CHALMERS
UNIVERSITY OF TECHNOLOGY

KandiDrone – Ett steg närmare autonom flygning

Utveckling och tester av autonoma funktioner för avsökning av område med quadrocopter

Kandidatarbete

Adam Josefsson
Andreas Johansson
Emil Rosenberg
Joachim Benjaminsson
Karl Svensson

Kandidatarbete 2015

KandiDrone – Ett steg närmare autonom flygning

Utveckling och tester av autonoma funktioner för avsökning av område med quadrocopter

ADAM JOSEFSSON
ANDREAS JOHANSSON
EMIL ROSENBERG
JOACHIM BENJAMINSSON
KARL SVENSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Avdelningen för signaler och system
Chalmers tekniska högskola
Göteborg, Sverige 2015

KandiDrone – Ett steg närmare autonom flygning
Utveckling och testning av autonoma funktioner för avsökning av område med quadrocopter

ADAM JOSEFSSON
ANDREAS JOHANSSON
EMIL ROSENBERG
JOACHIM BENJAMINSSON
KARL SVENSSON

Handledare: Patrik Bergagård, Signaler och System
Examinator: Petter Falkman, Signaler och System

Kandidatarbete 2015
Avdelningen för signaler och system
Chalmers tekniska högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Framsida: Quadrocoptern Ar.Drone 2.0 tillverkad av företaget Parrot SA. Publicerad med tillstånd av företaget Parrots svenska importör Ingeniörsfirman M Sjöberg AB.

Sammandrag

Under de senaste åren har flygfarkost-typen *quadrocopter* blivit allt mer känd hos allmänheten. Detta beror delvis på att konsumentvänliga och billiga produkter har introducerats på den privata marknaden. Manuell manövrering av en quadrocopter kräver dock mycket träning vilket i många fall gör att quadrocoptrar inte används trots att deras flexibilitet hade kunnat förenkla och möjliggöra nya typer av arbetsuppgifter. Således är det av stort intresse att utveckla väl fungerande styrsystem som möjliggör autonom användning av quadrocoptrar. På så sätt kan fokus flyttas från att manövrera quadrocoptern till att bestämma vad quadrocoptern ska utföra.

Den här rapporten är resultatet av ett kandidatarbete vid Chalmers tekniska högskola. Arbetets mål var att utveckla en plattform för autonom styrning av quadrocoptrar. Utifrån det definierades under arbetets gång ett användarscenario för en quadrocopter där grundläggande funktioner kopplades samman för att skapa en autonom sekvens. Användarscenariot innebar att quadrocoptern skulle flyga över ett område för att detektera markörer som låg på marken och sända deras koordinater tillbaka till styrdatorn.

För att implementera dessa funktioner utvecklades en modul, kallad *KandiDrone*, i programmeringsspråket JAVASCRIPT för plattformen NODE.JS. Modulen baserade sig på öppen källkod och kördes på en dator som tog emot och behandlade navigationsdata från, samt skickade styrkommandon till quadrocoptern.

De uppsatta målen uppfylldes delvis då de olika delmålen implementerades i kod och borde fungera i teorin. Tyvärr innebar hårdvaruproblem att ett praktiskt utförande av användarscenariot inte kunde genomföras med ett bra resultat.

Konceptet ansågs dock ha en stor förbättringspotential och med vidare utveckling skulle användning av quadrocoptrar för automatiserade avsökningar vara fullt möjlig.

Nyckelord: UAV, quadcopter, quadrocopter, AR.Drone, autonom, robotik, javascript, node.js.

Abstract

The unmanned aerial vehicle called quadrocopter have become more popular among the commercial market during the last years. Mostly because of less expensive and user friendly models which has been the key when introducing it to the commercial market. However, the maneuvering is quite complicated and a lot of practice is required which is why quadrocopters has not been used extensively, even though the flexibility and simplicity of the quadrocopter would enable new types of duties. Thus, it is in a great interest to develop a well functioning control system to enable autonomous maneuvering. In that way, focus can be on deciding the tasks for the quadrocopter, instead of on how to maneuver it.

This bachelor thesis was conducted at Chalmers University of Technology and its aim was to develop a platform for autonomously control a quadrocopter. During the project a user scenario for a quadrocopter was created, where basic functions were intertwined in order to create an autonomous sequence. The scenario involved having a quadrocopter fly over an area in order to detect tags on the ground and report their respective positions to a base station.

In order to implement these functions a Javascript module was developed for the platform Node.js. The module was based upon open source code and ran on a computer which received navigation data from the quadrocopter. The computer then calculated and returned control commands to the quadrocopter.

Since desired functionalities were implemented in code, the main goal was partly fulfilled. Sadly however, hardware related problems, mainly yaw drift, meant that the user scenario could not be executed with a satisfying result.

The concept was considered to have a great improvement potential and further development would enable an extended use of autonomous quadrocopters.

Keywords: UAV, quadcopter, quadrocopter, AR.Drone, autonomous, robotics, javascript, node.js.

Förord

Stora delar av det här projektet är baserat på öppen källkod skapad av Felix Geisendörfer och Laurent Eschenauer, även kända under användarnamnen *felixge* och *eschnou* på hemsidan GitHub. Vi är mycket tacksamma för det arbete de lagt ner till glädje för andra.

Vi vill även tacka vår projekthandledare Patrik Bergagård för att han vid våra handledningstillfällen tagit sig tid att lyssna på våra tankar och sedan ställt givande motfrågor som tagit oss närmre lösningarna på de otaliga problem som uppstått under projektet.

Slutligen tackar vi Hans Malmström på avdelningen för fackspråk och kommunikation på Chalmers för hans värdefulla synpunkter på vår rapport.

Adam Josefsson

Andreas Johansson

Emil Rosenberg

Joachim Benjaminsson

Karl Svensson

Göteborg, 16 maj 2015

Innehåll

Figurer

Tabeller

1

Inledning

1.1 Bakgrund

På senare år har intresset för obemannade flygfarkoster (eng. *Unmanned Aerial Vehicles*, förkortat *UAV*) ökat världen över [1]. I en rapport från 2013 [2] bedömde analysföretaget Teal Group att den globala marknaden för UAV:s inom en tioårsperiod skulle komma att omsätta 89 miljarder USD. Aerospace America förutspår i sin rapport *UAV Roundup 2013* [3] att den privata marknaden för UAV:s inom kort kommer att växa explosionsartat och bli mer omfattande än den i dagsläget större militära marknaden. Dessutom förutspår den amerikanska rapporten att försäljning inom den militära sektorn kommer öka till 61,47 miljarder USD mellan år 2011-2020. Det motsvarar en ökning med 60 % från år 2010 [3].

Möjliga användningsområden är bland andra att övervaka landområden och underlätta vid räddningsinsatser. I en spansk studie [4] användes en UAV för att inventera grödors tillväxt. Andra studier har med hjälp av UAV:er lokaliserat fallna träd i Japan [5], genomfört magnetiska undersökningar i Antarktis [6] och kartlagt urbana miljöer i Hamburg [7].

UAV:ernas potentiella mångsidighet till trots, har kritiska röster höjts kring en oro för att individer ska övervakas. I artikeln *Why Commercial Drones Are the Best or Worst Things to Happen to the World In a Long Time* skriver M. Rock [8] om de för- och nackdelar som finns med ett ökat och friare kommersiellt användande av UAV:er. Utöver frågan om den personliga integriteten nämns i artikeln att kommersiella UAV:er riskerar att störa vanlig flygtrafik. Förespråkare för UAV:er menar att risken för störningar är liten då UAV:er flyger på en betydligt lägre höjd än passagerarflygplan. De menar också att det redan vidsträckta användandet av smarta telefoner och sociala medier innebär en större övervakning än vad UAV:er kommer att kunna åstadkomma [9].

Den sorts UAV som användes i projektet var en så kallad *quadrocopter* som, vilket dess namn antyder, har fyra propellrar. En quadrocopter styrs i normala fall med en fjärrkontroll med vilken operatören kan reglera höjd och position. Att lära sig styra en quadrocopter är dock en tidskrävande process som gör att en quadrocopter inte kan användas av personer som saknar nödvändig kunskap. Ett sätt att komma

runt detta problem är att använda autonoma styrsystem som sköter stabilisering och förflyttning och istället låter användaren fokusera på att definiera och utforma arbetsuppgifter för quadrocoptern.

För att kunna implementera autonoma quadrocoptrar i vardagen måste deras styrning vara stabil och fungera i oförutsägbara miljöer där hinder kan dyka upp utan förvarning och starka vindbyar kan flytta quadrocoptern från sin bana. Att kunna undvika hinder är viktigt för att quadrocoptern inte ska vara en säkerhetsrisk för människor eller närmiljö, eftersom allvarliga skador kan orsakas av bland annat rotorbladen [10].

Det här projektet inspirerades av ett användarscenario från Volvo, där en quadrocopter ska lokalisera avfallsbehållare och soptunnor och rapportera deras positioner till sin sopbil. Med dessa positionsangivelser är sedan tanken att en markbaserad robot ska kunna ta sig dit autonomt för att tömma soptunnan och på så vis kunna effektivisera sophämtningen. I detta kandidatarbete förenklades användarscenariot till att lokalisera markörer på marken och rapportera deras positioner till en basstation.

Då mycket forskning rörande autonoma quadrocoptrar redan hade gjorts vid kandidatarbetets början, var meningen med arbetet att ge projektdeltagarna erfarenhet av och kunskap om tekniska detaljer och utförande av större projekt, snarare än att bidra med nya rön gällande autonom styrning av quadrocoptrar.

1.2 Syfte

Syftet med den här rapporten är att beskriva utvecklingen av autonoma funktioner till en quadrocopter. Detta genom att dokumentera utvecklingsprocessen, presentera den framtagna lösningen och analysera dess prestanda. Dessutom redogörs svårigheter, samt förslag på framtida förbättringar.

1.3 Projektmål

Det övergripande projektmålet var att kunna genomföra ett användarscenario där en quadrocopter lokaliserar markörer genom att autonomt söka av ett område och lagra deras positioner relativt en startpunkt. Med ”autonomt” avses här att användaren inte behöver ge några kommandon efter att uppdraget definierats och sökningen påbörjats.

Det tänkta användarscenariot var att sökuppdraget skulle definieras genom att användaren gav information om önskad storlek på avsökningsområdet, var sökningen skulle börja relativt quadrocopterns startposition, hur många markörer som skulle lokaliseras, samt arbetshöjd för quadrocoptern. Om datan från användaren låg inom tillåtna gränser skulle sedan quadrocoptern planera en flygrutt enligt definitionen av

sökområdet. Därefter skulle quadrocoptern autonomt påbörja flygningen och användaren fråntas kontrollen med undantag för ett manuellt nödlandingskommando. När antingen alla markörer hittats eller hela området sökts av, återvänder quadrocoptern till sin startposition.

För att uppnå detta projektmål formulerades följande delmål:

- Planera och exekvera flygrutt
- Lokalisera markörer och spara deras koordinater
- Systematiskt utvärdera prestandan hos kombinationen av hårvara och utvecklad mjukvara

1.4 Avgränsningar

Den första avgränsningen som gjordes var att modellen AR.Drone 2.0 tillhandahölls från institutionen bakom kandidatarbetet. Det ledde till att projektet utformades för flygning inomhus. Detta på grund av att AR.Drone 2.0 inte ansågs vara kraftfull nog för att kunna hantera vindpustar. Dessutom togs beslutet att inte söka ett flygtillstånd för att få flyga utomhus eftersom handläggningstiden var så lång att projektet hade varit i slutskedet när tillståndet hade varit klart.

För att komma så långt som möjligt i projektet valdes det att designa en enkel ruttplanering, som ej optimerades för att få en så kort flygrutt som möjligt. Anledningen var att det relevanta ansågs vara att täcka hela sökområdet och kunna lägga mer fokus på att förbättra prestanda. Avsökningsområdet begränsades till att vara rektangulärt och dessutom har inte användaren möjlighet att definiera icke tillåtna flygzoner inom området.

Ett antagande som gjordes var också att markörerna inte skulle ligga närmre än en meter från varandra och att quadrocoptern därmed ej kunde detektera flera markörer samtidigt. Detta för att förenkla hanteringen av markörerna genom att lättare kunna särskilja olika markörer.

I projektet behandlades inte heller bildbehandling utan den visuella detekteringen av markörer byggde helt på de funktioner som fanns inbyggda i quadrocopterns mjukvara från tillverkaren. Detta då bildbehandling är ett stort område i sig som skulle kräva ett omfattande utvecklingsarbete.

En ytterligare avgränsning var att inte utveckla ett användarvänligt gränssnitt för användning av quadrocoptern. Avgränsningen innebar att fokus kunde ligga på att skapa, och förbättra funktionerna för autonom styrning.

2

Teknisk beskrivning

Det här kapitlet presenterar quadrocoptermodellen som använts i projektet samt vilka sensorer som ingår i denna. Kapitlet behandlar även positionsreglering för quadrocoptern.

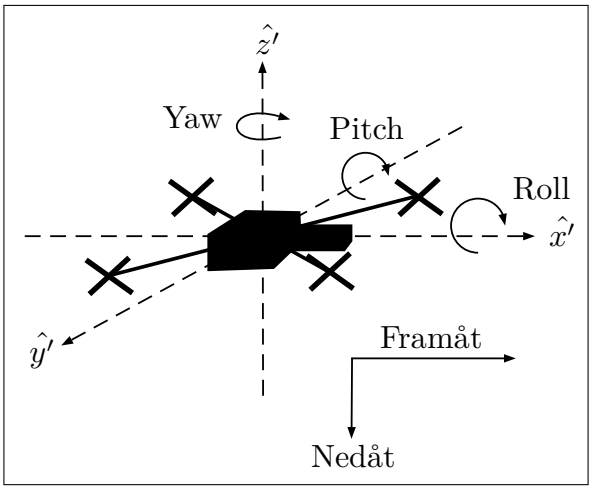
2.1 Quadrocopter – tekniska definitioner och begrepp

En quadrocopter har en förhållandevis enkel konstruktion jämfört med andra flygfarkoster, såsom helikoptrar som har många rörliga delar. Quadrocopterns kropp innehåller batteri och kretskort med diverse sensorer (se Avsnitt 2.2 för fullständig lista av sensorer som finns på quadrocoptern som användes i projektet) och en processor som beräknar styrsignaler till motorerna. Från kroppen sticker fyra armar ut och längst ut på dessa sitter fyra motorer varpå propellrar är monterade. Motorerna förses med matningsspänning och styrsignaler via kablar längs armarna.

Quadrocopterns tillstånd i rummet kan definieras med en *vinkeldel* och en *rumsdel*. Vinkeldelen består av de tre parametrarna *roll*, *pitch* och *yaw* som betecknar quadrocopterns rotation kring sina respektive symmetriaxlar (se Figur 2.1). Rumsdelen består av x'–, y'– och z'–koordinater vilkas orienteringar relativt quadrocoptern visas i Figur 2.1. Detta quadrocopter–fixa system av koordinater betecknas S' och är för den quadrocoptermodell som används i projektet icke–konventionellt på så sätt att det är vänsterorienterat[1]. Då målpositioner anges, sker det i ett jordfixt koordinatsystem, S, som även det är vänsterorienterat och vars origo placeras quadrocopterns startposition. Begreppet *hovra* innebär att quadrocoptern flyger på samma position i luften, det vill säga den står stilla i luften.

För att stabilisera en quadrocopter krävs kännedom om dess tillstånd i rummet. Detta åstadkoms genom att kombinera dess sensorer som tillsammans ger approximativ information om var quadrocoptern befinner sig, hur fort den förflyttar sig och roterar. Det är viktigt att understryka att den data som sensorerna producerar

[1] Ett vänsterorienterat koordinatsystem följer inte den så kallade *högerhandsregeln* som följer av den matematiska definitionen av vektorprodukten.



Figur 2.1: Definition av koordinatriktningar. Illustration över hur vinklarna *roll*, *pitch* och *yaw* samt koordinaterna $\hat{x}'$, $\hat{y}'$ och $\hat{z}'$ är definierade. Observera att koordinatsystemet är vänsterorienterat för den aktuella quadrocoptermodellen.

aldrig stämmer exakt överens med verkligheten. Vissa sensorer är känsliga för kortvariga fel, det vill säga små störningar som gör att deras utsignaler innehåller brus, men har ett medelvärde som inte avviker allt för mycket från verkligheten. Andra sensorer ger en signal med mindre brus, vilka dock, efter en tids mätning, avviker från de verkliga värdena och benämns *drift*.

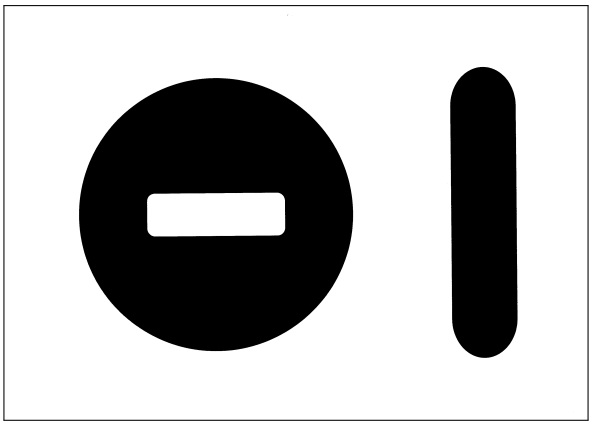
2.2 AR.Drone 2.0

Den modell av quadrocopter som använts i det här projektet är *AR.Drone 2.0* som tillverkas av företaget *Parrot*. Modellen är egentligen avsedd att styras via en applikation för mobila enheter, men kan via WiFi styras från en dator med trådlös nätverksanslutning. På AR.Drone 2.0 är elektroniken inkapslad i ett stöt– och väderskyddat hölje av hårdplast och cellplast. För att skydda propellrar och quadrocoptern används dessutom ett skydd av cellplast, vilket monteras utanpå quadrocoptern, se Figur 2.2.

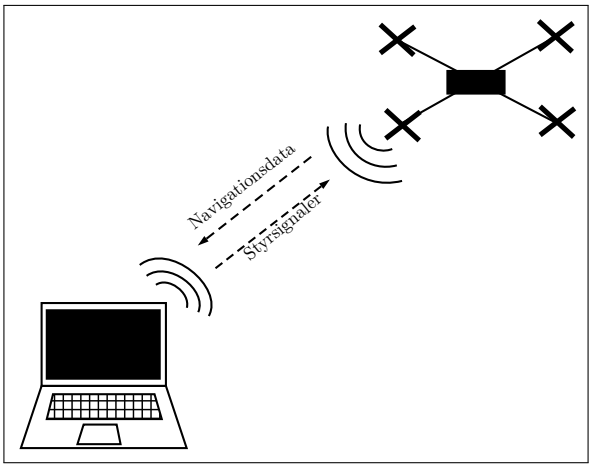
Quadrocoptern kan med hjälp av sina kameror och sin inbyggda mjukvara känna igen tre olika markörer. En av dessa markörer är en orienterad rundel som visas i Figur 2.3. Det är denna markör som använts i projektet.

I Figur 2.4 visas en schematisk skiss över hur quadrocoptern och styrdatorn samverkar. Via WiFi skickas navigationsdata från quadrocoptern till datorn, vilken behandlar navigationsdatan och sedan skickar styrsignaler till quadrocoptern. Denna kommunikation ungefär sker 15 gånger varje sekund. Navigationsdatan skickas i form av numeriska sekvenser, vilka måste översättas för att kunna tolkas på rätt

Figur 2.2: Quadrocoptern AR.Drone 2.0 med sitt skyddande hölje av cellplast påmonterat. Publicerad med tillstånd av företaget Parrots svenska importör Ingeniörsfirman M Sjöberg AB.



Figur 2.3: Markör att detektera. Denna markör är den som känns igen av quadrocoptern. Markören skrivs ut på A4–papper och tejpas fast på målen som ska hittas av quadrocoptern. Publicerad med tillåtelse av företaget Parrots svenska importör Ingeniörsfirman M Sjöberg AB.



Figur 2.4: Schematisk skiss över hur styrdatorn och quadrocoptern samverkar. Kommunikationen dem emellan sker via WiFi. På så sätt skickar quadrocoptern navigationsdata från sina sensorer till styrdatorn. Datorn behandlar dessa data och skickar sedan styrdata till quadrocoptern. Denna kommunikation sker 15 gånger per sekund.

sätt. Styrsignalerna består av UDP[1]–paket, som utgörs av dels ett sekvensnummer och dels en siffra. Denna siffra står för ett kommando, så som att lyfta eller landa, som quadrocoptern förstår. Anledningen till att ett sekvensnummer behövs är för att UDP inte garanterar att paketet når fram, utan det måste skickas ett flertal gånger.

2.2.1 Gyroskop

Ett gyroskop innehåller en skiva som konstant strävar efter att bibehålla sin orientering i rummet. Genom att använda jordens gravitationskraft i x–, y– och z–led kan därmed vinkelförändringar som systemet utsätts för mätas genom att mäta frekvensen som skivan vibrerar med [11].

2.2.2 Magnetometer

Magnetometern används för att bestämma quadrocopterns rotation relativt jordens magnetfält. Den gör detta genom att mäta mängden magnetiskt flöde som passerar genom magnetometern i x–, y– och z–led. I och med att jordens magnetfält lokalt har en konstant riktning och styrka inom tidsramen för en flygning[2] märker sensorn av en rotation genom att den relativa storleken på flödeskomponenterna varierar.

[1]User Datagram Protocol, ett protokoll för trådlös kommunikation.

[2]Jordens magnetfält varierar, beroende på vilken plats man befinner sig, och flyttar på sig över tid.

Den magnetometer som används i AR.Drone 2.0 har en mätosäkerhet på 6 grader [10].

2.2.3 Accelerometer

När en accelerometer utsätts för en acceleration, erhålls en elektrisk signal som är proportionell mot accelerationen. Genom att mäta och integrera accelerationerna i de tre rumsdimensionerna över tid, kan accelerometerns hastighetskomponenter bestämmas. Med hastigheten känd kan quadrocopterns position i de tre rumsdimensionerna följas genom ytterligare en tidsintegration. Accelerometern som används i AR.Drone 2.0 har en mätosäkerhet på ± 50 mg[1] [10].

2.2.4 Ultraljud

En ultraljudssensor mäter avstånd till närliggande objekt genom att skicka ut en serie ljudpulser med en bestämd frekvens och mäta tiden det tar för ljudvågens reflektion att återvända till sensorn. I och med att ljudfarten i luft är känd kan avståndet beräknas som $\textit{sträckan} = \textit{farten} \cdot \frac{\textit{tiden}}{2}$.

I AR.Drone 2.0 används ultraljudssensorn för att bestämma quadrocopterns höjd över marken då quadrocoptern är max sex meter över marken. När höjden är större än 6 meter fungerar inte ultraljudsmätningen och därför ersätts den med en mindre exakt barometer för höjdbestämning.

2.2.5 Barometer

Barometern används för höjdbestämning för altitud högre än 6 meter. Genom att mäta lufttryckets förändringar kan man bestämma höjden. Noggrannheten för barometern som används i AR.Drone 2.0 är $\pm$ 10 Pa [10]. Eftersom lufttryck kan variera lokalt medför det att höjdmätning med en barometer inte lika exakt som med en ultraljudssensor. Dessutom behöver barometern kalibreras ofta på grund av dessa förändringar i lufttrycket.

2.2.6 Bottenkamera

På quadrocoptern sitter två kameror monterade, varav en är riktad ner mot marken. Kameran har VGA-upplösning (640x360 pixlar) [10]. Med hjälp av bildbehandling används denna kamera för att hjälpa till vid quadrocopterns hastighetsbestämning [10]. Dessutom kan kameran användas till att leta efter fördefinierade markörer (se

[1] Enheten mg innebär tusendelar av tyngdaccelerationen

Avsnitt 2.1 för mer information om markör). Quadrocoptern kan förprogrammeras till att utföra särskilda kommandon när kameran upptäcker en markör, kan quadrocoptern programmeras till att utföra särskilda kommandon, exempelvis att rapportera markörens position till styrdatorn. Videoströmmen från kameran kan vid behov skickas till quadrocopterns styrenhet och sparas ner till en filmfil.

2.2.7 Framåtriktad kamera

Utöver den nedåtriktade bottenkameran finns en frontmonterad kamera med 720p-upplösning (1280x720 pixlar) och kapacitet till att filma med 30 fps[1]. Kameran har ett brett upptagningsområde på 92 grader över diagonalen. Även denna kamera kan användas till att leta efter markörer [10].

2.3 Positionsbestämning av quadrocopter

För att veta positionen för quadrocoptern har metoden odometri använts. Odometri–algoritmer använder sig av den data som sensorerna samlar in för att uppskatta den sträcka quadrocoptern har färdats. I det här projektet har hastigheten från navigationsdatan använts för att integreras upp över tiden, och på så vis kan positionen för quadrocoptern bestämmas.

2.4 Reglering av position

För att quadrocoptern ska kunna ta sig till en position eller behålla sig på samma position behövs en regulator. Genom att återkoppla utsignalen (befintlig position) till insignalen (önskad position) kan felet mellan befintlig och önskad position minimeras genom förflyttning av quadrocoptern. Regulatorn som använts under projektet är på formen PID, vilken består av tre delar. En proportionerlig del, en integrerande del och en deriverande del. Då regulatorn är implementerad digitalt, kan dess utsignal, F_{PID}, beskrivas matematiskt enligt

$$F_{PID} = k_p \cdot e[k] + k_i \cdot T \cdot \sum_{k=1}^{N} e[k] + k_d \cdot \frac{e[k] - e[k-1]}{T} \tag{2.1}$$

där $e[k]$ är felet $r[k] - y[k]$ och T är samplingstiden. Här är $r[k]$ börvärdet och $y[k]$ är det uppmätta tillståndet från det reglerade systemet. I projektet är börvärdet den position i rummet man vill att quadrocopter ska bege sig till och utdatan är nuvarande position. Denna form av PID–regulator kallas integrerande form, då den ackumulerar fel för beräkningar.

[1]Förkortning för eng. *frames per seconds*, på svenska *bildrutor per sekund*.

Ett stort värde på den proportionella delen (k_p) leder till ett snabbare system, men det genererar dock ett mer instabilt system. Den integrerande delen (k_i) eliminerar lågfrekventa fel, men även den försämrar stabiliteten av systemet. Därför införs en deriverande del (k_d) för att motverka de instabiliteter som uppstår som bieffekt [12]. Ett för högt värde på k_d gör dock systemet mer känsligt för brus. Genom en avvägning mellan snabbhet, kvarstående fel och stabilitet kan parametrarna för systemet erhållas. Detta kan göras genom att modellera systemet och utföra simuleringar för olika parametrar, vilket dock aldrig gjordes i det här projektet på grund av tidsbrist.

3

Mjukvara

För att underlätta för kommande kapitel förklaras i detta kapitel relevanta programmeringstermer och den mjukvaruplattform som använts – Node.js. Sedan följer en beskrivning av den mjukvarulösning som utvecklats under projektet – `KandiDrone`. I kapitlet används ordet *modul* som kan ses som Javascript:s motsvarighet till klasser i andra objektorienterade programmeringsspråk.

3.1 Programmeringstermer

Detta avsnitt ämnar att förklara programmeringstermer för att underlätta förståelsen för hur Javascript och de använda mjukvarumodulerna (hädanefter *modulerna*) fungerar.

***Callback–funktion*:** För att vara säker på att en funktion endast exekveras efter att en annan sekvens har utförts kan man i Javascript använda sig av så kallade callback–funktioner. I Javascript är funktioner *first–class* objekt, vilket bland annat betyder att nya funktioner kan konstrueras under exekvering av ett program och att funktioner kan användas som argument till andra funktioner [13]. När en funktion ges som argument kallas den för en callback–funktion och anropas i slutet av den första funktionen. Detta illustreras i följande exempel där funktionen `myFunc` ges argumentet `callback()` som är en callback–funktion:

```
function myFunc (function callback() {Kod som callback-funktionen
ska utföra})
{
    Kod som myFunc utför
    callback()
}
```

Från exemplet framgår att callback–funktionen exekveras efter att den första funktionen är genomförd [14] vilket är nödvändigt om en sekvens är beroende av att en tidigare sekvens har utförts, exempelvis på grund av att den första funktionen genererar indata till callback–funktionen. Framförallt är callback–funktioner användbara

i de fall då det inte är känt hur lång tid det tar för den tidigare sekvensen att exekvera och man under tiden inte vill låsa resten av programmet. Detta sistnämnda kallas vanligen *asynkron programmering* och är en av de starka sidorna hos JAVASCRIPT.

***Event*:** Programmeringsbegreppet *Event* (sv. händelse) används för att få en sekvens att utföras vid en viss händelse. För att detta ska fungera krävs två delar: en så kallad *event–emitter* och en eller flera *listners* (sv. lyssnare). Event–emittern skickar iväg en händelsenotis tillsammans med data, varpå en lyssnare som inväntar just denna händelsenotis tar emot datan och kan använda den i en funktion [15]. Denna metodik är känd som *händelsedriven programmering* och är användbar exempelvis då navigationsdata från en quadrocopter har tagits emot och ska användas i en annan metod som genererar styrsignaler.

***Objekt*:** I *objektorienterad programmering* används så kallade *objekt* för modellera och konkretisera egenskaper hos mer abstrakta idéer och koncept [16]. Objekt kan lagra data i form av variabler för att beskriva egenskaper och kan dessutom innehålla funktioner för att utföra vissa procedurer. Sådana funktioner som tillhör objekt kallas vanligen *metoder*. Ett annat centralt begrepp är *klasser* som fungerar som ritningar för objekten genom att definiera vilken data och vilka metoder som en viss typ av objekt ska innehålla. Utifrån en sådan klass kan sedan flera olika unika objekt instansieras, vilket innebär att flera objekt av samma typ kan existera men där varje enskilt objekt tillåts ha egna specifika värden lagrade.

JAVASCRIPT är ett objektorienterat programmeringsspråk men saknar klasser. Istället är JAVASCRIPT prototypbaserat vilket innebär att objekt ärver egenskaper från andra objekt snarare än från traditionella klasser [17]. Arvet sker genom att klona orginalobjektet som då bildar en *prototyp* för det nya objektet. Denna prototyp kan sedan byggas vidare på genom att lägga till nya egenskaper och metoder och därmed utöka funktionaliteten för den nya objekttypen. Det är dock möjligt att i JAVASCRIPT använda *moduler* för att definiera objekt på ett klass–liknande sätt. En modul är programkod som används modulärt för att definiera och skapa ett prototypobjekt.

3.2 Node.js

För att kunna programmera och kompilera koden till quadrocoptern måste en mjukvaruplattform användas. En plattform är en struktur för hur data behandlas och lagras. Den plattform som används i detta projekt är NODE.JS, vilken är en asynkron och händelsedriven (se Avsnitt 3.1) plattform för JAVASCRIPT–utveckling byggd på Google Chromes JAVASCRIPT–motor V8 [18].

I NODE.JS utnyttjas ickeblockerande in– och utdata, vilket innebär att flera insignaler kan behandlas samtidigt. Detta genom att en insignal börjar bearbetas direkt när den tagits emot istället för att invänta att programmet hunnit generera en utsignal från den föregående insignalen. Alltså blockerar inte en insignal programmet

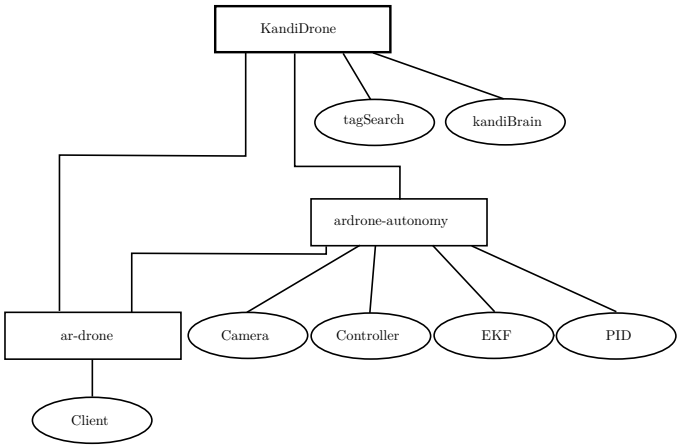
från att ta emot fler insignaler.

Node.js inkluderar pakethanteringsprogrammet *npm* (förkortning för eng. *node package manager*) [19]. Med npm tillåts användare av Node.js att dela öppen källkod i form av moduler (även kallade *paket*). Detta innebär att användare på ett enkelt sätt kan ladda ner programkod avsedd för att lösa specifika problem som redan lösts av andra användare. Npm hanterar dessa moduler genom att bland annat kontrollera versionsnummer och vilka moduler en viss modul i sig är beroende av.

Huvudsakligen är Node.js en plattform för att skapa webbapplikationer där indata från många olika användare behöver processeras så snabbt som möjligt. Node.js är till följd av sin goda förmåga att handskas med serveranrop även ett lämpligt val för att styra en quadrocopter såsom AR.Drone 2.0, vilken utnyttjar nätverkskommunikation via WiFi–anslutning, se Avsnitt 2.2. Detta då navigationsdata från quadrocoptern snabbt behöver bearbetas till styrsignaler.

3.3 Moduler och bibliotek

Under projektet användes färdigskriven kod i form av Node.js–moduler som installerats med hjälp av npm. En överblick av dessa moduler ges i Figur 3.1. Dessa moduler finns även tillgängliga för nedladdning från *GitHub* [20]. De moduler som använts är publicerad med licenser för öppen källkod, vilket innebär att koden är fri att använda och modifiera för icke–kommersiella ändamål utan att riskera upphovsrättsbrott.



Figur 3.1: Schematisk skiss över modulerna. Översikt över hur `KandiDrone` bygger på Node.js–modulerna `ar-drone` och `ardrone-autonomy`, samt deras undermoduler.

3.3.1 Ar–drone

Den Node.js–modul som utgör grunden för den utvecklade mjukvarulösningen är skapad av *Felix Geisendörfer* och heter `ar-drone`. Modulen är ett Javascript–bibliotek som sköter nätverkskommunikationen mellan AR.Drone 2.0 och Node.js på styrdatorn. Kommunikationen består främst av att skicka styrkommandon från Node.js och översätta den inkommande navigationsdatan från quadrocoptern. Navigationsdatan skickas i form av numeriska sekvenser. Därför använder sig Geisendörfer av binära masker[1] för att omvandla informationen till decimaltal lagrade i variabler. Styrningen sker genom att omvandla styrkommandon, så som att lyfta och landa, till UDP–paket, se Avsnitt 2.2.

Från `ar-drone` kan objekt skapas av undermodulen `Client`. Detta är ett högnivå–API (förkortning för eng. *Application Programming Interface*; på svenska känt som applikationsprogrammeringsgränssnitt [22]), vilket gör det lättare för användaren att skicka kommandon till quadrocoptern. Exempel på sådana kommandon är *lyfta*, *landa* och *flyga framåt*. `Client` använder sig i sin tur av ett lågnivå–API som i `ar-drone` kallas `UdpControl`. Det är detta lågnivå–API som sköter själva översättningen av den högre nivåns kommandon till råa UDP–paket som skickas via nätverket till quadrocoptern. `UdpControl` sköter även översättningen av de råa datapaket som skickas från quadrocoptern i form av navigationsdata, bildströmmar och data från visuell objektidentifiering.

Sammanfattningsvis består Geisendörfers Node.js–modul `ar-drone` av två primära delar: ett låg– och ett högnivå–API, där högnivå–API:et tillåter användaren att utnyttja abstrakta kommandon vilka sedan översätts av lågnivå–API:et till datapaket som skickas till quadrocoptern.

3.3.2 Ardrone–autonomy

En annan Node.js–modul som användes i projektet var `ardrone-autonomy`, skapad av *Laurent Eschenauer*. Denna modul bygger på Node.js–modulen `ar-drone` och består av ett antal undermoduler. Dock används inte alla i detta projekt då de ej är nödvändiga. En essentiell del är undermodulen som hanterar positionsreglering – `PID` – som används för att få quadrocoptern att flyga till en angiven position och även för att den ska behålla positionen och ej driva iväg, se Avsnitt 2.4. Vidare används en undermodul – `EKF` – som uppskattar quadrocopterns position. En del av Node.js–modulen som används är en kameramodul – `Camera` – som omvandlar en pixelposition från bottenkameran till en position relativt quadrocoptern, vilket är nödvändigt vid markördetektering. I `ardrone-autonomy` finns det även en styrmodul – `Controller` – som använder PID–regulatorn, positionsuppskattaren och kameramodulen. Hit skrivs bland annat en målposition varpå felet mellan målpostionen och quadrocopterns position beräknas. Med målposition avses den aktuella position

[1] En metod för att extrahera specifika bitar ur en sträng. För mer information rekommenderas den intresserade läsaren att läsa vidare på Wikipedias sida om *Bitmasker* [21].

som quadrocoptern ska flyga till. Från felet beräknas sedan styrsignaler med hjälp av PID–regulatorn. Styrsignalerna skickas sedan till `Client` som omvandlar dessa till UDP–paket och skickar dem till quadrocoptern.

Via `Index`–filen i denna NODE.JS-modul skapas objekt av `Controller`, `PID` och `Camera`. Ett objekt skapas även av `EKF` vilket dock endast används som positionsuppskattare och inte som ett utökat Kalmanfilter som Eschenauer tänkt. Detta på grund av att implementationen av Kalmanfiltret inte är applicerbart i projektet då koden bland annat förutsätter att alla markörer har känd position, vilket går rakt emot syftet med projektet.

En del ändringar har behövt göras i `ardrone-autonomy`. Markördetekteringen i `Controller` har tagits bort, eftersom detta nu sker i `KandiDrone`. Då tester av steg med quadrocoptern gjordes, se Avsnitt 4.2, upptäcktes att quadrocoptern flyger längre än den angivna sträckan. Därför lades en skalfaktor in i funktionen som hanterar målpositionen i x– och y–led, som finns i `Controller`–filen. Detta medförde att quadrocoptern flyttade sig närmare den verkliga positionen än tidigare.

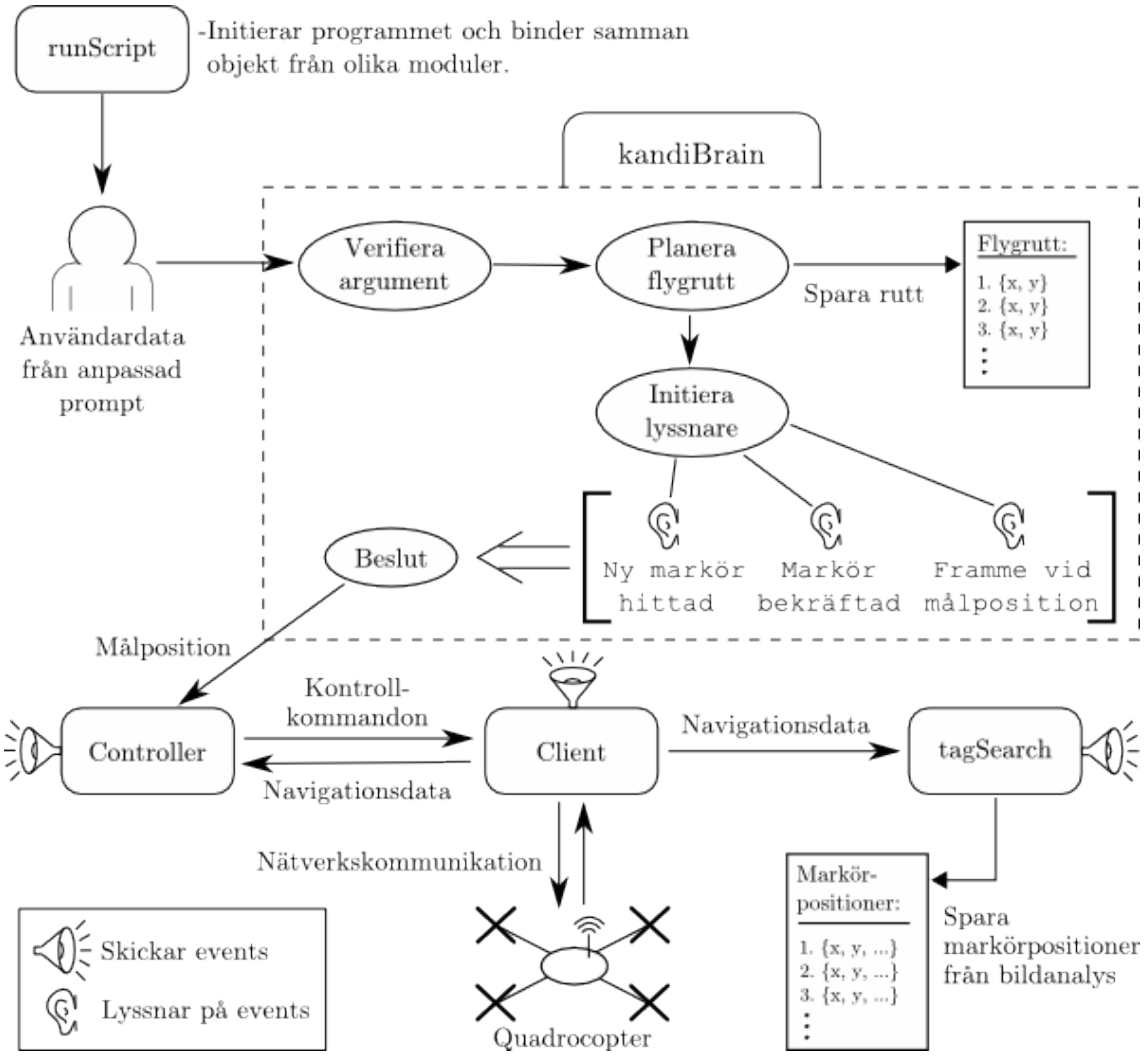
Även positionsuppskattaren i `EKF`–filen har justerats. Tidigare bestämdes yaw–vinkeln genom att integrera rörelsen i yaw–led, alltså summering av yaw–ändringar mellan tillståndsuppdateringar. Istället sparas quadrocopterns yaw–vinkel vid starttillfället, denna subtraheras från den avlästa yaw–vinkeln, som är angiven relativt jordens magnetfält, se Avsnitt 2.2.2. Denna metod innebär att en absolut yaw–vinkel används istället för en som beräknats från integrering vilket annars kan leda till att fel ackumuleras.

För positionsuppskattaren har också tidssteget mellan två efterföljande tillståndsuppdateringar ändrats. Ursprungligen var detta ett konstant värde som förutsatte att navigationsdatan från quadrocoptern alltid mottages 15 gånger per sekund. Nu beräknas istället tiden mellan två inkommande navigationsdatapaket för att få ett mer korrekt tidssteg i beräkningen av det nya tillståndet.

3.4 KandiDrone

Den modul som utvecklades under projektets gång kallas `KandiDrone`[1]. Modulen sköter hantering av markörpositioner, ruttplanering och exekvering av rutten. Huvuddelen och därmed hjärnan i modulen heter `kandiBrain`, se Figur 3.1, vilken använder `tagSearch` för detektering av markörer och hantering av deras positioner. En schematisk skiss över strukturen i `KandiDrone` och dess informationsflöde med undermodulerna återfinns i Figur 3.2.

[1] `KandiDrone` finns tillgänglig på https://github.com/andjohae/kandidrone



Figur 3.2: Översikt av programkoden. Schematisk skiss över hur komponenterna i modulen `KandiDrone` kommunicerar med varandra. Programmet initieras av den körbara filen `runScript` vilken också ger användaren möjlighet att ange inställningar för quadrocopterns sökområde. Därefter skickas användardatan till huvudmodulen `kandiBrain` vilken verifierar att argumenten är tillåtna och en flygrutt planeras utifrån det definierade sökområdet. Sedan initieras lyssnare som inväntar vissa events (några exempel syns i figuren). Beslutsfattandet i `kandiBrain` baseras på vilka events som registreras av lyssnarna och utifrån dessa uppdateras sedan den aktuella målpostionen till `Controller`–objektet. Detta skickar i sin tur kontrollkommandon till `Client`–objektet som sedan kommunicerar med quadrocoptern. Från quadrocoptern mottages navigationsdata som skickas till dels `Controller`–objektet som uppdaterar positionen och dels till `tagSearch`–objektet som hanterar markördetektering.

3.4.1 tagSearch – Markörhantering

Den stängda mjukvaran ombord på AR.Drone 2.0 är programmerad för att känna igen tre olika typer av markörer. Den markör som har använts i projektet ses i Figur 2.3. När quadrocoptern flyger över en sådan markör detekteras den med hjälp av bottenkameran och markörens pixelposition i kamerabilden skickas till styrdatorn. Pixelpositionen räknas om till en position relativt quadrocoptern med hjälp av kameramodulen i `ardrone-autonomy`. Sedan adderas denna relativa position till quadrocopterns position i rummet för att erhålla markörpositionen i det jordfixa koordinatsystemet, hädanefter kallade $x_{uppmätt}$ och $y_{uppmätt}$.

När quadrocoptern är i rörelse är den dock ej helt parallell med markplanet, det vill säga att vinklarna pitch och roll är nollskilda. Detta leder till att markörens position registreras felaktigt, eftersom bottenkameran då inte heller är parallell med marken, se Figur 3.3. För att kompensera för detta adderades en korrigeringsterm i var led till markörpositionen enligt:

$$\begin{aligned} x_{markör} &= x_{uppmätt} - a_x \\ y_{markör} &= y_{uppmätt} - a_y \end{aligned} \tag{3.1}$$

med korrigeringstermerna

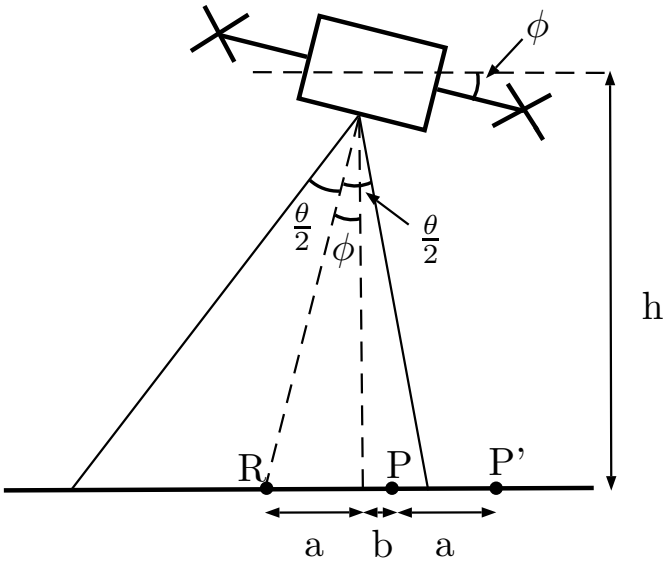
$$\begin{aligned} a_x &= h \cdot \tan(pitch) \\ a_y &= h \cdot \tan(roll) \end{aligned} \tag{3.2}$$

där vinkeln ϕ i Figur 3.3 är *pitch* eller *roll* beroende på vilken riktning som kompenseras.

Den del av projektets modul som hanterar markörpositioner heter `tagSearch`. När den första markören hittas skapas en lista, i vilken information om varje markör sparas. Informationen som sparas är markörens *position*, *antalet gånger den detekterats* och en *boolesk variabel* med värdet *sant* eller *falskt* beroende på om markörens position anses bekräftad eller inte. En markörposition anses bekräftad när markören detekterats tillräckligt många gånger. Antalet gånger valdes i projektet godtyckligt till 19, vilket ansågs ge tillräcklig precision och inte vara för tidskrävande.

När quadrocoptern detekterar ytterligare en markör kontrolleras om den nya markören redan är känd. Detta görs genom att undersöka om markörens position befinner sig inom en radie av 1 meter från tidigare funna markörer, se Figur 3.4d. Om det är en ny markör kommer denna att läggas till i den tidigare skapade listan.

Om markören redan är känd kommer istället en medelvärdesbildning att utföras: Vid andra avläsningen av markören adderas då de två positionerna och summan divideras med två. När nya markörpositioner registreras, adderas de med det tidigare beräknade medelvärdet för positionen och den nya summan divideras med två. Denna metod är beräkningseffektiv och innebär att senare avläsningar av markörpositionen viktas mer än de tidigare avläsningarna. Detta är en fördel eftersom att



Figur 3.3: Kompensering vid markördetektering. Quadrocoptern uppfattar att markören P ligger på avstånd a+b från R, dvs i P'. Därför måste a subtraheras, som beräknas enligt Ekvation (3.2).

quadrocoptern i detta användarscenario kommer att flyga ut till markören och därför befinna sig i ett stabilt tillstånd ovanför markören. Därmed kommer mer precisa positioner att återges för de senare avläsningarna.

Vid var 5:e detektion av samma markör skickas ett event om att den aktuella markörpositionen uppdaterats, tillsammans med datan för den medelvärdesbildade markörpositionen. När markören hittats 19 gånger anses dess position vara bekräftad varpå den booleska variabeln för markören sätts till *sann*. I samband med detta skickas ytterligare ett event och informationen om markörens position. I `kandiBrain` initieras lyssnare för dessa event och de medskickade markörpositionerna hanteras. Om en redan bekräftad markör åter detekteras, indikerar den logiska variabeln att markören kan ignoreras.

3.4.2 kandiBrain – Ruttplanering och exekvering

Huvuddelen i `KandiDrone` heter `kandiBrain` och är den undermodul som exekverar quadrocopterns flygrutt över sökområdet. I `kandiBrain` verifieras argumenten som definierar sökområdet och en flygrutt planeras utifrån dessa. Dessutom innehåller `kandiBrain` en rad olika lyssnare som registrerar events från bland andra `tagSearch`, `Client` och `Controller`. Beroende på vilka events som tas emot kan sedan `kandiBrain` se till att quadrocoptern flyger till rätt positioner genom att uppdatera den aktuella målpositionen till `Controller`–objektet.

3.4.2.1 Ruttplanering

Ruttplaneraren är som namnet antyder den funktion som beräknar rutten som quadrocoptern ska flyga. Flygrutten planeras med avsikt att söka av hela området med avseende på bottenkamerans upptagningsvinkel. Sökområdet begränsas till att endast ha en rektangulär form med mått som anges av användaren, se Figur 3.4a. Från Figur 3.4b framgår att quadrocoptern flyger en lång sträcka i x–led, flyttar sig ett kort steg i y–led, flyger tillbaka hela sträckan i x–led och sedan tar ytterligare ett steg i y–led, tills hela området är genomsökt. Avståndet mellan de längre sträckorna betecknas i figuren med b, som kallas *korridorsbredd* och beräknas utifrån en av användaren angiven arbetshöjd h meter över marken enligt

$$korridorsbredd = 0,9 \cdot 2 \cdot h \cdot \tan(\theta/2)$$

där θ är den nedåtriktade kamerans upptagningsvinkel. För att inte missa någon yta krävs ett visst överlapp av korridorsbredden, se Figur 3.4c, därför minskas korridorsbredden med 10 % vilket medför faktorn 0,9. Om ingen arbetshöjd anges används ett standardvärde på 1,5 m.

Vid en flygning roteras quadrocoptern inte i yaw–led, utan har alltid sin front i positiv x–led. Genom att göra på detta sätt fås en så bred korridorsbredd som möjligt eftersom att kamerans upptagningsvinkel är betydligt större i y–led än i x–led, se Avsnitt 5.3.

För att quadrocoptern ska kunna flyga den angivna rutten beräknas viapunkter utifrån de ovan beskrivna funktionsargumenten. Även antalet hela korridorsbredder på y–sträckan beräknas. Resten som blir kvar används för att sätta de två sista punkterna i rutten, som då kommer att bestå av ett något kortare steg i y–led än korridorsbredden. Detta görs för att undvika att quadrocoptern missar den sista delen av avsökningsområdet och inte heller flyger för långt bort. De beräknade viapunkterna sparas sedan i en matris.

Användaren kan även ange den startposition relativt quadrocopterns ursprungsposition där flygrutten ska börja, se Figur 3.4a, vilket är användbart om quadrocoptern inte lyfter från positionen där man vill börja avsökningen. Notera att quadrocoptern kommer landa på ursprungspositionen som den lyft ifrån, och inte på startpositionen för avsökningsområdet. För att undvika att quadrocoptern flyger längre bort från styrdatorn än räckvidden för WiFi, så är avståndet mellan avsökningsområdets startposition och quadrocopterns ursprungsposition begränsat. Detta förutsätter att quadrocoptern lyfter från en position nära styrdatorn. Kontrollen av sökområdets specifikationer sker i en verifieringsfunktion som sedan skickar vidare de kontrollerade argumenten till en callback–funktion, i detta fall ruttplaneraren.

3.4.2.2 Ruttföljning

Ruttföljningen inleds med att quadrocoptern lyfter, gör en kalibrering av magnetometern och flyger till startpositionen för avsökningsområdet varifrån den sedan flyger till första positionen i flygrutten. För att problem inte ska uppstå vid startsekvensen, alltså för att försäkra att förloppet inte sker i fel ordning, får quadrocoptern först kommandot att lyfta. Som callback–funktion anropas kalibreringen, som i sin tur har som callback–funktion att uppdatera den aktuella målpositionen till startpositionen för sökområdet. Därefter startar ruttföljningen och den aktuella målpositionen uppdateras till första positionen i flygrutten. `Controller`–objektet som hanterar positionsuppskattning skickar ett event när quadrocoptern nått den aktuella målpositionen, varpå denna uppdateras till nästa position i flygrutten. Om en markör hittas under en flygsträcka kommer ett event skickas från `tagSearch` till `kandiBrain`. Detta gör att rutten avbryts och quadrocoptern flyger till den hittade markören. När ett event mottages om att markörpositionen medelvärdesbildats 5 gånger uppdateras målpositionen till det medskickade medelvärdet för markörpositionen, för att placera sig så när rakt ovanför markören som möjligt. När `tagSearch` har bekräftat markörpositionen mottages ett event, då kommer quadrocoptern att fortsätta flygningen mot senaste viapunkten i rutten (positionen som quadrocoptern var på väg till när markören hittats), se Figur 3.4e. För en utförligare beskrivning av de mest centrala eventen som används under ruttföljningen, se Tabell 3.1.

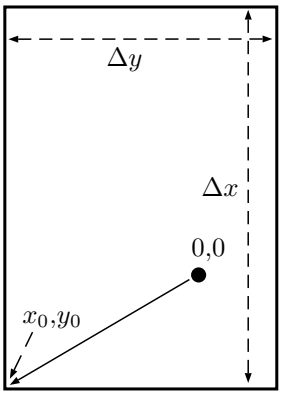
Ett problem som uppstod vid provflygningar var att när quadrocoptern flög rutten och upptäckte en markör, hände det ibland att den uppmätta positionen var så pass inkorrekt att quadrocoptern ställde sig för långt bort från markören för att se den igen. Detta ledde till att quadrocoptern stannade på positionen eftersom att inget nytt event togs emot i `kandiBrain`. För att undvika detta har en timeout lagts in i koden. Den fungerar på sådant sätt att om det dröjer längre än 5 sekunder efter att en markör har hittats utan att den hittas på nytt, sker ett funktionsanrop. Den anropade funktionen beräknar positioner för hörnen hos en kvadrat med sidlängden 1 meter och som är centrerad i quadrocopterns position. Quadrocoptern flyger därefter till vart och ett av hörnen. Om markören hittas uppdateras dess position som tidigare och quadropcoptern återgår därefter till sin rutt. Om markören däremot inte hittas inom denna kvadrat återgår quadrocoptern direkt till den senaste målpositionen i rutten, se Figur 3.4f.

Om quadrocoptern hittat lika många markörer som användaren har angivit kommer quadrocoptern direkt att flyga tillbaka till sin ursprungsposition. Quadrocoptern kommer även återgå till ursprungspositionen om hela rutten har flugits och den inte har hittat lika många markörer som användaren angivit.

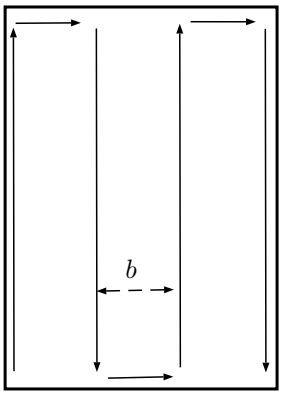
För att undvika att quadrocopterns batteri tar slut under rutten, har en lyssnare för batterinivån lagts in. Vid en batterinivå under 10 % återvänder quadrocoptern till sin startposition och landar.

Tabell 3.1: Centrala event. En lista över de mest centrala eventen som registreras av `kandiBrain` under följning av flygrutt.

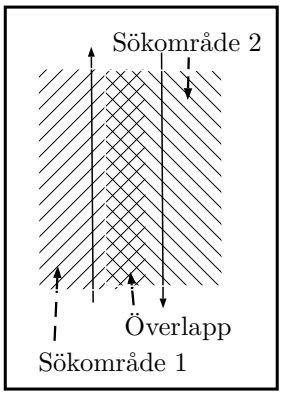
Typ av event	**Från objekt**	**Förklaring**
Ny markör	`tagSearch`	Ny markör detekterad. Uppdatera aktuell målpostion till markörpostionen.
Markör uppdaterad	`tagSearch`	Skickas var 5:e gång markörpositionen medelvärdesbildats. Uppdatera aktuell målposition till markörpositionen.
Markör bekräftad	`tagSearch`	Markörpositionen hittad 19 gånger, markören anses bekräftad. Återgå till aktuell position i flygrutt.
Markör borttappad	`tagSearch`	Det har dröjt mer än 5 sekunder från att en obekräftad markör hittats och inte detekterats igen. Anropa funktion för att hitta borttappad markör.
Framme vid målposition	`Controller`	Tar olika beslut beroende på tillfälle. Nås en position i flygrutten, uppdateras den aktuella målpositionen till nästa position i flygrutten, dock endast om alla markörer inte har detekterats. Om alla markörer detekterats, uppdateras målpositionen till ursprungspositionen och sedan landar quadrocoptern. Utnyttjas även vid borttappad markör för att flyga till alla punkter i kvadraten.
Batteriändring	`Client`	Vid batterinivå lägre än 10 % återvänd till ursprungsposition och landa.



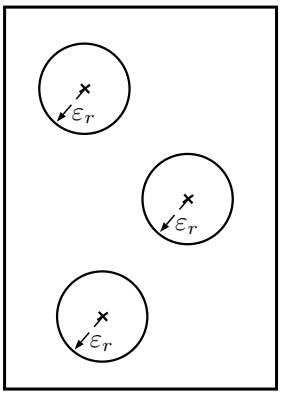
(a) Definition av sökområdets storlek och placering relativt quadrocoptern. Vid anrop av *runScript* ombeds användaren att ange det rektangulära sökområdets mått Δx och Δy samt sökområdets startposition (x_0,y_0) relativt quadrocoptern (0,0). Alla positioner anges relativt quadrocopterns startposition.



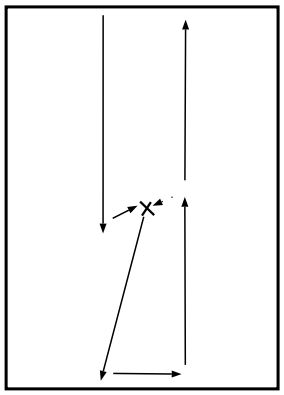
(b) Quadrocopterns avsökningsmetod. Avsökningen sker i ett kantigt mönster med en korridorsbredd b som beräknas utifrån den av användaren angivna arbetshöjden. Notera att de längre sträckorna flygs i x-led.



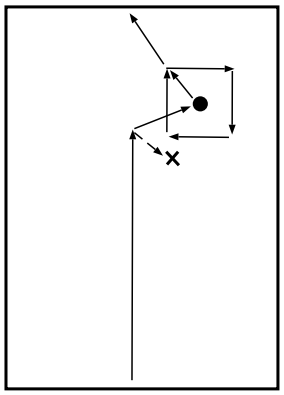
(c) Illustration av sökområdets överlapp. Quadrocopterns nedåtriktade kamera har en viss avsökningsbredd. För att minska risken att missa en markör har korridorsbredden b anpassats så att sökområdena överlappar.



(d) Feltolerans hos markörposition. För att förhindra att samma markör detekterades som flera olika, närliggande markörer till följd av quadrocopterns inexakta positionsbestämning, tilldelades varje detekterad markör ett osäkerhetsområde. Dessa hade formen av cirklar med radien ε_r.



(e) **Avbrott av rutt för fastställning av markörens position.** När en markör detekteras avbryts den planerade rutten och quadrocoptern börjar hovra över markören tills det att positionen är fastställd. Därefter flyger quadrocoptern till nästa punkt i den planerade rutten. Om quadrocoptern detekterar en redan verifierad markör avbryts inte rutten.



(f) **Sökning av borttappad markör.** Om quadrocoptern gör en felaktig positionsbestämning av en markör letar den efter markören på fel position och lyckas därför inte uppdatera markörens position enligt 3.4e. För att öka chansen att hitta markören flyger quadrocoptern därför i en kvadrat med sidlängden 1 m. Om markören därefter inte har hittats flyger quadrocoptern vidare till nästa punkt i den planerade rutten.

Figur 3.4: Schematiska illustrationer av quadrocopterns sätt att söka av en yta. De olika figurerna visar olika aspekter av den utvecklade mjukvaran.

3.4.3 Index – Definition av modulen

För att i Node.js definiera `KandiBrain` som en modul krävs en `index`–fil vilken explicit talar om för Node.js vilka undermoduler som `KandiBrain` består av och därmed vilka objekt som kan skapas. I detta fallet är de aktuella undermodulerna `kandiBrain` och `tagSearch`. Dessutom innehåller `index`–filen information om vilka andra moduler som `KandiDrone` är beroende av.

Utöver att definiera vilka objekt som kan skapas från `KandiDrone` innehåller `index`–filen även funktioner för att korrekt generera dessa objekt. Detta är nödvändigt eftersom både `kandiBrain` och `tagSearch` kräver objekt från Node.js–modulerna `ar-drone` och `ardrone-autonomy` för att fungera. Genom att generera objekt med hjälp av funktionerna i `index`–filen kan objekten från alla olika moduler bindas samman på korrekt sätt. Dessutom tillåter funktionerna att användaren skickar med tidigare skapade objekt från de andra Node.js–modulerna, men om detta inte

görs kommer `index`–filen se till att nya instanser av alla nödvändiga objekt skapas.

3.4.4 runScript – Initiering av avsökningsprocessen

För att använda modulen `KandiDrone` enligt det tänkta användarscenariot krävs den körbara filen `runScript`, vilken startas från kommandotolken. Denna fil initierar programmet genom att läsa in alla nödvändiga Node.js–moduler och skapa nödvändiga objekt. Dessutom innehåller `runScript` inställningar för vilken information som behöver skickas från quadrocoptern. Exempelvis måste man explicit be quadrocoptern skicka information om detekterade markörer och ställa in vilken kamera som ska användas. Detta sker genom att direkt använda `Client`–API:et från modulen `ar-drone`.

Under programmets initieringsfas presenteras en anpassad kommandoprompt för användaren som då får ange information om sökningen som ska genomföras av quadrocoptern. Den information som efterfrågas är storlek på avsökningsområdet, var detta område skall börja relativt quadrocopterns startposition, hur många markörer som skall detekteras, samt önskad arbetshöjd för quadrocoptern. Samtliga distanser anges i meter och endast storleken på avsökningsområdet måste specificeras för att programmet skall kunna starta. Övriga argument har standardvärden som presenteras för användaren i prompten och kommer därför anta värden även om användaren väljer att inte specificera dessa. Dessutom godtages endast argument som dels är giltiga (är siffror, heltal etc.) och dels ligger inom godkända intervall. Intervallen återfinns i Tabell 3.2 och har satts till värden som kan anses typiska för användarscenariot, samt bestämts med hänsyn till framförallt WiFi–räckvidd. Då ogiltiga argument anges kommer användaren meddelas om detta och nya argument efterfrågas.

Tabell 3.2: Tabell över godkända intervall för inargument.

Inargument	Undre gräns	Övre gräns	Standardvärde
Längd avsökningsområde (Δx)	0	30	–
Bredd avsökningsområde (Δy)	0	30	–
Antal markörer	1	10	1
Arbetshöjd	1	5	1,5
Startposition (x_0)	-40	10	0
Startposition (y_0)	-40	10	0

När användaren angivit all nödvändig information skickar `runScript` vidare informationen till `KandiDrone` som då tar över och exekverar sökningsprocessen. Dock bör påpekas att `runScript` innehåller en funktion för manuell nödlandning som kan aktiveras med ett tangentbordskommando. Anledningen till att den finns med är för att man ska kunna avbryta rutten om något fel inträffat. Till exempel för att hindra quadrocoptern från att flyga iväg, skada omgivningen eller skada sig själv.

4

Utforming av tester

Det här kapitlet avser att presentera de olika tester som under projektet utfördes för att utvärdera den aktuella hård– och mjukvaran. Värt att notera är att alla testflygningar skedde inomhus. Detta dels för att ett tillstånd för utomhusflygning saknades och dels för att tak och väggar tillsammans innebar en försäkran om att quadrocoptern inte skulle flyga iväg. För att minska risken för skador på människor, omgivningen och på quadrocoptern användes den medföljande cellplasthöljet vid samtliga flygningar.

4.1 PID–parametrar

Den öppna källkod som projektet baserades på använde sig av PID-parametrarna i Tabell 4.1 och en numerisk PID-algoritm på integrerande form enligt (2.1). För att få en uppfattning om hur väl de fungerade utfördes stegsvarstester där quadrocoptern instruerades att hovra under två sekunder för att sedan förflytta sig 3 och 5 meter i x– respektive y–led. I z–led testades ett steg från 0,5 till 1,5 meter. Valen av sträckor grundade sig i att de ansågs vara typiska sträckor för det tänkta användarscenariot.

Quadrocopterns navigationsdata sparades i loggfiler och ritades upp i MATLAB där sedan funktionen `Stepinfo` användes för att beräkna stigtiden τ_{stig}, insvängningstiden τ_{in} och översvänget Θ.

Tabell 4.1: Tabell över PID-parametrar.

Riktning	k_p	k_i	k_d
x	0.5	0.1	0.35
y	0.5	0.1	0.35
z	0.8	0.1	0.35

4.2 Precision i positionsbestämning

I och med att quadrocoptern bestämde sin position med odometri (se Avsnitt 2.3) var det naturligt att ett fel uppstod vid förflyttningar. För att få en uppskattning om hur väl den uppfattade sin position, genomfördes ett test där quadrocoptern kommenderades att flytta sig 3, respektive 5 meter i x– och y–led och sedan landa. Efter förflyttningen mättes avståndet från startpositionen till slutpositionen med ett måttband. Totalt gjorde fem försök för varje inställning för att kunna göra en medelvärdesbildning.

Efter den första omgången tester gjordes kalibreringar i mjukvaran genom att skala varje målposition med två korrektionsfaktorer k_{fx} och k_{fx}, varefter en ny serie tester genomfördes på samma sätt.

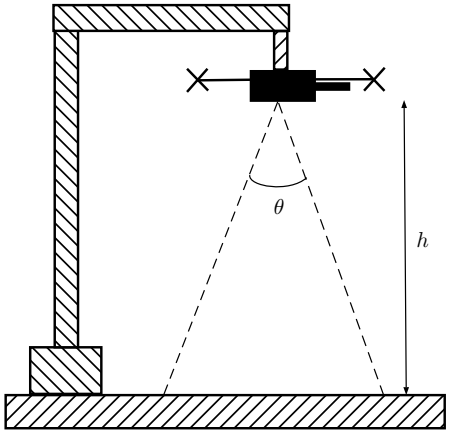
4.3 Vinkelupptagning av bottenkamera

För att kunna beräkna en lämplig korridorsbredd för avsökning av ett område (se Figur 3.4b) undersöktes den nedre kamerans vinkelupptagning. Testet utfördes genom att quadrocoptern med hjälp av ett stativ hölls på höjden h över golvet (se Figur 4.1a). Den nedåtriktade kamerans bild studerades samtidigt som tejpbitar markerade ut bildens gränser. Avstånden Δx och Δy (se Figur 4.1b) mättes upp med ett måttband varefter vinkeln θ kunde beräknas i x– och y–led enligt:

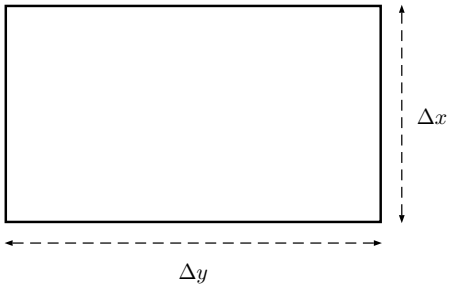
$$\begin{aligned} \theta_x &= 2 \cdot \arctan \frac{\Delta x/2}{h} \\ \theta_y &= 2 \cdot \arctan \frac{\Delta y/2}{h}. \end{aligned} \tag{4.1}$$

4.4 Yaw–drift över tid

För att noggrant undersöka quadrocopterns tendens att driva i yaw–led utfördes tre olika tester som fokuserade på olika faktorer. Gemensamt för alla tester var att den officiella iOS–applikationen för AR.Drone 2.0 användes för att på ett smidigt sätt få tillgång till den nedåtriktade kameran och för att säkerställa att kod som utvecklats under projektet inte skulle vara orsak till eventuella problem. För att kunna avgöra vinkelförändringen instruerades quadrocoptern att stå stilla i luften över ett golv med ett tydligt rutmönster. Under varje testflygning spelades en film från den nedåtriktade kameran in, från vilken bildrutor från olika tidpunkter kunde sparas ner och vinkeln dem emellan bestämmas med hjälp av ett bildredigeringsprogram.



(a) Uppställning för mätning av den nedåtriktade kamerans upptagningsvinkel, sedd från sidan. Quadrocoptern hölls på en känd höjd h med ett stativ. Kamerans upptagningsvinklar $\theta_{x,y}$ är vinklarna mellan kamerabildens kanter i $\hat{x}$- respektive $\hat{y}$-led.



(b) Kamerabilden från den nedåtriktade kameran. Sträckorna Δx och Δy mättes upp med ett måttband och användes för att beräkna vinklarna θ_x och θ_y.

Figur 4.1: Uppställning för mätning av den nedåtriktade kamerans upptagningsvinkel.

Det *första* testet gick ut på att genomföra många flygningar för att undersöka huruvida driften skedde på ett konsekvent vis. Totalt genomfördes 19 flygningar, vilka varade i 30 eller 60 sekunder. I och med att endast den totala vinkelförändringen var av intresse, bestämdes quadrocopterns rotation mellan den första och sista bildrutan i varje film.

Målet med det *andra* testet var att undersöka om quadrocoptern drev med konstant vinkelhastighet. Därför analyserades tre av flygningarna från det första testet i detalj. Rotationen delades in i tio sekunder långa sekvenser, mellan vilka rotationen mättes. På så vis erhölls en mer detaljerad bild av rotationens förlopp än i det första testet.

Det *tredje* testets mål var att undersöka quadrocopterns drift vid långa flygningar.

Därför genomfördes en åtta minuter lång flygning, vars första minut delades upp i tre stycken 20 sekunder långa sektioner och resterande minuter i delar om 30 sekunder. Anledningen till den något högre mätupplösningen i testets inledning var att driften skedde extra fort under den perioden.

4.5 Användarscenario

För att testa quadrocopterns förmåga att genomföra det tänkta användarscenariot (se Avsnitt 1.3), genomfördes kompletta flygtester där tre stycken markörer placerades ut inom en yta på 6 × 10 meter och quadrocoptern sökte av området från 2 meters höjd över marken. Quadrocoptern placerades på koordinaten (3,3) i sökområdet och behövde därför först flyga till dess ena hörn innan avsökningen påbörjades.

Målen med testet var att undersöka hur stor inverkan yaw–driften hade samt hur bra quadrocoptern var på att hålla sig inom sökområdet och på att detektera markörer. Till följd av avsaknaden av bra verktyg för att kvantitativt mäta upp hur väl quadrocoptern presterade, användes videoupptagning för att spela in de genomförda flygningarna. Utifrån dessa filmklipp kunde sedan okulära bedömningar av flygningarna utföras.

5

Testresultat

I det här kapitlet presenteras de uppmätta och observerade resultaten av de tester som beskrivits i Kapitel 4.

5.1 PID–parametrar

De uppmätta stegsvaren i x–, y– och z–led redovisas i Figur 5.1. Från figuren framgår att ett betydande översväng förekom i både x– och y–led. Däremot saknades helt översväng i z–led, vilket även märks tydligt hos de uppmätta, procentuella översvängen i Tabell 5.1: i x– och y–led var översvängen mellan 31 och 44 procent, men bara 3,5 procent i z–led.

Från tabellen framgår även att quadrocopterns stigtid τ_{stig} låg mellan 1,4 och 2,6 sekunder. Quadrocopterns insvängningstid τ_{in} låg mellan 11,8 och 14,2 sekunder.

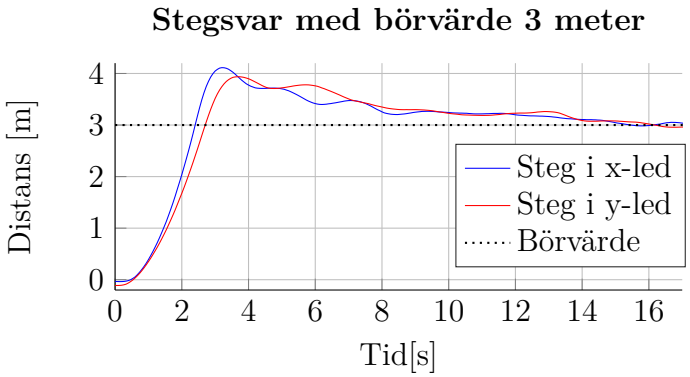
Tabell 5.1: Uppmätta stegsvarsparametrar.

Led	Börvärde [m]	Stigtid τ_{stig} [s]	Insvängningstid τ_{in} [s]	Översväng Θ [%]
x	3	1,4	13,2	37
x	5	1,8	12,4	44
y	3	1,7	13,6	31
y	5	2,1	14,4	37
z	1	2,6	11,8	3,5

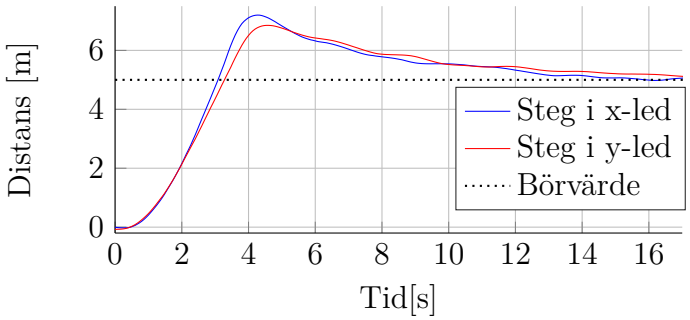
5.2 Precision i positionsbestämning

Vid det första testet av quadrocopterns förmåga att färdas en angiven sträcka, uppmättes stora avvikelser från de givna börvärdena 3 och 5 meter i både x- och y-led, vilket framgår tydligt i Figur 5.2a och 5.2b.

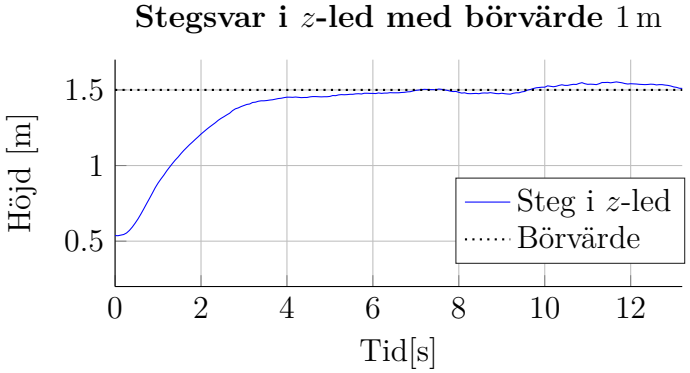
För att korrigera för felen användes de uppmätta medelvärdena; 4,1 meter i x-led vid börvärde 3 meter, respektive 5,8 meter i y-led vid börvärde 5 meter: I mjukvaran



(a) Stegsvar i x– och y–led med börvärde 3 meter. Stigtiden var kort, men översvänget stort och insvängningstiden lång. Insvängningen skedde med svårförklarliga oscillationer.

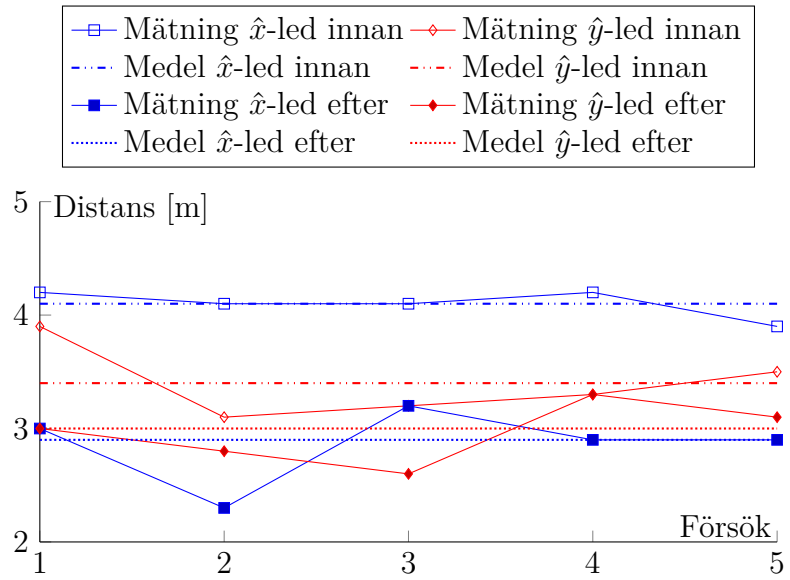


(b) Stegsvar i x– och y–led med börvärde 5 meter. Stigtiden var kort, men översvänget stort och insvängningstiden lång.

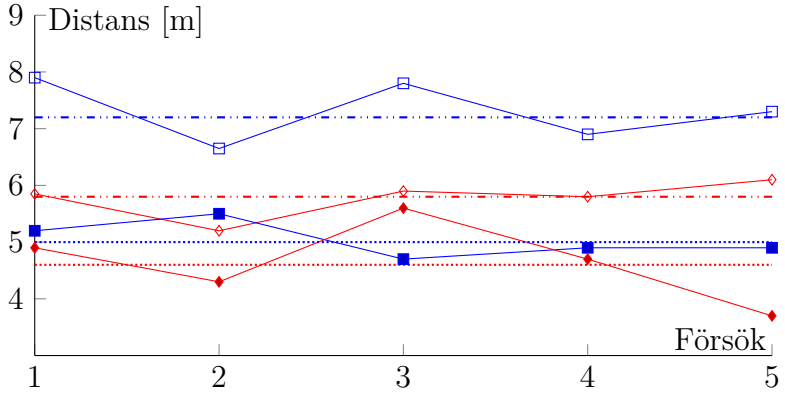


(c) Stegsvar i z–led med börvärde 1,5 meter. Inget översväng förekom, men insvängningstiden var lång.

Figur 5.1: Uppmätta stegsvar i x– och y–led med börvärden 3 respektive 5 meter och i z–led med börvärde 1,5 meter. I horisontalplanet förekom ordentliga översväng, men quadrocoptern stabiliserar sig efter en stund. I z–led var översvänget obefintligt, men insvängningstiden lång.



(a) Färdad sträcka med börvärde 3 meter innan och efter kalibrering. Notera att en viss varians förekom i både x- och y-led och att medelvärdena var närmre börvärdet 3 meter efter kalibrering än innan.



(b) Färdad sträcka med börvärde 5 meter innan och efter kalibrering. Både innan och efter kalibreringen förekom en viss varians i mätningarna, dock hamnade medelvärdena närmre börvärdet 5 meter då en kalibrering genomförts.

Figur 5.2: Färdade distanser med börvärden 3, respektive 5 meter. I båda fallen förekom viss varians, dock medförde en kalibrering att medelvärdena stämde väl överens med börvärdena.

multiplicerades given indata med faktorerna $^{3}/_{4,1}$ respektive $^{5}/_{5,8}$. När nya tester genomfördes upptäcktes att korrigeringen i x-led inte medförde en korrekt färdad sträcka. Efter att ha varierat korrigeringsfaktorn och genomfört testflygningar visade det sig att en faktor $^{3}/_{3,7}$ fungerade bättre än $^{3}/_{4,1}$. De faktorer som därefter användes i koden var därför:

$$k_{fx} = \frac{5}{5,8} \approx 0,86$$
$$k_{fx} = \frac{3}{3,7} \approx 0,81.$$

Resultatet av korrigeringarna blev som väntat att medelvärdena stämde bättre överens med börvärdena, vilket redovisas i Figur 5.2a och 5.2b. Däremot förändrades inte varianserna märkbart, vilket även det var väntat eftersom endast en translation av angiven sträcka utfördes.

5.3 Vinkelupptagning av bottenkamera

Vid testet i Avsnitt 4.3 placerades quadrocoptern 74 centimeter över golvet varefter avstånden $\Delta x = 40\,\text{cm}$ och $\Delta y = 69,5\,\text{cm}$ mättes upp. Med ekvation (4.1) beräknades sedan vinklarna:

$$\theta_x = 30,2\,^\circ$$
$$\theta_y = 50,3\,^\circ.$$

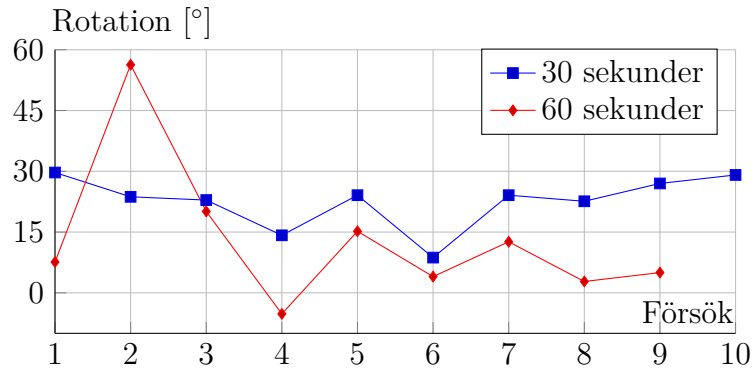
5.4 Yaw–drift över tid

Mätresultaten av quadrocopterns drift i yaw–led visas i Figur 5.3a, 5.3b och 5.3c.

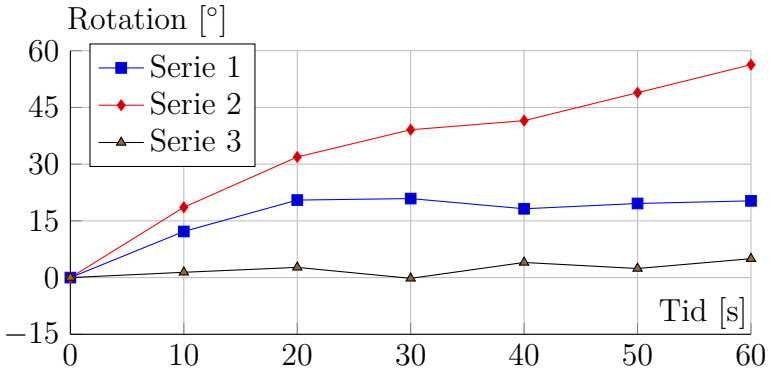
Det första testet visade att quadrocopterns drift inte skedde på ett konsekvent sätt, vilket syns i Figur 5.3a. Vid de kortare flygningarna var den totala driften större (i snitt 22,61 grader) än vid de längre flygningarna (i snitt 13,2 grader). Dessutom förekom både med- och moturs riktade rotationer utan något uppenbart mönster. Notera att quadrocopterns batteri blev helt urladdat i samband med den femte av de 60 sekunder långa flygningarna. De efterkommande mätningarna utfördes därför med ett fulladdat batteri.

Resultatet från det andra testet där flygningarna 2, 3 och 4 från det första testet analyserades redovisas i Figur 5.3b. Inte heller här kunde ett mönster urskiljas. Å ena sidan verkade *Serie 2* att efter 60 sekunder fortsätta sin drift med en bibehållen vinkelhastighet, men å andra sidan tenderade *Serie 1* och *Serie 3* att hålla ett konstant yaw–värde utan stora förändringar. Testet visade alltså att driften inte skedde med en konstant vinkelhastighet.

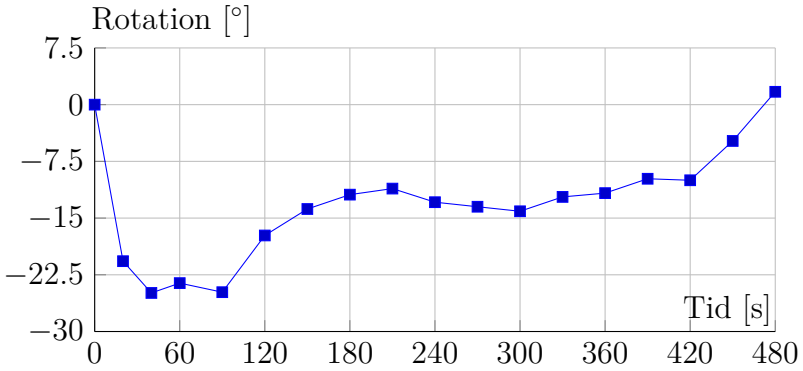
Det tredje testet visade att quadrocoptern vid en längre flygning inte stabiliserade



(a) Total drift i yaw-led vid 30 respektive 60 sekunders hovring. Notera att inget mönster går att urskilja, men att drift förekommer vid alla flygningar.



(b) Uppdelad drift i yaw-led för tre flygningar under 60 sekunder. En mer detaljerad redovisning av quadrocopterns drift under 60 sekunder långa flygningar. Noterbart är att driften sker jämnt och utan häftiga riktningsförändringar.



(c) Drift i yaw-led under en 8 minuter lång hovring. Den viktigaste observationen är att quadrocoptern inte stabiliserade sig kring en viss vinkel och att den drev i båda riktningarna.

Figur 5.3: Uppmätt drift i yaw-led i olika mätfall. Resultat från tre olika mätningar av quadrocopterns tendens att driva i yaw-led vid hovring.

sig vid ett visst yaw–värde, vilket syns i Figur 5.3c. Dessutom framgick det återigen tydligt att vinkelhastigheten varierade i både storlek och tecken.

5.5 Användarscenario

När det tänkta användarscenariot testades, visade det sig snabbt att quadrocopterns drift i yaw–led hade en stor negativ inverkan på resultatet: I enlighet med resultaten i Avsnitt 5.4, vred sig quadrocoptern ömsom kraftigt, ömsom inte alls. De flygningar då liten, eller ingen drift förekom, gick på det stora hela bra: quadrocoptern flög i det tänkta rutmönstret (se Figur 3.4b) och lyckades lokalisera de markörer som den flög över. Absolut vanligast förekommande var dock flygningar med mycket drift. I de fallen kom quadrocoptern fort ur kurs och försöket fick avbrytas.

I samtliga flygningar förekom betydande översväng, vilket också uppmättes i Avsnitt 5.1. För att inte riskera kollision med väggar behövde därför sökområdet definieras så att det fanns en säkerhetsmarginal på fem meter i den ledden där de längre flygningarna genomfördes, det vill säga i x–led (se Figur 3.4).

Gällande quadrocopterns förmåga att detektera markörer, så förekom inget fall då en markör flögs över utan att detekteras. Den valda arbetshöjden två meter fungerade alltså mycket bra för ändamålet.

6

Diskussion

I följande kapitel förs en diskussion kring mätvärden från Kapitel 5 och förbättringsförslag presenteras.

6.1 Utvärdering av testresultat

Detta underavsnitt är till för att tolka och utvärdera mätresultaten som erhölls från testerna i Kapitel 4.

6.1.1 PID–parametrar

Från stegsvaren i Avsnitt 5.1 framgick ett visst översväng uppkom vid förflyttning i horisontalplanet. Detta översväng tenderar utifrån mätresultaten att bli större ju längre sträcka som quadrocoptern färdas. Insvängningstiden verkar inte ha ett samband med sträcka utan kan vara olika beroende på riktning. En förklaring till dessa faktum kan vara att längre sträckor innebär att quadrocoptern hinner få upp en högre fart. En möjlig anledning till den långa insvängningstiden kan vara integratoruppvridning, vilket innebär att styrsignalen från regulatorn mättas och det tar lång tid för positionen att närma sig referensvärde. Detta är ett känt problem kring diskreta PID–regulatorer på integrerande form [12] och diskuteras vidare i Avsnitt 6.2.2.

Det kraftiga översvänget begränsade användarscenariot avsevärt då större avsökningsområden med längre sträckor krävde god säkerhetsmarginal runt området för att ske utan kollision. Vid mindre avsökningsområden (sidor kortare än fem meter långt) upplevdes dock parametrarna vara tillräckligt bra. Därför togs ett beslut om att inte lägga tid på att ta fram bättre lämpade parametrar genom antingen iterativa experiment eller en matematisk modellering. Fokus flyttades istället till att utveckla mjukvara för att nå de uppsatta målen.

6.1.2 Precision i positionsbestämning

Att avvikelserna blev så stora som i testresultaten från Avsnitt 5.2 tros vara en effekt av osäker positionsuppskattning med odometri. Faktumet att samtliga flygningar innan korrigeringen överskred de givna börvärdena tyder på att mindre fel vid varje uppskattning adderades ihop till ett större fel, vilket enligt boken *Robotics* [23] kan bero på allt för långsam uppdatering av positionsdata. Detta påstående styrks i rapporten *Low-latency event-based visual odometry* av A. Censi & D. Scaramuzza [24] där optisk odometri utvärderas. Författarna skriver bland annat följande:

> *"At the state of the art, the latency of a CMOS-based pipeline is at a minimum in the order of 50-250 ms and the sampling rate is in the order of 15-30 Hz. To obtain more agile robots, we need to switch to faster sensors."*
> A. Censi & D. Scaramuzza,

Istället använde författarna en sensor med uppdateringsfrekvensen 500 Hz, vilket gav ett bra resultat. I det här projektet skedde uppdateringar av positionsdata med frekvensen 15 Hz, vilket av citatet att döma var i långsammaste laget för snabba förflyttningar och alltså troligen orsaken till odometrins oförmåga att exakt uppskatta quadrocopterns position.

En tänkbar åtgärd var därför att sänka farten som quadrocoptern flög med. Dock påvisades ingen förbättring då den lägre farten innebar att flygningen tog så lång tid att det inte gick att genomföra något användarscenario. Därför beslutades det att återgå till den ursprungliga farten trots den bristande precisionen.

För att komma runt problemen med den låga uppdateringsfrekvensen och samtidigt göra quadrocoptern oberoende av en basstation kan beräkningar av position och styrkommandon ske i en mikroprocessor ombord på quadrocoptern. Detta undersöktes av *J. J. Lugo and A. Zell* från universitetet i Tübingen, Tyskland [25]. De studerade hur noggrann odometri var då beräkningar utfördes med en extern mikrokontroller monterad på en AR.Drone. Deras slutsats blev att odometri är tillräckligt noggrann för autonom flygning och att deras system var tillräckligt precis för att flyga i trånga miljöer, vilket inte stämmer överens med våra mätresultat. Att utföra beräkningar ombord på quadrocoptern verkar alltså medföra stora prestandavinster jämfört med att, som i det här projektet, sköta beräkningar på en markbaserad styrdator.

Eftersom att de framtagna korrektionsfaktorerna var anpassade för styrkommandon på 3 och 5 meter, är risken stor att korrektionen fungerar sämre vid avsökningsområden med andra dimensioner än dessa. Detta begränsar förstås användarens valmöjligheter gällande avsökningsområde, varför korrektionsfaktorer inte är en optimal lösning. Anledningen till att lösningen ändå användes i projektet var att den ansågs vara tillräckligt bra för att kunna genomföra användarscenariot och att förbättringar av odometrins noggrannhet inte var projektets huvudfokus.

6.1.3 Vinkelupptagning av bottenkamera

De uppmätta upptagningsvinklarna θ_x och θ_y var förvånansvärt små då artikeln *Framework for Autonomous Onboard Navigation with the AR.Drone* [25] berättade om en vinkel på 63°. Inte ens för kamerabildens diagonal var vinkeln så stor ($\approx 57°$). Detta visade på vikten av att utföra egna mätningar och inte lita blint på uppgifter från andra källor.

Förhållandet θ_x:θ_y = 1,67:1 föranledde en revidering av sättet på vilket quadrocoptern svepte över en yta; istället för att färdas de långa sträckorna i y–led, programmerades quadrocoptern att göra det i x–led, vilket illustreras i Figur 3.4b. På så vis maximerades avsökningsvinkeln, varför korridorsbredden kunde ökas och därmed avsökningen effektiviseras.

6.1.4 Yaw–drift över tid

Gemensamt för alla tester är att de individuella serierna genomfördes i en omgång, i samma miljö och vid samma tid. Eventuella systematiska fel bör alltså förekomma i alla mätningar och därmed ej spela någon roll för relativa jämförelser mellan individuella mätningar. Eftersom mätosäkerheten kunde uppskattas till $\pm 0,5°$, kan det antas att mätfel inte orsakade de variationer som uppstod. Inte heller kunde projektets mjukvara orsaka de fel som uppstod eftersom att den officiella applikationen användes. Således borde felet ha legat i hårdvaran, det vill säga magnetometern, eller i Parrots mjukvara, vilken sände ut de navigationsdata som användes till regleringen i projektets kod.

Innan testerna utfördes fanns funderingar kring om det kunde vara så att driften skedde med konstant fart eller bara under flygturens inledande fas. Om så vore fallet, hade mjukvarukorrektioner enkelt kunnat implementeras för att förbättra quadrocopterns flygegenskaper. Tyvärr visade resultaten (se Avsnitt 5.4) att ingen av hypoteserna stämde: Det första testet avslöjade att driften var inkonsekvent i både riktning och storlek och det andra att driftfarten inte var konstant. Dessa slutsatser underströks av resultatet från den tredje mätningen där driften bytte riktning och storlek under flygningens gång. Dessutom dokumenterades drift uppemot 56°, vilka vida översteg magnetometerns mätosäkerhet som enligt Parrot var ±6° (se Avsnitt 2.2.2). De fluktuationer som uppstod berodde med andra ord inte på magnetometerns mätosäkerhet.

En tänkbar förklaring till quadrocopterns oberäkneliga uppförande skulle istället kunna ligga i magnetometerns funktion, vilken beskrevs i Avsnitt 2.2.2. I och med att magnetometern mäter riktningen på det magnetiska flöde som passerar genom den, uppfattar den en förändring av magnetfältet som att en rotation har skett. Därför är det inte otänkbart att det magnetfält som alstras av elektroniken i quadrocoptern har en märkbar inverkan på det totala magnetfält som flödar genom magnetometern: Urladdning och uppvärmning av batteriet, uppladdning av kapa-

citanser, och elektriska motorer med hastigt varierande varvtal är alla källor till elektromagnetisk strålning. Denna teori styrks av en examensrapport skriven av *M. Wells* vid Carleton University i Ottawa, Kanada [26]. Wells konstaterar där att magnetiska störningar från själva UAV:n har en märkbar inverkan på mätningar av jordens magnetfält. Rapporten fokuserar sedan på att utvärdera olika metoder för att genom matematiska modeller kompensera för de misstänkta störningskällorna, vilket görs med ett gott resultat. Även utgivaren av den i UAV-kretsar välrenommerade mjukvaran *ArduPilot* skriver på sin hemsida [27] att magnetiska störningar från UAV:ns elektronik kan orsaka störningar som leder till att UAV:n börjar vrida sig i luften. För att råda bot på detta problem rekommenderas bland annat att magnetometern monteras på en mast som håller den borta från UAV:ns kropp och att aluminiumfolie används för att skärma av elektroniken.

Tänkbara lösningar på problemet kan alltså vara att förändra hårdvaran för att minska exponeringen för magnetiskt flöde eller att med hjälp av mjukvara kompensera för de störningar som förekommer. En annan lösning skulle kunna vara att använda en eller flera magnetometrar i kombination med andra sensorer såsom bildbehandling, GPS eller gyroskop för att minimera risken för fel från störningar. I det här projektet valdes att inte utforska detta ytterligare då tidsbrist rådde.

Klart är i alla fall att en precis och pålitlig mätmetod för bestämning av yaw-riktning är en förutsättning för autonoma flygningar med quadrocoptrar och att en sådan metod inte var möjlig med hårdvaran som användes i projektet.

6.1.5 Användarscenario

Att flygningarna oftast slutade med att quadrocoptern roterade ur kurs var väntat efter tidigare mätresultat. Inte heller förekomsten av översväng var överraskande. Positiv var dock quadrocopterns förmåga att detektera de markörer som den flög över: Att inte missa en enda markör under de tiotal testflygningar som utfördes sågs som en framgång.

Tänkbara lösningar till problemen med översväng och yaw-drift diskuteras i Avsnitten 6.1.1 och 6.1.4. För att arbeta runt problemet med översväng skulle de längre sträckorna ha kunnat delas in i kortare delsträckor. I och med att de procentuella översvängen var mindre för kortare flygsträckor (se Avsnitt 5.1), hade kortare flygsträckor inneburit mindre översväng och därmed en förbättrad säkerhet för omgivningen och quadrocoptern. Denna indelning skulle dock medföra en långsammare avsökning av området i och med att quadrocoptern skulle ha fler punkter att stabilisera sig vid, vilket i Avsnitt 5.1 visade sig ta lång tid även för kortare delsträckor. Den längre flygtiden skulle öka risken för förekomsten av drift, vilket bland annat Figur 5.3c visar. Således är det oklart om viapunkter skulle innebära en förbättring eller försämring av quadrocopterns prestanda.

Valet av avsökningsmetod innebar som tidigare nämnts att quadrocoptern hela tiden pekade åt samma håll och förflyttade sig i räta linjer. Detta val grundade sig dels

i att det ansågs vara enklare att implementera i kod och dels i att risken ansågs stor att upprepade förändringar i yaw–värde skulle öka felet i yaw-led ytterligare. För rektangulära sökområden var denna svepmetod inte heller särskilt ineffektiv: De enda ”onödiga” sträckorna som färdades var från quadrocopterns startposition till sökområdets hörn i $(x_0; y_0)$ och tillbaka till startpositionen när flygrutten var klar (se Figur 3.4a). För mer avancerade sökområden med oregelbundna former och icke tillåtna delområden skulle metoden dock inte fungera. Istället skulle en mer avancerad algoritm för ruttplanering behövas. Detta prioriteras dock bort på grund av tidsbrist.

6.2 Förbättringsförslag

I det här avsnittet har tankar kring vilka åtgärder som skulle kunna vidtas för att förbättra de autonoma funktionerna hos quadrocoptern sammanställts.

6.2.1 Positionsbestämning

Resultaten från mätningarna på precision vid förflyttning (Avsnitt 5.2) och yaw-drift (Avsnitt 5.4) visade att quadrocopterns förmåga att uppskatta sitt tillstånd var bristande. I Avsnitten 6.1.2 och 6.1.4 diskuteras därför olika lösningar för att förbättra precisionen och pålitligheten för AR.Drone 2.0 som endast använder sina interna sensorer. För att få en än mer exakt positionsbestämning behövs dock ett externt referenssystem istället för att förlita sig helt på odometri. Ett exempel på den precision som då kan uppnås ges i en rapport skriven av *D. Brescianini*, *M. Hehn* och *R. D'Andrea* vid ETH i Zürich [28]. Med hjälp av ett externt kamerasystem designade de ett system där två quadrocoptrar lyckades kasta en upprätt stav mellan varandra utan att tappa den. System som är beroende av externa och precisa kameror för positionsbestämning tillåter dock inte autonom användning i godtycklig miljö, varför de har ett begränsat användningsområde.

En lösning som skulle fungera i de flesta miljöer är att använda sig av GPS parallellt med odometri. Att endast använda GPS för positionsbestämning är inte särskilt noggrant i och med att dess upplösning enligt den officiella hemsidan för GPS–systemet är i storleksordningen en meter [29]. GPS, i kombination med andra sensorer kan dock ge quadrocoptern en bättre precision i positionsbestämning. Att montera en extern GPS–mottagare på AR.Drone undersöktes under projektets gång men ingen tillräckligt billig och samtidigt noggrann hittades. Dessutom var en av avgränsningarna att endast flyga inomhus, vilket skulle innebära dålig GPS–mottagning.

Ytterligare ett alternativ till odometri är att använda bildbehandling på data från quadrocopterns kameror. Genom att skapa referenspunkter i bilden kan information om quadrocopterns förflyttning i rummet återges mer exakt än med endast odometri. Detta gjordes med stor framgång i en tysk studie av *J. Engel* et al. [30]

där bildbehandling och en PID–regulator användes för att på ett mycket exakt sätt styra en AR.Drone 2.0. Lösningen fungerade bra i alla typer av kontrastrika miljöer och led inte av felfortplantning vilket möjliggjorde precisa manövrar.

6.2.2 Reglering

För att lösa problem med integratoruppvridning kan diskreta PID–regulatorer implementeras på differentiell form istället för integrerande form [12]. Under projektet testades en sådan form men då förbättring uteblev och tidsbrist rådde, valde gruppen att gå vidare med PID–regulatorn på integrerande form istället.

Istället för att använda en PID–regulator kan man använda sig av IMC[1] för att reglera quadrocoptern. I en studie utförd av *A. Hernandez et.al.*[31] jämförs PID–reglering mot IMC–reglering i x–, y– och z–led. De kom fram till att IMC erbjuder bättre styrmöjligheter gentemot PID, med kortare stigtid och insvängningstid.

6.2.3 Användargränssnitt

Projektets programvara körs uteslutande från kommandoprompten i UNIX–baserade operativsystem eller `PowerShell` i Windows. Detta är funktionellt men inte användarvänligt. Därför hade ett grafiskt gränssnitt varit ett trevligt tillskott. Förslagsvis hade man då kunnat klicka fram det område som ska sökas av och därefter i realtid se quadrocopterns position, avsökt område och upphittade markörer. Man hade också kunnat ställa in olika parametrar såsom arbetshöjd och korridorsbredd på ett enkelt sätt.

6.2.4 Robot Operating System

Ett alternativ till den mjukvaruplattform (Node.js) som användes under projektet är Robot Operating System (förkortat ROS), som är ett ramverk som är utvecklat för att skapa robusta system för robotar. ROS är en flexibel och modulbaserad plattform som kan användas för styrning av diverse avancerade robotar. Det stödjer programmering i en mängd olika språk och det finns därmed bibliotek för dessa, ROS fokus ligger dock på *C++* och *Python* [32]. Utifrån dessa bibliotek kan utvecklaren sedan skapa *noder* (moduler), i vilka olika beräkningar kan utföras. Ofta använder man ett flertal noder till en robot. Varje nod programmeras till att hantera ett visst område. Till exempel kan en nod hantera en sensor, en annan kontrollerar motorn och en tredje nod uppdaterar positionen. Noderna skickar respektive data till de noder som behöver den med hjälp av dataströmmar. Detta liknar sättet som Node.js fungerar med sina händelser som skickas och lyssnas efter. Dock kan kommunikationen ske på flera olika sätt när man använder ROS, alla med olika fördelar

[1] Internal Model Control

och användningsområden [33]. Detta kan vara bra om man är insatt i hur plattformen fungerar, men kan vara kritiskt om man inte är det. Att utvecklare delar med sig av sina noder och bibliotek stödjer samarbete mellan utvecklare. Men ROS har en betydligt högre inlärningskurva än Node.js. Detta är en av anledningarna till att Node.js valdes för projektet, ytterligare fördel var Node.js–modulerna `ar-drone` och `ardrone-autonomy`. Detta ledde till att fokus kunde läggas på mer avancerade funktioner. Ett förbättringsförslag är dock att använda `ROS` eftersom det är en plattform som är mer vedertagen, samt att det skulle underlätta implementation av mjukvarulösningen på andra quadrocoptrar än AR.Drone 2.0.

6.2.5 Optimerad flygruttsplanering

I Avsnitt 6.1.5 diskuteras hur utformningen av flygrutten för ett rektangulärt område fungerar bra, men för ett icke-rektangulärt område hade flygrutten inte varit tillräcklig. Ett förbättringsförslag är att optimera flygrutten så att man flyger en så kort sträcka som möjligt, men täcker upp en så stor area som möjligt. Detta är framförallt relevant för områden som har rundade hörn där flygrutten kan utgöras av krökta sträckor, vilka följer områdets kontur. Det hade kunnat genomföras genom att man tar hänsyn till bottenkamerans upptagningsvinkel, var man har flugit och vart man ska flyga, och omprogrammera rutten efter hand som man flyger. Dessutom hade man kunnat lagra alla markör–koordinater och medelvärdesbilda dessa, för att erhålla den mest sannolika positionen i det globala koordinatsystemet för olika markörer. Det innebär att nästa gång man flyger inom samma avsökningsområde kan quadrocoptern flyga till dessa positioner först och förhoppningsvis hitta markören. Om det inte hittas några markörer på dessa positioner läggs rutten om och quadrocoptern börjar söka av området. Ytterligare kan möjligheten att kunna modifiera rutten från basstationen under flygning implementeras, ifall användaren har en bra översikt över området och kan utgöra var en eller flera markörer befinner sig.

Ytterligare en utökning av ruttplaneraren hade varit att användaren kan välja områden som quadrocoptern inte ska flyga i. Detta är användbart för att undvika hinder. Användaren hade då till exempel kunnat ange områden där det finns träd och liknande. Vidare skulle flygrutten även kunna förändras dynamiskt under rutten, genom att fler sensorer installeras på quadrocoptern, så som IR–sensorer som kan observera omgivning. Även bildbehandling på data från quadrocopterns frontkamera skulle kunna användas för att identifiera hinder under färd.

6.2.6 Särskiljning av flera markörer

Koden som skrivits i projektet kunde endast uppfatta och behandla en markör åt gången och leta efter en sorts markör. Det finns dock stöd i den öppna källkod som projektet byggde på för att identifiera flera olika sorters markörer, vilket hade kunnat möjliggöra fler användningsområden. I en studie gjord av *R. Barták*, *A.*

Hraško och *D. Obdrzálek* [34], användes en egendesignad markör tillsammans med bildbehandling för att autonomt landa med hög precision. Genom att kunna definiera egna markörer möjliggör det flera användningsområden där det är intressant att kategorisera eller lokalisera olika objekt.

6.2.7 Landning med hög precision

I Avsnitt 6.1.2 fastslås det att precisionen i positionsbestämningen är bristande, troligen på grund av en sämre noggrannhet i positionsuppskattningen med odometri. Ifall quadrocoptern ska landa på en bestämd plats som till ytan är mycket liten, behövs en landningsteknik med hög precision. Det skulle kunna implementeras med hjälp av en markör och bildbehandling. Tanken är då att quadrocoptern placeras på en markör innan den sedan lyfter och utför sin flygrutt. När rutten är färdigflugen eller om alla markörer har hittats kommer quadrocoptern att flyga tillbaka till basmarkören, där den återigen stabiliserar sig och landar kontrollerat på markören.

För att kunna implementera en sådan lösning kan man använda sig av tekniken som *R. Barták*, *A. Hraško* och D. *Obdrzálek* beskriver i studien *A Controller for Autonomous Landing of AR.Drone* [34]. Deras metod var att använda bildbehandling och en PID–regulator för att positionera sig rakt över en markör och landa på den. Ett problem som de upptäckte var att när quadrocoptern kommer för nära marken så kunde den inte längre se markören. Detta löste de genom att använda sig av ett kontrollerat fall de sista 30 centimetrarna. Deras slutsats var att lösningen är pålitlig vid flygning inomhus.

7

Slutsatser

Projektet blev i sin helhet lyckat, då mjukvaran planerade och utförde flygrutten, samt upptäckte och lagrade markörers positioner. Användarscenariot kunde utföras, dock med vissa svårigheter, eftersom det förekom en drift i yaw-led.

Den valda mjukvaruplattformen Node.js visade sig vara väl lämpad för att styra en AR.Drone 2.0 i och med att kommunikationen skedde via ett trådlöst nätverk och Node.js är utvecklat i syfte att snabbt hantera nätverkstrafik.

För framtida projekt där autonoma flygningar eftersträvas bör dock en mer öppen hårdvaruplattform användas så att ruttplanering, bildbehandling med mera kan skötas ombord på quadrocoptern. En AR.Drone 2.0 kräver konstant kontakt med en markbaserad styrdator vilket begränsar räckvidden och minskar precisionen vid positionsuppskattning.

Under mjukvaruutvecklingen gjordes många prioriteringar, vilka medförde att funktionaliteten begränsades. Den utvecklade mjukvaran hade dock vid projektets slut alla komponenter som behövdes för att utföra ett enkelt avsökningsuppdrag och sedan återvända till startpositionen. Dessvärre var den valda metoden för bestämning av quadrocopterns position i rummet otillräcklig och orsakade felaktigheter vid förflyttningar vilket innebar att ingen komplett flygning kunde genomföras.

Trots avsaknaden av en komplett slutdemo anses projektet vara lyckat och vi påstår därför att det är fullt möjligt att efter vidare utvecklingsarbete konstruera ett system som använder sig av en quadrocopter för att utföra autonoma avsökningsuppdrag.

8

Litteratur

[1] (4 juni 2015). Google trender, URL: http://www.google.com/trends/explore#q=/m/0g2bc (hämtad 2015-06-04).

[2] (17 juni 2013). Teal group predicts worldwide uav market will total $89 billion in its 2013 uav market profile and forecast, URL: http://tealgroup.com/index.php/about-teal-group-corporation/press-releases/94-2013-uav-press-release (hämtad 2015-06-04).

[3] J. Wilson, "Uav roundup 2013", Aerospace America, Report, juli 2013. URL: http://www.aerospaceamerica.org/Documents/AerospaceAmerica-PDFs-2013/July-August-2013/UAVRoundup2013t-AA-Jul-Aug2013.pdf.

[4] R. Ballesteros, J. F. Ortega, D. Hernandez och M. A. Moreno, "Applications of georeferenced high-resolution images obtained with unmanned aerial vehicles. part ii: Application to maize and onion crops of a semi-arid region in spain", *Precision Agriculture*, vol. 15, nr 6, s. 593–614, 2014. DOI: 10.1007/s11119-014-9357-6.

[5] T. Inoue, S. Nagai, S. Yamashita, H. Fadaei, R. Ishii, K. Okabe, H. Taki, Y. Honda, K. Kajiwara och R. Suzuki, "Unmanned aerial survey of fallen trees in a deciduous broadleaved forest in eastern japan", *Plos One*, vol. 9, nr 10, s. 7, 2014. DOI: 10.1371/journal.pone.0109881.

[6] M. Funaki, S. I. Higashino, S. Sakanaka, N. Iwata, N. Nakamura, N. Hirasawa, N. Obara och M. Kuwabara, "Small unmanned aerial vehicles for aeromagnetic surveys and their flights in the south shetland islands, antarctica", *Polar Science*, vol. 8, nr 4, s. 342–356, 2014. DOI: 10.1016/j.polar.2014.07.001.

[7] B. Adler, J. H. Xiao och J. W. Zhang, "Autonomous exploration of urban environments using unmanned aerial vehicles", *Journal of Field Robotics*, vol. 31, nr 6, s. 912–939, 2014. DOI: 10.1002/rob.21526.

[8] M. Rock. (2015). Why commercial drones are the best or worst things to happen to the world in a long time, URL: http://2machines.com/184119/ (hämtad 2015-06-04).

[9] R. Gruber, "Commercial drones and privacy: Can we trust states with "drone federalism"?", *Richmond Journal of Law & Technology*, vol. 21, 4 2015.

[10] Parrot. (2015). Technical specifications, URL: http://ardrone2.parrot.com/ardrone-2/specifications/ (hämtad 2015-03-21).

[11] Mando. (2015). Gyroscope, URL: https://learn.sparkfun.com/tutorials/gyroscope (hämtad 2015-05-19).

[12] T. Glad och L. Ljung, *Reglerteknik*, 5:10. Lund: Studentlitteratur, 2004.

[13] Wikipedia. (2015). First-class, URL: http://en.wikipedia.org/wiki/First-class_function (hämtad 2015-05-12).

[14] W3schools.com. (2015). Callback-function, URL: http://www.w3schools.com/jquery/jquery_callback.asp (hämtad 2015-05-12).

[15] Node.js. (2015). Events — node.js, URL: https://nodejs.org/api/events.html (hämtad 2015-05-12).

[16] Wikipedia. (2015). Object-oriented programming — wikipedia, the free encyclopedia, URL: http://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=665327320 (hämtad 2015-06-04).

[17] F. Scholz. (11 maj 2015). Introduction to object–oriented javascript, Mozilla Developer Network, URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript (hämtad 2015-06-05).

[18] Wikipedia. (2015). Node.js — wikipedia, the free encyclopedia, URL: http://en.wikipedia.org/w/index.php?title=Node.js&oldid=661224879 (hämtad 2015-05-12).

[19] N. Inc. (26 nov. 2014). What is npm? *Youtube*, URL: https://www.youtube.com/watch?v=pa4dc480Apo (hämtad 2015-05-12).

[20] GitHub. (2015). Github, URL: https://github.com (hämtad 2015-05-18).

[21] Wikipedia. (2015). Mask (dator), URL: http://sv.wikipedia.org/wiki/Mask_(dator) (hämtad 2015-05-18).

[22] ——, (2015). Application programming interface — wikipedia, URL: //sv.wikipedia.org/w/index.php?title=Application_Programming_Interface&oldid=29941023 (hämtad 2015-05-14).

[23] A. knowledge solutions, *Robotics*, 1. utg. Hingham: Infinity Science Press LLC, 2007.

[24] A. Censi och D. Scaramuzza, "Low-latency event-based visual odometry", i *IEEE International Conference on Robotics and Automation (ICRA)*, maj 2014. URL: http://purl.org/censi/2013/dvsd.

[25] J. J. Lugo och A. Zell, "Framework for autonomous onboard navigation with the ar.drone", Cognitive Systems, Tübingen Germany, Report, maj 2013.

[26] M. Wells, "Attenuating magnetic interference in a uav system", Masters thesis, Carleton University, 2008.

[27] ArduPilot. (2015). Magnetic interference, URL: http://copter.ardupilot.com/wiki/common-appendix/automatic-compass-declination/ (hämtad 2015-05-16).

[28] D. Brescianini, M. Hehn och R. D'Andrea, "Quadrocopter pole acrobatics", i *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, IEEE, 2013, s. 3472–3479.

[29] GPS.gov. (2015). Gps accuracy, URL: `http://www.gps.gov/systems/gps/performance/accuracy/` (hämtad 2015-05-19).

[30] J. Engel, J. Sturm och D. Cremers, "Scale-aware navigation of a low-cost quadrocopter with a monocular camera", *Robotics and Autonomous Systems*, vol. 62, nr 11, s. 1646–1656, 2014.

[31] A. Hernandez, C. Copot, R. D. Keyser, T. Vlas och I. Nascu, "Identification and path following control of an ar.drone quadrotor", 2013, s. 583–588.

[32] wiki.ros.org. (2015). Ros, URL: `http://wiki.ros.org/Client%20Libraries` (hämtad 2015-05-08).

[33] ——, (2015). Ros, URL: `http://wiki.ros.org/Nodes` (hämtad 2015-05-08).

[34] R. Barták, A. Hraško och D. Obdrzálek, "A controller for autonomous landing of ar.drone", 2014, s. 329–334.