Projektarbeit in
System und Automatisierungstechnik

Flugregelung einer Kleindrohne für Forschungszwecke

Autoren: Oscar Battistelli
battiosc@students.zhaw.ch
Leandro Chelini
chelilea@students.zhaw.ch

Dozent: Prof. Dr. Walter Siegl
siew@zhaw.ch

Projektleiter: Oliver Ensslin
enso@zhaw.ch

Datum: 11. Januar 2011

1. Abstract

Diese Dokumentation soll dem Leser die Funktion des verwendeten Progammcodes verständlich erklären. Es wird davon ausgegangen, dass die Grundlagen der Regelungstechnik und das grobe Wissen über die Funktionsweise der Orientierung einer Drohne im Raum vorhanden sind. Damit diese Dokumentation kein Duplikat der Bachelorarbeit von Emilio Schmidhauser und René Chaney vom 13. August 2010 darstellt, wurde sie als Ergänzung zur vorhergehenden Arbeit geschrieben. Dies setzt somit gewisse Grundkenntnisse aus den alten Arbeiten über das Projekt voraus.
Die Hauptaufgabe war, eine Regelung zu entwickeln, welche die Geschwindigkeit gegenüber der Umgebungsluft (Airspeed) der Drohne in der Toleranz von +/-1 m/s hält. Nachdem der komplette alte Code verworfen wurde und die Drohne mit dem neuen Open-Source-Code des Projekts Paparazzi flog, wurde die Airspeed-Regelung erneut in Angriff genommen. Nach einigen Flugversuchen wurde das Ziel erreicht, da schlussendlich die Drohne mit einer Airspeed-Abweichung von nur +/-0.6 m/s geregelt wurde. Beim Testflug wurde jedoch festgestellt, dass die Drohne ein Problem mit dem Regeln der Flughöhen hat, wenn das aggressive Steigen aktiviert ist. Sehr wahrscheinlich handelt es sich dabei jedoch nur um falsch eingestellte Parameterwerte. Es konnte ebenfalls festgestellt werden, dass der verwendete analoge Differentialdrucksensor Probleme mit tiefen Temperaturen hat, dies wird jedoch zukünftig mit dem neuen digitalen Sensor nicht mehr der Fall sein.
Zusätzlich wurde noch die adaptive Regelstrecke untersucht, welche sich in der Paparazzi-Comunity zu einem brisanten Thema entwickelt hat und sich noch in der Entwicklungsphase befindet. Aus den Erkenntnissen der Codeanalyse wurden von uns dann zwei verschiedene Adaptive-Regelsysteme entwickelt, welche in den Code integriert werden können, sobald die Parameterprobleme mit dem neuen Code behoben sind.

Inhaltsverzeichnis

2. Aufgabenstellung

Zu Beginn des Projekts sollte lediglich die Geschwindigkeitsregelung verbessert werden. Dazu gehörte die Darstellung mit Simulink-Blöcke. Für diese Aufgabe hätte die alte Vassilis-Regelung und Airspeed-ONE und TWO als Vorlage gedient. Jedoch wurde während der Projektarbeit schnell klar, dass die alten Regler fehlerbehaftet und diese Fehler nur mit viel Aufwand zu lokalisieren sind. Aus Grund dieser Erkenntnissen wurde die Aufgabenstellung angepasst. Nun sollte die Geschwindigkeitsregelung des aktuellen Paparazzi-Codes in Betrieb genommen werden, da dies viele strukturelle und taktische Vorteile mit sich brachte, da nun wieder mit der Paparazzi-Comunity über den Code diskutiert werden konnte. Auch Möglichkeiten, welche die Parameter in Abhängigkeit der momentanen Geschwindigkeit anpassen, sollen danach erarbeitet und gegebenenfalls programmiert werden.

3. Übersicht Freheitsgrade

In der Abbildung 3.1 sind die Flugzeug-Achsen dargestellt. Die x-Achse entspricht der Rollbewegung. Diese wird von der Lage des Querruders beeinflusst und mit der `estimator_phi` Variable überprüft. Die Pitchbewegung entspricht der y-Achse. Diese wird von dem Höhenruder beeinflusst und mit der Variable `estimator_theta` abstrahiert. Die Yawbewegung (z-Achse) wird durch das Seitenruder gesteuert. Diese wird mit die Variable `estimator_hspeed_dir` verfolgt.
Bei die Maja wirkt die Motorleistung als ein Kippmoment um der Pitchachsse, dieses wird erzeugt, da die Motorlage vom Hersteller nicht optimal ausgewählt wurde. Für grössere Flugzeuge ist ebenfalls ein Drehmoment um die Yawachse zu berücksichtigen welches von der Propellerrotation erzeugt wird.
Im Kurvenflug wird zusätzlich die Auftriebskraft kleiner, weil das Flugzeug schräg in der Luft lieg und eine kleinere Fläche horizontal zum Boden aufweist. Um zu vermeiden, das dass Flugzeug in eine Vrille (Trudeln) verfällt wird mit dem Antrieb und dem Höhenruder ein Moment um der Pitchachse erzeugt.

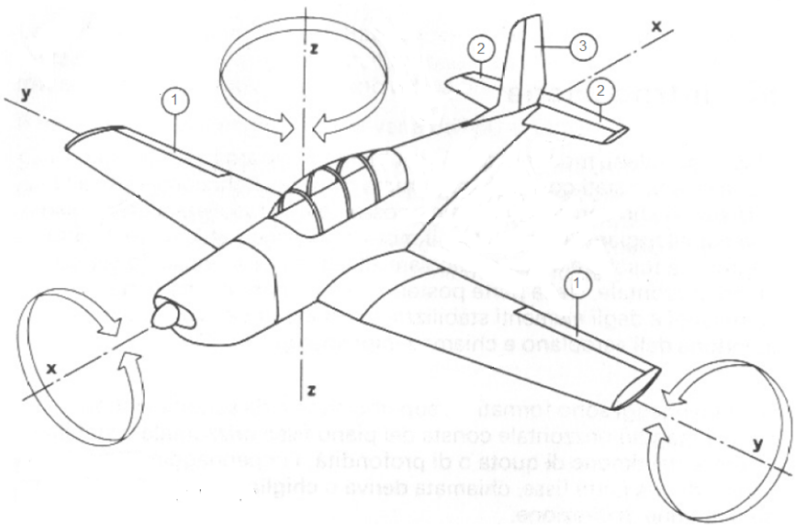


Abbildung 3.1.: Achsen eines Flugzeuges

x-Achse	Roll	1	Querruder
y-Achse	Pitch	2	Höhenruder
z-Achse	Yaw	3	Seitenruder

4. Neues Paparazzi Projekt

4.1. Entscheidungsgrund

Seit der Bachelorarbeit von Emilio Schmidhauser und René Chaney vom 13. August 2010 gab es grosse Veränderungen im Open Source Projekt "Paparazzi". Die gesamte Datenstruktur und das Modulmanagement wurde überarbeitet und bereinigt.
Im Bereich der Airspeed-Regelung gab es viele Erneuerungen. Der Code der Hauptregler befindet sich nun in der Datei `stabilizatio_attitude.c` und in `guidance_v.c`, wobei `stabilization_attitude.c` für die horizontale Stabilisierung und `guidance_v.c` für die vertikale Regelung zuständig ist. Die Airspeed-Regelung von Vasillis somit in `guidance_v.c` zu finden.
Diese Umstrukturierungen erleichtern den Überblick über die verschiedenen Regler. Da der alte Code durch die vielen Änderungen nur noch sehr schwer verständlich und mit Fehlern behaftet war, wurde entschieden, das Projekt mit dem neuen Paparazzi Code weiter zu führen. Dies verschlang, wie wir im Nachhinein feststellen müssen, sehr viel Zeit. Jedoch wurden die Änderungen, um die Testdrohne "Maja" mit Paparazzi fliegen zu lassen, nun so ausgebearbeitet, dass das Airframe mit einem frisch geladenen Source-Code funktioniert. Dies ist wichtig, da wir das sich ständig in Entwicklung befindende Open-Source-Projekt "On the fly" übernehmen können und somit von den Erneuerungen profitieren können.
Ebenfalls können so Codes, welche von uns geschrieben werden, veröffentlicht und von andern genutzt werden, was die Hauptidee eines Open-Source-Projekts widerspiegelt. Ebenfalls erhalten wir so nützliche Feedbacks über geschriebene Codes für zukünftige Arbeiten.

Abbildung 4.1.: Logo des Paparazzi-Projekts

4.2. Datenstruktur

Die wichtigsten Ordner und Dateien sind in der Abbildung 4.2 dargestellt. Auf die Konfigurationsdateien im Ordner `conf` wird im Kapitel 4.3.1 genauer eingegangen.

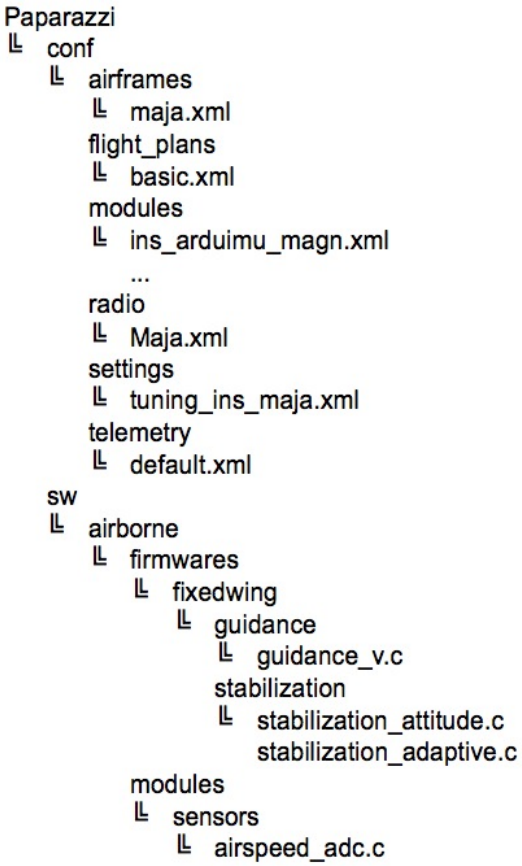


Abbildung 4.2.: Datenstruktur des aktuellen Paparazzi Projekts

- `guidance_v.c`

  Diese Datei beinhaltet den Programmcode für die vertikale Ausrichtung des Flugzeugs. In ihr wird unter Anderem der Throttle, Climb und Altitude-Loop abgearbeitet.

- `stabilization_attitude.c`

  Roll, Pitch und Course werden in dieser Datei geregelt. Diese Loops sind für die horizontale Stabilität verantwortlich.

- `stabilizaton_adaptive.c`

  Wird der `control` Variabel zusätzlich noch den Parameter `adaptive` mitgegeben, wird anstatt `stabilization_attitude.c` die Datei `stabilizaton_adaptive.c` kompiliert und bearbeitet. Wie der Name schon sagt, beinhaltet die Datei den Code für die adaptive Stabilisierung. Auf dieses Thema wird im Kapitel 6 genauer eingegangen.

- `airspeed_adc.c`

  Um die analogen Sensordaten des Differentialdrucksensors auszuwerten und um zu rechnen wird die Datei `airspeed_adc.c` benötigt. Diese wird aufgerufen, sobald im Airframe `USE_AIRSPEED` oder `MESURE_AIRSPEED` definiert ist.

4.3. Anpassungen für Maja

Um ein frisch heruntergeladenes Paparazzi-Projekt für die Drohne Maja lauffähig zu kompilieren, sind einige Anpassungen nötig. Die wichtigsten Änderungen werden hier beschrieben.

4.3.1. Airframe

Das Airframe eines Autopiloten ist die wichtigste Datei überhaupt. In ihr stehen neben vielen anderen Einstellungen alle zu ladenden Module wie Servoparameter und alle nötigen Regelparameter.
Im Airframe muss mindestens folgendes erweitert werden:

```
<firmware name="fixedwing">
<define name="USE_I2C0"/>
<subsystem name="radio_control" type="ppm"/>
   <subsystem name="telemetry" type="transparent">
      <configure name="MODEM_BAUD" value="B9600"/>
   </subsystem>
   <subsystem name="i2c"/>
   <subsystem name="control"/>
   <subsystem name="gps" type="ublox_lea5h">
      <configure name="GPS_BAUD" value="B38400"/>
   </subsystem>
   <subsystem name="navigation"/>
</firmware>

<modules>
   <load name="ins_arduimu_magn.xml"/>
</modules>

<servos>
   <servo name="AILERON_LEFT" no="7" min="1000" neutral="1500" max="2000"/>
   <servo name="ELEVATOR" no="0" min="1000" neutral="1500" max="2000"/>
   <servo name="MOTOR" no="6" min="1000" neutral="1000" max="2000"/>
   <servo name="RUDDER" no="2" min="2000" neutral="1500" max="1000"/>
   <servo name="AILERON_RIGHT" no="3" min="1000" neutral="1500" max="2000"/>
</servos>
```

4.3.2. Flugplan

Im Flugplan sind alle Punkte eingetragen, welche zur Berechnung des Flugkurses benötigt werden. Die `basic.xml` XML-Datei mit dem Standardflugplan wurde auf die Koordinaten vom Modellflugplatz Lommis angepasst. In einer der ersten Zeile werden die "Null-Koordinaten" festgelegt, von denen aus die restlichen Wegpunkte berechnet werden.
Diese Zeile sieht nach der Änderung folgendermassen aus:

```
<flight_plan alt="600" ground_alt="460" lat0="47.515217" lon0="8.975493"
max_dist_from_home="1500" name="Basic" security_height="25">
   <header>
   ...
```

- `alt` - legt die Standardhöhe (Höhe über Meeresspiegel) für die Wegpunkte fest.

- `ground_alt` - let die Höhe des Bodens fest.
- `lat0` - beschreibt den Breitengrad der Nullposition
- `lon0` - beschreibt den Längengrad der Nullposition
- `max_dist_from_home` - legt den Sicherheitsradius um den Homepunkt fest
- `security_height` - bestimmt die Höhe, welche für den Failsave (Circle-Home) benötigt wird

4.3.3. Radios

Damit die Funksignale bei manueller Steuerung direkt an die Servos übermittelt werden können, müssen in der `Radios` Datei die Kanäle richtig zugewiesen werden.
Der folgende Abschnitt zeigt die momentane Einstellung der verwendeten Radios Datei.

```
<radio name="cockpitMM" data_min="900" data_max="2100"
sync_min ="5000" sync_max ="15000" pulse_type="POSITIVE">
   <channel ctl="D" function="PITCH"     min="2000" neutral="1500" max="1000"
   average="0"/>
   <channel ctl="C" function="YAW"    min="2000" neutral="1500" max="1000"
   average="0"/>
   <channel ctl="A" function="THROTTLE" min="1000" neutral="1000" max="2000"
   average="1"/>
   <channel ctl="B" function="ROLL"      min="2000" neutral="1500" max="1000"
   average="0"/>
   <channel ctl="E" function="MODE"     min="2000" neutral="1500" max="1000"
   average="1"/> <!-- Top right switch (E) -->
   <channel ctl="F" function="RES1"     min="2000" neutral="1500" max="1000"
   average="0"/>
   <channel ctl="G" function="RES2"     min="2000" neutral="1500" max="1000"
   average="0"/>
   <channel ctl="H" function="RES3"     min="2000" neutral="1500" max="1000"
   average="0"/>
</radio>
```

4.3.4. Settings

Wenn keine Änderungen im GUI erwünscht sind, kann für die Settings die Datei `tuning_ins.xml` geladen werden. Diese Datei legt fest, welche Parameter im GCS (Ground Control Station) angezeigt bzw. verändert werden können. Nachfolgend ist der Beispielcode abgebildet, welcher die Manipulation der Parameter der Airspeed-Regelung während des Fluges ermöglicht.

```
<!--dl_settings NAME="airspeed">
      <dl_setting MAX="40" MIN="0" STEP="1.0" VAR="v_ctl_auto_airspeed_setpoint"
      shortname="airspeed_setpoint" param="V_CTL_AUTO_AIRSPEED_SETPOINT"
      module="guidance/guidance_v"/>
        <dl_setting MAX="10" MIN="0" STEP="0.01" VAR="v_ctl_auto_airspeed_pgain"
        shortname="airspeed_pgain" param="V_CTL_AUTO_AIRSPEED_PGAIN"/>
        <dl_setting MAX="10" MIN="0" STEP="10" VAR="v_ctl_auto_airspeed_igain"
        shortname="airspeed_igain" param="V_CTL_AUTO_AIRSPEED_IGAIN"/>
</dl_settings-->
```

- `NAME` - definiert den Namen des Tabs, unter welchem die Parameter aufgelistet werden
- `MAX,MIN,STEP` - definiert das Maximum, bzw. Minimum und die Schrittweite des Reglers
- `VAR` - Name der Variabel, welche verändert werden soll
- `shortname` - Name, welcher im GCS angezeigt wird
- `module` - gibt an, wo die Datei liegt, welche die Variabel beinhaltet. Dies muss nur im obersten Parameter definiert werden, wenn sich die folgenden Variablen in der selben Datei befinden.
- `param` - muss definiert werden, wenn die Datei nicht als `module` angegeben wird

4.3.5. Telemetry

In der Telemetriedatei wird festgelegt, in welchem Abstand die Datenpakete (Messages) an das Control-Terminal geschickt werden. Für die Drohne Maja kann die Standard-Datei `default.xml` verwendet werden.

4.3.6. IMU mit Magnetometer

Um die Inertial Measurement Unit (IMU) mit dem Magnetometer laufen zu lassen, muss im Airframe, wie im Unterkapitel 4.3.1 aufgelistet, der das Laden der Datei `ins_arduimu_magn.xml` eingetragen sein. Die Dateien wurden von Emilio Schmidhauser umgeschrieben und später dann im Paparazzi-Projekt zu finden sein.

Die benötigten Dateien sind wie folgt zu platzieren:
`.../sw/airborne/modules/ins/ins_arduimu_magn.c`
`.../sw/airborne/modules/ins/ins_arduimu_magn.h`
`.../conf/modules/ins_arduimu_magn.xml`

5. Regelsysteme

5.1. Alte Regelung

5.1.1. Funktion

Für die Funktion der alten Regelung verweisen wir hier auf die Bachelorarbeit - Flugregelung einer Kleindrohne für Forschungszwecke vom 13. August 2010 von Emilio Schmidhauser und René Chaney

5.1.2. Airspeed-Regelung

In der alten Arbeiten von Emilio Schmidhauser und René Chaney werden drei Geschwindigkeitsregelungen vorgestellt. Zum Einen gibt es die Vasillis-Regelung, welche in der Zwischenzeit weiterentwickelt und verbessert wurde und mit welcher die neuen Testflüge geregelt wurde. Zum Anderen existieren die zwei von Emilio Schmidhauser und René Chaney programmierten Airspeed ONE und TWO. Airspeed ONE baut auf dem selben Prinzip auf wie die Vasillis-Regelung. Bei Airspeed TWO wurde das Prinzip umgedreht. Das heisst, die Höhe wird statt mit dem Höhenruder mit der Motorleistung geregelt und die Geschwindigkeit dafür mit dem Höhenruder. Für eine genauere Funktionsbeschreibung wird hier auf die Bachelorarbeit von Emilio Schmidhauser und René Chaney verwiesen.

5.2. Neue Regelung

Die neue Flugregelung wurde vom C-Code zu Matlabgrafiken übersetzt und dann durch Matlabsymbole vereinfacht. Die übersetzten Matlabgrafiken finden sich im Anhang A.3. Die Abbildung 5.1 zeigt die Paparazzi Haupt-Regelkreise. Diese sind für die Navigation, Steuerung und Kontrolle zuständig.

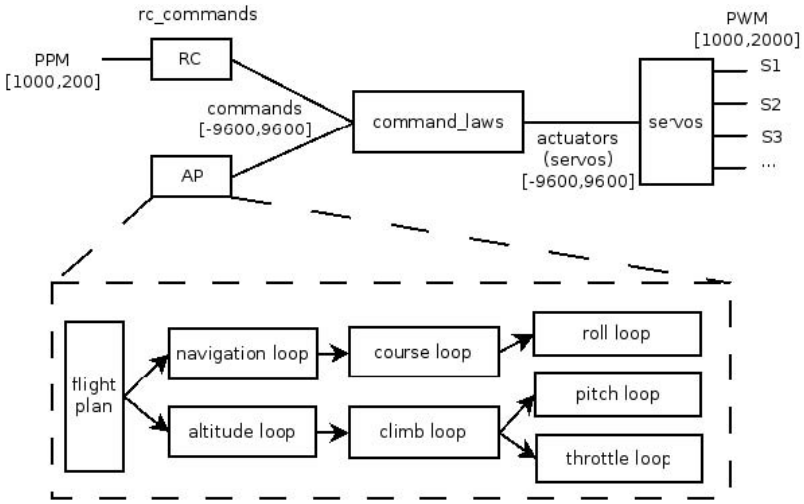


Abbildung 5.1.: Übersicht Paparazzicode

Die Zahl `+/-9600` entspricht dem im Code vorkommendem `+/-MAX_PPRZ`. Diese Paparazzi Einheit wird als genormter Wert als Eingang und Ausgang der Steuerungsblöke verwendet. Die Namen der Variablen sind die selben wie im Programm Code. Werden Variablen mit Grossbuchstaben geschrieben, bedeutet dies, dass ihnen einen festen Werte im Airframe zugeordnet wird.

5.2.1. Navigation Loop

Der Navigation Loop befindet sich im `w/airborne/nav.*`. Die Navigationsroutine wird im Flugplan aufgerufen.

5.2.2. Course Loop

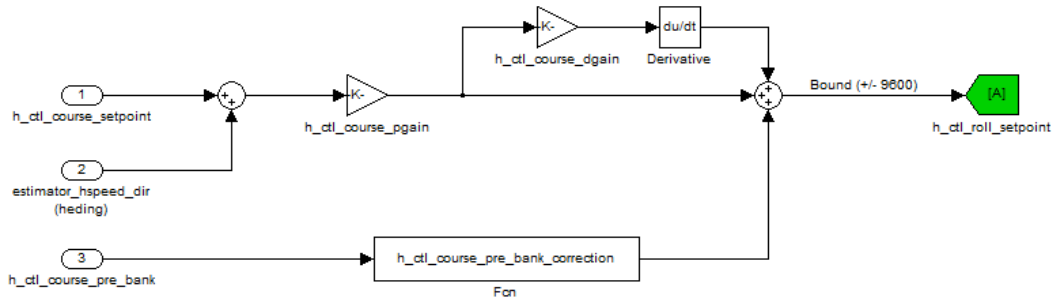


Abbildung 5.2.: Course Loop

Der Course Loop ist der erste Schritt der horizontalen Regelung. Er ist in `sw/airborne/firmwares/fixedwing/stabilization/stabilization_attitude.c` gespeichert. Mit einem PD-Regler wird die Abweichung zwischen dem Soll- und Istwert (Var1 und Var2) der Navigationsrichtung geregelt. Dazu wird ein Korrekturfaktor addiert, der aus der Schätzung der Position des Momentan-Navigationszielpunktes (Var3) berechnet wird.

5.2.3. Roll Loop

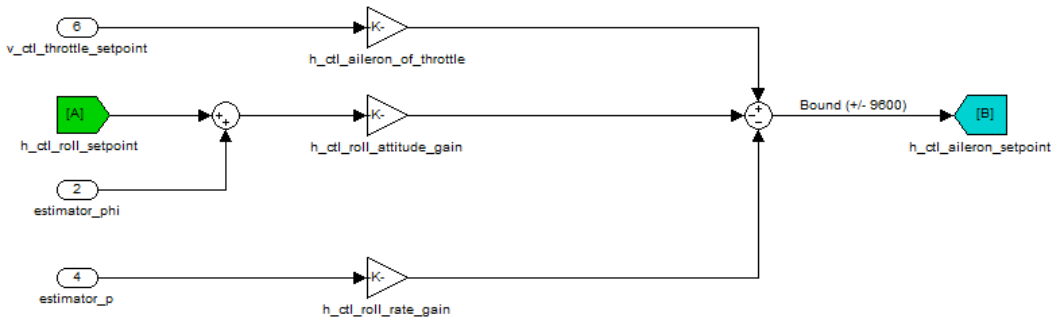


Abbildung 5.3.: Roll Loop

Der Roll Loop ist der letzter Schritt der horizontalen Regelung. Er ist in `stabilization_attitude.c` gespeichert. Mit einem PD-Regler wird das Rollverhalten geregelt. Die Abweichung zwischen Soll- und Istwert (VarA und Var4) wird mit einem Proportionalfaktor verstärket. Der D-Anteil wird mit der Änderungsrate des Neigungswinkels phi (Var5) und mit der `h_ctl_roll_rate_gain` berechnet. Falls der Propeller des Flugzeugs ein grosses Drehmoment auf die Rollachse auswirkt, kann zusätzlich die Querruder-Nullposition in der Funktion `h_ctl_aileron_of_throttle` mit dem Sollwert der Motordrehzahl (Var6) berechnet werden. Da dieser Fall bei

der Maja nicht zutrifft, ist `h_ctl_aileron_of_throttle` auf Null gesetzt und wird somit nicht behandelt. Wenn die Variable `H_CTL_ROLL_ALTITUDE_GAIN` im Airframe nicht definiert ist, wird der D-Anteil des Reglers nicht berücksichtigt. Der Proportionale Verstärkungsfaktor nimmt den Wert der Variable `H_CTL_ROLL_PGAIN` an.

5.2.4. Altitude Loop

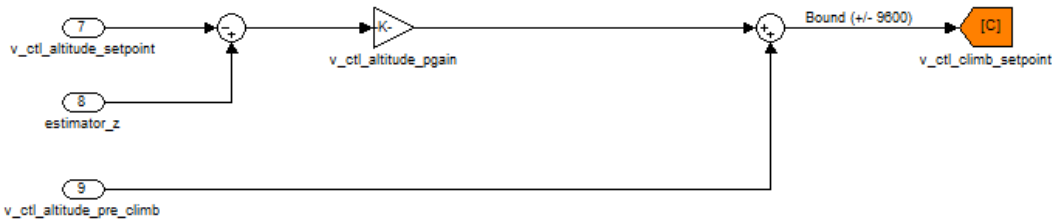


Abbildung 5.4.: Altitude Loop

Der Altitude Loop ist der erste Schritt der vertikalen Regelung. Er ist in `sw/airborne/firmwares/fixedwing/guidance/guidance_v.c` gespeichert. Mit einem P-Regler wird die Abweichung zwischen Soll- und Isthöhe (Var7 und Var8) geregelt. Dazu wird der gewünschte Steigungswert (Var9) Addiert, so dass es möglich ist ein 3D Weg zu fliegen. Die Variable `v_ctl_altitude_pre_climb` besitzt den Wert Null falls die Höhe konstant bleiben soll.

5.2.5. Auto Throttle Climb Loop und Pitch Loop

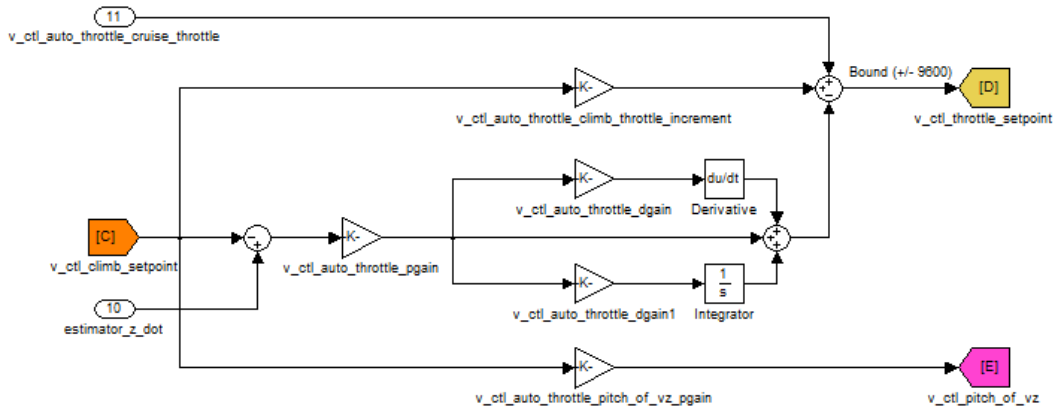


Abbildung 5.5.: Auto Throttle Loop

Mittels einem PID-Regelkreis wird die Motorleistung geregelt. Um eine wirksamer Steuerung zu gestalten, wird eine Vorsteuerung verwendet, so dass eine Erhöhung der Sollleistung schneller erkennbar ist und der Regler früher reagieren kann. Der Auto Throttle Climb Loop ist in `guidance_v.c` zu finden.

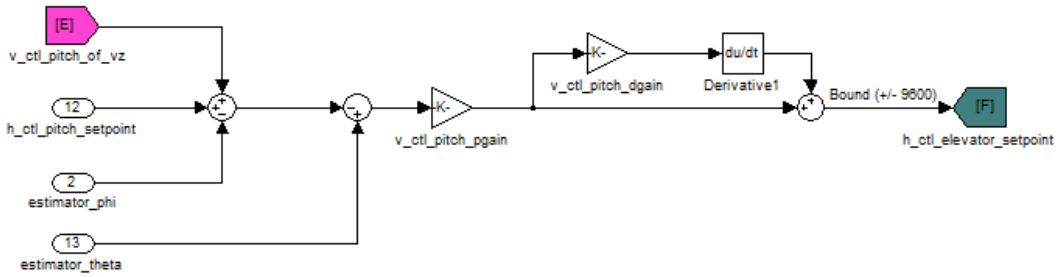


Abbildung 5.6.: Pitch Loop

Im Pitch Loop wird zuerst der Soll-Steigungswert (Var12) mit der Steigungsgeschwindigkeit (VarE) und dem Rollwinkel (Var2) verglichen. So ist zu erkennen, wenn sich das Flugzeug gleichzeitig im Kurven- und Steigflug befindet. Diese Erkennung ist sehr wichtig, weil die Gewichtskraft im Kurvenflug mit einem grösseren Anstellwinkel kompensiert werden muss. Danach wird die Lage des Flugzeug mit dem Istwert (Var13) und dessen Abweichung mit einem PD-Regler geregelt. Der Pitch Loop ist im `stabilization_attitude.c` zu finden.

5.3. Airspeed Regelung

Die Altitude und Airspeed Loops sind getrennt, wie in der Abbildung 5.7 zu erkennen ist. Grundsätzlich werden Throttle und Pitch unabhängig von einander gesteuert, jedoch beeinflussen sie sich wegen den physikalischen Gegebenheiten gegenseitig.

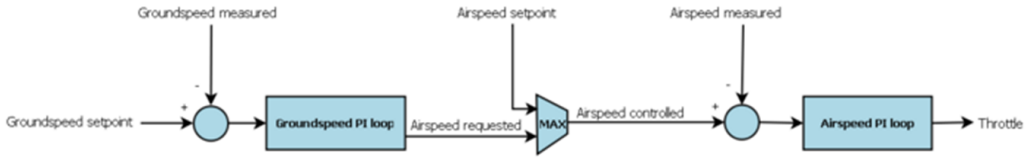


Abbildung 5.7.: Airspeed Control Loop

Die Fluggeschwindigkeit wird durch zwei kaskadierte PI-Regler gesteuert. Der erste wird verwendet, um den Groundspeed zu regeln und der zweite für den Airspeed. Mit der Groundspeed Regelung ist es möglich zu gewährleisten, dass, wenn die Airspeed unter einen bestimmten Wert fällt, die Airspeed-Regelung zu umgehen und mit der Groundspeed-Regelung zu regeln. So wird die Fluggeschwindigkeit positiv gehalten um immer eine gültige GPS Geschwindigkeit zu erhalten.

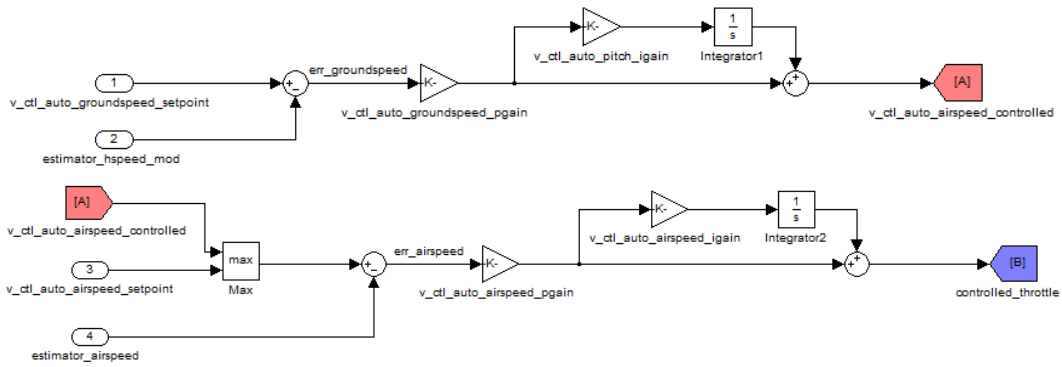


Abbildung 5.8.: Airspeed Control Loop

Diese zwei Regelkreise ersetzen bei aktivierten Airspeed-Regelung den Auto Throttle Climb Loop. Für die vollständige Steuerung wird noch eine Climb Loop Regelung im Vertikale Loop benötigt.

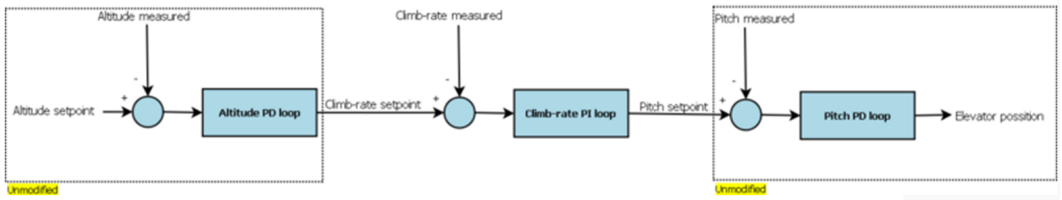


Abbildung 5.9.: Vertikal Control Loop

Aus der Abweichung zwischen Soll- und Istwert (VarC und Var10) wird mit einem PI-Regler der Wert (VarE) erzeugt, der im Pitch Loop verwendet wird, um das Höhenruder zu regeln.

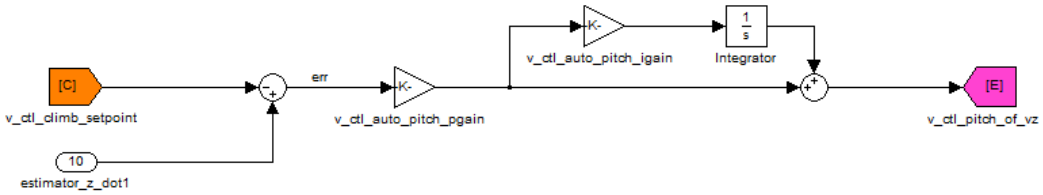


Abbildung 5.10.: Climb-rate Loop

6. Adaptive Regelung

Dieses Kapitel dient als ersten Einblick in den adaptiven Regelungscode, der von der Paparazzi-Comunity geschrieben wurde. Die entsprechende Steuerung befindet sich noch in der Testphase. Infolge wird die adaptive Regelung des Roll Loops dargestellt. Um den Überblick zu wahren, wird hier der Regler vereinfacht (in Matlabsymbolen) dargestellt und erklärt. Im Anhang A.2 ist die Matlab Darstellung des gesamten Codes zu finden, in welchem klar zu sehen ist, wie die benötigten Variabel berechnet oder definiert werden.

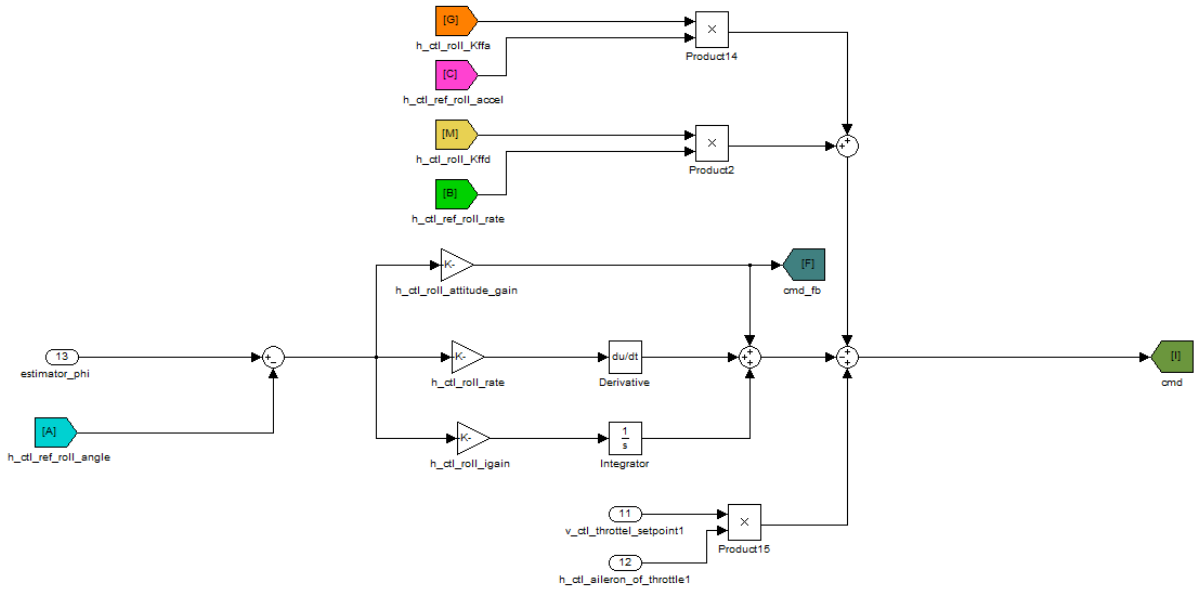


Abbildung 6.1.: Adaptive Voorsteuerung Roll Loop

Hier ist zu sehen, dass es sich um einen Vorsteuerungs-Regler handelt. Aus dem Vergleich zwischen Soll- und Istwert des Rollwinkels wird ein PID-Regler erstellt. Die Variablen `h_ctl_roll_Kffa` und `h_ctl_roll_Kffd` (VarG und VarM) sind die adaptiven Verstärkungsparameter und sorgen für die Vorsteuerung. Diese werden mit den Sollwerten der Rollwinkelbeschleunigung (VarC) und der Rollwinkelgeschwindigkeit (VarB) zusammengerechnet. Die Reglerausgangsvariabel `cmd` wird dann mit einem Filter 2. Ordnung filtriert und danach für die Querrudereinstellung verwendet.

7. Airspeed-Sensor

7.1. Beschreibung

Da das Ziel des gesamten Drohnen-Projekt der ZHAW, mit dem Autopiloten schlussendlich das Unmanned Modular Airborne Research System kurz UMARS zu fliegen ist, muss für die Luftmessung eine konstante Luftgeschwindigkeit vorgegeben sein. Um dies zu erreichen ist die Drohne mit einem Pitotrohr und einem Differentialdrucksensor ausgerüstet. Mit diesen Instrumenten ist es möglich, die Luftgeschwindigkeit um das Flugzeug (Airspeed) zu messen. Die GPS Daten liefern die Position im Bezug des Bodens und das Pitotrohr im Bezug zur Umgebungsluft. Das heisst die Geschwindigkeit gegenüber dem Boden kann negativ werden, obwohl die Umgebungsluftgeschwindigkeit (Airspeed) den korrekten Wert beibehält. Diese Situation wird programmtechnisch abgefangen, indem nach dem Unterschreiten eines gewissen Grenzwertes die Motorleistung unabhängig von der Airspeed mit dem Groundspeed Loop geregelt wird.

7.2. Sensor

Für die Messungen, welche für diese Arbeit durchgeführt wurden, wurde ein Differentialdrucksensor der Firma AMSYS verwendet. Es handelt sich um einen Niederdrucksensor mit der Modellnummer. AMS 4711-0020-D mit einem Druckbereich von 0 bis 20 mbar. Das Analoge Ausgangssignal geht von 0 bis 5 Volt.

7.3. Implementierung

Die Formel[1] für die Berechnung der Geschwindigkeit aus einem Druckunterschied sieht wie folgt aus.

$$v = \sqrt{\frac{2 \cdot \Delta p}{\rho}} \tag{7.1}$$

Um die Rohdaten programmtechnisch einfach verwenden zu können, wird diese Formel Arithmetisch wie folgt angepasst.

$$v = \sqrt{\frac{2}{\rho}} \cdot \sqrt{\Delta p} \tag{7.2}$$

Da die Druckdifferenz Δp nicht als Pascal vorhanden ist, muss diese zuerst umgerechnet werden. 0.02 beschreibt die maximale Druckdifferenz 0.02 Bar (20 mbar). Die 10^5 wandelt den Druck von Bar in Pascal. Mit dem Wert 1025 wird die Auflösung des Analogeingangs angegeben.

$$\Delta p = \frac{Analogwert \cdot 0.02 \cdot 10^5}{1025} \tag{7.3}$$

[1] Quelle: Titel: Grundlagen des Fluges, Autor: Eric Lindemann, Auflage: Februar 2008, Herausgeber: Segelflugverband der Schweiz

Werden die Formeln zusammengefasst erhält man:

$$v = \sqrt{\frac{2}{\rho}} \cdot \sqrt{Analogwert} \cdot \sqrt{\frac{0.02 \cdot 10^5}{1025}} = \sqrt{Analogwert} \cdot \sqrt{\frac{2 \cdot 0.02 \cdot 10^5}{\rho \cdot 1025}} \tag{7.4}$$

Wird nun für die Dichte $\rho = 1.2041 kg/m^3$ eingesetzt, erhält man folgenden Multiplikator.

$$v = \sqrt{Analogwert} \cdot 1.8 \tag{7.5}$$

Dieser Wert wird im Airframe in den Modulen wie folgt definiert:

```
<modules>
   <load name="airspeed_adc.xml">
      <configure name="ADC_AIRSPEED" value="ADC_3"/>
      <define name="AIRSPEED_QUADRATIC_SCALE" value="1.8"/>
<!--define name="AIRSPEED_SCALE" value="1"/-->
         <define name="AIRSPEED_BIAS" value="0"/>
      </load>
</modules>
```

Mit `load` wird das analoge Airspeed-Modul geladen. `ADC_3` beschreibt den Anschluss, an welchem der Sensor angehängt wurde. Der soeben berechnete Wert wird in `AIRSPEED_QUADRATIC_SCALE` geschrieben. Dieser Wert kann je nach Aufbau des Pitotrohrs noch ein bisschen angepasst werden. Ebenfalls hat auch die temperaturabhängige Dichte der Luft einen kleinen Einfluss auf den Multiplikator. Der Offset kann mit `AIRSPEED_BIAS` definiert werden.

8. Flugauswertung

Nach einem Fehlstart, bei dem die Testdrohne Maja beschädigt wurde, konnte zwei Tage später die Drohne mit dem neuen Paparazzi-Code mit aktiviertem `USE_AIRSPEED` geflogen werden. Wird im Airframe `USE_AIRSPEED` gesetzt, wird die Fluggeschwindigkeit im Bezug zur Umgebungsluft geregelt. Alternativ zu `USE_AIRSPEED` kann `MESURE_AIRSPEED` gesetzt werden, was bewirkt, dass die Windgeschwindigkeit zwar gemessen und geloggt wird, jedoch haben diese Daten dann keinen Einfluss auf die Regelung. Die Geschwindigkeit wird denn mit dem Groundspeed geregelt.
Die Messungen werden in diesem Bericht als erste und zweite Messung bezeichnet. Jedoch muss hier erwähnt werden, dass diese Messungen keines Falls die einzigen waren, welche durchgeführt wurden. Dies sind lediglich die professionellsten Messungen, welche brauchbare Informationen im Bezug auf die Airspeed-Messung enthalten.

8.1. Erste Messung

Abbildung 8.1.: Testflug Kreis bei Flugplatz Lommis

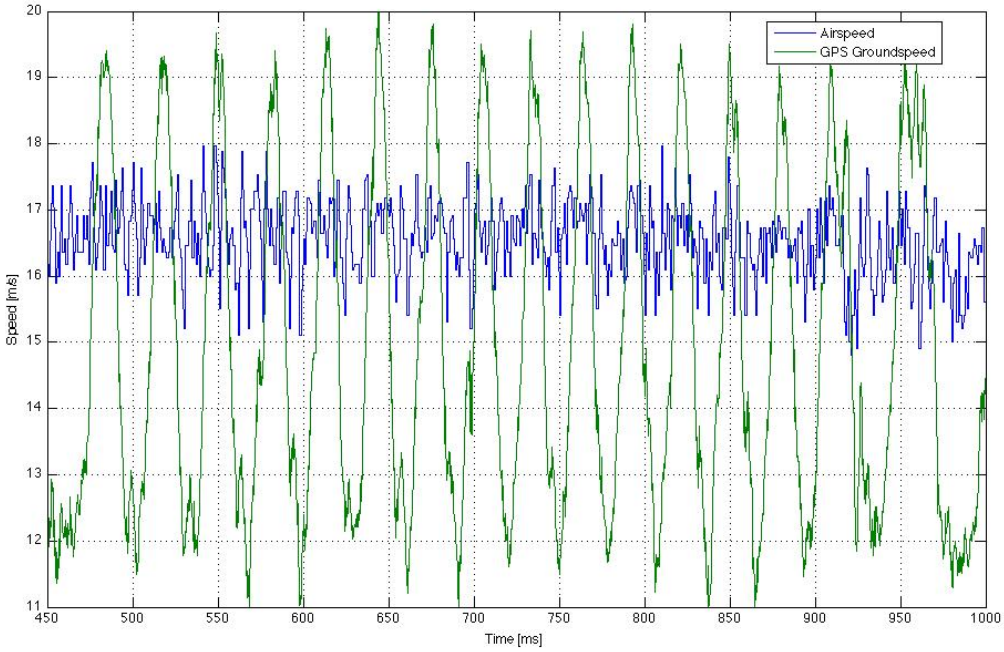


Abbildung 8.2.: Kreisflug mit Airspeed-Regelung

Bei der ersten Messung flog die Testdrohne einen Kreis mit dem Radius von 80 Meter (Abbildung 8.1). Auf der Abbildung 8.2 wird die Luftgeschwindigkeit (Airspeed) im Vergleich zur GPS Geschwindigkeit dargestellt. Obwohl bei diesem Testflug Standardwerte der Regelparamter verwendet wurden, liegt der Wert der Airspeed schon fast in der +/- 1 m/s Toleranz. Der grüne Graph, welcher die Geschwindigkeit gegenüber dem Boden darstellt, weist einen sinusförmigen Verlauf auf. Dies wird auf den Seitenwind zurückgeführt. Fliegt das Flugzeug mit dem Wind, muss es eine höhere Geschwindigkeit gegenüber dem Boden (Groundspeed) erreichen, damit es die gewünschte Umgebungswindgeschwindigkeit (Airspeed) erreicht. Muss das Flugzeug gegen den Seitenwind ankämpfen, braucht es einen kleineren Groundspeed damit der Airspeed erreicht wird.

8.2. Temperatureinflüsse auf Druckdifferenzsensor

Abbildung 8.3.: Höhenverlust bei Testflug (Flughöhe zu zurückgelegtem Weg)

Während der ersten Messung musste eine stetige Abnahme der Flughöhe festgestellt werden. Dieses Phänomen konnte im Verlauf der Messungen auf die Abkühlung des Sensors zurückgeführt werden. Nach dem ersten Flug wurde beim Stillstand eine Windgeschwindigkeit von 5 m/s gemessen.

Dies wurde dann für die nachfolgenden Messungen programmtechnisch korrigiert. Das Problem der steigenden Windgeschwindigkeit hatte einen indirekten Einfluss auf die Flughöhe. Da mit der Zeit eine überhöhte Airspeed gemessen wurde, hatte das Flugzeug zu wenig Geschwindigkeit um an Höhe zu gewinnen. Obwohl die Flugsteuerung das "Gefühl" hatte, dass das Flugzeug schneller wurde, verbohr dies an Geschwindigkeit. Am Anfang der Messungen hatte der Sensor noch Raumtemperatur. Mit der Zeit glich sich diese Temperatur der Umgebungstemperatur von ca. 2°C an. Dieses Problem wird in näheren Zukunft durch einen digitalen Differentialdurcksensor behoben. Der neue Sensor ist Temperaturkompensiert, was eine Messbeeinflussung durch tiefe Temperaturen verhindern sollte.

8.3. Zweite Messung

Abbildung 8.4.: Testflug Oval bei Flugplatz Lommis

Als zweiten Flugplan wurde ein Oval gewählt, da bei dieser Figur das Kurven- und Linienflugverhalten beobachtet werden kann. In der Abbildung 8.4 ist der Weg abgebildet, welcher für den Graphen der Abbildung 8.5 die Messdaten lieferte.

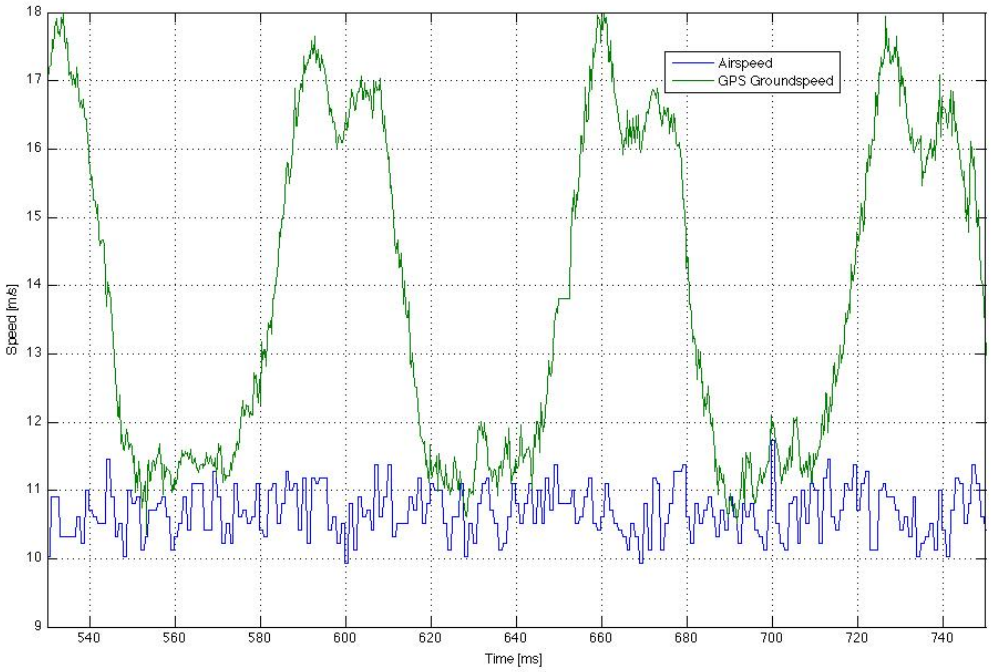


Abbildung 8.5.: Ovalflug mit angepasster Airspeed-Regelung

Nach Anpassungen des Programmcodes der Airspeed-Messung wurde ein zweiter Messflug gestartet. Nach einer Anpassung der `settings.xml` Datei war es uns nun möglich die Soll-Windgeschwindigkeit während des Fluges zu verändern. Die Umgebungswindgeschwindigkeit (Airspeed) ist in der Abbildung 8.5 viel zu tief. Dies sind die Auswirkungen der programmtechnischen Änderungen. Das Flugzeug hatte eine Umgebungswindgeschwindigkeit von ca. 14 m/s, jedoch wurde der Sollwert der Airspeed auf 10.6 m/s gesetzt. Regelungstechnisch hat dies keine Auswirkungen, lediglich der Graph wird "falsch" dargestellt. Diese Anpassungen werden mit dem neuen Digitalsensor nicht mehr benötigt.
Für diese Messung wurde der P-Gain der Airspeed-Regelung verkleinert, was eine deutliche Verbesserung der Konstanz der Windgeschwindigkeit zur Folge hatte. Obwohl das Flugzeug einem ständigem Wechsel der äusseren Einflüsse ausgesetzt war, konnte die Windgeschwindigkeit mit +/- 0.6 m/s gehalten werden.

8.4. Probleme bei der Messung

Wie im Kapitel 8.2 beschrieben wurde, gab es während der Messreihe Probleme mit der Temperatur. Diese Probleme können programmtechnisch gelöst werden, was wir in unserem Fall provisorisch auf dem Feld durchgeführt haben. Andererseits wird dieses Problem durch den neuen digitalen temperaturkompensierten Differentialdrucksensor behoben.
Ebenfalls wurde festgestellt, dass die Option `AGR_CLIMB` im Zusammenhang mit der Option `USE_AIRSPEED` keine Wirkung hat. Es ist sehr wahrscheinlich, dass dieses Problem hautsächlich von den Parameter abhängig ist, welche für den neuen Code neu definiert werden müssen. Zu den Wichtigsten Parameter gehören die `Course` und die `Auto-Throttle`-Parameter. Zum Teil konnte das Flugzeug, wenn der Soll-Airspeed genug hoch definiert wurde, die Höhe sehr genau halten, dabei musste jedoch der `Pitch-P-Gain` sehr stark verkleinert werden.
Die Regelparameter des Course-Loops mussten neu angepasst werden, da die Drohne sonst dem Kreis nicht folgte. Für die Weiterarbeit am Projekt ist auch der `nav_mode` ein Punkt, welcher abgeklärt werden muss. Brauchbare Regelreaktionen konnten nur mit dem `nav_mode` auf 1 erzielt werden.

9. Anpassung der Airspeed-Regelung

9.1. Problembeschreibung

Mit der neuen Airspeed-Regelung lassen sich schnell, gute Regelparameter finden, welche jedoch bei verschiedenen Soll-Fluggeschwindigkeiten angepasst werden müssen. Zwar war dieses Problem beim Testflug nicht offensichtlich, da die Drohne Maja an sich schon ein sehr instabiles Flugverhalten aufweist. Bei der Drohne Maja ist ebenfalls ein Problem, dass sich die Steifheit des Baumaterials extrem von der Umgebungstemperatur beeinflussen lässt. Da schlussendlich jedoch das UMARS mit diesem Regler geregelt wird, kann über dies hinweg gesehen werden. Da die Flugeingenschaften des UMARS viel weniger von der Aussentemperatur beeinflusst werden, können die Regelparameter adaptiv im Bezug zur Soll-Airspeed angepasst werden, ohne die Temperatur miteinzubeziehen.
Wir entwickelten zwei verschiedene adaptive Regelungen, welche die Beeinflussung der Sollwerte überbrücken soll und eine stabile Fluggeschwindigkeit bei jeder Vorgabe bieten soll.

9.2. Lookup

Um die Regler Parameter für Verschiedene Geschwindigkeiten adaptiv anzupassen, kann das System linearisiert werden. Für verschiedene Fluggeschwindigkeiten wird der Verstärkungwert `v_ctl_auto_airspeed_pgain` des Airspeed Loop bestimmt und im Airframe eingetragen.

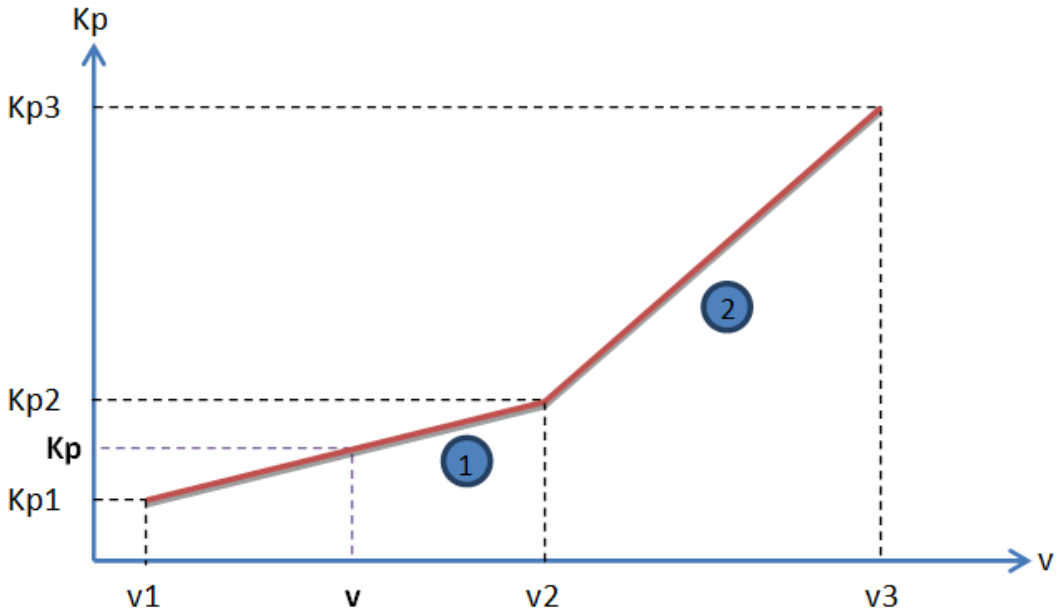


Abbildung 9.1.: Linearisierung einer P-Gain Funktion

Als Beispiel, in Abbildung 9.1 sind drei Arbeitspunkte aufgezeichnet: v1, v2 und v3. Falls die Sollgeschwindigkeit der Airspeed sich im Bereich 1 zwischen v1 und v2 befindet (Analog für Bereich 2), wird der Wert der Verstärkung interpoliert und gesetzt. Somit ist für eine relativ

flexible Regelung nur drei Arbeitspunkte zu bestimmen. Diese Methode lässt sich auf beliebig viele Punkte erweitern.
Dieses Verfahren kann mit folgendem Code implementiert werden.

```
float v1,v2,v3;
float Kp1, Kp2, Kp3;
//Code gehoert in "static viod v_ctl_climb_auto_loop(void)"
//Airspeed calculation
if (v1<=v && v>v2) {
   v_ctl_auto_airspeed_pgain=((kp1-kp2)*v_ctl_auto_airspeed_setpoint-kp1
   *v2+kp2*v1)/(v1-v2);
} if (v2<=v && v>v3) {
   v_ctl_auto_airspeed_pgain=((kp2-kp3)*v_ctl_auto_airspeed_setpoint-kp2
   *v3+kp3*v1)/(v2-v3);
}}
```

9.3. MRAC

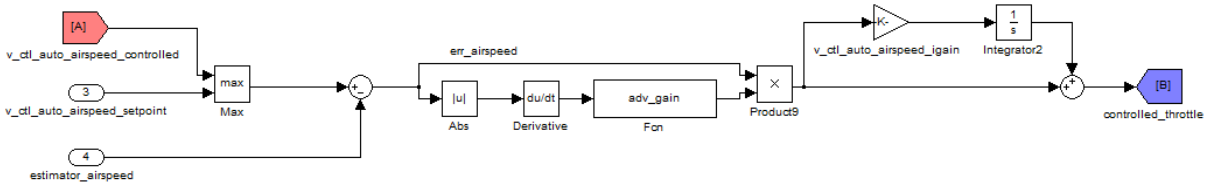


Abbildung 9.2.: Adaptive Airspeed-Regelung mit MRAC

Die zweite Variante ist eine Art Model Reference Adaptive Control (MRAC). Dieser Regler betrachtet die Veränderung der Abweichung zwischen dem Soll und dem Istwert. Die Veränderung wird mit einem Buffer-Array gefiltert damit keine ungewollten schlagartigen Änderungen auftreten. Wieviele der letzten Werte das gebuffert werden sollen, definiert man mit der Variabel `AIR_BUFFER_SIZE`. Danach wird jedes 15te mal geprüft ob als letztes der P-Gain erhöht oder verkleinert wurde. Je nachdem wie der Wert zuletzt verändert wurde, wird der Wert dann "auf die andere Seite" korrigiert. Mit `AIR_BOUNDARY` kann der Grenzwert definiert werden, ab wann der P-Gain verändert wird. Ist `AIR_BOUNDARY` positiv, lässt die Regelung den P-Gain unverändert, bis der Fehler sich verschlechtert. Mit einem negativen `AIR_BOUNDARY` erzwingt man eine ständige Verbesserung des Regelparameters.
Der Parameter `ADV_GAIN` beschreibt, wie stark sich der P-Gain pro Schritt ändern soll. Der Wertebereich liegt zwischen 1 und 2.5. Die IF Bedingung, welche den `airspeed_p_pos` abfragt, überprüft, ob als letztes der P-Gain erhöht oder verkleinert wurde. Ist der `ADV_GAIN` beispielsweise auf 1.6 definiert, wird der P-Gain mit 0.9 multipliziert, also verkleinert. Minimiert sich der Fehler in den nächsten Zyklen nicht, wird der P-Gain mit 1.6 multipliziert, was eine Verstärkung bedeutet.
Ist der `ADV_GAIN` über 1.5 definiert steigt der P-Gain mit der Zeit an, wenn sich keine Verbesserung des Errors zeigt. Bei einem `ADV_GAIN` unter 1.5 nähert sich der P-Gain gegen Null. Dieser Algorithmus soll das Schwingen verhindern. Zusätzlich könnte eine Grenze für den P-Gain definiert werden, welche den `ADV_GAIN` sobald sie überschritten wird.

```
#define AIR_BUFFER_SIZE 50
float airbuffer[AIR_BUFFER_SIZE];
int i;
i = 0;
float abserr;
flaot derr;
float derrhist;
float AIR_BOUNDARY = 0.01;
int airspeed_p_pos = 0;
float ADV_GAIN = 1.1;

//Code gehoert in "inline static void v_ctl_climb_auto_loop(void)"
abserr = abs(err_airspeed);
derr = error - lasterr;
airbuffer [i] = derr;
if (++i >= AIR_BUFFER_SIZE) {
   i = 0;}
sum = 0;
for (i = 0; i < AIR_BUFFER_SIZE; i++) {
     sum += airbuffer[i];
    }
derrhist =  sum /AIR_BUFFER_SIZE;
RunOnceEvery(15, airspeed_adaptive();

//Airspeed calculation
//Code als separate Methode
void airspeed_adaptive(void) {
   if (derrhist > AIR_BOUNDARY) {
      if (airspeed_p_pos > 0) {
         airspeed_p_pos = 0;
         v_ctl_auto_airspeed_pgain = v_ctl_auto_airspeed_pgain * ADV_GAIN;
      }
      else {
         airspeed_p_pos += 1;
         v_ctl_auto_airspeed_pgain = v_ctl_auto_airspeed_pgain * (2.5-ADV_GAIN);
      }
   }
}
```

10. Schreiben eines Treibers

Während der Projektarbeit wurde für den Barometer einen kleinen Treibercode programmiert. Um das zukünftige Treiberschreiben zu erleichtern werden hier kurz die Wichtigsten Punkte beschrieben.

10.1. Beschreibung

Bei dem nachfolgend beschriebenen Treiber handelt es sich um einen Code für die Auswertung eines analogen Barometers von AMSYS. Die Rohdaten sollen an das GCS gesendet werden, um die Werte überprüfen zu können. Die berechneten Daten sollen an den Estimator (Teil des Codes, bei welchem alle Messdaten wie GPS und Winkel zusammenlaufen) gesendet werden, welcher diese dann für die Positionsbestimmung verwenden kann.

10.2. Dateien

Folgende Dateien wurden erstellt oder Verändert:

- `baro_adc.c` - Programmcode
- `baro_adc.h` - Headerfile
- `baro_adc.xml` - Konfigurationsdatei
- `maja.xml` - Airframe
- `messages.xml` - Downlikmeldungen

10.2.1. baro_adc.c und baro_adc.h

Pfad: `.../sw/airborne/modules/sensors/`
Diese Dateien wurden von `airspeed_adc.c` und `airspeed_adc.h` abgeleitet und angepasst. Am Headerfile wurden hauptsächlich Variablen umbenannt. Aus diesem Grund wird diese Datei hier nicht näher beschrieben.
Interessanter ist jedoch der Inhalt der .c Datei. Im ersten Teil werden alle nötigen Header-Dateien geladen. Unter Anderem hat man dann auch Zugriff auf die Variablen, welche im Airframe definiert sind. Damit ein Wert dem Estimator übermittelt werden kann, muss dessen Header ebenfalls verknüpft sein. Um bestimmte Variablen im Messages-Tool Live mitverfolgen zu können, müssen `uart.h`, `messages.h` und `downlink.h` importiert werden.

```
#include "sensors/baro_adc.h"
#include "adc.h"
#include BOARD_CONFIG
#include "generated/airframe.h"
#include "estimator.h"
#include "uart.h"
#include "messages.h"
#include "downlink.h"
```

Im nächsten Abschnitt werden die nötigen Variablen und Argumente deklariert oder definiert.

```
#ifndef DOWNLINK_DEVICE
#define DOWNLINK_DEVICE DOWNLINK_AP_DEVICE
#endif
uint16_t adc_baro_val;
float pressure;
float altitude;
struct adc_buf buf_baro;
```

Als nächstes wird überprüft, ob die Argumente `ADC_CHANNEL_BARO` und `ADC_CHANNEL_BARO_NB_SAMPLES` definiert wurden. Falls dies nicht der Fall ist, erzeugt der eine Befehlt eine Fehlermeldung beim Kompilieren und der Andere definiert einen Standard-Wert.

```
#ifndef ADC_CHANNEL_BARO
#error "ADC_CHANNEL_BARO needs to be defined to use airspeed_adc module"
#endif
#ifndef ADC_CHANNEL_BARO_NB_SAMPLES
#define ADC_CHANNEL_BARO_NB_SAMPLES DEFAULT_AV_NB_SAMPLE
#endif
```

Die folgende Methode beinhaltet den Initialcode. Hier werden Werte auf Null zurückgesetzt und ein Kanal eröffnet.

```
void baro_adc_init( void ) {
pressure = 0;
altitude = 0;
adc_buf_channel(ADC_CHANNEL_BARO, &buf_baro, ADC_CHANNEL_BARO_NB_SAMPLES);
}
```

In dieser Methode werden die Werte am Analogeingang eingelesen und verarbeitet. Danach wird die Variable `altitude` dem Estimator übergeben. Zur Kontrolle wird danach die Variabel noch an die GCS gesendet.

```
void baro_adc_update( void ) {
adc_baro_val = buf_baro.sum / buf_baro.av_nb_sample;
pressure = adc_baro_val / 1024 * 500 + 700;
altitude = (1-pow(pressure * 100 / 1013.25,1/5.255))*288.15 / 0.0065;
EstimatorSetAlt(altitude);
DOWNLINK_SEND_ADC_BARO(DefaultChannel, &adc_baro_val, &pressure, &altitude);
}
```

10.2.2. maja.xml

Im Airframe muss folgenden Eintrag im Abschnitt `modules` hinzugefügt werden. Dabei ist zu beachten, dass der benutzte Analogeingang richtig definiert wurde.

```
<load name="baro_adc.xml">
<define name="ADC_BARO" value="ADC_4"/>
</load>
```

10.2.3. baro_adc.xml

In dieser Datei wird definiert, in welchem Verzeichnis die Programmdateien abgelegt sind. Ebenfalls wird hier festgelegt, welche Methoden zu welchem Zeitpunkt und in welcher Frequenz aufgerufen werden. Auch die Vorgaben für die Überprüfung des Airframes sind hier niedergeschrieben.

```
<module name="baro_adc" dir="sensors">
  <header>
    <file name="baro_adc.h"/>
  </header>
  <init fun="baro_adc_init()"/>
  <periodic fun="airspeed_adc_update()" freq="10."/>
  <makefile>
    <file name="airspeed_adc.c"/>
  </makefile>
  <makefile target="ap">
    <flag name="ADC_CHANNEL_BARO" value="$(ADC_BARO)"/>
    <flag name="USE_$(ADC_BARO)"/>
  </makefile>
</module>
```

10.2.4. messages.xml

Pfad: `.../conf/` Im `messages.xml` muss eine freie ID gefunden werden, um darunter die Nachricht definieren zu können. In diesem Beispiel wird die ID 65 verwendet. In diesem Abschnitt werden alle Variablen aufgelistet, welche im DOWNLINK-Befehl in der .c Datei versendet werden. Ebenfalls müssen dessen Datentypen definiert werden. Nach der Veränderung der `messages.xml` muss ein `make` (neu kompilieren) über das gesamte Paparazzi-Projekt ausgeführt werden.

```
<message name="ADC_BARO" id="65">
   <field name="adc_baro_val" type="uint16" unit=""/>
   <field name="pressure" type="uint16" unit=""/>
   <field name="altitude" type="uint16" unit=""/>
 </message>
```

11. Checkliste

Da während dem Projekt mehrmals Flugzeuge zu Bruch gingen, erstellten wir eine kurze Checkliste, welche zukünftige Schäden vermeiden sollte. Diese sieht zu diesem Zeitpunkt folgendermassen aus.

- Künstlicher Horizont
- GPS
- Altitude
- Launch + Auto 2 = Throttle
- Querruder
- Höhenruder
- Seitenruder
- Fernsteuerung auf Manuell
- Flugrute

Der Sinn der Liste ist, dass vor jedem Start eine Person die Punkte herunterliest und eine Zweite alles prüft. Momentan sind die wichtigsten Punkte aufgeführt, welche je nach Flugzeug oder Situation noch ergänzt werden können.

12. Flashvorgang

Arbeitet man mit dem Paparazzi-Projekt ist es unumgänglich einen Flashvorgang (Projekt auf den Speicher der Drohne laden) durchführen zu müssen. Hier wird kurz der einfache Flashvorgang zum Laden des Codes beschrieben, welcher bei zum Beispiel einer Änderung des Airframes nötig wird.

- Code über das Paparazzi-Programm kompilieren (Kontrolle ob die Kompilation fehlerfrei ausgeführt wurde)
- Stromversorgung des Flugzeugs unterbrechen
- Drohne über ein USB-Kabel mit dem Notebook verbinden
- Stromversorgung des Flugzeugs anschlissen
- Mit Upload den Code auf den Chip laden
- USB-Kabel abhängen
- USB-Kabel an das Funkmodem anschlissen
- Ca. 8 Sekunden warten (Änderung seit neuem Code)
- Richtige Verbindungsart im Dropdownmenue auswählen und Ausführen
- Kontrollieren ob Verbindung zu Stande gekommen ist (Kontrolle in der Terminalausgabe)

13. Fazit

Es ist uns gelungen die neue Airspeed-Regelung erfolgreich in Betrieb zu nehmen. Die Messungen lieferten schnell stabile Geschwindigkeitsdaten, jedoch traten andere Probleme auf, welche noch behoben werden müssen. Zu den Hauptprobleme gehört das nicht vorhandene Verständnis der vielen verschiednen Regelparameter. Als nächster Schritt wäre es hilfreich, wenn zu jeder im GCS einstellbaren Variable (Abbildung A.1) die nötigen Informationen über dessen Wirkung bekannt ist. Sind diese Probleme dann behoben und die Drohne hält die Höhe problemlos, kann bei Bedarf der zusätzlich vorbereitete Code der adaptiven Geschwindigkeits-Regelung eingefügt und getestet werden.
Die Entwicklung der adaptiven Stabilisierung, welche auf der Paparazzi-Comunity in aller Munde ist, soll weiterverfolgt werden, da der momentane Stand vielversprechende Testergebnisse liefert. Mit den erlebten Erfahrungen und Erkenntnisse kann aus unserer Sicht behauptet werden, dass das UMARS in näherer Zukunft autonom mit dem neuen Code fliegen wird.
Die Erfahrungen, welche bei dieser Projekt-Arbeit gemacht wurden, legten das Fundament für die kommende Bachelorarbeit und erleichtern somit dessen Einstieg. Gemachte Erkenntnisse wurden für zukünftige Arbeiten am Autopiloten in dieser Dokumentation ausführlich dokumentiert, damit die Dokumentation später als Nachschlagewerk benutzt werden kann.

A. Anhang

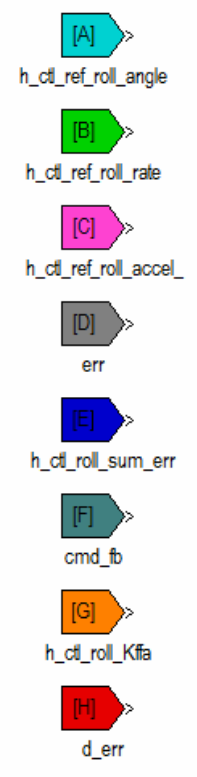
A.1. Variablenliste

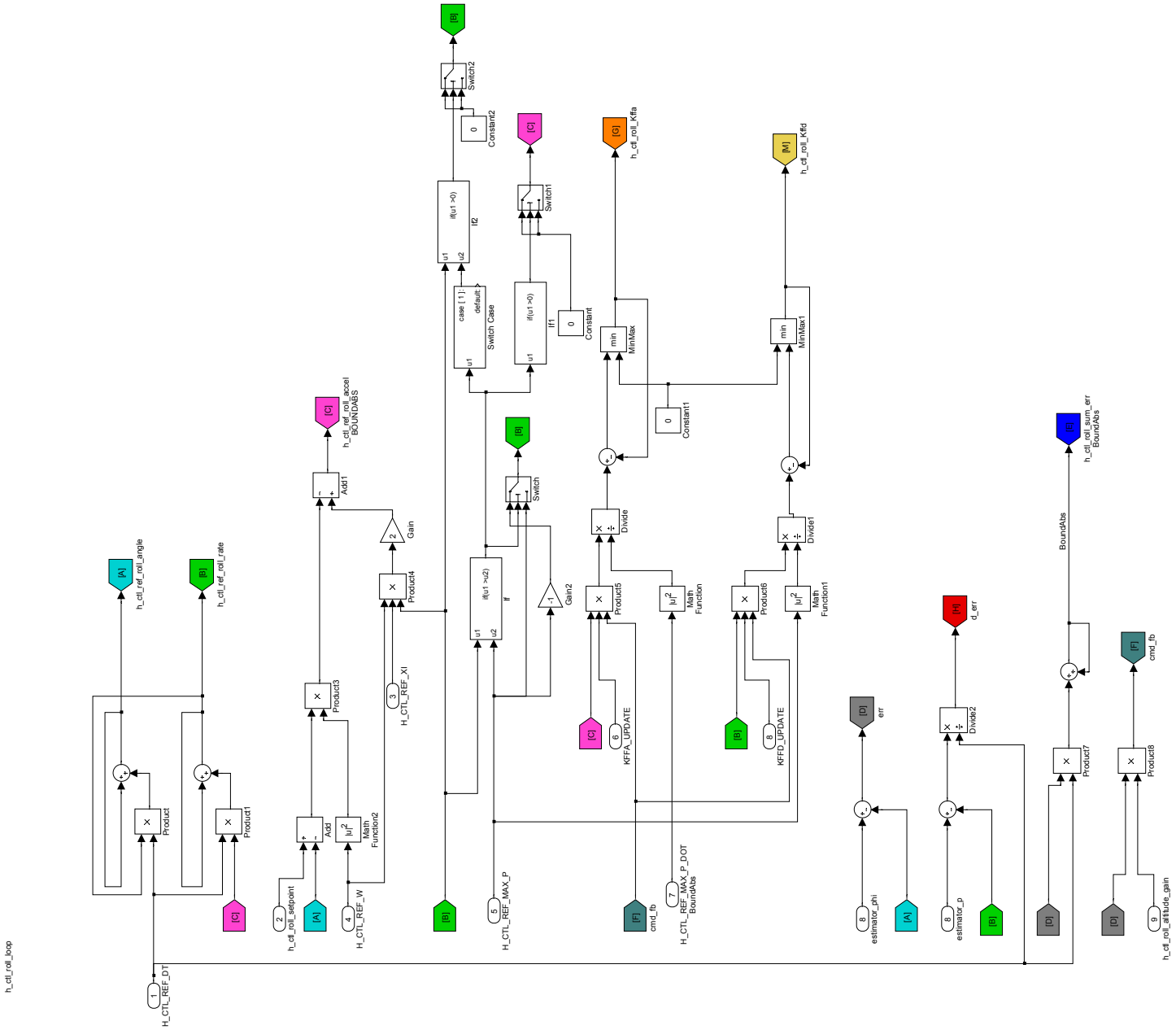
flight params			
	altitude		
	wind_east		
	windnorth		
mode			
	pprz_mode		
	alt_kalman		
	flight time		
	stage_time		
	launch		
	kill_trottle		
	tele_AP		
	tele_FBW		
	GPS reset		
	nav_radius		
control			
	ins		
		roll_neutral	
		pitch_neutral	
	attitude		
		roll_pgain	
		max_roll	
		pitch_pgain	
		pitch_dgain	
		elevator of roll	
		aileron_of_throttle	
		roll attitude pgain	
		roll rate gain	
	alt		
		alt_pgain	
	auto_throttle		
		cruise throttle	
		throttle_pgain	
		throttle_igain	
		throttle_dgain	
		dash trim	
		loiter trim	
		throttle_incr	
		pitch_of_vz	
		pitch_of_vz (d)	
	auto_pitch		
		pgain	
		igain	
	nav		
		crouse pgain	
		crouse dgain	
		pre bank cor	
		glide pitch trim	
		roll slew	
		nav_radius	
		nav_course	
		nav_mode	
		nav_climb	
		fp_pitch	
		inc. shift	
		ground speed	
		ground speed p	
		nav_svrcey_shi	
Modules			
	ArdulMU_priod		

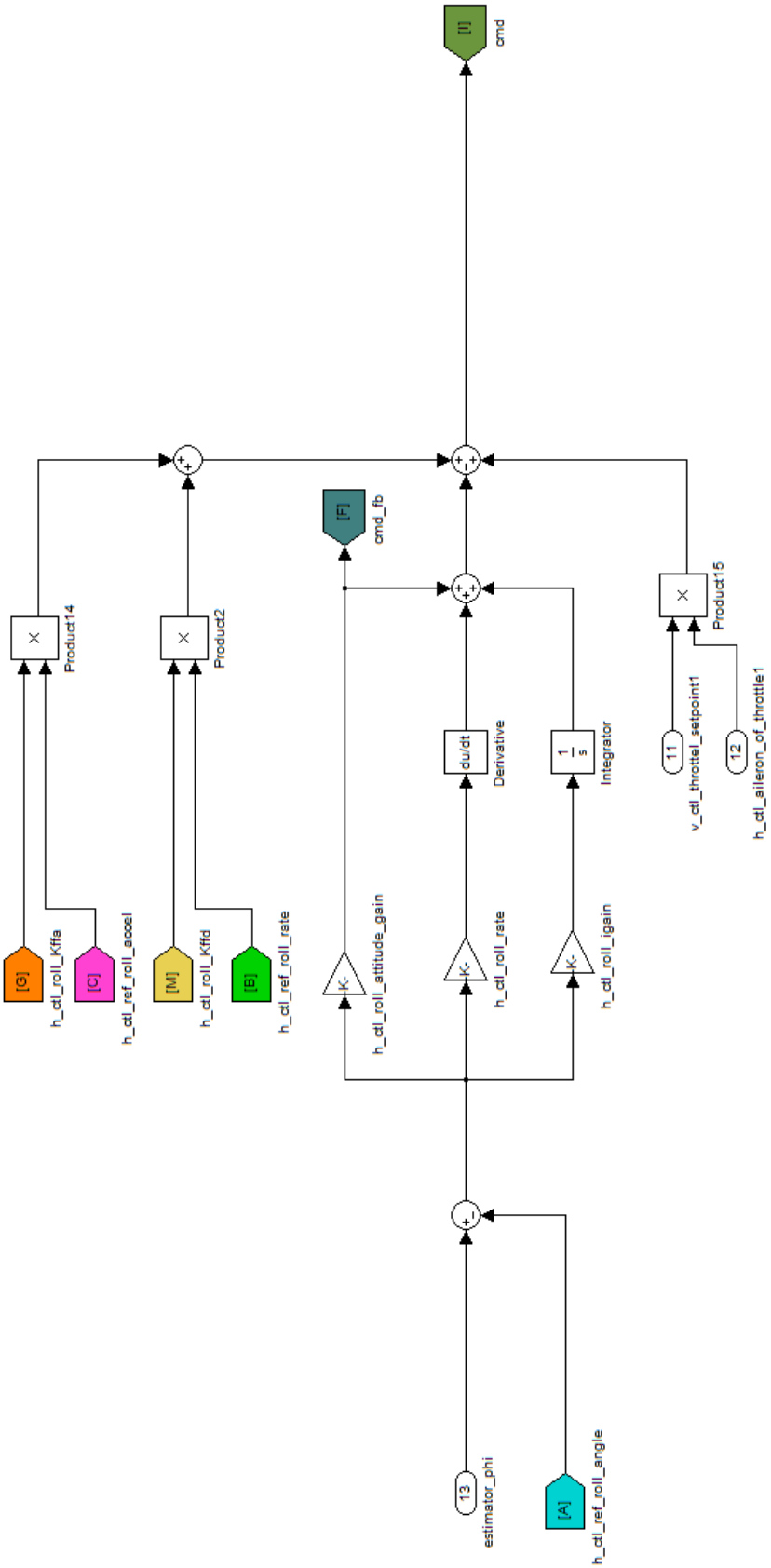
Abbildung A.1.: Liste der Standardparameter im GCS

A.2. Simulink Model adaptive Regelung

Legende




h_ctl_roll_loop
h_ctl_ref_roll_angle
h_ctl_ref_roll_rate
h_ctl_ref_roll_accel BOUNDABS
Product
Product1
Add1
Gain
Product4
Product3
Add
Math Function2
H_CTL_REF_XI
H_CTL_REF_W
h_ctl_roll_setpoint
H_CTL_REF_DT
Switch2
Constant2
if(u1 >0) If2
Switch Case
Switch1
if(u1 >0) If1
Constant
if(u1 >=2) If
Gain2
Switch
H_CTL_REF_MAX_P
h_ctl_roll_Kffa
MinMax
Constant1
Divide
Product5
Math Function
KFFA_UPDATE
cmd_fb
H_CTL_REF_MAX_P_DOT BoundAbs
h_ctl_roll_Kffd
MinMax1
Divide1
Product6
Math Function1
KFFD_UPDATE
err
estimator_phi
d_err
Divide2
estimator_p
h_ctl_roll_sum_err BoundAbs
BoundAbs
Product7
Product8
cmd_fb
h_ctl_roll_altitude_gain


[I]
cmd
[F]
cmd_fb
Product14
Product2
Product15
du/dt
Derivative
1
s
Integrator
11
v_ctl_throttel_setpoint1
12
h_ctl_aileron_of_throttle1
[G]
h_ctl_roll_Kffa
[C]
h_ctl_ref_roll_accel
[M]
h_ctl_roll_Kffd
[B]
h_ctl_ref_roll_rate
K-
h_ctl_roll_attitude_gain
K-
h_ctl_roll_rate
K-
h_ctl_roll_igain
13
estimator_phi
[A]
h_ctl_ref_roll_angle

A.3. C-Code der neuen Regelung als Matlab-Grafiken

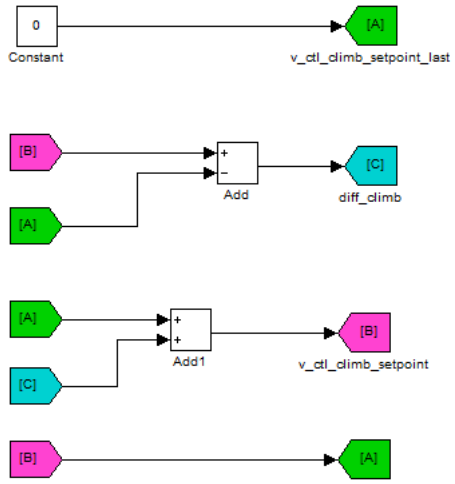


Abbildung A.2.: Limite rate of change of climbe setpoint

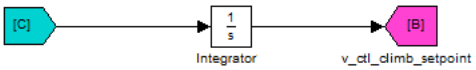


Abbildung A.3.: Limite rate of change of climbe setpoint - übersetzt

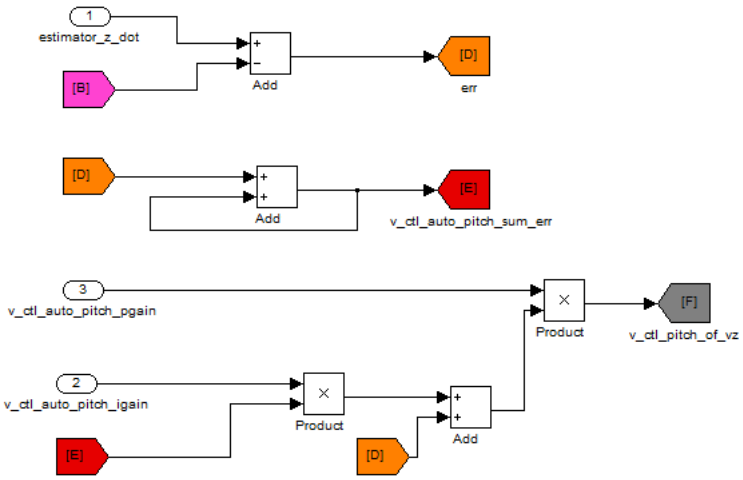


Abbildung A.4.: Pitch control

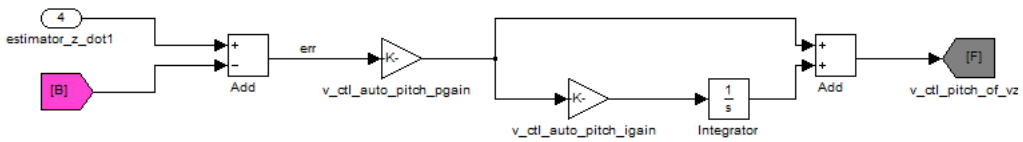


Abbildung A.5.: Pitch control - übersetzt

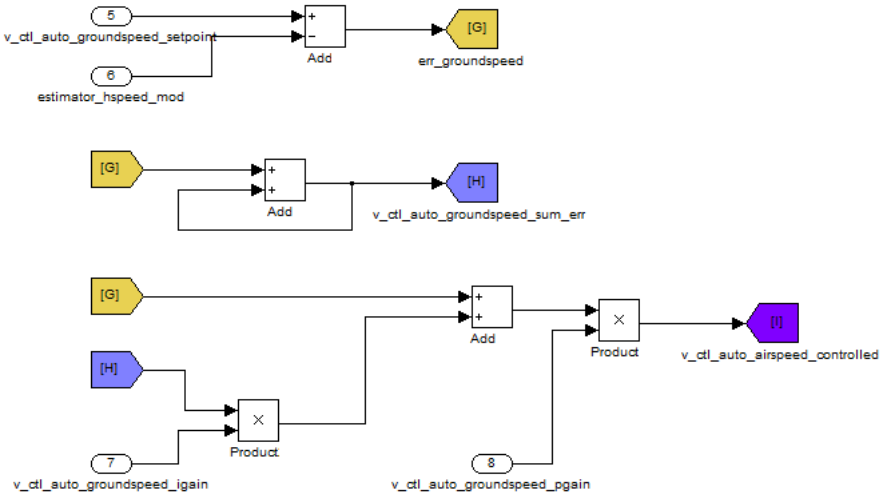


Abbildung A.6.: Ground speed control loop

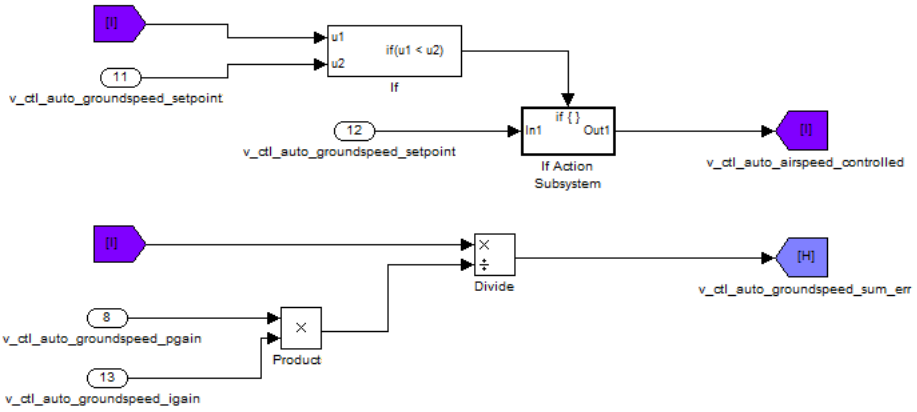


Abbildung A.7.: Do not allow controlled airspeed below setpoint

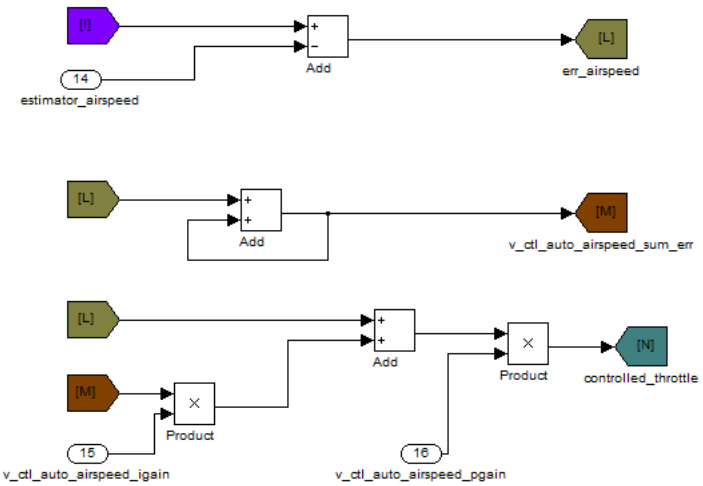


Abbildung A.8.: Airspeed control loop

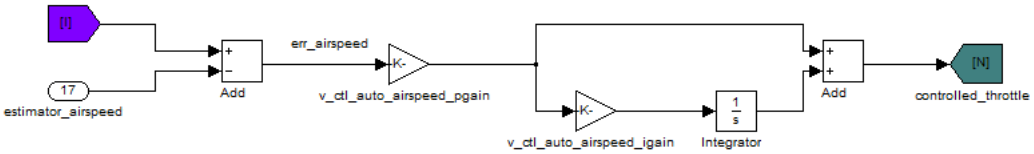


Abbildung A.9.: Airspeed control loop - übersetzt

A.4. Source Code

A.4.1. Neue Regelung

Course Loop

```
void h_ctl_course_loop ( void ) {
  static float last_err;
  // Ground path error
  float err = estimator_hspeed_dir - h_ctl_course_setpoint;
  NormRadAngle(err);
  float advance = cos(err) * estimator_hspeed_mod / reference_advance;
  if (
       (advance < 1.)  &&
       (estimator_hspeed_mod < reference_advance)
     )
  {
    // Heading error
    float herr = estimator_psi - h_ctl_course_setpoint; //+crab);
    NormRadAngle(herr);
    if (advance < -0.5)
    {
      err = herr;
    }
    else if (advance < 0.)           //<!
    {
      err = (-advance)*2. * herr;
    }
    else
    {
      err = advance * err;
    }
    // Reset differentiator when switching mode
    //if (h_ctl_course_heading_mode == 0)
    //  last_err = err;
    //h_ctl_course_heading_mode = 1;
  }
/*  else
  {
    // Reset differentiator when switching mode
    if (h_ctl_course_heading_mode == 1)
      last_err = err;
    h_ctl_course_heading_mode = 0;
  }
```

```
*/
#endif
  float d_err = err - last_err;
  last_err = err;
  NormRadAngle(d_err);
  float speed_depend_nav =
  estimator_hspeed_mod/NOMINAL_AIRSPEED;
  Bound(speed_depend_nav, 0.66, 1.5);
  float cmd = h_ctl_course_pgain *
  speed_depend_nav * (err + d_err * h_ctl_course_dgain);
  float roll_setpoint = cmd + h_ctl_course_pre_bank_correction
  * h_ctl_course_pre_bank;
#ifdef H_CTL_ROLL_SLEW
  float diff_roll = roll_setpoint - h_ctl_roll_setpoint;
  BoundAbs(diff_roll, h_ctl_roll_slew);
  h_ctl_roll_setpoint += diff_roll;
#else
  h_ctl_roll_setpoint = roll_setpoint;
#endif
  BoundAbs(h_ctl_roll_setpoint, h_ctl_roll_max_setpoint);
}
```

Roll Loop

```
inline static void h_ctl_roll_loop( void ) {
  float err = estimator_phi - h_ctl_roll_setpoint;
  float cmd = h_ctl_roll_pgain * err
    + v_ctl_throttle_setpoint * h_ctl_aileron_of_throttle;
  h_ctl_aileron_setpoint = TRIM_PPRZ(cmd);

#ifdef H_CTL_RATE_LOOP
  if (h_ctl_auto1_rate) {
    /** Runs only the roll rate loop */
    h_ctl_roll_rate_setpoint = h_ctl_roll_setpoint * 10.;
    h_ctl_roll_rate_loop();
  } else {
    h_ctl_roll_rate_setpoint = h_ctl_roll_rate_setpoint_pgain * err;
    BoundAbs(h_ctl_roll_rate_setpoint, H_CTL_ROLL_RATE_MAX_SETPOINT);

    float saved_aileron_setpoint = h_ctl_aileron_setpoint;
    h_ctl_roll_rate_loop();
    h_ctl_aileron_setpoint = Blend(h_ctl_aileron_setpoint,
    saved_aileron_setpoint, h_ctl_roll_rate_mode) ;
  }
#endif
}
```

Altitude Loop

```
void v_ctl_altitude_loop( void ) {
  float altitude_pgain_boost = 1.0;
```

```
#if defined(USE_AIRSPEED) && defined(AGR_CLIMB)
  // Aggressive climb mode (boost gain of altitude loop)
  if ( v_ctl_climb_mode == V_CTL_CLIMB_MODE_AUTO_THROTTLE) {
    float dist = fabs(v_ctl_altitude_error);
    altitude_pgain_boost = 1.0 + (V_CTL_AUTO_AGR_CLIMB_GAIN-1.0)
    *(dist-AGR_BLEND_END)/(AGR_BLEND_START-AGR_BLEND_END);
    Bound(altitude_pgain_boost, 1.0, V_CTL_AUTO_AGR_CLIMB_GAIN);
  }
#endif

  v_ctl_altitude_error = estimator_z - v_ctl_altitude_setpoint;
  v_ctl_climb_setpoint = altitude_pgain_boost * v_ctl_altitude_pgain
  * v_ctl_altitude_error + v_ctl_altitude_pre_climb;
  BoundAbs(v_ctl_climb_setpoint, V_CTL_ALTITUDE_MAX_CLIMB);

#ifdef AGR_CLIMB
  if ( v_ctl_climb_mode == V_CTL_CLIMB_MODE_AUTO_THROTTLE) {
    float dist = fabs(v_ctl_altitude_error);
    if (dist < AGR_BLEND_END) {
      v_ctl_auto_throttle_submode = V_CTL_AUTO_THROTTLE_STANDARD;
    }
    else if (dist > AGR_BLEND_START) {
      v_ctl_auto_throttle_submode = V_CTL_AUTO_THROTTLE_AGRESSIVE;
    }
    else {
      v_ctl_auto_throttle_submode = V_CTL_AUTO_THROTTLE_BLENDED;
    }
  }
#endif
}
```

Auto Throttle Climb Loop und Pitch Loop

```
inline static void v_ctl_climb_auto_throttle_loop(void) {
  float f_throttle = 0;
  float controlled_throttle;
  float v_ctl_pitch_of_vz;

  // Limit rate of change of climb setpoint
  //(to ensure that airspeed loop can catch-up)
  static float v_ctl_climb_setpoint_last = 0;
  float diff_climb = v_ctl_climb_setpoint - v_ctl_climb_setpoint_last;
  Bound(diff_climb, -V_CTL_AUTO_CLIMB_LIMIT, V_CTL_AUTO_CLIMB_LIMIT);
  v_ctl_climb_setpoint = v_ctl_climb_setpoint_last + diff_climb;
  v_ctl_climb_setpoint_last = v_ctl_climb_setpoint;

  // Pitch control (input: rate of climb error, output: pitch setpoint)
  float err  = estimator_z_dot - v_ctl_climb_setpoint;
  v_ctl_auto_pitch_sum_err += err;
  BoundAbs(v_ctl_auto_pitch_sum_err, V_CTL_AUTO_PITCH_MAX_SUM_ERR);
```

```
  v_ctl_pitch_of_vz = v_ctl_auto_pitch_pgain *
    (err + v_ctl_auto_pitch_igain * v_ctl_auto_pitch_sum_err);

  // Ground speed control loop (input: groundspeed error,
  output: airspeed controlled)
  float err_groundspeed = (v_ctl_auto_groundspeed_setpoint
   - estimator_hspeed_mod);
  v_ctl_auto_groundspeed_sum_err += err_groundspeed;
  BoundAbs(v_ctl_auto_groundspeed_sum_err,
  V_CTL_AUTO_GROUNDSPEED_MAX_SUM_ERR);
  v_ctl_auto_airspeed_controlled = (err_groundspeed
  + v_ctl_auto_groundspeed_sum_err * v_ctl_auto_groundspeed_igain)
  * v_ctl_auto_groundspeed_pgain;

  // Do not allow controlled airspeed below the setpoint
  if (v_ctl_auto_airspeed_controlled < v_ctl_auto_airspeed_setpoint) {
    v_ctl_auto_airspeed_controlled = v_ctl_auto_airspeed_setpoint;
    v_ctl_auto_groundspeed_sum_err = v_ctl_auto_airspeed_controlled/
    (v_ctl_auto_groundspeed_pgain*v_ctl_auto_groundspeed_igain);
    // reset integrator of ground speed loop
  }

  // Airspeed control loop
  //(input: airspeed controlled, output: throttle controlled)
  float err_airspeed = (v_ctl_auto_airspeed_controlled - estimator_airspeed);
  v_ctl_auto_airspeed_sum_err += err_airspeed;
  BoundAbs(v_ctl_auto_airspeed_sum_err, V_CTL_AUTO_AIRSPEED_MAX_SUM_ERR);
  controlled_throttle = (err_airspeed + v_ctl_auto_airspeed_sum_err
  * v_ctl_auto_airspeed_igain) * v_ctl_auto_airspeed_pgain;

  // Done, set outputs
  Bound(controlled_throttle, 0, V_CTL_AUTO_THROTTLE_MAX_CRUISE_THROTTLE);
  f_throttle = controlled_throttle;
  nav_pitch = v_ctl_pitch_of_vz;
  v_ctl_throttle_setpoint = TRIM_UPPRZ(f_throttle * MAX_PPRZ);
  Bound(nav_pitch, V_CTL_AUTO_PITCH_MIN_PITCH, V_CTL_AUTO_PITCH_MAX_PITCH);
}
```

A.4.2. Airspeed Regelung

```
inline static void v_ctl_climb_auto_throttle_loop(void) {
  float f_throttle = 0;
  float controlled_throttle;
  float v_ctl_pitch_of_vz;

  // Limit rate of change of climb setpoint
  //(to ensure that airspeed loop can catch-up)
  static float v_ctl_climb_setpoint_last = 0;
  float diff_climb = v_ctl_climb_setpoint - v_ctl_climb_setpoint_last;
  Bound(diff_climb, -V_CTL_AUTO_CLIMB_LIMIT, V_CTL_AUTO_CLIMB_LIMIT);
  v_ctl_climb_setpoint = v_ctl_climb_setpoint_last + diff_climb;
```

```
  v_ctl_climb_setpoint_last = v_ctl_climb_setpoint;

  // Pitch control (input: rate of climb error, output: pitch setpoint)
  float err  = estimator_z_dot - v_ctl_climb_setpoint;
  v_ctl_auto_pitch_sum_err += err;
  BoundAbs(v_ctl_auto_pitch_sum_err, V_CTL_AUTO_PITCH_MAX_SUM_ERR);
  v_ctl_pitch_of_vz = v_ctl_auto_pitch_pgain *
    (err + v_ctl_auto_pitch_igain * v_ctl_auto_pitch_sum_err);

  // Ground speed control loop
  //(input: groundspeed error, output: airspeed controlled)
  float err_groundspeed = (v_ctl_auto_groundspeed_setpoint
  - estimator_hspeed_mod);
  v_ctl_auto_groundspeed_sum_err += err_groundspeed;
  BoundAbs(v_ctl_auto_groundspeed_sum_err,
  V_CTL_AUTO_GROUNDSPEED_MAX_SUM_ERR);
  v_ctl_auto_airspeed_controlled = (err_groundspeed
  + v_ctl_auto_groundspeed_sum_err * v_ctl_auto_groundspeed_igain)
  * v_ctl_auto_groundspeed_pgain;

  // Do not allow controlled airspeed below the setpoint
  if (v_ctl_auto_airspeed_controlled < v_ctl_auto_airspeed_setpoint) {
    v_ctl_auto_airspeed_controlled = v_ctl_auto_airspeed_setpoint;
    v_ctl_auto_groundspeed_sum_err = v_ctl_auto_airspeed_controlled/
    (v_ctl_auto_groundspeed_pgain*v_ctl_auto_groundspeed_igain);
    // reset integrator of ground speed loop
  }

  // Airspeed control loop
  //(input: airspeed controlled, output: throttle controlled)
  float err_airspeed = (v_ctl_auto_airspeed_controlled
  - estimator_airspeed);
  v_ctl_auto_airspeed_sum_err += err_airspeed;
  BoundAbs(v_ctl_auto_airspeed_sum_err,
  V_CTL_AUTO_AIRSPEED_MAX_SUM_ERR);
  controlled_throttle = (err_airspeed + v_ctl_auto_airspeed_sum_err
  * v_ctl_auto_airspeed_igain) * v_ctl_auto_airspeed_pgain;

  // Done, set outputs
  Bound(controlled_throttle, 0, V_CTL_AUTO_THROTTLE_MAX_CRUISE_THROTTLE);
  f_throttle = controlled_throttle;
  nav_pitch = v_ctl_pitch_of_vz;
  v_ctl_throttle_setpoint = TRIM_UPPRZ(f_throttle * MAX_PPRZ);
  Bound(nav_pitch, V_CTL_AUTO_PITCH_MIN_PITCH, V_CTL_AUTO_PITCH_MAX_PITCH);
}
```

A.4.3. Adaptive Regelung

```
void h_ctl_course_loop ( void ) {
  static float last_err;

  // Ground path error
  float err = estimator_hspeed_dir - h_ctl_course_setpoint;
  NormRadAngle(err);

  float d_err = err - last_err;
  last_err = err;
  NormRadAngle(d_err);

  float speed_depend_nav = estimator_hspeed_mod/NOMINAL_AIRSPEED;
  Bound(speed_depend_nav, 0.66, 1.5);

  h_ctl_roll_setpoint = h_ctl_course_pre_bank_correction * h_ctl_course_pre_bank
    + h_ctl_course_pgain * speed_depend_nav * err
    + h_ctl_course_dgain * d_err;

  BoundAbs(h_ctl_roll_setpoint, h_ctl_roll_max_setpoint);
}

static float airspeed_ratio2;

static inline void compute_airspeed_ratio( void ) {
    float throttle_diff =  v_ctl_throttle_setpoint / (float)MAX_PPRZ
    - v_ctl_auto_throttle_nominal_cruise_throttle;
    float airspeed = NOMINAL_AIRSPEED; /* Estimated from the throttle */
    if (throttle_diff > 0)
      airspeed += throttle_diff / (V_CTL_AUTO_THROTTLE_MAX_CRUISE_THROTTLE
      - v_ctl_auto_throttle_nominal_cruise_throttle)
      * (MAXIMUM_AIRSPEED - NOMINAL_AIRSPEED);
    else
      airspeed += throttle_diff / (v_ctl_auto_throttle_nominal_cruise_throttle
      - V_CTL_AUTO_THROTTLE_MIN_CRUISE_THROTTLE)
      * (NOMINAL_AIRSPEED - MINIMUM_AIRSPEED);

    float airspeed_ratio = airspeed / NOMINAL_AIRSPEED;
    Bound(airspeed_ratio, 0.5, 2.);
    airspeed_ratio2 = airspeed_ratio*airspeed_ratio;
}

void h_ctl_attitude_loop ( void ) {
  if (!h_ctl_disabled) {
    // compute_airspeed_ratio();
    h_ctl_roll_loop();
    h_ctl_pitch_loop();
  }
}
```

```
#define H_CTL_REF_DT (1./60.)
#define KFFA_UPDATE 0.1
#define KFFD_UPDATE 0.05

inline static void h_ctl_roll_loop( void ) {

  static float cmd_fb = 0.;

  if (pprz_mode == PPRZ_MODE_MANUAL)
    h_ctl_roll_sum_err = 0;

  // Update reference setpoints for roll
  h_ctl_ref_roll_angle += h_ctl_ref_roll_rate * H_CTL_REF_DT;
  h_ctl_ref_roll_rate += h_ctl_ref_roll_accel * H_CTL_REF_DT;
  h_ctl_ref_roll_accel = H_CTL_REF_W*H_CTL_REF_W
  * (h_ctl_roll_setpoint - h_ctl_ref_roll_angle) - 2*H_CTL_REF_XI
  *H_CTL_REF_W * h_ctl_ref_roll_rate;
  // Saturation on references
  BoundAbs(h_ctl_ref_roll_accel, H_CTL_REF_MAX_P_DOT);
  if (h_ctl_ref_roll_rate > H_CTL_REF_MAX_P) {
    h_ctl_ref_roll_rate = H_CTL_REF_MAX_P;
    if (h_ctl_ref_roll_accel > 0.) h_ctl_ref_roll_accel = 0.;
  }
  else if (h_ctl_ref_roll_rate < - H_CTL_REF_MAX_P) {
    h_ctl_ref_roll_rate = - H_CTL_REF_MAX_P;
    if (h_ctl_ref_roll_accel < 0.) h_ctl_ref_roll_accel = 0.;
  }

#ifdef USE_KFF_UPDATE
  // update Kff gains
  h_ctl_roll_Kffa -= KFFA_UPDATE * h_ctl_ref_roll_accel * cmd_fb
  / (H_CTL_REF_MAX_P_DOT*H_CTL_REF_MAX_P_DOT);
  h_ctl_roll_Kffd -= KFFD_UPDATE * h_ctl_ref_roll_rate  * cmd_fb
  / (H_CTL_REF_MAX_P*H_CTL_REF_MAX_P);
#ifdef SITL
  printf("%f %f %f\n", h_ctl_roll_Kffa, h_ctl_roll_Kffd, cmd_fb);
#endif
  h_ctl_roll_Kffa = Min(h_ctl_roll_Kffa, 0);
  h_ctl_roll_Kffd = Min(h_ctl_roll_Kffd, 0);
#endif

  // Compute errors
  float err = estimator_phi - h_ctl_ref_roll_angle;
#ifdef SITL
  static float last_err = 0;
  estimator_p = (err - last_err)/(1/60.);
  last_err = err;
#endif
  float d_err = (estimator_p - h_ctl_ref_roll_rate) / H_CTL_REF_DT;
```

```
  h_ctl_roll_sum_err += err * H_CTL_REF_DT;
  BoundAbs(h_ctl_roll_sum_err, H_CTL_ROLL_SUM_ERR_MAX);

  cmd_fb = h_ctl_roll_attitude_gain * err;// + h_ctl_roll_rate_gain * derr;
  float cmd = h_ctl_roll_Kffa * h_ctl_ref_roll_accel
    + h_ctl_roll_Kffd * h_ctl_ref_roll_rate
    - cmd_fb
    - h_ctl_roll_rate_gain * d_err
    - h_ctl_roll_igain * h_ctl_roll_sum_err
    + v_ctl_throttle_setpoint * h_ctl_aileron_of_throttle;
  //float cmd = h_ctl_roll_Kffp * h_ctl_ref_roll_accel
  //  + h_ctl_roll_Kffd * h_ctl_ref_roll_rate
  //  - h_ctl_roll_attitude_gain * err
  //  - h_ctl_roll_rate_gain * derr
  //  - h_ctl_roll_igain * h_ctl_roll_sum_err
  //  + v_ctl_throttle_setpoint * h_ctl_aileron_of_throttle;

  //x  cmd /= airspeed_ratio2;

  // Set aileron commands
  h_ctl_aileron_setpoint = TRIM_PPRZ(cmd);
}
```

B. Abbildungsverzeichnis