Parallel realtime optimal control of a quadcopter

Projektreport: Advanced Topics in High Performance Scientific Computing

Benedikt König

1. April 2016

1. Einleitung

1.1. Motivation

Die autonome Steuerung eines Fluggerätes ist durch die starken und schnellen Änderungen der Umgebung - besonders durch die Windeinflüsse - eine schwierige Aufgabe. Daher können die optimalen Steuerungssignale nicht vor dem Start berechnet werden. Die passende Steuerungstrategie muss während des Fluges gefunden werden. Hierzu werden Methoden von real-time Optimierung und Strategien von model predictive control kombiniert, um einen Quadrocopter zu steuern.

1.2. Übersicht

Für das dynamische System des Quadrocopters wird die optimale Steuerungsstrategie gesucht, um Unsicherheiten in der Umgebung und im Modell, wie zum Beispiel Windeinflüsse, auszugleichen. In diesem Kontext wird zwischen den Steuerungssignalen u und States x unterschieden. Eine Beziehung zwischen beiden wird durch die ODE $\dot{x(t)} = f(x(t), u(t))$ erreicht. Das Ziel ist es, eine Steuerung u zu finden, welche eine gegebene Kostenfunktion J, die die ODE und deren Randwertbedingungen als Nebenbedingung besitzt, zu minimieren. D.h.

$$\min_{x,u} J(x(t), u(t)) \text{ u.d.N. } \dot{x(t)} = f(x(t), u(t)) \text{ und } x(0) = x_0,\ x(1) = x_1$$

Es existieren verschiedene Techniken, um dieses unendlich dimensionale Optimierungsproblem zu lösen. Diese indirekten Methoden sind aber nicht realtime fähig. Daher wird in diesem Projekt die direkte Methode benutzt: Dabei wird das unendliche dimensionale Optimierungsproblem durch ein endlich nichtlineares Problem ersetzt, um die Steuerungssignale sowie die Zustände zu approximieren. Für die Approximation wird das sogenannte multiple shooting Verfahren verwendet, welches anschließend mit Hilfe der SQP - Methode gelöst wird. Bei den gegebenen Modell- und Umgebungsparametern erreicht diese Methode eine gute numerische Approximationen für u in der Kürze der Zeit. Der Ansatz dieses Projektes ist angelehnt an die Methoden, welche entwickelt worden sind von Diehl et al. [6, 7]

Für die Implementierung des Projektes wurden folgende Bereiche berücksichtigt: Modellierung des Quadrocoptermodelles, Implementierung der Diskretierung und der SQP - Methode sowie der Realtime Ansatz.

1.3. Entwicklung und Entscheidungen

Das Projekt "Parallel realtime optimal control of a quadcopter" ist aus den Case Studies for Nonlinear Optimization des Sommersemesters 2015 entstanden. Dazu mussten wir in einem Team, bestehend aus Philipp Fröhlich, Simon Kick, Annika Stegie und mir (Benedikt König) einen Quadrocopter mit Hilfe der Ansätze von Diehl et al.[6, 7] in Echtzeit optimieren. Wir entschieden uns für die Entwicklungsumgebung MATLAB, da alle mit jenen vertraut waren und sich besonders das Debuggen als einfach

herausgestellt hatte. Nachdem Erfolg des Projektes "rtopt" kam der Wunsch auf, dieses weiterzuführen und die Ergebnisse in eine konkurrenzfähige Umgebung zu transformieren. Zunächst wurde von mir versucht das Projekt unter FORTRAN zu übersetzten, dies ist leider aufgrund von Compilierverfügbarkeit (Stand: 31.03.2016 ist PGI CUDA Fortran Compiler der einzige CUDA Compiler für Fortran -> teuer) gescheitert. Für den zweiten Versuch wurde das Projekt nun in "c" erstellt. Dafür sprachen unter anderem die hohe Plattformverfügbarkeit, Performance und benutzte OpenSource Projekte. Dies waren unter anderem das OpenSource Projekt "SuiteSparse" mit dem Paket "CSparse" von Timothy A. Davis [5] und das Projekt CVODE von Alan C. Hindmarsh [3]. Das Paket CSparse beinhaltet eine Vielzahl von Sparsematrixoperationen, sowie Direktlöser (lu, qr, ..) von Gleichungssystemen mit Sparsematrizen. CVODE hingegen löst Anfangswertprobleme von gewöhnlichen Differentialgleichungen. Im Laufe des Projektes musste besonders das Paket "CSparse", welches auf den compressed sparse column(csc) Format basiert, angepasst und erweitert werden. Zudem stellte sich heraus, dass das Schreiben von Testfunktionen in "C" eine Herausforderung war und die Kombination aus den MATLAB Projekt "rtopt" und "c"-Projekt eine schnellere Alternative darstellte. Dabei wurden, durch die vorhandenen und korrekt funktionieren MATLAB - Funktionen, Daten generiert und mit den neuen Funktionen des "c"- Projektes verglichen.

1.4. Quadrocopterdynamik

In der Literatur gibt es verschiedene dynamische Modelle für Quadrocopter mit unterschiedlicher Komplexität. In diesem Projekt wurde das Modell von [9] verwendet.

1.5. Herleitung

Für die Herleitung des Modelles hat sich als passender physikalischer Ansatz die Newton'schen - Bewegungsgleichungen bewährt.

Dazu lässt sich der Multikopter in drei Körpergruppen (B_0, B_1, B_2) einteilen: Bezugssystem (Startpunkt), "nicht rotierende Körper" (Grundplatte, Arme, Controller, Batterie, etc) und "rotierende Körper" (Motor, Rotor, Rotorschraube). In der Abbildung 1 lassen sich die Schubkräfte $F_1, ...F_4$, wirkend im jeweiligen rotierenden System $B_{M1}, ..., B_{M2}$, erkennen.

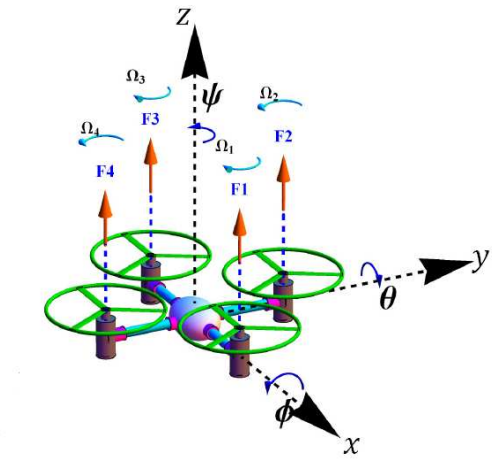

Abbildung 1: Schubkräfte der Motoren

Im Folgendem entspricht

m_{ges}	Gesamtmasse des Multikopters
g	Gewichtskraft
I_{ges}	Gesamter Trägheitsmoment des Quadrocopters im System B_1
${}_{M_i}I_{M_i}$	Trägheitmoment der Motoren im i-ten Motorsystem
${}_1v_1$	Translatorische Geschwindigkeit des Körpers B_1 relativ zum B_0 System im B_1 System
${}_1\omega_1$	Rotationsgeschwindigkeit des Körpers B_1 relativ zum B_0 System im B_1 System
${}_{M_i}\omega_{1,M_i}$	Analog wie ${}_1\omega_1$
${}_{M_i}r_{1,M_i}$	Abstand zwischen B_1 und B_2 mit Länge d
$A_{0,1}$	Rotationsmatrix von Körper B_1 ins B_0 System
A_{1,M_i}	Rotationsmatrix von Körper B_i ins B_1 System

Mit der nicht eingezeichneten Gewichtskraft führt dies zu folgendem Kräftegleichgewicht im KOS(B_0).

$$\begin{aligned}\frac{d}{dt}F_{ges} &= m_{ges}\cdot\frac{d}{dt}{}_1v_1 = m_{ges}\frac{d_1}{dt}{}_1v_1 + {}_1\omega_1 \times m_{ges}{}_1v_1\\ &= {}_1F_g + \sum_{i=1}^{4}{}_1F_i\\ &= m_{ges}A_{1,0}\cdot\begin{pmatrix}0\\0\\-g\end{pmatrix} + \sum_{i=1}^{4}A_{1,M_i}\cdot_{M_i}F_{M_i}\\ &= m_{ges}A_{1,0}\cdot\begin{pmatrix}0\\0\\-g\end{pmatrix} + \sum_{i=1}^{4}\begin{pmatrix}0\\0\\F_i\end{pmatrix}\end{aligned}$$

Die Rotationsmatrix A_{1,M_i} ändert sich je nach Konfiguration des Multikopters. Die Abbildung 2 zeigt die behandelten Konfigurationen. Je nach Konfiguration entstehen folgende Konfigurationsmatrizen:

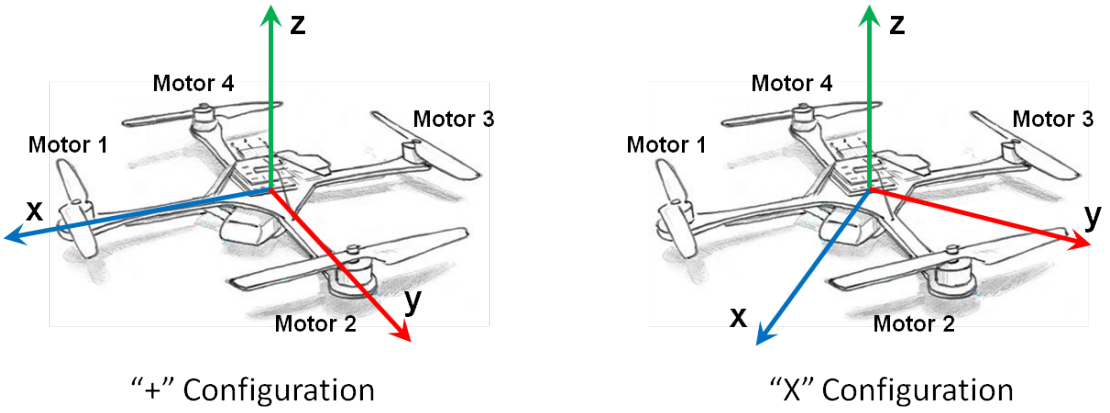

Abbildung 2: Konfigurationen

Für die "+" - Konfiguration werden folgende Matrizen verwendet:

$$A_{1,M_1} = \begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\ A_{1,M_2} = \begin{pmatrix}0&-1&0\\1&0&0\\0&0&1\end{pmatrix}\ A_{1,M_3} = \begin{pmatrix}-1&0&0\\0&-1&0\\0&0&1\end{pmatrix}\ A_{1,M_4} = \begin{pmatrix}0&1&0\\-1&0&0\\0&0&1\end{pmatrix}$$

Für die “x” - Konfiguration werden folgende Matrizen verwendet:

$$A_{1,M_1} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad A_{1,M_2} = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$A_{1,M_3} = \begin{pmatrix} -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad A_{1,M_4} = \begin{pmatrix} -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Für das Kräftegleichgewicht sind die Matrizen nicht relevant, jedoch für den folgenden Drallsatz im körperfesten System B_1. Zuvor wird ein fest montierter Motor betrachtet mit dem Drehmoment M. Diesem wirkt ein Strömungswiderstand τ_{drag} entgegen und es gilt:

$$I_{rot} \cdot \dot{\omega} = M - \tau_{drag}$$

Dabei ist I_{rot} das Trägheitsmoment des Rotors entlang seiner z-Achse. Der Strömungswiderstand ist in der Literatur [9] folgenderweise definiert als:

$$\tau_{drag} = \frac{1}{2}\rho A_r v^2$$

ρ ist die Luftdichte, A_r die Fläche, die der Rotor bei der Umdrehung überschreitet und v ist die Geschwindigkeit relativ zur Luft. Näherungsweise gilt: $\omega \approx \frac{v}{r}$ und es folgt:

$$\tau_{drag} \approx k_{drag}\omega^2$$

Die Konstante $k_{drag} > 0$ ist abhängig von der Luftdichte, dem Radius, der Form des Propellers und anderen Faktoren. Für quasistationär Manöver ist ω konstant und es gilt:

$$M = \tau_{drag} \approx k_{drag}\omega^2$$

Neben dem Drehmoment der Rotoren wird auch deren Schubkraft benötigt. Die Literatur [9] gibt folgende Formel an:

$$F_s = C_T \rho A_r r^2 \omega^2$$

C_T ist der Schubkoeffizient für einen speziellen Rotor, ρ, A_r ist wie oben, die Dichte der Luft bzw. die Fläche die, der Rotor bei der Umdrehung überschreitet. Analog führen wir einen vereinfachten Koeffizienten ein:

$$F_s \approx k_T \omega^2$$

Im Folgenden wird die Annahme getroffen, dass sich der Quadrocopter in einem quasistationären Zustand befindet, d.h. $\omega = const$

Dann gilt für Drallsatz im B_{M_i} System:

$$\begin{aligned} {}_{M_i}I_{M_iM_i}\dot{\omega}_{M_i} + {}_{M_i}\omega_1 \times {}_{M_i}I_{M_iM_i}\omega_{M_i} &= {}_{M_i}I_{M_i}\left({}_{M_i}\dot{\omega}_1 + {}_{M_i}\dot{\omega}_{1,M_i}\right) \\ &+ {}_{M_i}\omega_1 \times {}_{M_i}I_{M_i}\left({}_{M_i}\omega_1 + {}_{M_i}\omega_{1,M_i}\right) \underbrace{\approx}_{{}_{M_i}\omega_1 << {}_{M_i}\omega_{1,M_i}} {}_{M_i}I_{M_iM_i}\dot{\omega}_{1,M_i} \\ &+ {}_{M_i}\omega_1 \times {}_{M_i}I_{M_iM_i}\omega_{1,M_i} = M_i - {}_{M_i}\tau_{M_i} \end{aligned}$$

Zudem folgt wegen Stationärflug:

$$\underbrace{{}_{M_i}I_{M_iM_i}\dot{\omega}_{1,M_i}}_{=0,\text{ da }\omega=\text{ const}} + {}_{M_i}\omega_1 \times {}_{M_i}I_{M_iM_i}\omega_{1,M_i} = {}_{M_i}\omega_1 \times {}_{M_i}I_{M_iM_i}\omega_{1,M_i} = M_i - {}_{M_i}\tau_{M_i}$$

Dem Drehmoment der Motoren M_i wird ein Gegendrehmoment ${}_{M_i}\tau_{M_i}$ entgegen gesetzt. Dieses Drehmoment findet sich auch wieder im Drallsatz des System B_1

$$I_{ges1}\dot{\omega}_1 + {}_1\omega_1 \times I_{ges1}\omega_1 = (\tau_R + \tau_P) - \sum_{i=1}^{4} {}_1\tau_{M_i}$$

Die Drehmomente τ_R und τ_P ergeben sich aus den Roll $f_2 + f_4$ - und Nickkräfte $f_1 + f_3$:

$$\tau_R = A_{1,M_1M_1}r_{1,M_1} \times F_1 + A_{1,M_3M_3}r_{1,M_3} \times F_3$$
$$\tau_P = A_{1,M_2M_2}r_{1,M_2} \times F_2 + A_{1,M_4M_4}r_{1,M_4} \times F_4$$

Das Gegendrehmoment ${}_{M_i}\tau_{M_i}$ ist äquivalent mit ${}_1\tau_{M_i}$, da der Übergang von System B_{M_i} ins B_1 System eine Rotation um die z-Achse darstellt und $M_i\tau_{M_i}$ nur eine z-Komponente besitzt. Somit folgt für den ganzen Drallsatz:

$$\begin{aligned} I_{ges1}\dot{\omega}_1 + {}_1\omega_1 \times I_{ges1}\omega_1 + \sum_{i=1}^{4} {}_1\omega_1 \times {}_{M_i}I_{M_iM_i}\omega_{1,M_i} \\ = -\sum_{i=1}^{4} M_i + (\tau_R + \tau_P) \end{aligned}$$

I_{ges} stellt das Trägheitsmoment des Körpers 1, d.h. des Quadrokopters ohne rotierende Objekte, dar. ${}_{M_i}I_{M_i}$ hingegen ist das Trägheitsmoment, eines einzelnen Rotors i. Bei Brushlesh Motoren ist es so, das sich die "Wand" mitdreht. Dies muss die Kalkulation des Trägheitsmomentes ${}_{M_i}I_{M_i}$ mit einbezogen werden.
Da ${}_{M_i}\omega_{1,M_i}$ nur eine z - Komponente besitzt mit ω_{M_i}, lässt sich Gleichung folgendermassen vereinfachen.

$$\begin{aligned} I_{ges1}\dot{\omega}_1 + {}_1\omega_1 \times I_{ges1}\omega_1 + \sum_{i=1}^{4} ({}_1\omega_1 \times e_z) \cdot I_M\omega_{M_i} \\ = \begin{bmatrix} 0 \\ 0 \\ -\sum_{i=1}^{4} M_i \end{bmatrix} + (\tau_R + \tau_P) \end{aligned}$$

Für die + Konfiguration entsteht mit 1.5 nun folgendes Modell:

$$
\begin{aligned}
{}_0\dot{r}_1 &= A_{0,11} v_1 \\
m_{ges} \frac{d_1}{dt} {}_1v_1 + {}_1\omega_1 \times m_{ges 1} v_1 &= m_{ges} A_{1,0} \cdot \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} + \sum_{i=1}^{4} \begin{pmatrix} 0 \\ 0 \\ F_i \end{pmatrix} \\
\dot{A}_{0,1} &= A_{0,11} \tilde{\omega}_1 \\
I_{ges 1} \dot{\omega}_1 + {}_1\omega_1 \times I_{ges 1} \omega_1 + \sum_{i=1}^{4} ({}_1\omega_1 \times e_z) \cdot I_{MM_i} \omega_{M_i} & \\
= \begin{bmatrix} 0 \\ 0 \\ -\sum_{i=1}^{4} M_i \end{bmatrix} + (\tau_R + \tau_P) &= \begin{bmatrix} d(F_2 - F_4) \\ d(F_3 - F_1) \\ -\sum_{i=1}^{4} M_i \end{bmatrix}
\end{aligned}
$$

Mit 1.5 und 1.5

$$
\begin{aligned}
{}_0\dot{r}_1 &= A_{0,11} v_1 \\
m_{ges} \frac{d_1}{dt} {}_1v_1 + {}_1\omega_1 \times m_{ges 1} v_1 &= m_{ges} A_{1,0} \cdot \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} k_{TM_i} \omega_{1,M_i}^2 \end{pmatrix} \\
\dot{A}_{0,1} &= A_{0,11} \tilde{\omega}_1 \\
I_{ges 1} \dot{\omega}_1 + {}_1\omega_1 \times I_{ges 1} \omega_1 + \sum_{i=1}^{4} ({}_1\omega_1 \times e_z) \cdot I_{MM_i} \omega_{M_i} & \\
= \begin{bmatrix} 0 & d \cdot k_T & 0 & -d \cdot k_T \\ -d \cdot k_T & 0 & d \cdot k_T & 0 \\ -k_{drag} & k_{drag} & -k_{drag} & k_{drag} \end{bmatrix} \cdot \begin{bmatrix} M_1 \omega_{M_1}^2 \\ M_2 \omega_{M_2}^2 \\ M_3 \omega_{M_3}^2 \\ M_4 \omega_{M_4}^2 \end{bmatrix} &
\end{aligned}
$$

1.5.1. Quaternionen

Statt mit Eulerwinkel wird die Rotationsmatrix $A_{0,1}$, welche die Vektoren vom Bezugssystem ins Körpersystem transformiert, mit Hilfe von Quaternionen berechnet. Dies bringt den Vorteil das kritische Punkte, also Punkte dessen Zuordnung nicht lokal umkehrbar, nicht spezifisch betrachtet werden müssen. Es bringt aber den Nachteil mit sich, dass für eine sinnvolle Auswertung der Quaternionen, diese normiert sein müssen.
Die Rotationsmatrix für Quaternionen lautet:

$$
A_{0,1} := R_{0,1}(q) = \begin{bmatrix} 1 - 2(q_2^2 + q_3^2) & -2q_0q_3 + 2q_1q_2 & 2q_0q_2 + 2q_1q_3 \\ 2q_0q_3 + 2q_1q_2 & 1 - 2(q_1^2 + q_3^2) & -2q_0q_1 + 2q_2q_3 \\ -2q_0q_2 + 2q_1q_3 & 2q_0q_1 + 2q_2q_3 & 1 - 2(q_1^2 + q_2^2) \end{bmatrix}
$$

nach [10].

Die Zeitableitung der Quaternionen q lässt nach [2], wie folgt berechnen: $\dot{q} = \frac{1}{2} q \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix}$.

Somit folgt für das dynamische System des Quadrocoptors in der "+"- Konfiguration:

$$
\begin{aligned}
{}_0\dot{r}_1 &= R_{0,1}(q)\,{}_1v_1 \\
\dot{q} &= \frac{1}{2}q \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix} \\
m_{ges}\frac{d_1}{dt}{}_1v_1 + {}_1\omega_1 \times m_{ges}{}_1v_1 &= R(q)^T \begin{pmatrix} 0 \\ 0 \\ -m_{ges}\cdot g \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} k_{TM_i}\omega_{1,M_i}^2 \end{pmatrix} \\
I_{ges1}\dot{\omega}_1 + {}_1\omega_1 \times I_{ges1}\omega_1 + \sum_{i=1}^{4} ({}_1\omega_1 \times e_z)\cdot I_{MM_i}\omega_{M_i} & \\
= \begin{bmatrix} 0 & d\cdot k_T & 0 & -d\cdot k_T \\ -d\cdot k_T & 0 & d\cdot k_T & 0 \\ -k_{drag} & k_{drag} & -k_{drag} & k_{drag} \end{bmatrix} \cdot \begin{bmatrix} {}_{M_1}\omega_{M_1}^2 \\ {}_{M_2}\omega_{M_2}^2 \\ {}_{M_3}\omega_{M_3}^2 \\ {}_{M_4}\omega_{M_4}^2 \end{bmatrix} &
\end{aligned}
$$

Da die Schrittweite des ODE - Lösers in unserem Lösungsansatz deutlich kleiner als 1 ist, wird die Normierung der Quaternionen mit Hilfe des Korrekturterms

$$
\begin{aligned}
q_1 &= ... + \lambda\dot{q}_1 \\
q_2 &= ... + \lambda\dot{q}_2 \\
q_3 &= ... + \lambda\dot{q}_3 \\
q_4 &= ... + \lambda\dot{q}_4 \\
\text{mit } \lambda &= 1 - (q_1^2 + q_2^2 + q_3^2 + q_4^2)
\end{aligned}
$$

gelöst [4].
Für das Projekt wurde nun $\dot{x} = f(x,u) \in \mathbb{R}^{13}$ wie folgt gewählt:

$$
\begin{aligned}
f(x,u) &:= \begin{bmatrix} R_{0,1}(q)\,{}_1v_1 \\ \frac{1}{2}q \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix} \\ M^{-1}(T-\Theta) \end{bmatrix} \\
\text{mit } T &:= \begin{bmatrix} 0 \\ 0 \\ -k_T \cdot \sum_{i=0}^{4} {}_{M_i}\omega_{M_i}^2{}^2 \\ K \cdot \begin{bmatrix} {}_{M_1}\omega_{M_1}^2{}^2 \\ {}_{M_2}\omega_{M_2}^2 \\ {}_{M_3}\omega_{M_3}^2 \\ {}_{M_4}\omega_{M_4}^2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times {}_1\omega_1 \cdot I_M \cdot ({}_{M_1}\omega_{M_1}^2 - {}_{M_2}\omega_{M_2}^2 + {}_{M_3}\omega_{M_3}^2 - {}_{M_4}\omega_{M_4}^2) \end{bmatrix}, \\
\Theta &:= \begin{bmatrix} m_{ges}\cdot {}_1\omega_1 \times {}_1v_1 + R(q)^T \cdot \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \end{bmatrix} \text{ und} \\
M &:= diag(m, m, m, I_{ges_1}, I_{ges_2}, I_{ges_3}) \text{ sowie für} \\
x &:= [r_1, r_2, r_3, q_1, q_2, q_3, q_4, v_1, v_2, v_3, \omega_1, \omega_2, \omega_3] \text{ und} \\
u &:= [{}_{M_1}\omega_{M_1}, {}_{M_2}\omega_{M_2}, {}_{M_3}\omega_{M_3}, {}_{M_4}\omega_{M_4}] \\
K &:= \begin{bmatrix} 0 & d\cdot k_T & 0 & -d\cdot k_T \\ -d\cdot k_T & 0 & d\cdot k_T & 0 \\ -k_{drag} & k_{drag} & -k_{drag} & k_{drag} \end{bmatrix}
\end{aligned}
$$

1.6. Sensitivitäten

Das Ziel ist es später das SQP - Verfahren zu lösen. Dazu benötigt man die Jacobimatrix und Hessematrix der Zustände $h(t_{x_k}, x_{k-1}, u_{k-1})$, welche durch das Lösen der ODE berechnet werden. Aber anstatt zuerst die Differentialgleichung zu lösen und dann die Lösung nach x bzw. u abzuleiten, leitet man zuerst die ODE nach x bzw. u ab und löst erst dann die Differentialgleichung. Damit das nachfolgende Vorgehen überhaupt erlaubt ist, bzw. alle Ausdrücke wohldefiniert sind, muss $x(t)$ in allen Variablen stetig differenzierbar sein, damit man den Satz von Schwarz anwenden kann.

1.6.1. Jacobimatrix

Sei $x \in \mathbb{R}^{k\cdot 13}$, $u \in \mathbb{R}^{k\cdot 4}$, $f(x,u) \in \mathbb{R}^{k\cdot 13}$ wie in 1.5.1 gewählt und in k Zeitpunkten diskretisiert, so ist $f \in C^\infty$. Dann gibt es für die ODE $\dot{x} = f(x,u)$, wegen ihrer Lipschitzstetigkeit, nach Poincare eine eindeutige Lösung $h(x,u) \in \mathbb{R}^{k\cdot 13}$ mit

$$\dot{h}(x,u) = f(h(x,u),u)$$

Man betrachte nun die Differenzialgleichung zum Zeitpunkt k, es folgt für die Differenzierung nach der x_i Komponente

$$\frac{d}{dx_i^k}\frac{d}{dt^k}h(x^k,u^k) = \frac{d}{dx_i^k}f(h(x^k,u^k),u^k)$$

Da $f \in C^\infty$ folgt nun, dass die Ableitungen vertauscht werden können.

$$\begin{aligned}\frac{d}{dt^k}\frac{d}{dx_i^k}h(x^k,u^k) = \frac{d}{dx_i^k}f(h(x^k,u^k),u^k) &= \frac{d}{dx^k}\left[f(h(x^k,u^k),u^k)\right]\cdot\frac{d}{dx_i^k}h(x^k,u^k)\\ &+ \frac{d}{du^k}\left[f(h(x^k,u^k),u^k)\right]\cdot\underbrace{\frac{du^k}{dx_i^k}}_{=0}\end{aligned}$$

mit $A^k := \frac{d}{dx_i^k}h(x^k,u^k) \in \mathbb{R}^{13\times 13}, i = 1..13$ folgt die ODE

$$\frac{d}{dt^k}A^k = \underbrace{\frac{d}{dx^k}\left[f(h(x^k,u^k),u^k)\right]}_{\in\mathbb{R}^{13\times 13}}\cdot A^k := a(x^k,u^k,A^k)$$

mit der Anfangsbedingung $A_0^k = I \in \mathbb{R}^{13\times 13}$

Analog folgt für die Differenzierung nach der u_i Komponente

$$\begin{aligned}\frac{d}{dt^k}\frac{d}{du_i^k}h(x^k,u^k) &= \frac{d}{dx^k}\left[f(h(x^k,u^k),u^k)\right]\cdot\frac{d}{du_i^k}h(x^k,u^k)\\ &+ \frac{d}{du^k}\left[f(h(x^k,u^k),u^k)\right]\cdot\frac{du^k}{du_i^k}\end{aligned}$$

mit $B^k := \frac{d}{du_i^k} h(x^k, u^k) \in \mathbb{R}^{13\times 4}, i = 1..4$ folgt die ODE

$$\frac{d}{dt^k} B^k = \underbrace{\frac{d}{dx^k}\left[f(h(x^k,u^k),u^k)\right]}_{\in\mathbb{R}^{13\times 13}} \cdot B^k + \underbrace{\frac{d}{du^k}\left[f(h(x^k,u^k),u^k)\right]}_{\in\mathbb{R}^{13\times 4}} \cdot \underbrace{I^k}_{\in\mathbb{R}^{4\times 4}} := b(x^k, u^k, B^k)$$

mit folgender Anfangsbedingung $B_0^k = 0 \in \mathbb{R}^{13\times 4}$.

Sei Φ_1^k, Φ_2^k Lösungen für $\frac{d}{dt^k}\Phi_1^k = a(x^k, u^k, \Phi_1^k)$ und $\frac{d}{dt^k}\Phi_2^k = b(x^k, u^k, \Phi_2^k)$ Für die Jacobimatrix $J^k \in \mathbb{R}^{13\times 17}$ von $h(t_{x_k}, x_{k-1}, u_{k-1})$ folgt aus den Lösungen Φ_1^k, Φ_2^k; $J^k = [\Phi_1^k, \Phi_2^k]$

1.6.2. Hessematrix

Im Verlauf des Projektes hat sich herausgestellt, dass der Aufwand für die Berechnung der Hessematrix mit dem Butzen kollidiert. Anstatt diese zu berechnen wird sie durch eine Diagonalmatrix approximiert.

1.7. Implementierung

Das Modell wurde zunächst mit Hilfe des Mapleskriptes "GenerateFunktionJacobi.mw" generiert. Dazu wurde zunächst die Funktion $f(x, u)$ sowie die Matrizen A und B wurden 1.6.1 berechnet. Die Matrizen A und D wurden durch die Umgruppierung $Bv := [A_{1,1}, \dots, A_{13,1}, A_{1,2}, \dots, A_{13,2}, \dots, A_{13,13}]$ und $Bv := [B_{1,1}, \dots B_{13,1}, B_{1,2}, \dots, B_{13,4}]$ in die Vektorform gebracht. Für das gesamte zu lösende Differentialgleichungssystem gilt nun: $[\dot{x}, \dot{Av}, \dot{Bv}] := [f(x, u), Av, Bv] =: F \in \mathbb{R}^{234}$. Nach dem Lösen der ODE, können die Ergebnisse wieder zu Matrizen Φ_1, Φ_2 transformiert werden, um die Jacobimatrix $J := [\Phi_1, \Phi_2]$ zu erhalten.

Durch Analyse verschiedener Löser für gewöhnliche Differentialgleichungen in Matlab (ode45, ode23, ode15s, ode23s), wurde festgestellt, dass das System steif ist. Somit wird für die Lösung der ODE, die Jacobimatrix J_F für F benötigt, dazu wurde in MAPLE F nach x, Av, Bv abgeleitet.

Abschließend wurde mit Hilfe des CodeGeneration Package von MAPLE C Code generiert und mit Hilfe der Pythonskripte "GenerateScript.py" und "GenerateDyn.py" Ersetzungen (Bsp.: Warnungen, Datentypen) durchgeführt und mit der Templatedatei "rtopt_gen.template" in festgelegte Form gebracht. Die Ausgabe der Codegenerierung wird in die C - Datei "rtopt_gen.c" geschrieben. Die generierten Funktionen werden durch die Wrapperfunktionen "f", "jac" in "dyn.c" aufgerufen.

Zum Lösen der ODE wurden verschiedene Softwarepakete ausgetestet (Matlab, odeint, CVODE) mit verschiedenen Lösern (Matlab: ode45, ode15s, ode23s; odeint: rosenbrock4 ohne und mit cuda; CVODE: CV_BDF, CV_ADAMS). Als schnellster Löser/Methode hat sich die Methode CV_ADAMS des Softwarepaketes CVODE von Alan. C. Hindmarsh herausgestellt [8]. Die Methode "integrate" in "solver.c" ruft die Funktionen des Softwarepaketes auf und gibt als Ergebnis die Zustände $h(t_{x_k}, x_{k-1}, u_{k-1})$ sowie deren Jacobimatrix $J = [\Phi_1, \Phi_2]$ zurück.

1.7.1. Parameterfindung

Als nächster Schritt wurden realistische Parameter mit Hilfe des Softwaretools; GUI_Modeling des Softwarepaket Quad-Sim Quadrocopter gefunden:

m_{ges}	$1{,}022\,\mathrm{kg}$
g	$9{,}81\,\mathrm{kg\,s^{-2}}$
I_{ges}	$\mathrm{diag}(0.0093886, 0.0093886, 0.018406)\,\mathrm{kg \cdot m^2}$
kT	$1{,}5 \cdot 10^{-7}\,\mathrm{N\,RPM^{-2}}$
kQ	$3{,}0 \cdot 10^{-9}\,\mathrm{N\,m\,RPM^{-2}}$
d	$0{,}22\,\mathrm{m}$
I_M	$4{,}4466 \cdot 10^{-6}\,\mathrm{kg\,m^2}$

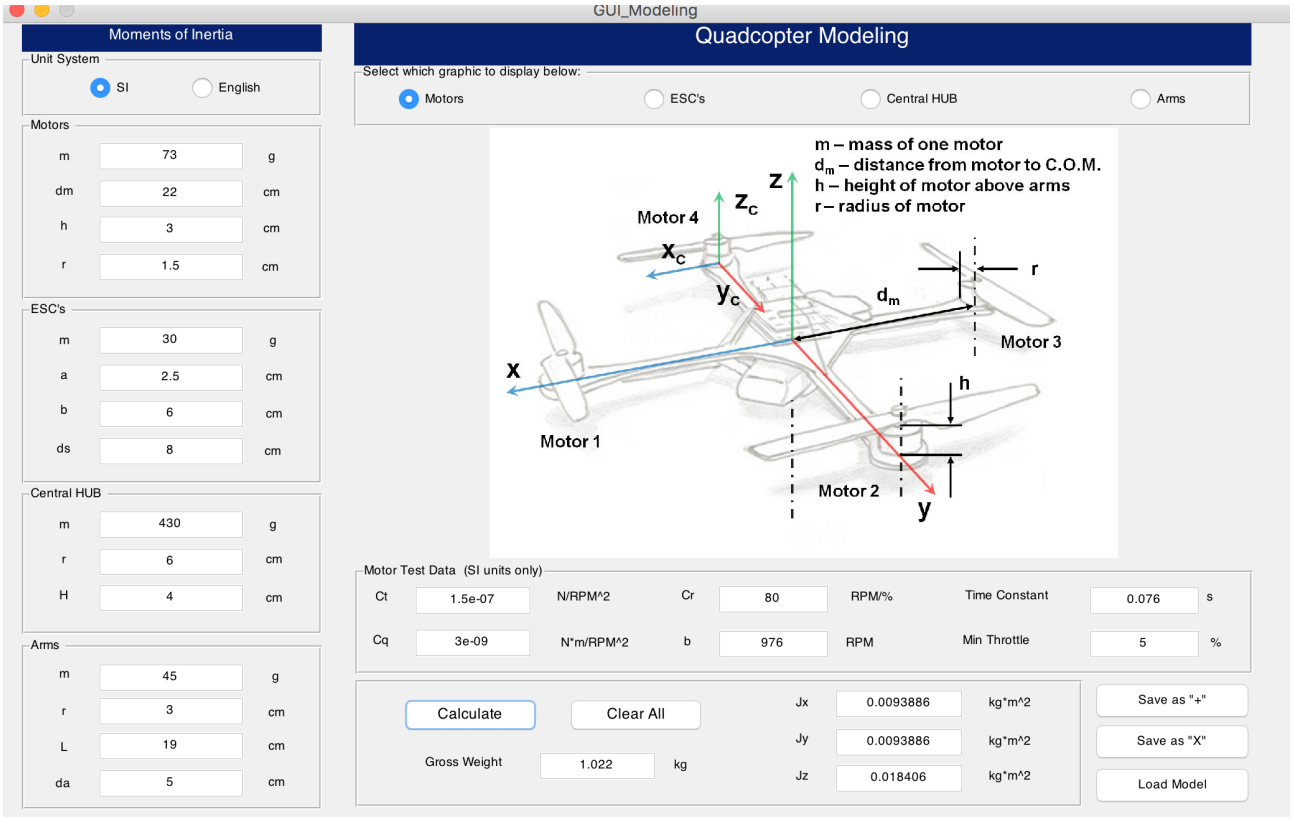

Abbildung 3: Konfigurationsmanager

1.7.2. Tests

Um die Implementierung der Funktionen “integrate” und “jac”zu ratifizieren wurden und die Testfunktionen “numDiff_nDvec” in “solver.c” und “testJac” in “dyn.c” geschrieben. Für die Funktion “integrate” konnte nur ein qualitativer Test geschrieben werden, da die numerische Differenzierung zu ungenau für die exakte Lösung des nichtlinearen ODE System war.

1.8. Diskretisierung

Zu Beginn des Fluges ist der benötige Zeitraum Δt, aufgrund von der Unsicherheiten (Wind, Änderung der Flugbahn, etc.), ungekannt. Deshalb wird ein prediction horizon (Vorhersagehorizont) eingeführt. Dieser umspannt einen fixen Zeitrahmen und ist in N äquidistante Punkte unterteilt. In der aktuellen Konfiguration beträgt der Abstand der einzelnen Gitterpunkte eine Sekunde. Die Zeitspanne des Horizonts darf nicht zur kurz sein, da die Berechnungen besonders bei plötzlichem Kurswechsel und starken äußeren Einflüssen zu ungenau werden. Bei zu großem Horizont benötigen die Berechnungen zu viel Zeit, so dass die Updaterate der Steuerungssignale nicht eingehalten werden kann.

1.9. Multiple shooting

In den $N-1$ Intervallen werden $N-1$ Anfangswertprobleme gelöst mit:

$$\dot{s} = f(s,u) \quad s(t_{k-1}) = s_{k-1}$$

Dabei wird der Anfangswert eines jeden Intervalls jedoch nicht fest vorgegeben, sondern als eine Variable behandelt. Da die Lösung h der Differentialgleichung auch von der Steuerung u in dem jeweiligen Intervall abhängig ist, kann man die Lösung jedes Anfangswertproblems als eine Funktion $h = h(t, s_{i-1}, q_{i-1})$ angeben. Hierbei ist also s_{i-1} der Anfangswert als Parameter, außerdem gehen wir davon aus, dass die Steuerung im Intervall $[t_{i-1}, t_i]$ konstant gleich $q(t_{i-1}) = q_{i-1}$ ist.
Damit x eine stetige Funktion ist, soll der Anfangswert eines jeden Intervalls mit dem dem Endwert des vorigen Intervalls übereinstimmen. Dies wird als Nebenbedingung folgendermaßen in das bereits bestehenden Optimierungsproblem eingefügt:

$$h(t_{x_i}, s_{i-1}, q_{i-1}) - s_i = 0$$

Diese Methode wird in der Literatur [1] als multiple shooting method oder auch Mehrfachschießverfahren bezeichnet. Dort wird die Gleichung 1.9 aber nicht mit Hilfe eines SQP - Verfahren gelöst, sondern mit Hilfe eines Newton-Verfahrens mit geschätzten Anfangswerten x_i, $i \in [1, N-1]$

1.10. SQP - Methode

Mit der Diskretisierung des Optimierungsproblems und der multiple shooting method als Nebenbedingung führt dies zu dem Problem $P^k(x_k)$

$$\min_{\substack{s_k,...,s_N \\ q_k,...,q_N}} \sum_{i=k}^{N-1} j_i(s_i,q_i) \quad s.t. \begin{cases} x_k - s_k = 0 \\ h_i(s_i,q_i) - s_{i+1} = 0 \\ g_i(s_i,q_i) \leq 0 \quad \forall i = k,...,N-1 \end{cases}$$

in jedem Zeitschritt k.

Die zu den Problemen $P^k(x_k)$ gehörenden Lagrangegleichungen lauten wie folgt:

$$L^k(y) = \sum_{i=k}^{N-1} j_i(s_i,q_i) + \lambda_k^T(x_k - s_k) + \sum_{i=k}^{N-1} \lambda_{i+1}^T(h_i(s_i,q_i) - s_{i+1}) + \sum_{i=k}^{N-1} \mu_{i,j}^T(g_{i,j}(s_i,q_i))$$

mit $j \in \mathscr{A}(s_i,q_i) := \{j : 1 \leq j \leq m; g_{i,j}(s_i,q_i) \geq 0\}$
In dieser Lagrangegleichung wird $y := (\lambda_k, s_k, q_k, \mu_k, \lambda_{k+1}, s_{k+1}, q_{k+1}, \mu_{k+1}, ..., \lambda_N, s_N)$ verwendet. Mit KKT-Bedingung

$$\nabla_y L^k(y) = 0$$

und das exakte Newton-Raphson-Verfahren

$$y_{i+1} = y_i + \Delta y_i$$

bei dem jedes Δy_i die Lösung des linearen approximierten Systems

$$\nabla_y L^k(y_i) + J^k(y_i)\Delta y_i = 0$$

ist.

Der von Diehl [6] vorgestellte Algorithmus verwendet das oben vorgestellte Newton-Raphson-Verfahren nicht exakt. Die zweite Ableitung $\nabla_y^2 L^k$, die Hesse-Matrix $\nabla_{q,s}^2 L^k$, wird ersetzt durch eine (symmetrische) Approximierung. Die Approximierung von $\nabla_y^2 L^k(y)$ wird im Folgenden mit $J^k(y)$ bezeichnet. Ebenso wird das Newton-Type-Verfahren approximiert:

$$\nabla_y L^k(y_i) + J^k(y_i)\Delta y_i = 0$$

J^k wird auch als Karush-Kuhn-Tucker Matrix bezeichnet.

$$\nabla_y^2 L^k(y) = \begin{pmatrix}
 & -E & & & & & & & \\
-E & Q_k & M_k & & A_k^T & & & & \\
 & M_k^T & R_k & C_{i,j}^T & B_k^T & & & & \\
 & & C_{i,j} & & & & & & \\
 & A_k & B_k & & & \ddots & & & \\
 & & & & \ddots & Q_{N-1} & M_{N-1} & & A_{N-1}^T \\
 & & & & & M_{N-1}^T & R_{N-1} & C_{N,j}^T & B_{N-1}^T \\
 & & & & & & C_{N,j} & & \\
 & & & & & A_{N-1} & B_{N-1} & & & -E \\
 & & & & & & & -E & Q_N
\end{pmatrix}$$

Mit $A_i := \dfrac{\partial h_i}{\partial s_i}$, $B_i := \dfrac{\partial h_i}{\partial q_i}$, $\begin{pmatrix} Q_i & M_i \\ M_i^T & R_i \end{pmatrix} := \nabla_{s_i,q_i}^2 L^i$, $C_{i,j} := \dfrac{\partial g_{i,j}}{\partial q_i}$, $j \in \mathscr{A}$ und $Q_N := \nabla_{s_N}^2 L^i$.

In der Approximierung werden Q_i,R_i und M_i ersetzt durch $Q_i^H(s_i, q_i, \lambda_{k+1}, \mu_{k+1})$, $R_i^H(s_i, q_i, \lambda_{k+1}, \mu_{k+1})$ und $M_i^H(s_i, q_i, \lambda_{k+1}, \mu_{k+1})$.
Man betrachte $y = (\lambda_k, s_k, q_k, \mu_k, \tilde{y})$, dass $\tilde{y}$ direkt zum nächsten Problem $P_{k+1}(x_{k+1})$ gehört.

Im Paper [7] wird erwähnt, dass diese vorteilhafte Form von $\nabla_y^2 L^k(y)$, bzw. $J^k(y)$ eine effiziente Lösung der Gleichung $J^k(y)x = b$ durch die Riccati Recursion ermöglicht.

1.10.1. Approximation

Ein wichtiger Spezialfall des Newton-Type Verfahrens ist die Constrained Gauss-Newton Methode, welche sich auf die LEAST SQUARES Form der Funktion

$$\sum_{i=k}^{N-1} \frac{1}{2}\|l_i(s_i, q_i)\|_2^2 + \frac{1}{2}\|e(s_N)\|_2^2$$

anwenden lässt. In diesem Fall lässt sich die Approximierung wie folgt berechnen:

$$\begin{pmatrix} Q_i^H & M_i^H \\ (M_i^H)^T & R_i^H \end{pmatrix} := \left(\frac{\partial l_i(s_i,q_i)}{\partial(s_i,q_i)}\right)^T \left(\frac{\partial l_i(s_i,q_i)}{\partial(s_i,q_i)}\right), \quad Q_N := \left(\frac{\partial e(s_N)}{\partial s_N}\right)^T \left(\frac{\partial e(s_N)}{\partial s_N}\right)$$

In der Praxis hat sich eine einfache Approximation von $\begin{pmatrix} Q_i^H & M_i^H \\ (M_i^H)^T & R_i^H \end{pmatrix}$ durch $\nabla^2 l_i(s_i,q_i) + \alpha \cdot \mathrm{E}$ sehr effizient gegenüber der exakten Berechung sowie deren obrigen Approximation herausgestellt. Zudem folgt, dass die Matrix $M = 0$ ist und die Matrizen Q und R eine Diagonalmatrix darstellen.

1.11. Implementierung

Die Diskretisierung wird in der Initialisierung des Programmes in der Datei "realtimesolver.c" mit Funktion "initialize_rtsolver" durchgeführt. Die Funktion wird im Kapitel 3 Programmablauf näher behandelt.

1.11.1. Lagrange

Die SQP - Methode und die damit verbundene Lagrangeableitung bzw. Approximation($\nabla_{y_i} L^k(y_i)$, $J^k(y_i)$) werden in den Funktionen "getLD" bzw. "getLDD" berechnet bzw. wie in 1.10.1 approximiert. Jene Funktionen greifen auf die Jacobi und Hessematrix der Kostenfunktion zu. Die zugehörigen Funktionen "costD" und "costDD" befinden sich in der Datei "cost.c" und wurden auch mit Hilfe des MAPLE - "GenerateFunktionJacobi.mw" und Pythonskriptes "GenerateDyn.py" generiert. Zudem benötigt man für die Berechnung von $\nabla_{y_i} L^k(y_i)$ und $J^k(y_i)$ die Ableitungen der Nebenbedingungen.

1.11.2. Nebenbedingungen

Wie in 1.10 festgelegt, wird als einzige Gleichheitsnebenbedingung das Mehrfachschießverfahren gesetzt. Das Berechnen der $N-1$ Differentialgleichungen wird im Kapitel Parallelisierung 3.1 erläutert.

Damit die Steuerungssignale der Motoren nicht in einen unerlaubten Bereich fallen, wurde eine untere u_{min} und eine obere u_{max} Begrenzung eingeführt. Daraus ergeben sich dann acht Ungleichungsnebenbedingungen für die vier Motoren. Da man von einer Lösung nahe eines KKT - Punktes ausgehen kann, sind nur aktive Constraints interessant. Davon können aber wegen $0 \leq u_{min} < u_{max}$, nur vier Ungleichungsnebenbedingungen aktiv sein, was die Sache vereinfacht hat. Die Ungleichungsnebenbedingungen gelten aktiv, wenn gilt:

$$g_i(s_i,q_i) = \begin{pmatrix} u_{min} - q_i \\ q_i - u_{max} \end{pmatrix} \geq 0$$

Die Implementierung der Ableitung der Ungleichungsnebenbedingungen und die Implementierung der Abfrage der aktiven Menge befinden sich in den Funktionen "get_ineq_con_at_t_act" und "checkIfActive".

1.11.3. Test

Mit Hilfe der Matlab Funktion "generateTestData" des Matlab Projektes "rtopt" und der Funktion "test_lagrange" in "lagrange.c" wurden die Ergebnisse der Funktionen "getLD" und "getLDD" validiert.

2. Riccati - Rekursion

In diesem Projekt wird die Gleichung $L^k(y_i) + J^k(y_i)\Delta y_i = 0$ mit Hilfe der Riccati Rekursion gelöst. Dies ist aufgrund der speziellen Form der Matrix $J^k(y^k)$ (1.10) möglich. Um die Übersicht zu wahren erfolgt die Herleitung der Riccati - Rekursion für die Gleichung $J^k(y^k)\delta y^k = -\nabla_{y_k} L^k(y^k)$ ohne Ungleichungsnebenbedingungen und zudem wird die Teilmatrix M_i auf 0 gesetzt, d.h.

$$J^k(y) = \begin{pmatrix} & -E & & & & & & & \\ -E & Q_k & 0 & A_k^T & & & & & \\ & 0 & R_k & B_k^T & & & & & \\ & A_k & B_k & & \ddots & & & & \\ & & & \ddots & Q_{N-1} & 0 & A_{N-1}^T & & \\ & & & & 0 & R_{N-1} & B_{N-1}^T & & \\ & & & & A_{N-1} & B_{N-1} & & & -E \\ & & & & & & & -E & Q_N \end{pmatrix}$$

Betrachte nun

$$\begin{pmatrix} -E & Q_{N-1} & 0 & A_{N-1}^T & \\ & 0 & R_{N-1} & B_{N-1}^T & \\ & A_{N-1} & B_{N-1} & & -E \\ & & & -E & Q_N \end{pmatrix} \begin{pmatrix} \Delta\lambda_{N-1} \\ \Delta s_{N-1} \\ \Delta q_{N-1} \\ \Delta\lambda_N \\ \Delta s_N \end{pmatrix} = - \begin{pmatrix} \nabla_{s_{N-1}} L^k(y^k) \\ \nabla_{q_{N-1}} L^k(y^k) \\ \nabla_{\lambda_N} L^k(y^k) \\ \nabla_{s_N} L^k(y^k) \end{pmatrix}$$

Zur einfacheren Schreibweise ist ab jetzt $\nabla_{s_N} := -\nabla_{s_N} L^k(y^k)$

$$\begin{pmatrix} 0 & A_{N-1} & B_{N-1} \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\lambda_{N-1} \\ \Delta s_{N-1} \\ \Delta q_{N-1} \end{pmatrix} + \begin{pmatrix} 0 & -E \\ -E & Q_N \end{pmatrix} \begin{pmatrix} \Delta\lambda_N \\ \Delta s_N \end{pmatrix} = \begin{pmatrix} \nabla_{\lambda_N} \\ \nabla_{s_N} \end{pmatrix}$$

Für das Verfahren setzt man $P_N = Q_N$.

$$\begin{aligned} \begin{pmatrix} \Delta\lambda_N \\ \Delta s_N \end{pmatrix} &= \begin{pmatrix} 0 & -E \\ -E & P_N \end{pmatrix}^{-1} \left[\begin{pmatrix} \nabla_{\lambda_N} \\ \nabla_{s_N} \end{pmatrix} - \begin{pmatrix} 0 & A_{N-1} & B_{N-1} \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\lambda_{N-1} \\ \Delta s_{N-1} \\ \Delta q_{N-1} \end{pmatrix} \right] \\ &= \begin{pmatrix} -P_N & -E \\ -E & \end{pmatrix} \left[\begin{pmatrix} \nabla_{\lambda_N} \\ \nabla_{s_N} \end{pmatrix} - \begin{pmatrix} 0 & A_{N-1} & B_{N-1} \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\lambda_{N-1} \\ \Delta s_{N-1} \\ \Delta q_{N-1} \end{pmatrix} \right] \end{aligned}$$

Dann werden $\Delta\lambda_{N-1}$,Δs_{N-1} und Δq_{N-1} gelöst.

$$\begin{pmatrix} -E & Q_{N-1} + A_{N-1}^T P_N A_{N-1} & A_{N-1}^T P_N B_{N-1} \\ 0 & B_{N-1}^T P_N A_{N-1} & R_{N-1} + B_{N-1}^T P_N B_{N-1} \end{pmatrix} \begin{pmatrix} \Delta\lambda_{N-1} \\ \Delta s_{N-1} \\ \Delta q_{N-1} \end{pmatrix} = \begin{pmatrix} \nabla_{s_{N-1}} \\ \nabla_{q_{N-1}} \end{pmatrix} + \begin{pmatrix} A_{N-1}^T P_N & A_{N-1}^T \\ B_{N-1}^T P_N & B_{N-1}^T \end{pmatrix} \begin{pmatrix} \nabla_{\lambda_N} \\ \nabla_{s_N} \end{pmatrix}$$

Zuerst wird Δq_{N-1} gelöst

$$\Delta q_{N-1} = (R_{N-1} + B_{N-1}^T P_N B_{N-1})^{-1} (\nabla_{q_{N-1}} + B_{N-1}^T P_N \nabla_{\lambda_N} + B_{N-1}^T \nabla_{s_N} - B_{N-1}^T P_N A_{N-1} \Delta s_{N-1})$$

Für $\Delta\lambda_{N-1}$ und Δs_{N-1} ergibt sich dann

$$-\Delta\lambda_{N-1} + P_{N-1}\Delta s_{N-1} = \nabla^*_{s_{N-1}}$$

mit

$$\begin{aligned} P_{N-1} = \;& Q_{N-1} + A_{N-1}^T P_N A_{N-1} - A_{N-1}^T P_N B_{N-1}(R_{N-1} + B_{N-1}^T P_N B_{N-1})^{-1} B_{N-1}^T P_N A_{N-1} \\ \nabla^*_{s_{N-1}} = \;& \nabla_{s_{N-1}} + A_{N-1}^T P_N \nabla_{\lambda_N} + A_{N-1}^T \nabla_{s_N} \\ & - A_{N-1}^T P_N B_{N-1}(R_{N-1} + B_{N-1}^T P_N B_{N-1})^{-1}(\nabla_{q_{N-1}} + B_{N-1}^T P_N \nabla_{\lambda_N} + B_{N-1}^T \nabla_{s_N}) \end{aligned}$$

Damit ergibt sich für das anfängliche System $J^k(y^k)\Delta y^k = -\nabla_{y^k} L^k(y^k)$

$$\begin{pmatrix} & -E & & & & & & \\ -E & Q_k & 0 & A_k^T & & & & \\ & 0 & R_k & B_k^T & & & & \\ & A_k & B_k & & \ddots & & & \\ & & & \ddots & Q_{N-2} & 0 & A_{N-2}^T & \\ & & & & 0 & R_{N-2} & B_{N-2}^T & \\ & & & & A_{N-2} & B_{N-2} & & -E \\ & & & & & & -E & P_{N-1} \end{pmatrix} \begin{pmatrix} \Delta\lambda_k \\ \Delta s_k \\ \Delta q_k \\ \vdots \\ \Delta\lambda_{N-1} \\ \Delta s_{N-1} \end{pmatrix} = \begin{pmatrix} \nabla_{\lambda_k} \\ \nabla_{s_k} \\ \nabla_{q_k} \\ \vdots \\ \nabla_{\lambda_{N-1}} \\ \nabla^*_{s_{N-1}} \end{pmatrix}$$

Die weiteren P_i ergeben sich für $i = k+1, ..., N-1$

$$\begin{aligned} P_{i-1} = \;& Q_{i-1} + A_{i-1}^T P_i A_{i-1} - A_{i-1}^T P_i B_{i-1}(R_{i-1} + B_{i-1}^T P_i B_{i-1})^{-1} B_{i-1}^T P_i A_{i-1} \\ \nabla^*_{s_{i-1}} = \;& \nabla_{s_{i-1}} + A_{i-1}^T P_i \nabla_{\lambda_i} + A_{i-1}^T \nabla^*_{s_i} \\ & - A_{i-1}^T P_i B_{i-1}(R_{i-1} + B_{i-1}^T P_i B_{i-1})^{-1}(\nabla_{q_{i-1}} + B_{i-1}^T P_i \nabla_{\lambda_i} + B_{i-1}^T \nabla^*_{s_i}) \end{aligned}$$

Schließlich ergibt sich

$$\begin{pmatrix} \Delta\lambda_k \\ \Delta s_k \end{pmatrix} = \begin{pmatrix} -P_k & -E \\ -E & 0 \end{pmatrix} \begin{pmatrix} \nabla_{\lambda_k} \\ \nabla^*_{s_k} \end{pmatrix}$$

Bis zum jetzigen Zeitpunkt der Riccati - Rekursion wird x_k das Eingabesignal (bestehend aus aktueller Position, Lage, etc.) nicht benötigt. Ist dies nun bekannt, berechne $\nabla_{\lambda_k} = x_k - s_k$ und

$$\Delta q_k = (R_k + B_k^T P_{k+1} B_k)^{-1}(\nabla_{q_k} + B_k^T P_{k+1} \nabla_{\lambda_{k+1}} + B_k^T \nabla^*_{s_{k+1}} - B_k^T P_{k+1} A_k \Delta s_k)$$

Sende an die Steuerungseinheiten der Motoren das Signal $u_k = q_k + \Delta q_k$ und berechne mit der Forward Recursion die restlichen Werte von Δy^k

$$\begin{pmatrix} \Delta\lambda_{i+1} \\ \Delta s_{i+1} \end{pmatrix} = \begin{pmatrix} -P_{i+1} & -E \\ -E & 0 \end{pmatrix} \left[\begin{pmatrix} \nabla_{\lambda_{i+1}} \\ \nabla^*_{s_{i+1}} \end{pmatrix} - \begin{pmatrix} 0 & A_i & B_i \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\lambda_i \\ \Delta s_i \\ \Delta q_i \end{pmatrix} \right]$$

Abschließend ergibt sich y^{k+1} aus $y^{k+1} = \prod^{k+1}(y^k + \Delta y^k)$.

Dieses Verfahren empfiehlt sich nicht wenn N zu groß ist.
Eine Alternative wäre, für große N, ein fixes n zu wählen und das Verfahren darauf ohne kleiner werdenden Horizont anzuwenden.

2.1. Implementierung

Die Implementierung des Riccatialgorithmus ist in zwei Funktionen unterteilt: Einmal die Funktion "doStep", welche rückwärts rekursiv, d.h. $i = N \dots 1$ P_{i-1} und $\nabla^*_{s_{i-1}}$ berechnet. Zum anderen die

Funktion “solveStep” die vorwärts rekursiv die Werte $\Delta\lambda_k$, Δs_k, Δq_k und $\Delta\mu_k$ berechnet. Wie oben erwähnt, werden bei den implementierten Algorithmus die Nebenbedingungen $\Delta\mu_k$ berücksichtigt. Da die aktiven Ungleichungsnebenbedingungen nur sehr selten auftreten und die Version ohne Constraints deutlich performanter ist wie mit, wurden zwei verschiedene Versionen des Riccatialgorithmus implementiert.

Da der Algorthmus sehr rekursiv ist, wurden wiederverwendete Ergebnisse in der Struktur “riccati_step_tmp” zwischengespeichert, um die Performance zu erhöhen.

2.1.1. Test

Mit Hilfe der Matlab Funktionen “generateHesse” und “doTestWithHorizon2” des Matlab Projektes “rtopt” in der Klasse “Lagrange” und der Funktion “test_lagrange” in “lagrange.c” wurden die Ergebnisse der Funktionen “getLD” und “getLDD” validiert.

3. Programmablaufplan

Das Hauptprogramm “main” ruft die Funktion “initialize_rtsolver” auf, diese führt eine Diskretisierung in N äquidistante Schritten durch. Zudem allokiert sie den benötigten Speicher und initialisiert den Vektor y_0 mit übergebenen Werten. Ein Beispiel für Initialisierung für den Zustandsvektor liefert die Funktion “getSteadyPoint” in “config.c”. Der zurückgegebene Zustandsvektor liefert für einen stationären Flug eine Lösung des Optimierungsproblems. Nach der Initialisierung werden die Werte in der Struktur “realtimesolver_struct” gespeichert und an die Hauptroutine “fminrt” übergeben. Diese führt den Programmablauf wie Abbildung 3 durch. Nach erfolgreichem Ablauf wird der Speicher in “free_rtsolver” wieder freigeben.

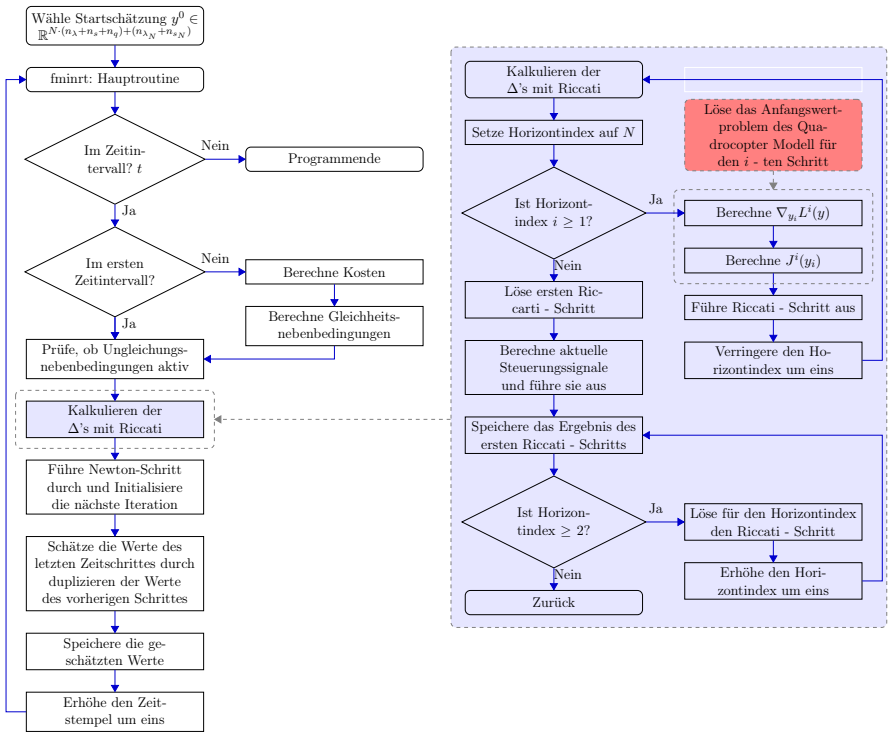

Abbildung 4: Serieller Programmablaufplan

3.1. Parallelisierung

3.1.1. Riccati

Bei der Analyse des Riccatialgorithmus stellt man fest, dass sich der Punkt “Kalkulieren der Δ’s mit Riccati” in der Abildung 3 nur sehr schwer parallelisieren lässt, da die Berechnungen von P, $\nabla^*_{s_{i-1}}$ und Co rekursiv sind, d.h. es wird zwingend einen Vorgänger bzw. Nachfolger benötigt. Einen Performancegewinn erzielt man, wie bereits erwähnt, mit dem Zwischenspeichern von wiederkehrenden Ergebnissen. Besonders die im Schritt “Führe Riccati - Schritt aus” erzielten Ergebnisse der lu - Zerlegung können in den Schritten “Löse Riccati Schritt” wiederverwendet werden. Durch die einfache Approximation der Hessematrix durch $J^i(y_i)$ ist der Aufwand zur Lösung des Gleichungssystem $\nabla_y L^k(y_i) + J^k(y_i)\Delta y_i = 0$ gegenüber dem multiple shooting in Schritt “Paralleles Lösen der $N-1$ Anfangswertprobleme” in der Abbildung 3.2 vernachlässigbar.

3.2. Multiple Shooting

Das Lösen des Anfangswertproblems des Quadrocopter - Modells für den i - ten Schritt im parallelisierten Programmablauf 3.2 wird nicht mehr durch die Schritte “Berechne $\nabla_{y_i} L^i(y)$” und “Berechne $J^i(y_i)$” aufgerufen (siehe Abbildung 3), sondern in dem Punkt “Löse $N-1$ Anfangswertprobleme des Quadrocopter Modell” ausgelagert, um eine Parallelisierung zu erreichen. Die Ergebnisse werden durch den Vektor der Struktur “multiple_shooting” an die Funktionen “getLD” und “getLDD” übergeben, die für die Berechnung der Ableitung bzw. der Approximation Jacobi - bzw. Hessematrix der Lagrangefunktion zuständig sind.

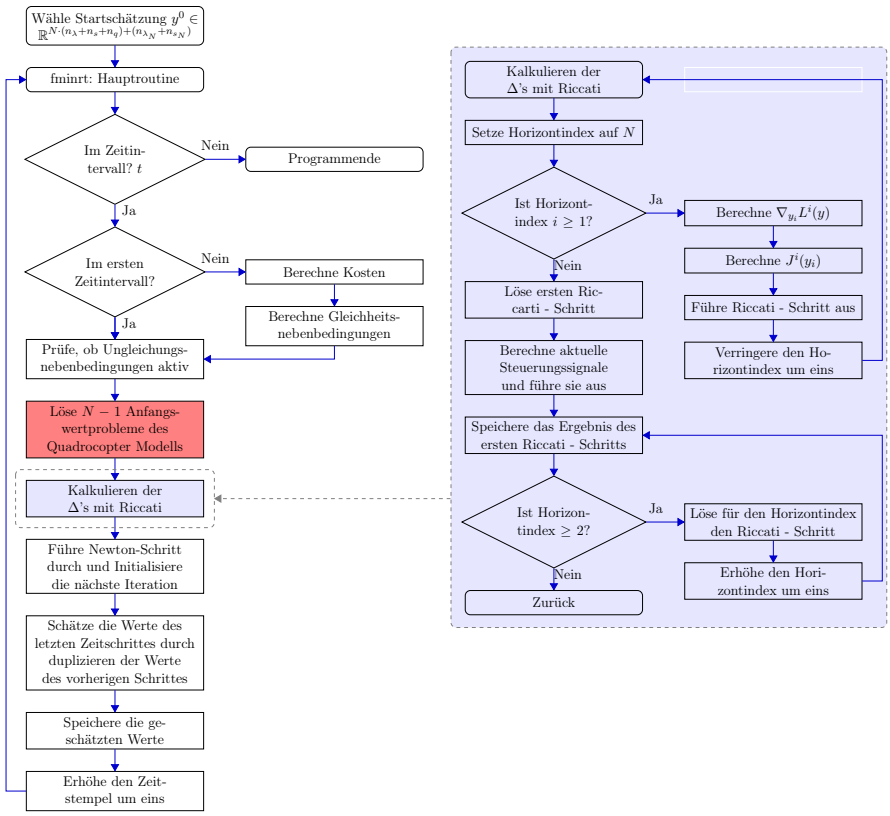

Abbildung 5: Paralleler Programmablaufplan

3.2.1. Implementierung

Für die Parallelisierung wurde das Message Passing Interface (MPI) gewählt, dar die Designziele die Bedürfnisse des Projektes erfüllen. Portabilität, Effizenz und Flexibilität. MPI ist kein IEEE oder ISO Standard, aber es ist tatsächlich so, dass dieses Protokoll ein Industriestandard geworden ist. Zudem unterstützen verschiedene MPI Produkte MVAPICH2, OpenMPI, CRAY, und IBM Platform MPI die Schnittstelle CUDA-aware. Dies Schnittstelle bietet eine einfache Kommunikation zwischen CUDA - GPU's und CPU, sodass dieses Projekt von reiner CPU Parallelisierung in einen Mix aus GPU und CPU Parallelisierung umgebaut werden kann. Im Nachfolgenden wird angenommen, dass die Anzahl zu lösender Differentialgleichungen $N-1$ deutlich höher ist als die Anzahl der Prozesse P, d.h. $P << N-1$. Zudem existiert ein Booleanarray "bwork" mit $N-1$ Werten. Ist der i-te Eintrag wahr, so wurde bzw. wird das i te Differentialgleichungssystem gelöst. Pro Prozess existiert ein temporäres "itmp_work" Array mit $N-1$ ganzzahligen Elementen. Hat ein Prozess eine ODE gelöst fügt es dem "itmp_work" Array in kommender Reihefolge die Ranking-Id hinzu. Dabei beginnen die Ranking-Id's mit 1. Ist an der k - ten Stelle eine Null, so ist das Array zu Ende. Die Parallisierung hat folgenden Ablauf:

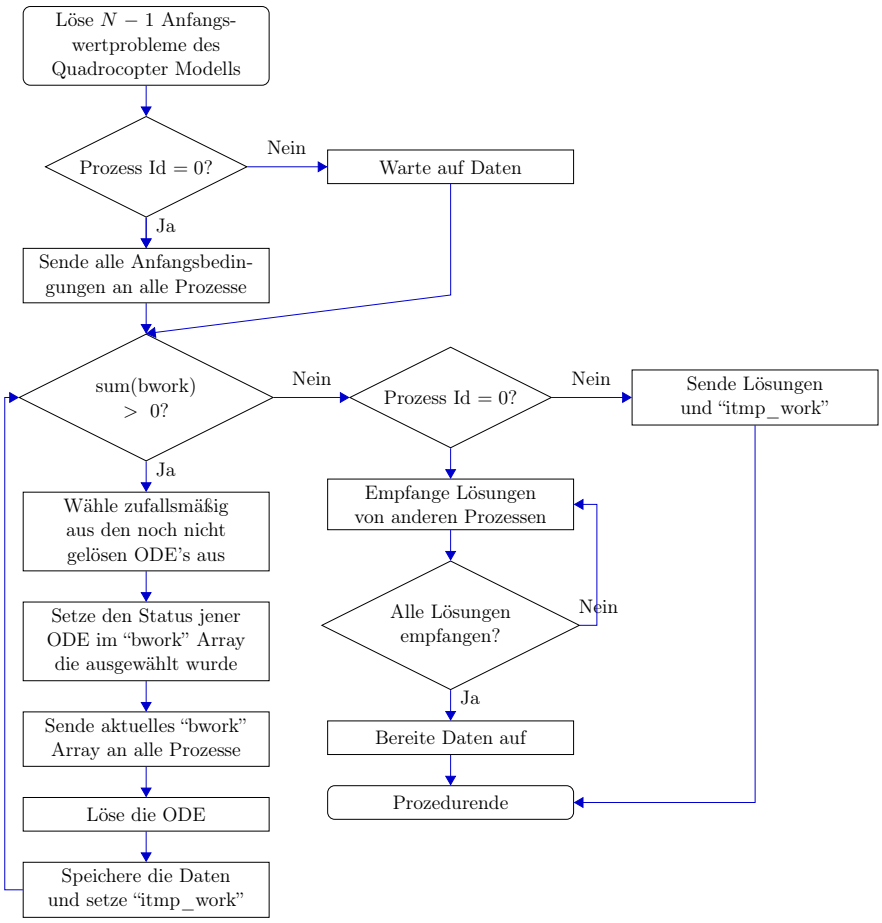

Abbildung 6: MPI - Implementierung

4. Ausblick

Nach der Implementierung des Projektes kann als nächster Schritt das Parallelisieren mit MPI CUDA Adware durchgeführt werden. Es besteht die Möglichkeit den Code Embedding fähig zu machen. Als

Plattform könnte sich die Embedding Platform Jetson TK1 eignen. Es müssten aber noch weitere Nachforschungen getätigt werden. Abschließend sollte ein passende Konfiguration für einen Quadrocopter gefunden und eine Testumgebung mit Trackingsystem aufgebaut werden.

A. Anhang

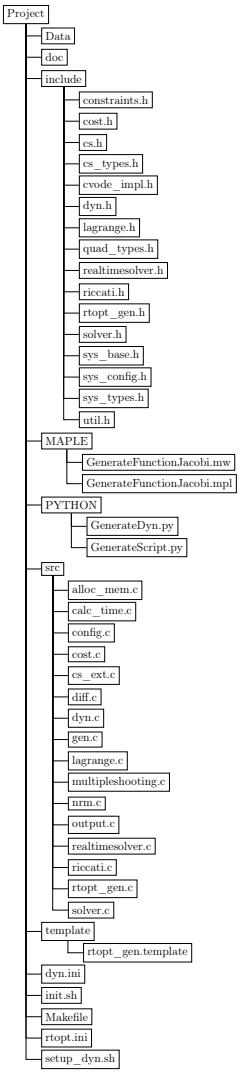

Abbildung 7: Dateistruktur

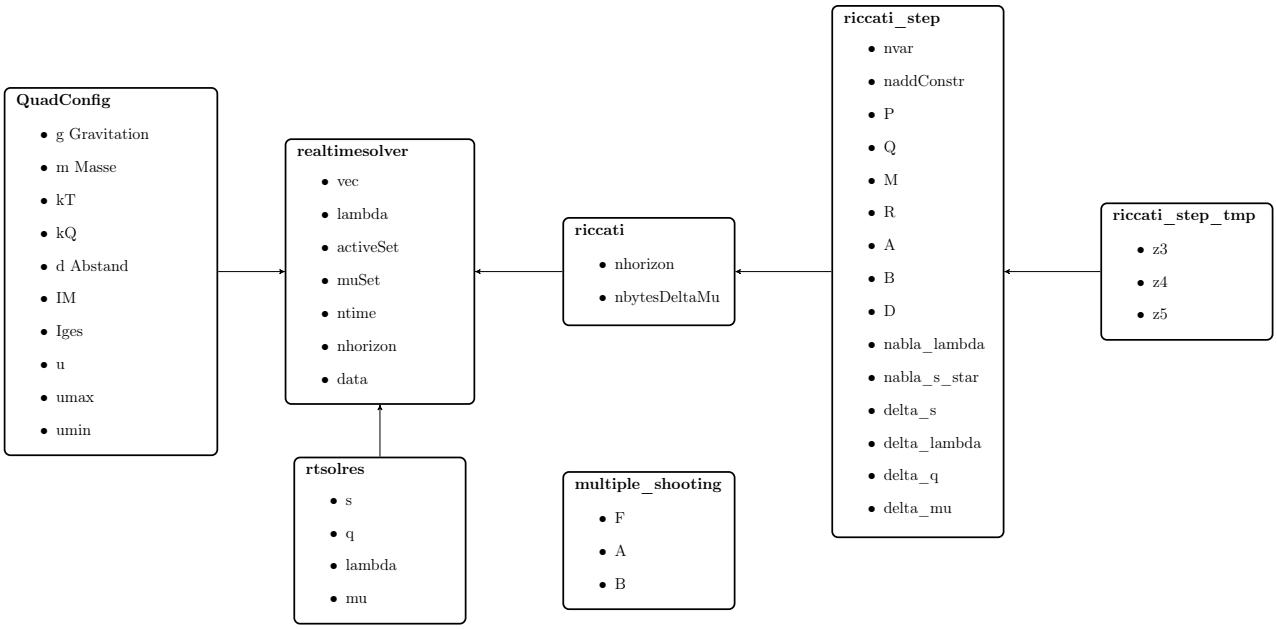

Abbildung 8: Stukturdiagramm

Literatur

[1] Bulirsch, Roland ; Stoer, Josef: *Numerische Mathematik 2*. Springer-Verlag Berlin Heidelberg, 2005

[2] Chou, J. C. K.: Quaternion kinematic and dynamic differential equations. In: *IEEE Transactions on Robotics and Automation* 8 (1992), Feb, Nr. 1, S. 53–64. `http://dx.doi.org/10.1109/70.127239`. – DOI 10.1109/70.127239. – ISSN 1042–296X

[3] Cohen, Scott D. ; Hindmarsh, Alan C.: CVODE, a Stiff/Nonstiff ODE Solver in C. In: *Comput. Phys.* 10 (1996), März, Nr. 2, 138–143. `http://dx.doi.org/10.1063/1.4822377`. – DOI 10.1063/1.4822377. – ISSN 0894–1866

[4] Cooke, Joseph M. ; Zyda, Michael J. ; Pratt, David R. ; McGhee, Robert B.: NPSNET: Flight Simulation Dynamic Modeling Using Quaternions. In: *Presence: Teleoper. Virtual Environ.* 1 (1992), Oktober, Nr. 4, 404–420. `http://dx.doi.org/10.1162/pres.1992.1.4.404`. – DOI 10.1162/pres.1992.1.4.404. – ISSN 1054–7460

[5] Davis, Timothy A.: *Direct methods for sparse linear systems*. Bd. 2. Siam, 2006

[6] Diehl, Moritz: *Real-Time Optimization for Large Scale Nonlinear Processes*, Universität Heidelberg, Faculty of Mathematics and Computer Science, Diss., 2001

[7] Diehl, Moritz: Realtime optimization and nonlinear model predictive control of processes governed by differential algebraic equations. In: *Journal of Process Control* 12 (2002), Jun, Nr. 4, S. 577–585

[8] Hindmarsh, A. C. ; Serban, R.: User Documentation for CVODE v2.8.2 / LLNL. 2015 (UCRL-MA-208108). – technical report. – ODE - Solver

[9] Reyes-Valeria, E. ; Enriquez-Caldera, R. ; Camacho-Lara, S. ; Guichard, J.: LQR control for a quadrotor using unit quaternions: Modeling and simulation. In: *Electronics, Communications and Computing (CONIELECOMP), 2013 International Conference on*, 2013, S. 172–178

[10] Richter-Gebert, J. ; Orendt, T.: *Geometriekalküle*. Springer Berlin Heidelberg, 2009 (Springer-Lehrbuch). `https://books.google.ca/books?id=e69BmAEACAAJ`. – ISBN 9783642025297