AN3461.pdf
28 // It is then converted from radians to degrees
29 #ifdef RESTRICT_PITCH // Eq. 25 and 26
30     roll = atan2(accY, accZ) * RAD_TO_DEG;
31     pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) * \
32             RAD_TO_DEG;
33 #else // Eq. 28 and 29
34     roll = atan(accY / sqrt(accX * accX + accZ * accZ)) * \
35             RAD_TO_DEG;
36     pitch = atan2(-accX, accZ) * RAD_TO_DEG;
37 #endif
38 }
39
40
41 uint8_t i2cWrite(uint8_t address, uint8_t registerAddress, uint8_t data, \
42 bool sendStop) {
43     return i2cWrite(address, registerAddress, &data, 1, sendStop);
44 }
45
46
47 uint8_t i2cWrite(uint8_t address, uint8_t registerAddress, uint8_t *data, \
48 uint8_t length, bool sendStop) {
49     Wire.beginTransmission(address);
50     Wire.write(registerAddress);
```

```
1     Wire.write(data, length);
2     uint8_t rcode = Wire.endTransmission(sendStop);
3     if (rcode) {
4         Serial.print(F("i2cWrite failed: "));
5         Serial.println(rcode);
6     }
7     return rcode;
8     // See: http://arduino.cc/en/Reference/WireEndTransmission
9 }
10
11
12 uint8_t i2cRead(uint8_t address, uint8_t registerAddress, uint8_t *data, \
13 nbytes) {
14     uint32_t timeOutTimer;
15     Wire.beginTransmission(address);
16     Wire.write(registerAddress);
17     uint8_t rcode = Wire.endTransmission(false); // Don't release the bus
18     if (rcode) {
19         Serial.print(F("i2cRead failed: "));
20         Serial.println(rcode);
21     }
22     return rcode;
23     // See: http://arduino.cc/en/Reference/WireEndTransmission
24 }
25
26 Wire.requestFrom(address, nbytes, (uint8_t)true);
27 for (uint8_t i = 0; i < nbytes; i++) {
28     if (Wire.available()) data[i] = Wire.read();
29     else {
30         timeOutTimer = micros();
31         while (((micros() - timeOutTimer) < I2C_TIMEOUT) && \
32             !Wire.available());
33         if (Wire.available()) data[i] = Wire.read();
34         else {
35             Serial.println(F("i2cRead timeout"));
36             return 5;
37         }
38     }
39     return 0; // Success
40 }
```