ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jet Propulsion Laboratory
California Institute of Technology

Autonomous Systems Lab
Prof. Roland Siegwart

Master Thesis

Guidance, Navigation, Control and Mission Logic for Quadrotor Full-cycle Autonomy

Spring Term 2018

Supervised by:
Roland Brockers, Dr.-Ing.
Thomas Stastny
Timo Hinzmann

Author:
Danylo Malyuta

This page is intentionally left blank.

To my family, who taught me to live, to my teachers, who taught me to learn, and to the explorers, who taught me to dream.

This page is intentionally left blank.

Contents

Acknowledgment

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Zdeněk and Michaela Bakala Foundation, the Swiss-American Society and the National Aeronautics and Space Administration.

This page is intentionally left blank.

Preface

" *We are embarked as pioneers upon a new science and industry in which our problems are so new and unusual that it behooves no one to dismiss any novel idea with the statement that 'it can't be done!'* " – William E. Boeing

This project was undertaken in the Micro Aerial Vehicle Laboratory at JPL. I owe my deep gratitude to Dr. Roland Brockers for accepting and for supporting me throughout this thesis. I am immensely grateful for the liberty that you have given me to develop my own engineering solutions and for your insistence to keep my solutions simple due to their inherently superior robustness and commercial value. I am incredibly lucky to have been allowed to develop an end-to-end autonomous system and I am in debt for having had this opportunity.
Of course, it takes a village to complete such a large project. At JPL, I am very thankful to Christian Brommer for his fantastic systems knowledge, his technical skill, his presence during long nights of flight testing and his patience with fixing a myriad of bugs that are so characteristic of a young system. I am grateful to Jeff Delaune and Rohan Thakker for critically insightful discussions about guidance, navigation and control. I am equally thankful to Benjamin Companeitz and Bhairav Mehta for their electronics and software engineering experience. Thank you to Theodore Tzanetos and Dillon Azzam for organizing the motor identification test stand and for generously helping me to get up and running with it. To the Unnamed JPL Engineer, thank you for your help and for being part of a vibrant, ambitious and truly world-changing community that is the Jet Propulsion Laboratory.
I would like to thank my excellent ETH Zurich supervisors Thomas Stastny and Timo Hinzmann from the Autonomous Systems Lab for their much appreciated advice. I am also thankful to Prof. Roland Siegwart not only for agreeing to supervise this thesis but also for his welcome encouragement to pursue team-based project work dating as far back as my first semester at ETH Zurich.
Finally, I would like to thank my family. Ever since I was 6 years old I have been in a pursuit to become a maverick aerospace engineer. You have supported me throughout my life and this has been as true as ever for this thesis.
The project presented herein has been a monumentally challenging and infinitely rewarding endeavor. To keep spirits high, I had poems and quotes from inspiring leaders and people whom I simply respect pasted above my JPL desk. The above quote by William Boeing is one of them. This project has reinforced in me the core principle that continues to guide my engineering career – if you work hard enough, any problem is at some point bound to succumb to your will. As mavericks and inventors have shown and continue to show, nothing is impossible and we should all be foolish enough to remember it.

This page is intentionally left blank.

Abstract

Autonomous multirotor Micro Aerial Vehicles (MAVs) are currently being actively developed for personal, commercial and military applications. Three of the most rapidly growing use cases are precision farming, environmental surveying and surveillance. These markets require full-cycle autonomous MAVs i.e. robots that can not only perform single-flight autonomous data collection but that can also automatically recharge so as to sustain long term operation with no human in the loop. No such suitable commercial MAV exists today.
Motivated by this market gap, this thesis pioneers a full-cycle autonomous MAV system using an AscTec Pelican quadrotor. Designed for a data collection mission, the MAV takes off, flies a waypoint trajectory, lands on a contact-based charging pad, recharges and then repeats the cycle all without human presence. This thesis develops the entire guidance, visual landing navigation, control and autonomy engine subsystems from the ground up.
After describing the hardware and software architectures, a visual landing navigation pipeline is developed based on Recursive Least Squares AprilTag bundle pose estimation. Then, guidance is explained which consists of polynomial trajectory generators, a real time trajectory sequencer and a trajectory tracker. A full cascaded control loop is designed next and this report explains its complete development starting from a flight dynamics model, then system identification, then controller theoretical design followed by implementation peculiarities, parameter tuning and finally an enlightening performance and robustness analysis. With the guidance, navigation and control system available, a state machine based autonomy engine is developed to carry out a full-cycle autonomous data collection mission. We finally present how these subsystems perform individually and as a complete, working system. Data is taken from a Software In The Loop high fidelity simulation as well as indoor and outdoor real life flight tests. The highlights of this body of work are:

- An AprilTag 2 based visual landing navigation pipeline, an evaluation of bundle pose measurement accuracy and a definition of ideal bundle geometry;
- A guidance trajectory sequencer that implements a well-defined set of real time trajectory sequencing mechanics;
- A complete three stage cascaded control loop with novel quaternion-based yaw control;
- A hierarchical autonomy engine based on self developed, Unified Modeling Language compatible state machines;
- A modified RotorS simulator that can run a Software In The Loop simulation of the entire system presented herein.

Keywords: Guidance, Navigation, Control, Autonomy engine, Takeoff, Vision-based landing, Polynomial trajectories, State machines, System identification, Tuning, AprilTag, PID.

This page is intentionally left blank.

Nomenclature

Variables

Symbol	Description	Units
$\alpha_{\text{tilt,max}}$	Maximum thrust vector reference tilt angle	rad
α_{tilt}	Thrust vector reference tilt angle	rad
$\bar{f}_i$	Load-cell thrust of the i-th propeller	N
$\bar{f}_{\text{max}}$	Maximum load-cell thrust	N
$\bar{f}_{\text{min}}$	Minimum load-cell thrust	N
$\bar{\boldsymbol{a}}_{\text{ref}}$	Gravity-compensated reference acceleration	$^{\text{m}}/_{\text{s}^2}$
β_{axis}	Boolean which is **true** to activate control along an axis (one such Boolean for each translation axis)	-
$\beta_{\text{emergency}}$	Boolean which activates the emergency landing controller when **true**	-
$\beta_{\text{prefilter}}$	Boolean which is **true** to use the reference pre-filters	-
$\beta_{\text{reset,ext}}$	Boolean which is **true** during control mode switching (externally triggered)	-
$\beta_{\text{reset,int}}$	Boolean which is **true** during translation control parameter changes	-
β_{reset}	Boolean which, when **true**, triggers a quasi-bumpless translation controller reset	-
$\boldsymbol{\eta}_i$	Net (aerodynamic) torque generated by the i-th propeller reduced to the propeller geometric center	N·m
$\boldsymbol{\eta}_{\text{ex}}$	Total exogenous torque at the quadrotor CoG	N·m
$\boldsymbol{\eta}_{\text{other},i}$	Unmodelled component of $\boldsymbol{\eta}_{\text{i}}$	N·m
$\boldsymbol{\eta}_{\text{P},i}$	Pitching torque of i-th propeller	N·m
$\boldsymbol{\eta}_{\text{Q},i}$	Drag torque of i-th propeller	N·m
$\boldsymbol{\eta}_{\text{R},i}$	Rolling torque of i-th propeller	N·m
$\boldsymbol{\eta}_{\text{total}}$	Total torque acting at the quadrotor CoG	N·m
$\boldsymbol{\omega}_{\text{ff}}$	Control feed-forward body rates (in the body frame)	$^{\text{rad}}/_{\text{s}}$

$\boldsymbol{\omega}_{\text{ref}}$	Reference body rates	$^{\text{rad}}/_{\text{s}}$
$\boldsymbol{\xi}$	Quadrotor body torque	N·m
$\boldsymbol{\zeta}_{\text{rotor}}$	Angular momentum of all propellers and motor rotors	$^{\text{kg·m}^2}/_{\text{s}}$
$\boldsymbol{\zeta}_{\text{total}}$	Quadrotor total angular momentum	$^{\text{kg·m}^2}/_{\text{s}}$
$\boldsymbol{a}_{\text{ff}}$	Control feed-forward acceleration (in the world frame)	$^{\text{m}}/_{\text{s}^2}$
$\boldsymbol{a}_{\text{ref}}$	Reference acceleration output by the translation controller	$^{\text{m}}/_{\text{s}^2}$
$\boldsymbol{b}_{\omega}$	Gyro measurement bias ($\in \mathbb{R}^3$)	$^{\text{rad}}/_{\text{s}}$
$\boldsymbol{b}_{\text{a}}$	Accelerometer measurement bias ($\in \mathbb{R}^3$)	$^{\text{rad}}/_{\text{s}}$
$\boldsymbol{e}_1$	Unit vector $(1, 0, 0)$	-
$\boldsymbol{e}_2$	Unit vector $(0, 1, 0)$	-
$\boldsymbol{e}_3$	Unit vector $(0, 0, 1)$	-
$\boldsymbol{e}^{\text{b}}_{\text{z,ref}}$	Reference thrust unit vector	-
$\boldsymbol{f}_i$	Net (aerodynamic) force generated by the i-th propeller reduced to the propeller geometric center	N
$\boldsymbol{f}_{\text{d}}$	Stochastic disturbance force modeling wind for RotorS precision landing experiment	N
$\boldsymbol{f}_{\text{ex,other}}$	$\boldsymbol{f}_{\text{ex}}$ contributors other than aerodynamic drag	N
$\boldsymbol{f}_{\text{ex}}$	Net exogenous force acting at the quadrotor CoG	N
$\boldsymbol{f}_{\text{H},i}$	Drag force or H-force of i-th propeller	N
$\boldsymbol{f}_{\text{other},i}$	Unmodelled component of $\boldsymbol{f}_i$	N
$\boldsymbol{f}_{\text{T},i}$	Thrust force of i-th propeller	N
$\boldsymbol{f}_{\text{Y},i}$	Side force of i-th propeller	N
$\boldsymbol{g}$	Gravity vector $(0, 0, -9.81)$	$^{\text{m}}/_{\text{s}^2}$
$\boldsymbol{k}_{\text{p},i}$	Propeller plane normal which points in the propeller rotation sense (via right hand rule)	-
$\boldsymbol{k}'_{\text{p},i}$	Propeller plane normal which always points up (i.e. $\boldsymbol{k}'_{\text{p},i} \cdot \boldsymbol{e}^{\text{b}}_{\text{z}} > 0$)	-
$\boldsymbol{l}_i$	Vector from quadrotor CoG to the i-th propeller geometric center	m
$\boldsymbol{p}_{\text{align}}$	Pad-aligned body frame position for landing	m
$\boldsymbol{p}_{\text{ref}}$	Control reference position (in the world frame)	m
$\boldsymbol{p}_{\text{rth}}$	Return to home $(\boldsymbol{e}^{\text{w}}_{\text{x}}, \boldsymbol{e}^{\text{w}}_{\text{y}})$ position	m
$\boldsymbol{p}_{w_i}$	Position of waypoint i in the data acquisition trajectory	m
$\boldsymbol{q}_{\text{align}}$	Pad-aligned body frame attitude for landing	-
$\boldsymbol{q}_{\text{e,rp}}$	Roll-pitch error quaternion (also known as the reduced error quaternion)	-

$\boldsymbol{q}_{\mathrm{e,y,h}}$	Heuristic yaw error quaternion	-
$\boldsymbol{q}_{\mathrm{e,y}}$	Yaw error quaternion	-
$\boldsymbol{q}_{\mathrm{e}}$	Total error quaternion	-
$\boldsymbol{q}_{\mathrm{I}}$	Identity quaternion $(1, 0, 0, 0)$	-
$\boldsymbol{q}_{\mathrm{ref}}$	Full reference attitude	-
$\boldsymbol{r}_{\mathrm{motor}}$	Motor RPM measurement	$^{1}/_{\mathrm{min}}$
$\boldsymbol{v}_{\mathrm{air}}$	Quadrotor air velocity	$^{\mathrm{m}}/_{\mathrm{s}}$
$\boldsymbol{v}_{\mathrm{A},i}$	Air velocity of i-th propeller	$^{\mathrm{m}}/_{\mathrm{s}}$
$\boldsymbol{v}_{\mathrm{A},i}^{\perp}$	Projection of $\boldsymbol{v}_{\mathrm{A},i}$ onto the propeller plane of rotation	$^{\mathrm{m}}/_{\mathrm{s}}$
$\boldsymbol{v}_{\mathrm{ref}}$	Control reference velocity (in the world frame)	$^{\mathrm{m}}/_{\mathrm{s}}$
$\boldsymbol{v}_{\mathrm{wind}}$	Local wind velocity	$^{\mathrm{m}}/_{\mathrm{s}}$
$\boldsymbol{x}_{\mathrm{state,core}}$	SSF state vector core part	-
$\boldsymbol{x}_{\mathrm{state}}$	SSF state vector	-
Δt_{d}	Stochastic disturbance force sampling time	s
$\epsilon_{\mathrm{tracking,thresh}}$	Position tracking error threshold triggering the trajectory tracker to generate a realignment trajectory	m
η_i	Torque of i-th propeller in the simplified FDM	N·m
γ_i	Thrust factor of the i-th propeller	-
$\Gamma_{\mathrm{list,landing}}$	Landing autopilot's trajectory list	-
$\Gamma_{\mathrm{list,mission}}$	Mission autopilot's trajectory list	-
$\Gamma_{\mathrm{list,takeoff}}$	Takeoff autopilot's trajectory list	-
Γ_{list}	Trajectory list (i.e. trajectory *sequence*)	-
$\hat{\boldsymbol{a}}_{\mathrm{acc}}$	Bias-compensated accelerometer measurement	$^{\mathrm{m}}/_{\mathrm{s}^2}$
λ	RLS exponential forgetting factor	-
$\mathcal{A}$	Thrust allocation matrix	-
$\mathcal{B}_{\hat{\psi}}$	Measurement buffer for $\hat{q}_{\mathrm{w,z}}^{\mathrm{l,yaw}}$	-
$\mathcal{B}_{\hat{x}}$	Measurement buffer for $\hat{p}_{\mathrm{w,x}}^{\mathrm{l}}$	-
$\mathcal{B}_{\hat{y}}$	Measurement buffer for $\hat{p}_{\mathrm{w,y}}^{\mathrm{l}}$	-
$\mathcal{B}_{\hat{z}}$	Measurement buffer for $\hat{p}_{\mathrm{w,z}}^{\mathrm{l}}$	-
$\mathcal{B}_{\tilde{v}_{\mathrm{w,z}}^{\mathrm{b}}}$	FIFO buffer of $\tilde{v}_{\mathrm{w,z}}^{\mathrm{b}}$ for the emergency landing controller	-
$\mathcal{D}$	Set of detected tag IDs by the AprilTag algorithm	-
$\mathcal{I}_{\mathrm{pin}}$	Undistorted downfacing camera image	-
$\mathcal{I}_{\mathrm{raw}}$	Raw, distorted image taken by the downfacing camera	-

$\mathcal{L}$	Landing bundle set composed of constituent tag IDs	-
$\mathcal{L}_{\mathcal{D}}$	Set of detected landing bundle tags by the AprilTag algorithm	-
$\mathcal{P}$	A polynomial object	-
$\mathcal{P}_{\mathrm{l}}^{\mathrm{t}_i}$	Set of position measurements of tag i with respect to the landing pad frame.	-
$\mathcal{Q}_{\mathrm{l}}^{\mathrm{t}_i}$	Set of quaternion attitude measurements of tag i with respect to the landing pad frame.	-
$\mathcal{S}$	A segment object	-
$\mathcal{S}_{\mathrm{T}}$	Segment duration	-
$\mathcal{T}_j$	Set of rigid transforms for tags visible in the j-th $\mathcal{I}_{\mathrm{pin}}$ during bundle calibration	-
$\mathcal{T}_{\mathrm{C}}$	Set of $\mathcal{T}_j$ collected during bundle calibration	-
$\mathcal{W}$	Tuple of user-defined waypoints for routing the data acquisition trajectory	-
$\mathcal{W}_{\mathrm{mission}}$	Waypoints list defining the mission trajectory	-
ω_{m}	Generic DC motor rotor angular speed	rad/s
$\omega_{\mathrm{p},i}$	Angular speed of the i-th propeller	rad/s
$\omega_{\mathrm{z,min}}$	Minimum existing body rate for yaw minimum-time tracking heuristic	rad/s
ψ	Quadrotor body frame yaw angle	rad
ψ_{d}	Stochastic disturbance force yaw	N
ψ_{l}	Landing bundle yaw	rad
ψ_{ref}	Control reference yaw	rad
ρ	Air density	kg/m^3
τ_{actuator}	Actuator time constant (attitude loop or motors) used for translation control robustness analysis	s
τ_{att}	Attitude control time constant	s
τ_{l}	Generic DC motor exogenous load torque	N·m
$\tau_{\mathrm{m,dn}}$	Motor spin-down time constant	s
$\tau_{\mathrm{m,up}}$	Motor spin-up time constant	s
τ_{m}	Generic DC motor torque	N·m
θ	Quadrotor body frame pitch angle	rad
$\theta_{\mathrm{feedback}}$	Feedback delay used for translation control robustness analysis	s
φ	Quadrotor body frame roll angle	rad
$\xi_{\mathrm{z,assured}}$	Assured body torque about $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}$ in the prioritized thrust saturation algorithm	N·m

$a_{\text{em,z}}$	Emergency landing controller target open-loop vertical acceleration (expressed > 0)	m/s^2
$A_{\text{p},i}$	Geometric center of i-th propeller	-
a_{takeoff}	Target acceleration for acceleration/deceleration to/from v_{takeoff}	m/s^2
c_{D}	Quadrotor conglomerated drag coefficient	m^2
$c_{\text{H},i}$	Drag force coefficient of i-th propeller	kg
$c_{\text{P},i}$	Pitching torque coefficient of i-th propeller	kg·m
$c_{\text{R},i}$	Rolling torque coefficient of i-th propeller	kg·m
c_{x}	Camera image principal point coordinate along $\boldsymbol{e}_{\text{x}}^{\text{im}}$	px
$c_{\text{Y},i}$	Side force coefficient of i-th propeller	kg
c_{y}	Camera image principal point coordinate along $\boldsymbol{e}_{\text{y}}^{\text{im}}$	px
f	Quadrotor collective thrust	N
f_i	Thrust of i-th propeller in the simplified FDM	N
$f_{\text{d,r}}$	Stochastic disturbance force radial component (i.e. projection onto $(\boldsymbol{e}_{\text{x}}^{\text{w}}, \boldsymbol{e}_{\text{y}}^{\text{w}})$ plane)	N
$f_{\text{d,z}}$	Stochastic disturbance force vertical component	N
$f_{\text{em,buffer}}$	Rate of addition of new measurements into $\mathcal{B}_{\tilde{v}_{\text{w,z}}^{\text{b}}}$	Hz
f_{guidance}	Guidance frequency at which all autonomy engine state machine and trajectory tracker loops are executed	Hz
f_{max}	User-defined maximum collective thrust	N
f_{min}	User-defined minimum collective thrust	N
f_{ref}	Reference collective thrust for the body rate controller	N
f_{sm}	State machine execution frequency	Hz
f_{x}	Camera focal length along $\boldsymbol{e}_{\text{x}}^{\text{c}}$	px
f_{y}	Camera focal length along $\boldsymbol{e}_{\text{y}}^{\text{c}}$	px
$f_{i,\text{ref}}$	Desired thrust of i-th propeller in the simplified FDM	N
H_i	Homography matrix of tag with ID i detected in $\mathcal{I}_{\text{pin}}$	-
$h_{\text{im,ground}}$	Downfacing camera image physical height on the ground	m
h_{rth}	Return to home target height (and therefore the starting height for landing)	m
h_{spiral}	Spiral grid search trajectory height above the ground	m
h_{takeoff}	Target takeoff height above the landing pad	m
$h_{\text{touchdown}}$	Touchdown detection height threshold	m
i_{m}	Generic DC motor current draw	A

J	Quadrotor total inertia tensor	$\mathrm{kg{\cdot}m^2}$
J_{m}	Generic DC motor rotor moment of inertia (scalar) including any attached components	$\mathrm{kg{\cdot}m^2}$
$J_{\mathrm{p},i}$	Inertia tensor of i-th propeller and motor rotor assembly	$\mathrm{kg{\cdot}m^2}$
K	Downfacing camera calibration matrix corresponding to $\mathcal{I}_{\mathrm{pin}}$	-
$k_{\omega,\mathrm{x}}$	Proportional gain for ω_{x} control	-
$k_{\omega,\mathrm{y}}$	Proportional gain for ω_{y} control	-
$k_{\omega,\mathrm{z}}$	Proportional gain for ω_{z} control	-
k_{emf}	Generic DC motor back-EMF (and motor torque) constant	$\mathrm{V{\cdot}s}$
l_{arm}	Propeller geometric centers' radial distance away from the body frame origin, i.e. the arm length	m
L_{m}	Generic DC motor internal inductance	H
m	Quadrotor total mass	kg
M_{buffer}	Measurement history buffer length	-
M_{init}	Number of initial measurements required to initialize the landing bundle pose RLS filter	-
N_τ	RLS time constant expressed as a number of iterations	-
N_{calib}	Total number of images processed during bundle calibration	-
n_{mission}	Number of times to re-fly the mission trajectory	-
$n_{\mathrm{motor,check}}$	Maximum number of motor nominal performance check trials	-
p_{att}	Yaw error fraction used for yaw control	-
p_{search}	Horizontal and vertical advance fraction of the image ground size for the spiral grid search trajectory	-
R	Data acquisition trajectory cornering radius	m
R_{mission}	Cornering radius at pass-through waypoints	m
$r_{\mathrm{motor,check}}$	Motor RPM tolerance for validating motor nominal performance	$1/\mathrm{min}$
R_{m}	Generic DC motor internal resistance	Ω
$s_{\mathrm{tag},i}$	Tag side half-length	m
$t_{0,\mathrm{takeoff}}$	Takeoff start time	s
$t_{\mathrm{em,init}}$	$\mathcal{B}_{\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}}$ buffer duration to use for $v_{\mathrm{em},0,\mathrm{z}}$ computation	s
t_{em}	$\mathcal{B}_{\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}}$ buffer total time duration	s
$t_{\mathrm{motor,check}}$	Motor nominal performance check duration	s
$t_{\mathrm{rls,init,max}}$	Timeout for initializing the AprilTag RLS estimator	s
$t_{\mathrm{sync,max}}$	Timeout for mission data synchronization	s
$t_{\mathrm{takeoff,max}}$	Timeout for climbing to h_{takeoff}	s

t_{w_i}	Time to hover at waypoint i in the data acquisition trajectory	s
$u_{\text{esc,motor,check}}$	Motor ESC command for checking motor nominal performance	-
$u_{\text{esc},i}$	i-th motor ESC command (integer $\in [0, 200]$)	-
v_{align}	Speed of the position alignment trajectory with the charging pad origin	m/s
V_{charged}	Battery charged voltage	V
V_{critical}	Battery critical voltage	V
v_{descend}	Landing final descent speed	m/s
$v_{\text{em,z}}$	Emergency landing controller target final descent velocity	m/s
v_{emf}	Generic DC motor back-EMF	V
v_{mission}	Target speed between mission trajectory waypoints	m/s
v_{m}	Generic DC motor input voltage	V
v_{m}	Mission speed	m/s
V_{rth}	Battery low voltage	V
v_{search}	Speed of the spiral grid search segments	m/s
v_{spiral}	Spiral grid search trajectory speed	m/s
v_{takeoff}	Target takeoff velocity	m/s
$v_{\text{touchdown}}$	Touchdown detection velocity threshold	m/s
v_{transfer}	Realignment trajectory reference speed	m/s
$w_{\text{im,ground}}$	Downfacing camera image physical width on the ground	m
x_{im}	Image x-coordinate	px
x_{tag}	Tag local frame x-coordinate	m
y_{im}	Image y-coordinate	px
y_{tag}	Tag local frame y-coordinate	m

Functions

$\kappa(f_i)$	Map from the i-th propeller thrust to its torque η_i
$\mathcal{D}_{\mathcal{P}}(t, j)$	Polynomial derivative mapping function (evaluates $\frac{\text{d}^j \mathcal{P}(t)}{\text{d}t^j}$)
$\mathcal{M}_{\eta}(u_{\text{esc}})$	Full quadratic propeller torque map, $k_{\eta,2} u_{\text{esc}}^2 + k_{\eta,1} u_{\text{esc}} + k_{\eta,0}$
$\mathcal{M}_{\text{f}}(u_{\text{esc}})$	Full quadratic propeller thrust map, $k_{\text{f},2} u_{\text{esc}}^2 + k_{\text{f},1} u_{\text{esc}} + k_{\text{f},0}$
$\mathcal{S}_{\mathcal{P}}(t)$	Segment value function which evaluates at time t the polynomial $\mathcal{P}$ associated with the segment $\mathcal{S}$

$F_{\dot{\mathrm{p}}}(s)$	Position control position reference's time derivative prefilter
$F_{\mathrm{p}}(s)$	Position control position reference prefilter
$G_{\omega}(s)$	Body rate single-axis dynamics plant
$G_{\mathrm{p}}(s)$	Position dynamics plant
$G_{\mathrm{v}}(s)$	Velocity dynamics plant
$K_{\mathrm{p}}(s)$	Position PID controller
$K_{\mathrm{v}}(s)$	Velocity PI controller
$T_{\mathrm{p,ideal}}(s)$	Ideal position control loop complementary sensitivity (a second-order system)
$T_{\mathrm{v,ideal}}(s)$	Ideal velocity control loop complementary sensitivity (a first-order system)

Acronyms and Abbreviations

ADC	Analogue to Digital Converter
BLDC	Brushless Direct Current
Caltech	California Institute of Technology
CoG	Center of Gravity
CPU	Central Processing Unit
DAC	Digital to Analogue Converter
DGPS	Differential GPS
EMF	Electromotive Force
ESC	Electronic Speed Controller
ETFE	Estimate Empirical Transfer Function
FAA	Federal Aviation Administration
FDM	Flight Dynamics Model
FFRDC	Federally Funded Research and Development Center
FIFO	First-In First-Out
FSM	Finite State Machine
GALCIT	Guggenheim Aeronautical Laboratory
GNC	Guidance, Navigation and Control
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GSD	Ground Sampling Distance

GUI	Graphical User Interface
HIL	Hardware In the Loop
HLP	High Level Processor
I^2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
JPL	Jet Propulsion Laboratory
KF	Kalman Filter
LHP	Left Half Plane
LLP	Low Level Processor
LRF	Laser Range Finder
MAV	Micro Aerial Vehicle
MIMO	Multi-Input Multi-Output
MPC	Model Predictive Control
NASA	National Aeronautics and Space Administration
NRMSE	Normalized RMSE
NTP	Network Time Protocol
OCP	Open-Closed Principle
ODE	Open Dynamics Engine
PC	Personal Computer
PID	Proportional Integral Derivative
PI	Proportional Integral
PnP	Perspective-n-Point
RAM	Random Access Memory
RANSAC	Random Sample Consensus
RC	Radio Control
RHP	Right-Half Plane
RLS	Recursive Least Squares
RMSE	Root Mean Square Error
ROS	Robot Operating System
RPM	Rotations Per Minute
RTK	Real Time Kinematic
SIL	Software In the Loop
SISO	Single-Input Single-Output

SMC	State Machine Compiler
SSF	Single Sensor Fusion
UART	Universal Asynchronous Receiver-Transmitter
UHF	Ultra High Frequency
UML	Unified Modeling Language
UWB	Ultra Wide Band
V&V	Verification & Validation
VTOL	Vertical Takeoff and Landing
ZOH	Zero Order Hold

Coordinate Frames

$\{b\}$	Body (i.e. IMU) frame
$\{b_{des}\}$	Pad-aligned body frame for landing
$\{c\}$	Camera frame
$\{c\}$	Yawed world frame
$\{c_2\}$	Intermediate frame between the $\{c\}$ and $\{b\}$ for Euler angle Z-X-Y convention
$\{c_{des}\}$	Pad-aligned camera frame for landing
$\{im\}$	Camera image (2 dimensional) frame
$\{l\}$	Landing pad frame
$\{m\}$	Calibration master tag frame (just like $\{t_i\}$)
$\{s\}$	Spiral grid search trajectory construction frame
$\{t\}$	Inertia tensor identification test frame
$\{t_i\}$	Tag i local frame
$\{wi\}$	Wind frame
$\{w\}$	World frame

Chapter 1

Introduction

This chapter provides context to the thesis. After an introduction to the Jet Propulsion Laboratory (JPL) in Section 1.1, the project is motivated in Section 1.2. This is done in the form of a target market specification in Section 1.2.1 and a research of existing work in Section 1.2.2. The **thesis goal** is stated in Section 1.3. Finally, this report's conventions and structure are described in Section 1.4.

1.1 The Jet Propulsion Laboratory

JPL is an aerospace robotics and Earth science research and development facility located in Pasadena, California. JPL is managed by the California Institute of Technology (Caltech) and is under contract from the National Aeronautics and Space Administration (NASA) as a Federally Funded Research and Development Center (FFRDC) Jet Propulsion Laboratory [1]. JPL is the primary NASA center for the construction and operation of robotic planetary spacecraft which have visited every planet in our solar system as illustrated in Figure 1.1.
The history of JPL dates back to 1936 when six students from Caltech's Guggenheim Aeronautical Laboratory (GALCIT) carried out a set of rocket engine experiments in the Arroyo Seco, California. Managed by then-director of GALCIT, Theodore von Kármán, and with financial support from the United States Army, the students became part of the GALCIT Rocket Project in 1939. During World War 2, work was done on solid and liquid rockets for assisted takeoff of heavy aircraft. In 1943, the

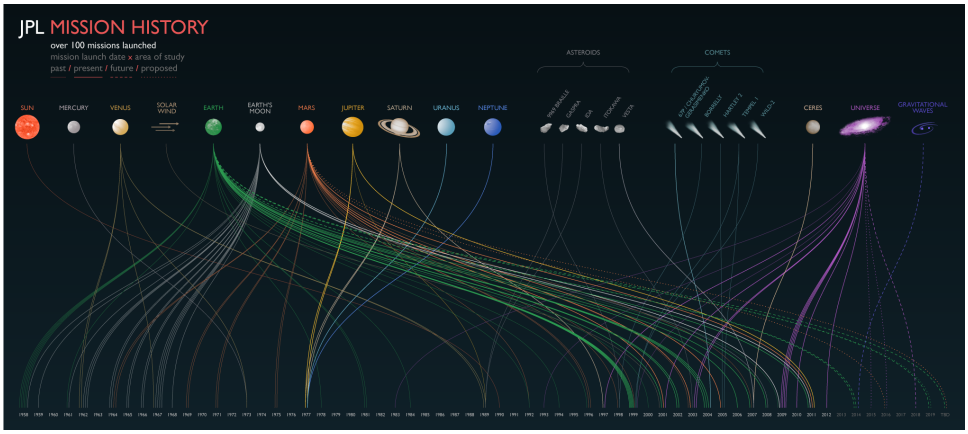


Figure 1.1: JPL mission history. Source: JPL Infographics [2].

project was reorganized under the name *Jet Propulsion Laboratory* and became a formal Army facility managed by Caltech and carrying out research in aerodynamics and missile guidance in support of the United States Army Ordnance.
JPL was transferred to NASA's control in December 1958. In partnership with Wernher von Braun's team at the Army's Redstone Arsenal, JPL launched the United States' first satellite, Explorer 1, on January 31, 1958. During Apollo, JPL operated the Ranger and Surveyor missions that paved the way for future manned lunar landings. JPL then pioneered interplanetary exploration with the Mariner missions to Mercury, Venus and Mars. On September 5, 2017, JPL celebrated the 40 year anniversary of the still-operating interstellar spacecraft Voyager 1.

1.2 Motivation

This thesis took place in the context of an ongoing JPL research and development project on fully autonomous and ultra long term aerial data collection with quadrotors, specifically targeted for agricultural science where such a platform is needed. For the purposes of this report we define a *full-cycle autonomous robot*:

A full-cycle autonomous robot is a machine that is capable of fully autonomous mission execution and autonomous maintenance of its energy supply such that it can continuously operate for indefinite periods of time without human intervention.

1.2.1 Target Market

The robotics market is rapidly expanding with spending on robotics and related services set to reach $135 billion by 2019 and some banks predicting as much as a $100 billion opportunity for drones alone between the years 2016 and 2020 Goldman Sachs [3], The Economist [4], Andrew Meola [5], Jonathan Vanian [6]. A full-cycle autonomous quadrotor platform is foreseen to have three primary use cases.

Use Case 1: Precision Agriculture. The Food and Agriculture Organization of the United Nations predicts that "food production will have to increase by 70 percent" by 2050 in order to feed a projected additional 2.2 billion people [7]. *Precision agriculture*, also known as *smart farming*, is a data-driven approach that helps to address this challenge by using sensing technology to increase farming efficiency. Data is collected about plant and animal health, crop yields, organic matter content, moisture, nitrogen and pH levels, etc.
To complement farming vehicle-mountable sensors, Micro Aerial Vehicle (MAV) multirotor and fixed-wing drones can be equipped with RGB, thermal and hyperspectral cameras and flown over fields in order to create cost-effective and on-demand orthomosaic maps of biophysical parameters like Figure 1.2. The market for such robots is analyzed to be the second-largest in the commercial drone sector, which is also the fastest growing out of the military, business and personal drone market segments Goldman Sachs [3].
Full-cycle long-term autonomy as developed in this project has the particular advantage of capturing time-variant crop properties such as plant water usage over the diurnal cycle. In this case, a single flight during the day would not capture the diurnal water usage pattern while multiple flights requiring human pilot presence would be expensive and impractical for capturing the pattern at a high temporal resolution. This thesis develops a full-cycle autonomous system which offers a cost-effective solution to this challenge.

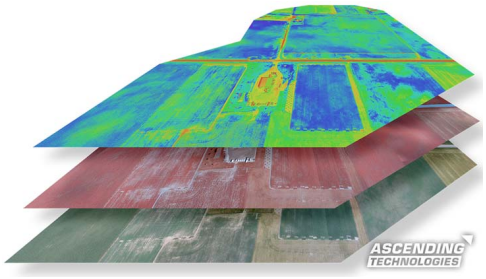


Figure 1.2: A thermal, multispectral and RGB map generated by an AscTec Falcon 8 UAS over two human-piloted flights. Source: Ascending Technologies [8].

Use Case 2: Environmental Surveying. Similar to precision agriculture, environmental surveying involves data collection about certain properties of an area of land. This includes assessments of flood risk, fire risk, methane and other greenhouse gas emissions, contaminated land, etc. While satellites and aircraft have been historically used for this task, autonomous MAVs offer a more flexible, real-time and cost-effective data acquisition platform SenseFly [9]. While fixed-wing MAVs can capture a global map of a large area, multirotor MAVs can supplement the global map with local precision maps due to their hovering capability.

Use Case 3: Surveillance. The natural benefit of full-cycle autonomy is that it may be used to deploy drones equipped with RGB or grayscale cameras for surveillance of an area of land. This is particularly useful for surveillance over water and open, natural landscapes where security cameras are not readily installable due to lack of mounting infrastructure, electricity or sheer operating cost. Surveillance in these cases can be used to e.g. combat illegal hunting and fishing.

1.2.2 Existing Commercial Platforms

Both established and start-up companies propose MAV platforms targeted at the three applications presented in Section 1.2.1. These are illustrated in Figure 1.3 and are discussed and compared below.

Ascending Technologies. The AscTec Falcon 8 octorotor, shown in Figure 1.3a, is advertised as a versatile solution for applications including precision agriculture and land surveying Ascending Technologies [10, 11]. The Falcon 8 can autonomously fly waypoint-based flight routes, offers robustness features such as a pre-takeoff sensor verification and provides tools for working with the collected data. However, full-cycle autonomy features such as automatic data synchronization and automatic recharging are missing.

senseFly. The senseFly eBee fixed-wing MAV, shown in Figure 1.3b, is developed specifically for aerial mapping and is capable of autonomously flying predefined trajectories as well as doing on-board re-planning. Its sensor package is sufficient for creating RGB, thermal and multispectral maps [12] however the sensor choice is limited to the relatively restricted palette from senseFly's online marketplace due to weight and airframe compatibility constraints. Furthermore, the eBee is not capable of autonomous recharging and thus is not full-cycle autonomous.

(a) AscTec Falcon 8 octorotor, capable of autonomous flight of up to 22 minutes and data collection, but not full-cycle autonomy.

(b) senseFly eBee, capable of autonomous flight of up to 50 minutes, but not full-cycle autonomy nor carrying heavy sensors.

(c) The DJI Matrice 100 quadrotor is capable of up to 35 minutes of fully autonomous flight on a single charge. It cannot recharge autonomously.

(d) The fixed-wing WintraOne VTOL MAV is capable of fully autonomous flight and aerial mapping, but has a limited sensor suite and cannot autonomously recharge.

(e) The AeroVinci drone and docking station promise to deliver full-cycle autonomy, but are not yet at the product delivery stage.

(f) Matternet's M2 drone and station are the world's first operational full-cycle autonomous drone system, but target goods delivery.

Figure 1.3: Existing MAV systems targeting the applications in Section 1.2.1.

	Type	Flight time [min]	Area coverage[a]	Sensors[b]	Auto. flight	Auto. charge	Product readiness
Asctec Falcon 8	Octorotor	12–22	N/A	RGB, T, M	Yes	No	Mature
senseFly eBee	Fixed-wing	50	12 km^2 at 974 m altitude	RGB, T, M	Yes	No	Mature
Wingtra	Fixed-wing VTOL	55	1.4 km^2 at 8 cm/px	RGB, M	Yes	No	Early stage, can purchase
DJI and PrecisionHawk	Quadrotor	35	N/A	RGB, M	Yes	No	Mature
AeroVinci	Fixed-wing VTOL	N/A	2.4-4.8 km^2	RGB, M[c]	Yes	Yes	Research and development
Matternet	Quadrotor	20[d]	20 km[e]	RGB[f]	Yes	Yes	Early stage trials in Zürich, Switzerland

[a] Area coverage data is sparse and is not an appropriate comparison metric since Ground Sampling Distance (GSD) is not standardized across different MAV platform advertisements.
[b] RGB: visible spectrum camera; T: thermal camera; M: multispectral camera.
[c] A modular payload interface is advertised, so more sensors are likely possible.
[d] Estimated from published range and cruising speed Thuy Ong [17].
[e] Matternet drones are designed for goods delivery, so *distance* coverage is more important than *area* coverage.
[f] Speculation based purely on existing M2 drone images showing what appears to be a downfacing camera lens on the drone underside.

Table 1.1: Comparison of several data acquisition UAS platforms.

DJI and PrecisionHawk. The Smarter Farming Package [13] is a joint effort by the DJI and PrecisionHawk companies to provide an all-in-one platform for precision agriculture. It combines a DJI Matrice 100 quadrotor (Figure 1.3c) equipped with RGB and multispectral cameras and a comprehensive data processing software suite (PrecisionHawk [14, 15, 16]) to deliver a very capable platform for precision agriculture. It is possible to create orthomosaic maps for crop counting, biomass measurements, plant health monitoring, etc. Nevertheless, while the Matrice 100 is fully autonomous for a single flight, it is not able to recharge autonomously and therefore it is not full-cycle autonomous.

Wingtra. The WingtraOne Vertical Takeoff and Landing (VTOL) fixed-wing MAV, shown in Figure 1.3d, is developed specifically for mapping and surveying. Its advertised sensor suite is capable of creating RGB and multispectral orthomosaic maps, however sensor choice appears to be limited by the airframe form factor like for the eBee. It is also not full-cycle autonomous due to lack of autonomous recharging.

AeroVinci. This start-up company is developing a VTOL fixed-wing MAV and a drone docking station, DroneDock (Figure 1.3e), which will be capable of full-cycle autonomous operation AeroVinci [18]. Targeted applications are precision agriculture and infrastructure inspection, however the product appears to be in the early research and development stages with no commercially deployed product as of yet.

Matternet. This start-up company develops a delivery drone network with the goal to "make access to goods as frictionless and universal as access to information" Matternet [19]. Their proprietary M2 drone and a docking station for payload and battery exchange are claimed to form the world's first operational autonomous drone network deployed over Zürich, Switzerland, in the form of 2 drones and 4 routes. Importantly, the drones are full-cycle autonomous. However, their target market for goods delivery makes the technology neither sold nor likely to be directly suitable without modifications for the applications in Section 1.2.1.
Table 1.1 summarizes how these platforms compare, from which one can see that:

- Multirotor MAVs such as the DJI Matrice 100 show that comparable flight times to fixed-wing MAVs can be achieved. Rotorcraft offer the additional advantage of precision sampling (lower GSD) thanks to their slow flight/hovering capability;

- While all commercial platforms offer autonomous flight, no drone available for the applications described in Section 1.2.1 can autonomously recharge.

1.3 Thesis Goal

It is concluded from Section 1.2.2 that **no currently commercially available drone for precision agriculture, environmental surveying or surveillance is full-cycle autonomous**. Yet these applications, by their long-term data collection nature, require full-cycle autonomous drones. There is a clear market opportunity to be addressed with such a system, the development of which is the subject herein.

Thesis Goal. To develop the control, guidance, precision landing navigation and the autonomy engine for a full-cycle autonomous quadrotor MAV. The prototypical mission scenario is to execute a pre-defined data collection mission over an open area (e.g. a field) for an indefinite period of time.

The value of this thesis is in an **end-to-end system development and implementation**. In addition, the algorithms presented herein are deployable on virtually any multirotor MAV and the core ideas are applicable even to fixed-wing VTOL MAVs. This is the fortunate result of using a recharging rather than a battery swapping approach. This enables the system to transform any drone into a full-cycle autonomous machine, provided an available charging pad. Battery swapping systems, meanwhile, require compatible airframe/docking station combinations.

1.4 Report Organization

This report aims to be an all-encompassing document for system design and deployment. As a research document, it formalizes every system aspect in a rigorous and theoretical framework. As an engineering document, the theory is linked to implementation via source code references throughout the text. As a practitioner document, system operation tutorials are provided where necessary.

1.4.1 Writing Conventions

Navigational Aids

In order to facilitate navigation between this report and the software implementation, the following structural aids are used[1]:

- Icon  represents the flight software directory, `~/alure_flight_software/src`;
- Icon  represents the testing software directory, `~/alure_testing_software/src`;
- Icon  represents the analysis scripts directory, `~/scripts`.

These icons shall be encountered throughout the text to reference implementation locations. See Appendix H for further details.

Pseudocode Conventions

Although the implementation is written in the C, C++, Python and MATLAB languages, the body of this report (i.e. what you are reading now) is made as programming language agnostic as possible such that the reader can focus on the ideas. The following pseudocode conventions shall be adopted:

- Self-written methods are written in small-caps and end with parentheses, e.g. myMethod();
- Appendix A provides several generic methods that are used throughout the report;
- Variables that are not mathematically typeset use either regular or small-caps font (the lack of parentheses differentiating them from methods), e.g. foo or BAR;

[1] Let `~` represent the source code base directory, `<path_to_thesis_folder>/material`.

- Programming language native methods use verbatim font, e.g. `push_back` (C++) and `diag` (MATLAB). The reader is assumed to be familiar with these methods and their usage shall be clear in context;
- Unless stated otherwise, indexing is one-based for mathematical convenience;
- Algorithm captions shall reference the corresponding source code implementation method via `<directory_path>:<method_name>`.

Most methods in the implementation are written in classes, meaning that multiple methods have access to the same set of variables without needing to return them as outputs nor pass between each other some kind of data object. Pseudocode in this report benefits from this to simplify code and make it look more like the implementation. In particular, it is assumed that pseudocode algorithms in this report have shared access to the same set of variables (except internal variables to the individual algorithms).

Mathematical Notation

Mathematical notation is mostly standard. The nomenclature provides a central repository defining all non-trivial variables except for temporary/placeholder variables and those from standard notation (e.g. rotation matrices). Where nomenclature symbols are duplicated with different definitions, their usage shall be clear from context. The following precisions are made:

- Only variables are italicized. A variable is anything which may be substituted with a numerical value. Consider t_f vs. x_k: "f" indicates the final time while "k" is a variable;
  - Under this rule, axes x, y and z are not italicized since they are not variables.
- All vectors are boldface and all matrices are upper-case. Scalars are typically lower-case, but when they are upper case it is clear from context that they are not matrices;
- Notation (t) denotes time-dependency, $(\cdot)$ denotes "function of $\cdot$", $[k]$ denotes discrete-time index dependency and $[\cdot]$ denotes element $\cdot$ selection operation ($[\cdot,\cdot]$ for matrices). Element selection uses the standard MATLAB notation (e.g. $M[:,2]$ selects the second column of M). Element selection removes the boldface from a vector, since it is now a scalar. t is reserved for the current time while k is reserved for the current iteration index;
- The time derivative of x is denoted $\frac{\mathrm{d}x}{\mathrm{d}t} \equiv \dot{x}$;
- Subscripts x, y and z are used to select the first, second and third component of $\boldsymbol{x} \in \mathbb{R}^3$ e.g. $x[3] \equiv x_\mathrm{z}$. Subscripts w, x, y and z are used to select the first, second, third and fourth elements of a quaternion. Similar notation (but in 2 dimensions) applies for a matrix M, e.g. $M_\mathrm{xy} = M[1,2]$;
- Column vectors $\begin{bmatrix} a & b & c & \cdots \end{bmatrix}^\mathsf{T}$ can be written inline as $(a, b, c, \ldots)$;
- {a} denotes "(coordinate) frame a". Its standard basis is denoted as $\{\boldsymbol{e}_\mathrm{x}^\mathrm{a}, \boldsymbol{e}_\mathrm{y}^\mathrm{a}, \boldsymbol{e}_\mathrm{z}^\mathrm{a}\}$. Figures in this report use the typical red/green/blue color code for the x/y/z axes respectively;
- $\boldsymbol{p}_\mathrm{a}^\mathrm{b}$ denotes the vector from frame {a} origin to frame {b} origin, expressed in the frame {a}. The *velocity* $\boldsymbol{v}_\mathrm{a}^\mathrm{b} := \dot{\boldsymbol{p}}_\mathrm{a}^\mathrm{b}$ and the *acceleration* $\boldsymbol{a}_\mathrm{a}^\mathrm{b} := \ddot{\boldsymbol{p}}_\mathrm{a}^\mathrm{b}$. Instances of this standard notation are not listed in the nomenclature;

- $\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}}$ denotes the quaternion representation of a *passive* rotation (i.e. change of reference) from frame {b} to frame {a}. This is, equivalently, the *active* rotation of frame {a} into frame {b}. Pure yaw rotations are denoted $\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b,yaw}}$. Instances of this standard notation are not listed in the nomenclature;

- $C_{(\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}})}$ denotes the rotation matrix version of $\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}}$. Instances of this standard notation are not listed in the nomenclature;

- The body angular velocity $\boldsymbol{\omega}$ is always expressed in the body-fixed frame, therefore does not specify the frame in subscript/superscript;

- A "pose" of a frame refers to the values of $\boldsymbol{p}_{\mathrm{a}}^{\mathrm{b}}$ and $\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}}$ which spatially fully define the frame;

- The rigid transformation matrix T_{ab} transforms homogeneous vectors in frame {b} into homogeneous vectors in frame {a}. Because $T_{\mathrm{ab}} = [C_{(\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}})}, \boldsymbol{p}_{\mathrm{a}}^{\mathrm{b}}; 0, 0, 0, 1]$, the conversion between the T_{ab} and $\boldsymbol{p}_{\mathrm{a}}^{\mathrm{b}}$, $\boldsymbol{q}_{\mathrm{a}}^{\mathrm{b}}$ representations of pose is assumed to be available. Instances of this standard notation are not listed in the nomenclature;

- A unit vector carries a check, e.g. $\check{\boldsymbol{x}}$, when it is deemed necessary to explicitly specify this. The {a} standard basis $\{\boldsymbol{e}_{\mathrm{x}}^{\mathrm{a}}, \boldsymbol{e}_{\mathrm{y}}^{\mathrm{a}}, \boldsymbol{e}_{\mathrm{z}}^{\mathrm{a}}\}$ is always composed of three unit vectors, so we do not use the check notation for them;

- Consider a variable x (or $\boldsymbol{x}$ or X). x denotes the actual value, $\hat{x}$ denotes the measured value and $\tilde{x}$ denotes the filtered value (e.g. by a Kalman Filter or from a calibration procedure). Note that throughout this report x, $\hat{x}$ and $\tilde{x}$ shall invariably refer to the same quantity, but simply carry slightly different ideal/measured/estimated values;

- $\mathbb{R}_{+} = \{x \in \mathbb{R} \mid x \geq 0\}$ and $\mathbb{R}_{++} = \{x \in \mathbb{R} \mid x > 0\}$. Similarly for $\mathbb{Z}$;

- Elements of a set are enclosed in braces, e.g. $\{a, b, c, \ldots\}$. Elements of a tuple, which is a finite ordered list, are enclosed in angle brackets, e.g. $\langle a, b, c, \ldots \rangle$ Kane [20]. A new element a shall be added to some tuple $\mathcal{T}$ via $\mathcal{T} = \langle T, a \rangle$, forming a new tuple with all the previous elements and the new element at the end;

- $I_n \in \mathbb{R}^{n \times n}$ represents the identity matrix;

- $1_{n \times m} \in \mathbb{R}^{n \times m}$ represents a matrix of ones. 1_n is used when $n = m$ or the subscript is dropped entirely when it is obvious from context (e.g. it is the scalar one). The same notation is used for 0 (usually a scalar, generally a matrix of zeros). Boldface is not used for the zero and one vectors, $0_{n \times 1}$ and $1_{n \times 1}$.

1.4.2 Report Structure

The following report is structured as follows. Chapter 2 provides a hardware and software systems-level description in order to give context to the individual subsystems. The Guidance, Navigation and Control (GNC) subsystems are then tackled. Navigation is described first in Chapter 3, followed by guidance in Chapter 4 and control in Chapter 5. The order follows from the fact that both control and guidance require navigation while the input to control is the output of guidance. Chapter 6 then describes the autonomy engine, which deploys the GNC system for the special case of a full-cycle autonomous data collection mission as required by Section 1.3.

Results are presented in Chapter 7 and demonstrate successful full-system operation. Finally, Chapter 8 concludes with a future outlook and a prioritized set of future improvements. The appendices support and expand on concepts introduced in these chapters.

Chapter 2

System Description

This chapter gives an overview of the hardware and software architecture of the full-cycle autonomous quadrotor developed in this thesis. This makes the chapter an important entry point for understanding how the individual subsystems explained in the rest of the report fit together.
System schematics in this chapter (e.g. Figure 2.3) use the following convention:

- Solid lines (———) represent information flow and are labeled whenever the communication protocol is known;
- Dashed lines (– – – –) represent energy flow;
- Dash-dotted lines (-·-·-·-) represent wireless information flow.

Arrows represent flow direction. No arrows represents bidirectional flow.

2.1 Hardware Architecture

Figures 2.1 and 2.2 label the following quadrotor main hardware elements:

Q1. AscTec Pelican quadrotor airframe (carbon fiber balsa sandwich arms, pure carbon fiber cross-brace plates in the midsection and custom 3D printed, carbon infused, legs to accommodate Q14);

Q2. APC propeller with a 10 in diameter and a 4.7 in pitch (2×left- and 2×right-turning);

Q3. 4× standard AscTec Pelican Hacker Brushless Direct Current (BLDC) motors;

Q4. 4× standard AscTec Pelican Electronic Speed Controller (ESC) units;

Q5. Downfacing camera, the USB 2.0 board-level mvBlueFOX–MLC200wG camera [21];

Q6. FLIR Ax5-series thermal camera [22];

Q7. 5 GHz WiFi module [23];

Q8. Trimble BD930-UHF Global Navigation Satellite System (GNSS) receiver [24];

Q9. 3-axis magnetometer Honeywell [25];

Q10. Trimble UHF antenna for Q8;

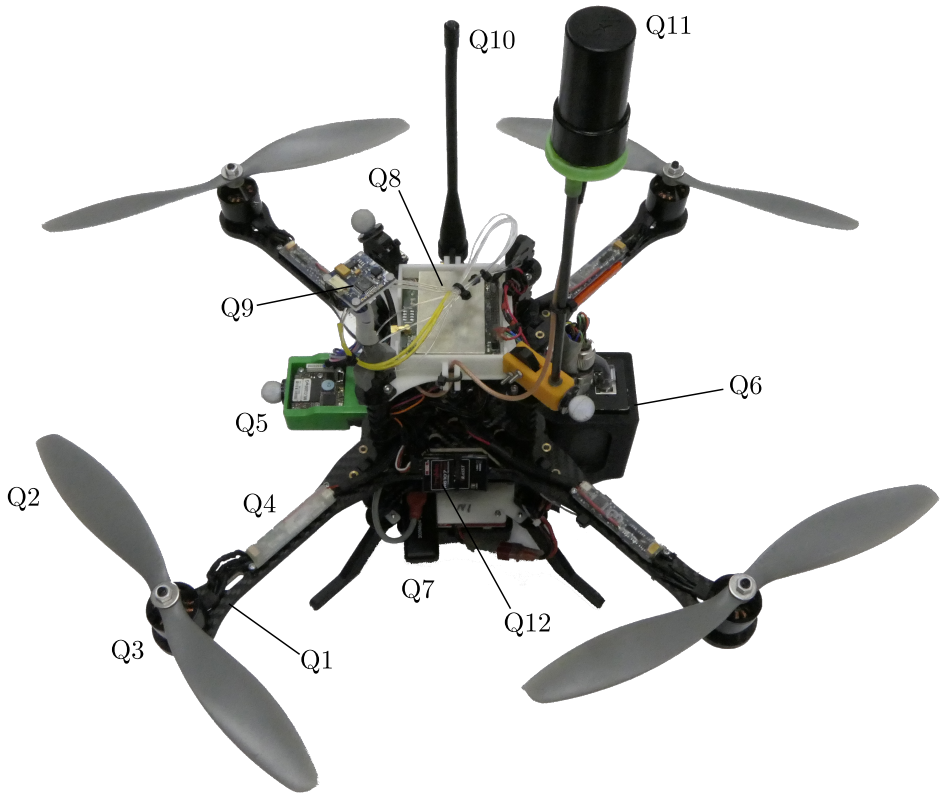


Figure 2.1: Top view of the AscTec Pelican quadrotor with labeled hardware.

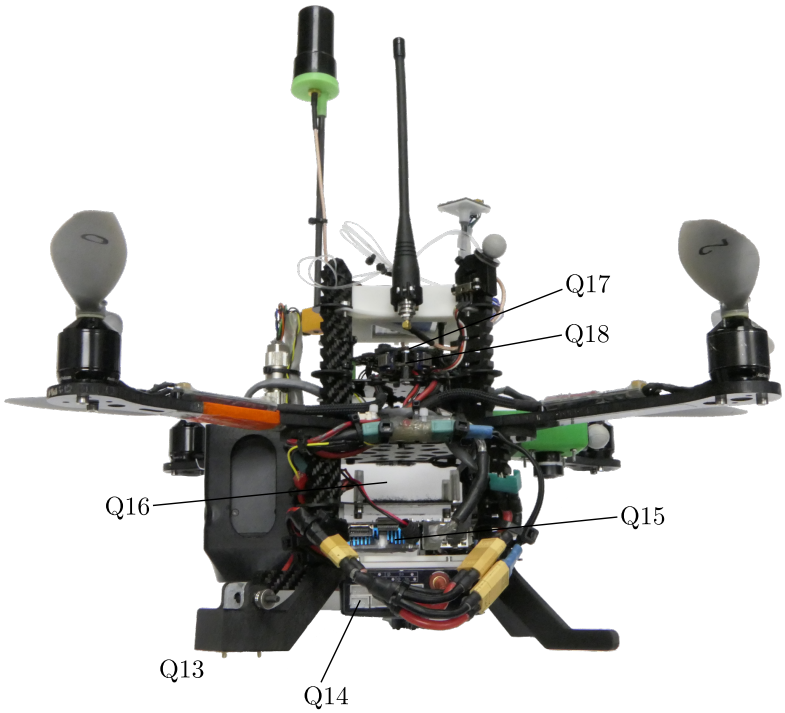


Figure 2.2: Side view of the AscTec Pelican quadrotor with labeled hardware.

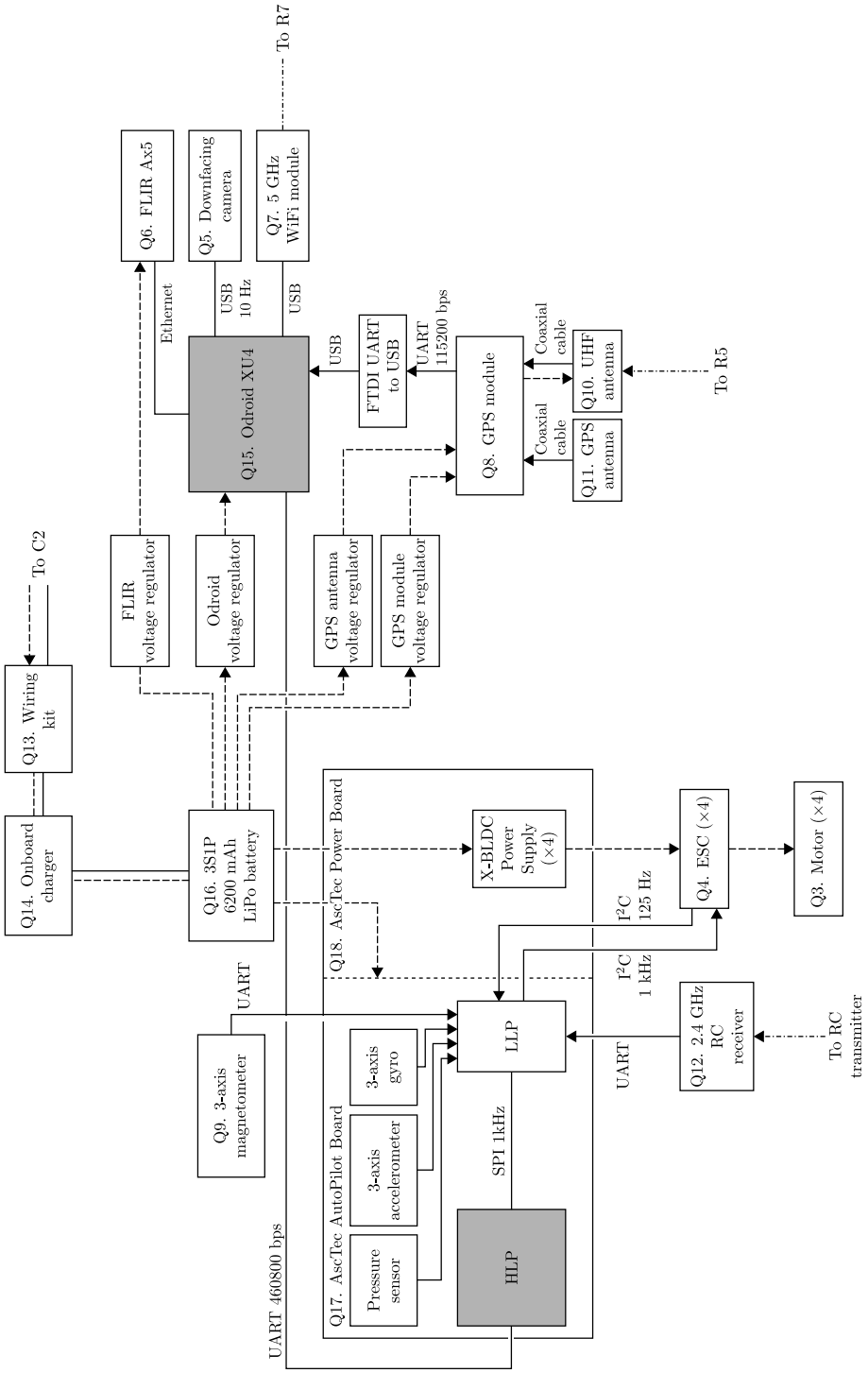


Figure 2.3: On-board quadrotor hardware architecture corresponding to Figures 2.1 and 2.2.

Q11. Global Positioning System (GPS) antenna for Q8;

Q12. Robbe R6107SP 2.4 GHz 7 channel RC receiver;

Q13. Skysense wiring kit [26];

Q14. CRC C6S LiPo balance charger [27];

Q15. Odroid XU4 [28];

Q16. 3S1P 6200 mAh LiPo battery holder (used: ZIPPY Compact battery [29]);

Q17. AscTec AutoPilot Board [30]. Computations are split between two ARM7-type micro-controllers NXP [31] known as the Low Level Processor (LLP) (closed-source proprietary) and the High Level Processor (HLP) (open for custom code). The board also carries a 3-axis accelerometer Memsic [32], a 3-axis gyroscope Analog Devices [33] and a pressure sensor NXP [34];

Q18. AscTec Power Board [30].

Composed of these elements, Figure 2.3 displays the quadrotor on-board hardware architecture. The gray blocks (the Odroid XU4 and the HLP) are highlighted to emphasize that they host all the flight software developed in this thesis. This is discussed further in Section 2.2.
Figure 2.3 shows that the quadrotor has interfaces to a base station PC, to a charging pad and (optionally) to a Real Time Kinematic (RTK) GPS ground station. RTK GPS allows outdoor flight testing with a more precise GPS signal. When RTK GPS is used, an information flow exists over a wireless Ultra High Frequency (UHF) band to an RTK ground station which provides corrections for centimeter-level quadrotor position measurement accuracy. Figure 2.4 labels the RTK GPS ground station setup and Figure 2.5 draws a corresponding hardware schematic.
The contact-based link to the Skysense charging pad is established automatically once the quadrotor is on the charging pad. Figure 2.6 labels the charging pad setup and Figure 2.7 draws a corresponding hardware schematic. Finally, the wireless link to a ground station Personal Computer (PC) happens through the WiFi router in Figure 2.5[1].

2.2 Software Architecture

Figure 2.8 illustrates the full system as an information flow diagram. Sensors feed information to the Single Sensor Fusion (SSF) state estimator and visual landing navigation subsystems. These supply the guidance, operated by the autonomy engine, and the control with navigational information necessary for stabilization, trajectory tracking, trajectory generation and general decision making. The control subsystem is then responsible for actual motor speed computation which gets applied on the quadrotor. What happens to the quadrotor as a function of control's actions gets picked up by the sensor array and is fed back into the system. The LLP and AscTec Power Board are in this case part of the "Quadrotor" block as they are the ones responsible for motor actuation as a function of the control block's output. Going one level of detail further, Figure 2.9 shows the spatial layout of the software components with respect to the hardware and provides a more detailed view of the subsystem interconnections. On the Odroid XU4 side, algorithms are programmed in C++ and are interconnected using the Robot Operating System (ROS) middleware Quigley et al. [35]. On the AscTec AutoPilot Board side, custom control

[1]The Network Time Protocol (NTP) is used via `chrony` in order to synchronize the clocks of Q15 and R1.

Figure 2.4: RTK GPS ground station with labeled hardware.

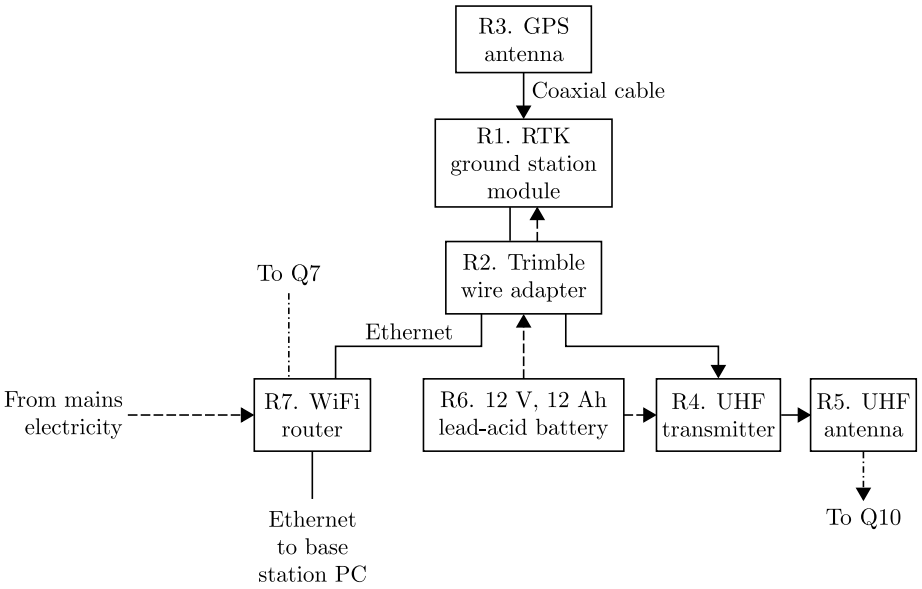


Figure 2.5: RTK GPS ground station hardware architecture corresponding to Figure 2.4.

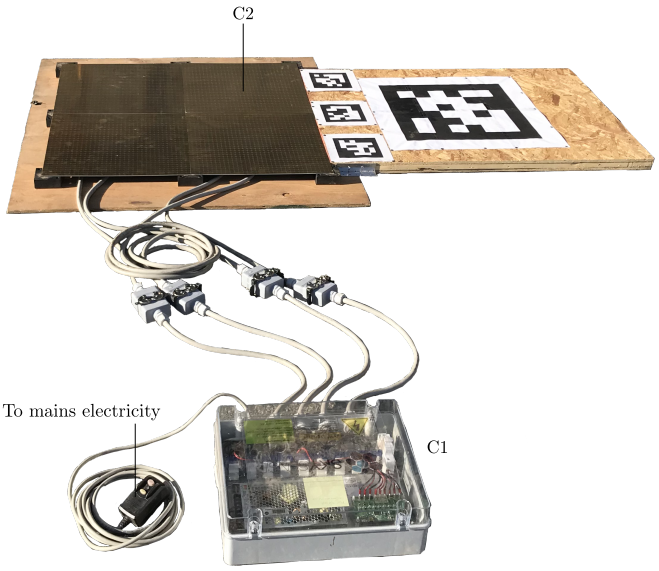


Figure 2.6: Skysense charging pad with labeled hardware.

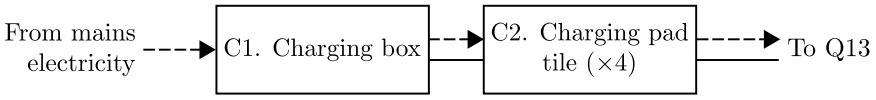


Figure 2.7: Charging pad hardware architecture corresponding to Figure 2.6.

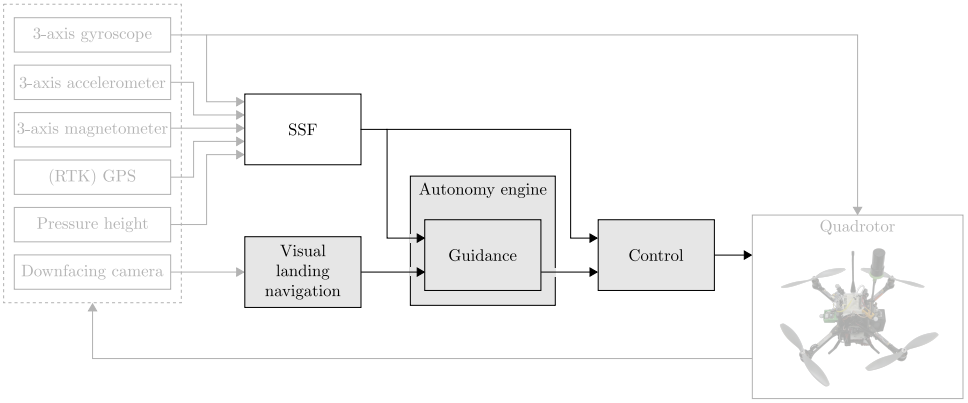


Figure 2.8: GNC and autonomy system information flow diagram. Blocks and lines with black stroke form the software subsystem while blocks with light gray stroke are part of the hardware subsystem described in Section 2.1. Blocks with gray background were developed in this thesis and are described in this report.

algorithms in bare-metal C are flashed onto the HLP by modifying the original `asctec_hl_firmware` Achtelik et al. [36]. The LLP runs closed-source firmware that reads sensors and sends motor commands to the ESC units. The serial communication between the AutoPilot and the Odroid is abstracted away by the AscTec AutoPilot interface which is a modified version of the `asctec_hl_interface` Achtelik et al. [36]. Sensor data, control commands, etc. are fed into/out of the ROS network from/to the AutoPilot via this interface. Note that the wireless link to the base station PC is purely for mission data synchronization and interaction with the system (e.g. starting and stopping the autonomy engine). **The quadrotor is fully self-contained such that full-cycle autonomy is not contingent on any external elements other than the charging pad**. Appendix H gives an overview of the actual source code directory structure.

Figure 2.9: Flight software architecture, showing subsystem interconnections and what hardware the subsystems run on. Blocks with gray background were developed in this thesis and are described in this report.

Chapter 3

Navigation

The navigation subsystem is responsible for estimating the quadrotor state and the landing pad pose in the world frame[1]. Its design specifications are given in Specification List 1. The remainder of this chapter explains the two navigation subsystems – SSF and visual landing navigation.

S1	Provide an accurate enough landing pad pose estimate to be able to land on a 90×90 cm charging pad in visible conditions (no rain/snow/fog/nighttime, potentially cloudy).
S2	Provide a robust and reliable state estimate in an open-sky outdoor environment (e.g. a crop field).

Specification List 1: Navigation subsystem design specifications.

3.1 Visual Landing Navigation

The *landing navigation system* concerns itself with landing pad pose estimation. The visual landing navigational subsystem developed herein is responsible for satisfying S1 in Specification List 1. Following a literature review of possible methods, an approach is selected, motivated and its implementation is detailed.

3.1.1 Literature Review

This section reviews research on *landing in a structured environment*, i.e. an environment which the operator can specially design for the task (unlike landing in unknown, unstructured environments). MAV structured landing research dates back to the very early 2000s Sharp et al. [37], Saripalli et al. [38]. To date, the dominating majority of this research has used a camera setup for core navigation which is sometimes supplemented by other sensors such as Ultra Wide Band (UWB) radios[2]. Shaogang et al. [39] explain that Differential GPS (DGPS) and RTK GPS systems suffer from precision degradation and signal loss in occluded urban or canyon environments while Laser Range Finder (LRF) systems are heavy and power hungry. Stereo camera setups have limited range as a function of their baseline versus quadrotor-to-pad distance. Monocular cameras, meanwhile, are lightweight, cheap

[1] Throughout this report, the terms *landing pad* and *charging pad* shall be used interchangeably. They are the same thing.

[2] Note that we speak strictly about the landing navigation sensors. GPS may well be used for state estimation – but it is not a suitable sensor for landing pad localization.

and power efficient sensors that have attracted much of research on structured landing (i.e. on a landing pad).

Yang et al. [40] present a monocular visual landing method based on estimating the 6 DOF pose of an (H) marker. The same authors extend this work to enable autonomous landing site *search* by using a scale-corrected PTAM algorithm Klein and Murray [41] and relax the landing site structure to an arbitrary but feature-rich image that is matched using the ORB algorithm Rublee et al. [42]. Forster et al. [43] use SVO Forster et al. [44] and a downfacing monocular camera's motion to build a probabilistic 2 dimensional elevation map of unstructured terrain and to detect safe landing spots based on a score function. Huber [45] extended the landing spot detection work by using a stereo camera (Intel RealSense R200) to create a 2.5 dimensional elevation map, fitting planes to it via the Random Sample Consensus (RANSAC) algorithm and using a legged robotics-inspired safe spot detection scoring function. Brockers et al. [46, 47] develop a fully self-contained visual landing navigation pipeline using a single camera. With application for rooftop landing, the planar character of a rooftop is leveraged to do landing site detection via a planar homography decomposition using RANSAC to distinguish the ground and elevated landing surface planes. Desaraju et al. [48] employ a similar rooftop landing site detection approach. The 3 dimensional world below the MAV is reconstructed via dense motion stereo and, from this, a binary landing map is created. In the world frame, a confidence measure of landing site quality is provided.

A dedicated landing pad, however, is a structured environment over which the designer has a high degree of control. Landing algorithms which build elevation maps and perform generic landing spot detection trade their robustness for the ability to land in *un*structured environments. However, placing visual fiducial markers at or near the landing zone allows one to use visual fiducial detector algorithms to much more robustly detect the landing pad pose. Furthermore, landing pad detection using visual fiducials is limited virtually only by the camera resolution Fnoop and Hadaway [49]. Even with small resolution cameras and a wide angle lens, it is fine for the maximum detection height to be under 10 m as GPS is accurate enough for the landing approach prior to landing pad visual detection.

Dating from the early 2000s, Sharp et al. [37] demonstrated helicopter landing on a visual fiducial composed of black and white squares of various sizes while Saripalli et al. [38] used a large white H on a black background for the visual fiducial. Following these early experiments, an explosion of visual fiducial patterns has occured and is covered by survey papers like Shaogang et al. [39] and the literature reviews of Yang et al. [40], Fiala [50], Olson [51]. Patterns include letters, circles, concentric circles and/or polygons, letters, 2 dimensional barcodes and special patterns based on topological Bencina et al. [52], detection range maximizing Xu and Dudek [53] and blurring/occlusion robustness considerations Bergamasco et al. [54].

Currently one of the most popular visual fiducial detectors and patterns is the AprilTag algorithm Olson [51] which is renowned for its speed, robustness and extremely low false positive detection rates thanks to a Hamming distance guarantee for tag families. The algorithm was updated by Wang and Olson [55] to further improve computational efficiency, enable detection of even smaller tags (down to the single-pixel level) but remove robustness features like the detection of partially occluded tags. AprilTag has been applied multiple times for MAV landing Ling, Kevin [56], Kyristsis et al. [57], Borowczyk et al. [58], Chaves et al. [59]. Similar visual fiducials are seen around landing zones in the (scarce) video content of both Amazon and Google delivery drones Amazon [60], Vincent [61] as well as surveying drones of some start-up companies Wingtra [62].

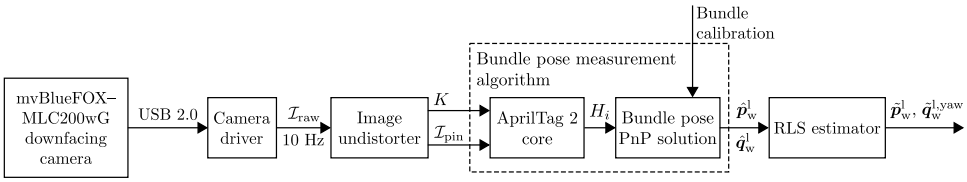


Figure 3.1: Visual landing navigation block diagram.

3.1.2 System Overview

In light of the literature review, we chose a landing navigation approach based on the AprilTag 2 algorithm[3] Wang and Olson [55]. In particular, landing navigation is done purely based on AprilTag detections and is entirely decoupled from state estimation. This makes for a simple, robust and cost-effective system which in practice works in even the most dire conditions like deep night (i.e. very low visibility).
Figure 3.1 shows the landing navigation pipeline. The downfacing camera feeds a distorted 752×480 px grayscale image, $\mathcal{I}_{\text{raw}}$, into the image undistorter. The latter removes radial distortion in the image via the ideal fish-eye lens model Devernay and Faugeras [63], producing the un-distorted image $\mathcal{I}_{\text{pin}}$ assumed to be produced by a pinhole camera with calibration matrix K. The unmodified AprilTag 2 implementation APRIL Laboratory [64] identifies AprilTag markers[4] in $\mathcal{I}_{\text{pin}}$ and for each tag with ID i produces a homography matrix H_i (see below). Given that at least one tag is detected, the Perspective-n-Point (PnP) solver produces a bundle pose measurement $\hat{\boldsymbol{p}}^{\text{l}}_{\text{w}}$, $\hat{\boldsymbol{q}}^{\text{l}}_{\text{w}}$ which gets filtered by the RLS estimator to output $\tilde{\boldsymbol{p}}^{\text{l}}_{\text{w}}$ and $\tilde{\boldsymbol{q}}^{\text{l,yaw}}_{\text{w}}$. The latter variables are then used for landing navigation. This pipeline is explained in the following sections as follows:

- Section 3.1.3 appends a PnP solver after the AprilTag algorithm in order to handle pose measurement of tag bundles. The resulting measurement noise is analyzed in Appendix C;
- Section 3.1.4 describes the tag bundle calibration process;
- Section 3.1.5 proposes an RLS estimator for obtaining a smoother bundle pose signal useful for precision landing.

3.1.3 AprilTag Bundle Pose Measurement

Before estimating the landing pad pose, one must measure it. Let {l} represent the landing pad frame whose pose is given by $\boldsymbol{p}^{\text{l}}_{\text{w}}$ and $\boldsymbol{q}^{\text{l}}_{\text{w}}$. As shown in Figure 3.1, the *bundle pose measurement algorithm*[5] outputs $\hat{\boldsymbol{p}}^{\text{l}}_{\text{w}}$ and $\hat{\boldsymbol{q}}^{\text{l}}_{\text{w}}$ based on the input $\mathcal{I}_{\text{pin}}$. Assume that $\mathcal{I}_{\text{pin}}$ corresponds to the image of the world that would be generated by an ideal pinhole camera with a calibration matrix K Hartley and Zisserman [65]:

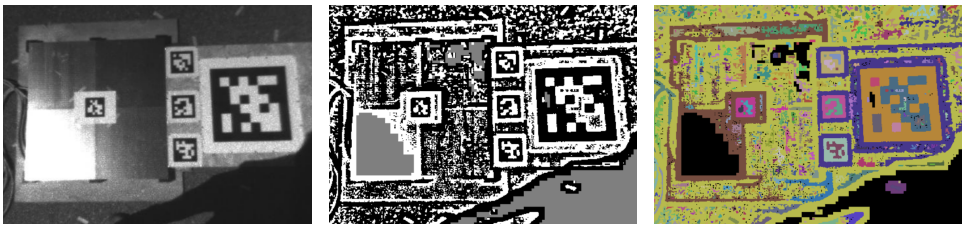
$$K = \begin{bmatrix} f_{\text{x}} & 0 & c_{\text{x}} \\ 0 & f_{\text{y}} & c_{\text{y}} \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.1}$$

where f_{x} and f_{y} are respectively the focal lengths along $\boldsymbol{e}^{\text{c}}_{\text{x}}$ and $\boldsymbol{e}^{\text{c}}_{\text{y}}$, where {c} is the camera frame (see Figure 3.6). c_{x} and c_{y} are respectively the camera principal point coordinates along $\boldsymbol{e}^{\text{im}}_{\text{x}}$ and $\boldsymbol{e}^{\text{im}}_{\text{y}}$, where {im} is the image frame (see Figure 3.3b).

[3] In the remainder of this report, the AprilTag 2 algorithm shall be referred to simply as the *AprilTag algorithm*.

[4] From here on, AprilTag markers shall be referred to as *tags*.

[5] `/sensing/apriltags2_ros/apriltags2_ros/include/apriltags2_ros/continuous_detector.h`



(a) Original image. (b) Adaptive thresholding. (c) Segmentation.

(d) Clustered pixels. (e) Detected quads. (f) Tag detections.

Figure 3.2: Intermediate steps of the AprilTag 2 detection algorithm (on a sample image from a landing bundle calibration). High contrast parts of the original image (a) are binarized in (b) into black and white regions while the rest of the image is discarded (marked gray). Black and white regions are segmented using the union-find algorithm in (c). For every pair of adjacent black and white segments, in (d) the pixels on their boundary are lumped into clusters using a hash table. Quads are fitted to sufficiently quad-like clusters in (e). Each quad is sampled for its payload and those matching a known tag are returned in (f). Note that the false quad inside the large tag in (f) is eliminated by this last step, which shows the importance of a good payload design that can eliminate quads that are not tags.

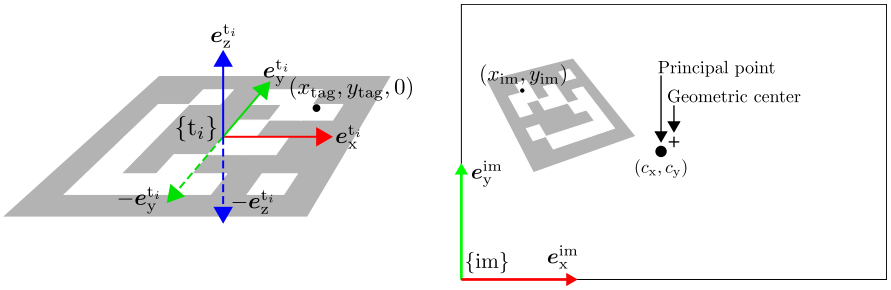
The landing pad is surrounded by a *landing bundle* $\mathcal{L}$ composed of one or more unique tags of the 36h11 family[6]. Each tag in the family has a unique *ID* (ranging from 0 to 586 for the 36h11 family). Landing bundle $\mathcal{L}$ is a set composed of the tag IDs which compose it.

Given $\mathcal{I}_{\text{pin}}$ as input, the core AprilTag algorithm[7] will produce a set of detected tags $\mathcal{D}$. The internal workings of this are summarized in Figure 3.2 (similar to Figure 3 of Wang and Olson [55]). Assume from here on that $\mathcal{L}_{\mathcal{D}} := \mathcal{D} \cap \mathcal{L} \neq \varnothing$, i.e. a non-empty subset of the landing bundle is detected. When $\mathcal{L}_{\mathcal{D}} = \varnothing$, the landing pad pose obviously cannot be measured.

Let $H_i \in \mathbb{R}^{3\times3}$ be the homography matrix of tag $i \in \mathcal{D}$ where i is the tag ID. As shown in Figure 3.3, H_i transforms a point $(x_{\text{tag}}, y_{\text{tag}}, 0)$ in the tag i local frame, $\{\text{t}_i\}$, to its projection's pixel coordinates $(x_{\text{im}}, y_{\text{im}})$ in the $\{\text{im}\}$ frame. Let $s_{\text{tag},i}$ be the tag side half-length such that $x_{\text{tag}}, y_{\text{tag}} \in [-s_{\text{tag},i}, s_{\text{tag},i}]$. The AprilTag algorithm, meanwhile, internally assumes that $s_{\text{tag},i} = 1$ and this assumption must be respected when using H_i. The relationship between local tag and image coordinates is therefore:

[6] This means that the tags have 6×6 bit payloads and an inter-payload Hamming distance of 11. In practice these tags have a very low false positive detection rate while remaining computationally inexpensive to decode.

[7] `/sensing/apriltags2_ros/apriltags2`



(a) Tag local frame. Solid axes show the frame used in this report (with tag side half-length $s_{tag,i}$), dashed axes show the frame used by the AprilTag 2 core algorithm (with unity tag side half-length).

(b) Image frame, showing an example projection of tag i in (a). The principal point is generally different from the image geometric center due to real camera defects Hartley and Zisserman [65]. The projection of the point indicated by the black dot in (a) is also shown.

Figure 3.3: Coordinate frames used for bundle pose measurement.

Algorithm 1 Standalone tag pose in the camera frame measurement given $\mathcal{I}_{pin}$.
`/sensing/apriltags2_ros/apriltags2_ros/src/common_functions.cpp:detectTags`

1: Run AprilTag detection on $\mathcal{I}_{pin}$ ▷ Obtain H_i for $i \in \mathcal{D}$
2: Remove all $i \in \mathcal{D}$ appearing more than once ▷ Prevents failure from rogue duplicate tag presence
3: **for each** $i \in \mathcal{L}_{\mathcal{D}}$ **do**
4: $\mathcal{P}_O \leftarrow \varnothing$ ▷ Initialize the tuple of corner coordinates in {l} frame
5: $\mathcal{P}_I \leftarrow \varnothing$ ▷ Initialize the tuple of corner coordinates in {im} frame
6: Execute Algorithm 3 with $\tilde{T}_{lt_i} = I$ to update $\mathcal{P}_O$
7: Execute Algorithm 4 to update $\mathcal{P}_I$
8: $\hat{\boldsymbol{r}}, \hat{\boldsymbol{p}}_c^{t_i} \leftarrow$ `cv::solvePnP`$(\mathcal{P}_O, \mathcal{P}_I, K,$ no distortion$)$
9: $C_{(\hat{\boldsymbol{q}}_c^{t_i})} \leftarrow$ `cv::Rodrigues`$(\hat{\boldsymbol{r}})$ ▷ Convert rotation vector to rotation matrix
10: Convert $C_{(\hat{\boldsymbol{q}}_c^{t_i})}$ to $\hat{\boldsymbol{q}}_c^{t_i}$
11: **end for**

$$\begin{bmatrix} x'_{im} \\ y'_{im} \\ z'_{im} \end{bmatrix} = H_i \begin{bmatrix} x_{tag}/s_{tag,i} \\ -y_{tag}/s_{tag,i} \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} x_{im} \\ y_{im} \end{bmatrix} = \begin{bmatrix} x'_{im}/z'_{im} \\ y'_{im}/z'_{im} \end{bmatrix}, \tag{3.2}$$

where y_{tag} is negated because the AprilTag core algorithm uses a tag local frame where the y-axis points down (see Figure 3.3a). Note that because $H_i \in \mathbb{R}^{3\times3}$, out-of-plane local tag coordinates are not allowed. The third column of the usually $\mathbb{R}^{3\times4}$ homography matrix is removed for computational efficiency and the reasonable assumption that the tags are flat – otherwise they would likely not be detected in the first place.

Consider first the simple case of measuring the pose of single tags. AprilTag computes H_i from the four tag (black border) corners, thus these are the points to use for the PnP solver. Algorithm 1 explains how this is done. It is seen that pose measurement boils down to providing the $\{t_i\}$ frame to {im} frame point correspondences to the `cv::solvePnP` iterative solver based on Levenberg-Marquardt optimization OpenCV [66]. This returns the tag i pose in the camera frame based on the calibration matrix K.

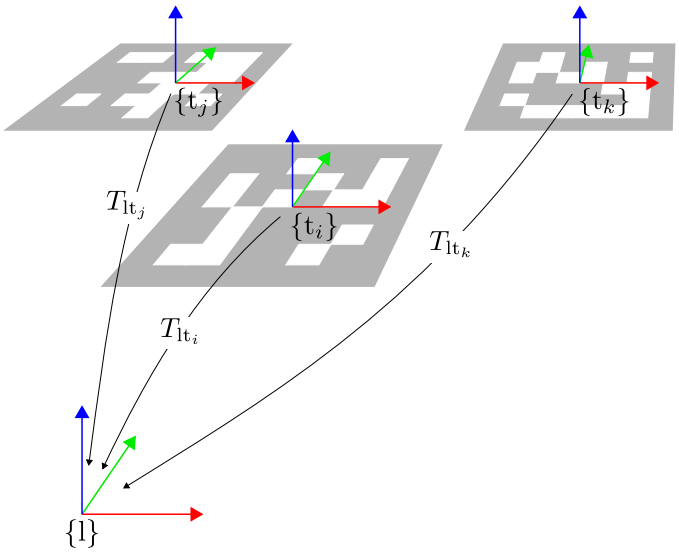


Figure 3.4: Rigid body transforms from each tag's local frame to the landing pad frame. Generic tag IDs i, j and k are shown.

Algorithm 2 Landing bundle pose measurement given $\mathcal{I}_{\text{pin}}$.
`/sensing/apriltags2_ros/apriltags2_ros/src/common_functions.cpp:detectTags`

Run AprilTag detection on $\mathcal{I}_{\text{pin}}$ ▷ Obtain H_i for $i \in \mathcal{D}$
Remove all $j \in \mathcal{D}$ appearing more than once ▷ Prevents failure from rogue duplicate tag presence
$\mathcal{P}_{\text{O}} \leftarrow \varnothing$ ▷ Initialize the tuple of corner coordinates in {l} frame
$\mathcal{P}_{\text{I}} \leftarrow \varnothing$ ▷ Initialize the tuple of corner coordinates in {im} frame
for each $i \in \mathcal{L}_{\mathcal{D}}$ **do**
    Execute Algorithm 3 to update $\mathcal{P}_{\text{O}}$
    Execute Algorithm 4 to update $\mathcal{P}_{\text{I}}$
end for
$\hat{\boldsymbol{r}}, \hat{\boldsymbol{p}}^{\text{l}}_{\text{c}} \leftarrow$`cv::solvePnP`$(\mathcal{P}_{\text{O}}, \mathcal{P}_{\text{I}}, K, \text{no distortion})$
$C_{(\hat{\boldsymbol{q}}^{\text{l}}_{\text{c}})} \leftarrow$`cv::Rodrigues`$(\hat{\boldsymbol{r}})$ ▷ Convert rotation vector to rotation matrix
$\hat{T}_{\text{cl}} \leftarrow \begin{bmatrix} C_{(\hat{\boldsymbol{q}}^{\text{l}}_{\text{c}})} & \hat{\boldsymbol{p}}^{\text{l}}_{\text{c}} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$
$\hat{T}_{\text{wl}} \leftarrow \tilde{T}_{\text{wb}} \tilde{T}_{\text{bc}} \hat{T}_{\text{cl}}$
$\hat{\boldsymbol{p}}^{\text{l}}_{\text{w}}, \hat{\boldsymbol{q}}^{\text{l}}_{\text{w}} \leftarrow \hat{T}_{\text{wl}}$

Algorithm 3 Addition of tag i corners to the set of points in the {l} frame.
`/sensing/apriltags2_ros/apriltags2_ros/src/common_functions.cpp:addObjectPoints`

$\mathcal{P}_{\text{O}} \leftarrow \langle \mathcal{P}_{\text{O}}, \tilde{T}_{\text{lt}_i}[1:3,:](-s_{\text{tag},i}, -s_{\text{tag},i}, 0, 1) \rangle$
$\mathcal{P}_{\text{O}} \leftarrow \langle \mathcal{P}_{\text{O}}, \tilde{T}_{\text{lt}_i}[1:3,:](\ s_{\text{tag},i}, -s_{\text{tag},i}, 0, 1) \rangle$
$\mathcal{P}_{\text{O}} \leftarrow \langle \mathcal{P}_{\text{O}}, \tilde{T}_{\text{lt}_i}[1:3,:](\ s_{\text{tag},i}, \ \ s_{\text{tag},i}, 0, 1) \rangle$
$\mathcal{P}_{\text{O}} \leftarrow \langle \mathcal{P}_{\text{O}}, \tilde{T}_{\text{lt}_i}[1:3,:](-s_{\text{tag},i}, \ \ s_{\text{tag},i}, 0, 1) \rangle$

Consider now the case of bundle pose measurement. Using a bundle provides a less noisy pose measurement because it uses a greater number of points for the PnP problem (four points corresponding to the four corners of each detected tag), which compensates for the noise in each point in a similar way as using more data would

Algorithm 4 Addition of tag i corners to the set of points in the {im} frame.
`/sensing/apriltags2_ros/apriltags2_ros/src/common_functions.cpp:addImagePoints`

1: $\mathcal{P}_\mathrm{I} \leftarrow \langle \mathcal{P}_\mathrm{I}, (x_\mathrm{im}, y_\mathrm{im})$ obtained from (3.2) with $(x_\mathrm{tag}, y_\mathrm{tag}) = (-s_{\mathrm{tag},i}, -s_{\mathrm{tag},i}) \rangle$
2: $\mathcal{P}_\mathrm{I} \leftarrow \langle \mathcal{P}_\mathrm{I}, (x_\mathrm{im}, y_\mathrm{im})$ obtained from (3.2) with $(x_\mathrm{tag}, y_\mathrm{tag}) = (\ \ s_{\mathrm{tag},i}, -s_{\mathrm{tag},i}) \rangle$
3: $\mathcal{P}_\mathrm{I} \leftarrow \langle \mathcal{P}_\mathrm{I}, (x_\mathrm{im}, y_\mathrm{im})$ obtained from (3.2) with $(x_\mathrm{tag}, y_\mathrm{tag}) = (\ \ s_{\mathrm{tag},i}, \ \ s_{\mathrm{tag},i}) \rangle$
4: $\mathcal{P}_\mathrm{I} \leftarrow \langle \mathcal{P}_\mathrm{I}, (x_\mathrm{im}, y_\mathrm{im})$ obtained from (3.2) with $(x_\mathrm{tag}, y_\mathrm{tag}) = (-s_{\mathrm{tag},i}, \ \ s_{\mathrm{tag},i}) \rangle$

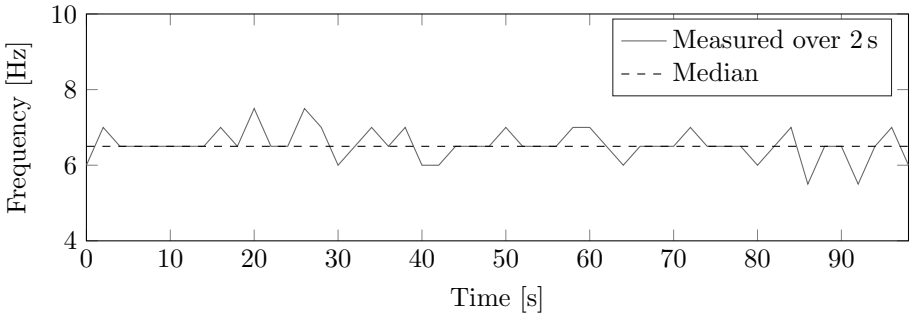


Figure 3.5: Landing bundle pose measurement frequency. The median is 6.5 Hz.

for a least squares regression (see Section 7.2). For this purpose, $\mathcal{L}$ is assumed to be *calibrated*. As shown in Figure 3.4, this means that an estimated rigid transformation matrix $\tilde{T}_{\mathrm{lt}_i}$ from each tag $i \in \mathcal{L}$ to the landing pad frame is available. Section 3.1.4 describes the calibration process. Algorithm 2 then explains how the bundle pose is measured. Note that obtaining the bundle pose in the world frame requires knowledge of the camera-IMU calibration $\tilde{T}_\mathrm{bc}$. As shown in Figure 3.5, the Odroid XU4 can execute Algorithm 2 at ≈ 7 Hz.

3.1.4 Landing Bundle Calibration

Bundle calibration computes the estimated pose $\tilde{T}_{\mathrm{lt}_i}$ of each tag $i \in \mathcal{L}$ with respect to the landing pad frame. Calibration is a one-time process that must be done for a new bundle or when the bundle is moved with respect to the landing pad[8].
Let {m} be the local frame of a new *master* tag $m \notin \mathcal{L}$. Let the master tag define the landing frame, i.e. {m} = {l}. As will be explained in Chapter 6, the quadrotor will attempt to land with the same position and yaw as {l}. **The calibration thus determines the desired landing pose**. Therefore, the master tag position and yaw are set to match the desired quadrotor landing position and yaw. In Figure 3.2a, the master tag is placed in the middle of the charging pad.
With the master tag placed, the downfacing camera is pointed at the tag bundle $\mathcal{L} \cup \{m\}$ for e.g. 1 minute. During this time, Algorithm 1 is run with the master-tag-augmented $\mathcal{L} \cup \{m\}$ set instead of just $\mathcal{L}$. As a result, several hundred $\hat{T}_{\mathrm{ct}_i}$ and $\hat{T}_\mathrm{cm}$ are collected.
Consider a single time instance in the collected data where tags m and i were among the detected tags in the corresponding $\mathcal{I}_\mathrm{pin}$. Figure 3.6 shows the ideal rigid transform triad which is created between these tags' frames and the camera frame. From Algorithm 1, $\hat{T}_\mathrm{cm}$ and $\hat{T}_{\mathrm{ct}_i}$ in Figure 3.6 are known. The unknown T_{lt_i}, i.e. the calibration transform, can be easily solved for (i.e. be "measured"):

[8]Since the desired landing yaw is often set by the bundle (e.g. keep a yaw with the camera closest to the bundle), yawing the bundle about the {l} frame origin does not require a new calibration.

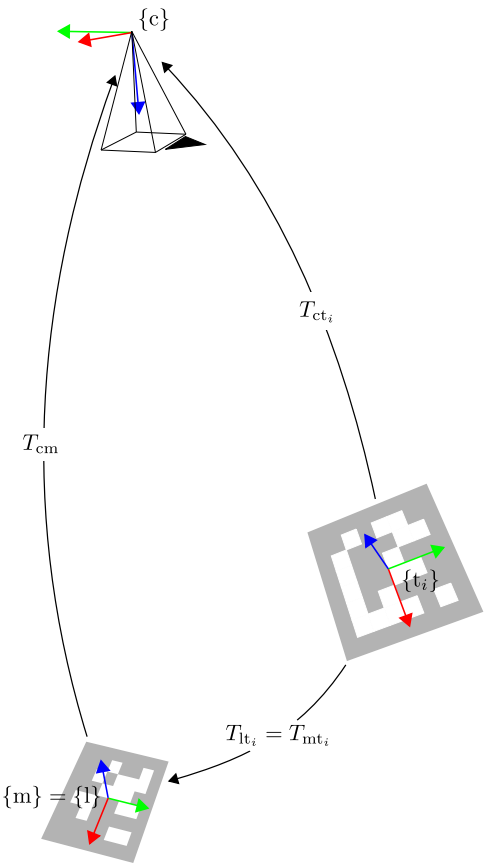


Figure 3.6: Rigid transform triad formed during the calibration process between the camera frame {c}, the master tag frame {m} and the tag i frame {t_i}. Tag i is shown to have a full translation and a full rotation with respect to tag m. In practice, tag i is placed such that it has a full translation and eventually a yaw. The complication of a non–zero relative roll and pitch is not necessary, although it is supported by Algorithm 5. The Blender [67] camera symbol is used where the black triangle denotes the image top edge.

$$\hat{T}_{\mathrm{lt}_i} = \hat{T}_{\mathrm{mt}_i} = \hat{T}_{\mathrm{cm}}^{-1} \hat{T}_{\mathrm{ct}_i}. \tag{3.3}$$

However, every standalone tag pose measurement has noise due to effects from image blur, pixelation, etc. The result of the single-measurement computation (3.3) is therefore not reliable. For this reason, the several hundred collected rigid transforms are combined into a geometric median position and a mean quaternion attitude of the {l} frame. To do this, let $\mathcal{T}_{\mathrm{C}} := \{\mathcal{T}_j,\, j = 1, \ldots, N_{\mathrm{calib}}\}$ where $\mathcal{T}_j$ is the set of rigid transforms for tags visible in the j-th $\mathcal{I}_{\mathrm{pin}}$ and N_{calib} is the total number of images processed during calibration. Calibrating the full $\mathcal{L}$ requires that $\forall i \in \mathcal{L}\ \exists \mathcal{T}_j \in \mathcal{T}_{\mathrm{C}}$ which contains both $\hat{T}_{\mathrm{cm}}$ and $\hat{T}_{\mathrm{ct}_i}$. This requirement follows from (3.3) which uses both quantities to compute a single $\hat{T}_{\mathrm{lt}_i}$ over whose instances the median and mean are then taken.
The set $\mathcal{T}_{\mathrm{C}}$ is all that is required for calibration and the user would provide this via a ROS bag file to our bundle calibration script[9]. Algorithm 5 then provides the

[9] `/bundle_calibration/apriltag_bundle_calibration.m`

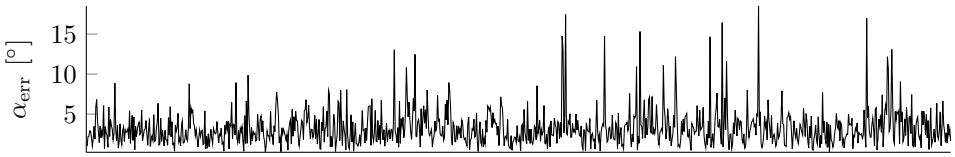


Figure 3.7: Calibration attitude variation angle α_{err} of (3.4) for a 1000-element subset of all possible quaternion measurement combinations. Shown only for the noisiest tag in $\mathcal{L}$ in this particular calibration.

completely automated calibration algorithm which outputs $\tilde{T}_{\text{l}t_i}\ \forall i \in \mathcal{L}$.
Note that Algorithm 5 uses the Markley et al. [68] method for computing the average quaternion (see e.g. Trawny and Roumeliotis [69], Solà [70] for a mathematical background on quaternions). This is a robust and accurate method which has the important property of generating a unique $\tilde{\boldsymbol{q}}_{\text{l}}^{\text{t}_i}$ under the mild, sufficient, condition that:

$$\alpha_{\text{var}} := 2\arccos(q_{\text{var,w}}) < \frac{\pi}{2}, \text{ where } \boldsymbol{q}_{\text{var}} := \hat{\boldsymbol{q}}_1 \otimes \hat{\boldsymbol{q}}_2^{-1},\ \forall \hat{\boldsymbol{q}}_1, \hat{\boldsymbol{q}}_2 \in \mathcal{Q}_{\text{l}}^{\text{t}_i}, \tag{3.4}$$

where $\mathcal{Q}_{\text{l}}^{\text{t}_i}$ is the set of quaternion attitude measurements of tag i with respect to the master tag frame. This condition states that no two attitude measurements of tag i with respect to tag m may have more than 90° discrepancy (in the axis-angle rotation sense). Figure 3.7 validates this condition for a typical calibration on one of the tags in the bundle. It is seen that noise in the AprilTag detector satisfies this condition by a large margin.

3.1.5 Recursive Least Squares Bundle Pose Estimation

Flight testing revealed that raw landing bundle pose measurements are still noisy and have occasional jumps due to motion blur and motor-induced vibration (see Section 7.2). This is especially true for larger $\|\boldsymbol{p}_{\text{l}}^{\text{c}}\|$ values. This noise causes undesirable uncertainty about where the quadrotor should land, causing the craft to move horizontally more than necessary as it targets the measured {l} frame origin. A Recursive Least Squares (RLS) filter is used to reduce this noise.

General RLS Formulation[10]

RLS is a digital adaptive filter[11] which is the optimal estimator for constant parameters with Gaussian noise in their measurements. In particular, standard RLS computes the optimal estimate $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ of a constant $\boldsymbol{x} \in \mathbb{R}^n$. Note that the RLS algorithm is identical to a Kalman Filter (KF) applied to a static system.
Optimality of the estimate is achieved in the weighted least squares sense[12]:

$$\underset{\tilde{\boldsymbol{x}}}{\text{minimize}} \quad \sum_{j=1}^{k} \big(\lambda^{j-k}(\hat{\boldsymbol{x}}[j] - H[j]\tilde{\boldsymbol{x}})^{\mathsf{T}} R[j]^{-1}(\hat{\boldsymbol{x}}[j] - H[j]\tilde{\boldsymbol{x}})\big), \tag{3.6}$$

where k represents the current time step, $\lambda \in (0, 1]$ is an *exponential forgetting factor* and $H[j] \in \mathbb{R}^{m\times n}$ defines the measurement model at time step j such that:

$$\boldsymbol{z}[j] = H[j]\boldsymbol{x} + \boldsymbol{w}[j] \in \mathbb{R}^m \quad \text{where } \boldsymbol{w}[j] \sim \mathcal{N}(0, R[j]) \in \mathbb{R}^m \text{ and } n \geq m. \tag{3.7}$$

[10] Generic variables such as x, H, etc. are local to this section.

[11] `/general/alure_common/include/alure_common/rls_estimator.h`

[12] Strictly speaking, like the KF, optimality is contingent on a correct initialization of the RLS filter.

Algorithm 5 Bundle calibration.
`/bundle_calibration/apriltag_bundle_calibration.m`

Initialize:
for each $i \in \mathcal{L}$ **do**
  $\mathcal{P}_\mathrm{l}^{\mathrm{t}_i} \leftarrow \varnothing$ ▷ Set of measured relative translations $\hat{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i}$
  $\mathcal{Q}_\mathrm{l}^{\mathrm{t}_i} \leftarrow \varnothing$ ▷ Set of measured relative orientations $\hat{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i}$
end for
Fill $\mathcal{P}_\mathrm{l}^{\mathrm{t}_i}$ and $\mathcal{Q}_\mathrm{l}^{\mathrm{t}_i}$:
for each $\mathcal{T} \in \mathcal{T}_\mathrm{C}$ **do**
  if $\hat{T}_\mathrm{cm} \notin \mathcal{T}$ **then**
    continue ▷ Without master, the triad in Figure 3.6 is incomplete
  end if
  for each $\hat{T}_{\mathrm{ct}_i} \in \mathcal{T}$ **do**
    $\hat{T}_{\mathrm{lt}_i} \leftarrow \hat{T}_\mathrm{cm}^{-1} \hat{T}_{\mathrm{ct}_i}$ ▷ Apply (3.3)
    $\mathcal{P}_\mathrm{l}^{\mathrm{t}_i} \leftarrow P_\mathrm{l}^{\mathrm{t}_i} \cup \{\hat{T}_{\mathrm{lt}_i}[1:3,4]\}$
    Extract $\hat{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i}$ from $\hat{T}_{\mathrm{lt}_i}[1:3,1:3]$ using Martin John Baker [71]
    $\mathcal{Q}_\mathrm{l}^{\mathrm{t}_i} \leftarrow \mathcal{Q}_\mathrm{l}^{\mathrm{t}_i} \cup \{\hat{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i}\}$
  end for
end for
Compute geometric median position estimates $\tilde{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i}$:
for each $\mathcal{P}_\mathrm{l}^{\mathrm{t}_i} \neq \varnothing$ **do** ▷ Calibration requirement
  $\boldsymbol{p}_0 \leftarrow |\mathcal{P}_\mathrm{l}^{\mathrm{t}_i}|^{-1} \sum_{\boldsymbol{p} \in \mathcal{P}_\mathrm{l}^{\mathrm{t}_i}} \boldsymbol{p}$ ▷ Mean position
  Compute geometric median via MATLAB's `fminsearch`, with $\boldsymbol{p}_0$ as initial solution: ▷ This is a convex problem, thus its solution is unique

$$\underset{\tilde{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i}}{\text{minimize}} \quad \sum_{\boldsymbol{p} \in \mathcal{P}_\mathrm{l}^{\mathrm{t}_i}} \|\boldsymbol{p} - \hat{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i}\|_2 \tag{3.5}$$

end for
Compute mean attitude estimates $\tilde{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i}$: ▷ Apply Markley et al. [68]
for each $\mathcal{Q}_\mathrm{l}^{\mathrm{t}_i} \neq \varnothing$ **do** ▷ Calibration requirement
  $M \leftarrow \sum_{\boldsymbol{q} \in \mathcal{Q}_\mathrm{l}^{\mathrm{t}_i}} \boldsymbol{q}\boldsymbol{q}^\mathsf{T}$
  $V, D \leftarrow$ eigendecomposition $M = VDV^{-1}$ ▷ Use MATLAB's `eig`
  $\boldsymbol{\lambda} \leftarrow \mathtt{diag}(D)$ ▷ Vector of eigenvalues of M
  $\boldsymbol{v} \leftarrow V[\arg\max_i \lambda[i], :]$ ▷ Eigenvector of M for the largest eigenvalue
  $\tilde{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i} \leftarrow \boldsymbol{v} / \|\boldsymbol{v}\|$
end for
Compute calibration rigid body transforms:
for each i such that $\tilde{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i}$, $\tilde{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i}$ are available **do**

$$\tilde{T}_{\mathrm{lt}_i} \leftarrow \begin{bmatrix} C_{(\tilde{\boldsymbol{q}}_\mathrm{l}^{\mathrm{t}_i})} & \tilde{\boldsymbol{p}}_\mathrm{l}^{\mathrm{t}_i} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

end for

Figure 3.8: Input-ouput view of the RLS filter.

RLS makes the following assumptions:

- The linear measurement model (3.7) is the correct representation of reality. In particular, the model is linear in $\boldsymbol{x}$ and the measurement error $(\boldsymbol{z}[j] - H[j]\boldsymbol{x})$ is an additive zero-mean Gaussian noise $\boldsymbol{w}[j]$ with covariance $R[j] \in \mathbb{R}^{m\times m}_{++}$;
- The noise samples $\{\boldsymbol{w}[1], \boldsymbol{w}[2], \ldots\}$ are independent.

The λ term serves two roles:

1. It relaxes the assumption that $\boldsymbol{x}$ is constant, allowing it to vary with slower dynamics than the algorithm's convergence speed;
2. It makes the filter forget old measurements. This is useful if e.g. the old measurements are known to be less precise or somehow become "outdated" with respect to the more recent measurements.

In both cases, a good way to set λ is by relating it to the algorithm convergence speed Derek [72]:

$$N_\tau = \frac{\lambda}{1-\lambda}, \tag{3.8}$$

where N_τ is the time constant[13] expressed as a number of iterations. Usually $\lambda \in [0.95, 0.995]$ is chosen which yields $N_\tau \in [19, 199]$. After $3N_\tau$ the old measurement is weighed at 5% and may be assumed for practical purposes to have been entirely forgotten.
Figure 3.8 shows an input-output black box view of the RLS filter. The estimate mean, $\tilde{\boldsymbol{x}}[k]$, and covariance, $P[k]$, are output. Algorithm 6 explains one RLS iteration. Note that, thanks to recursion, only $\tilde{\boldsymbol{x}}[k-1]$ and $P[k-1]$ need to be known at iteration k which drastically improves efficiency as the full (3.6) need not be recomputed. For $\lambda = 1$, Algorithm 6 exactly matches the measurement update step of the KF.

Application of RLS to Bundle Pose Estimation

The general formulation (3.7) is specialized to the problem of estimating the landing bundle pose[14]. In this special case, pose shall be defined as the regular position $\boldsymbol{p}^{\mathrm{l}}_{\mathrm{w}}$ and the *only-yaw quaternion* $\boldsymbol{q}^{\mathrm{l,yaw}}_{\mathrm{w}}$. Let ψ_{l} denote the landing pad yaw angle, then $\boldsymbol{q}^{\mathrm{l,yaw}}_{\mathrm{w}} = (\cos(\psi_{\mathrm{l}}/2), 0, 0, \sin(\psi_{\mathrm{l}}/2))$. Because $\boldsymbol{q}^{\mathrm{l,yaw}}_{\mathrm{w}}$ is a unit quaternion, only one of its two non-zero components needs to be estimated. The $q^{\mathrm{l,yaw}}_{\mathrm{w,z}}$ component is chosen. Note that estimation of an only-yaw quaternion component, rather than the yaw itself, is crucial to avoid wrap-around issues associated with e.g. a yaw angle defined on $[0, 2\pi)$. In the latter case, a $1\,^\circ$ and a $359\,^\circ$ yaw are physically almost equal (with only a minor $2\,^\circ$ difference) but their intermixed presence as measurements would lead to an incorrect estimate somewhere between $0\,^\circ$ and $360\,^\circ$. The RLS measurement model (3.7) then becomes:

[13] The time constant is the time for a disturbance to decay to e^{-1} of its initial value.

[14] `/autonomy_engine/alure_landing/include/alure_landing/charging_pad_pose_estimator.h`

Algorithm 6 Iteration k of the RLS filter.
`/general/alure_common/src/rls_estimator.cpp:initialize`
`/general/alure_common/src/rls_estimator.cpp:update`

1: **if** $k = 0$ **then**
Initialization:
2: $\tilde{\boldsymbol{x}}[0] \leftarrow \bar{\boldsymbol{x}}$ ▷ Initial knowledge of the mean
3: $P[0] \leftarrow P_{\boldsymbol{x}}$ ▷ Initial knowledge of the variance
4: **else**
Recursion:

$$K[k] \leftarrow P[k-1]H[k]^{\mathsf{T}}(H[k]P[k-1]H[k]^{\mathsf{T}} + \lambda R[k])^{-1}$$
$$\tilde{\boldsymbol{x}}[k] \leftarrow \tilde{\boldsymbol{x}}[k-1] + K[k](\boldsymbol{z}[k] - H[k]\tilde{\boldsymbol{x}}[k-1])$$
$$P[k] \leftarrow \frac{1}{\lambda}(I - K[k]H[k])P[k-1]$$

5: **end if**

$$\boldsymbol{x} = \begin{bmatrix} p^{\mathrm{l}}_{\mathrm{w,x}} \\ p^{\mathrm{l}}_{\mathrm{w,y}} \\ p^{\mathrm{l}}_{\mathrm{w,z}} \\ q^{\mathrm{l,yaw}}_{\mathrm{w,z}} \end{bmatrix} \quad \boldsymbol{z}[j] = \begin{bmatrix} \hat{p}^{\mathrm{l}}_{\mathrm{w,x}}[j] \\ \hat{p}^{\mathrm{l}}_{\mathrm{w,y}}[j] \\ \hat{p}^{\mathrm{l}}_{\mathrm{w,z}}[j] \\ \hat{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}}[j] \end{bmatrix} \quad H[j] = I_4, \tag{3.9}$$

During landing, bundle pose measurements become more precise as the quadrotor approaches the landing pad (see Appendix C). It is desirable then to smoothly replace the noisy and error-polluted old measurements with the more accurate recent ones. Exponential forgetting is leveraged for this purpose. Figure 3.5 shows that bundle pose measurements arrive at $\approx 7\,\mathrm{Hz}$. The filter sampling period is thus $T \approx 7^{-1}\,\mathrm{s}$. Using $\lambda = 0.8$ the filter time constant is computed using (3.8) to be

$$\tau = N_\tau T = \frac{\lambda T}{1-\lambda} \approx 0.6\,\mathrm{s},$$

such that $3\tau \approx 1.8\,\mathrm{s}$ (aforementioned "time to forget" a measurement) is much shorter than the typical landing duration ($\approx 30\,\mathrm{s}$ from a $10\,\mathrm{m}$ initial altitude at $0.3\,\mathrm{m/s}$ descent speed, which is the current setting). Exponential forgetting thus helps to keep an up-to-date pose estimate using more accurate recent measurements without error-prone old measurements compromising the estimate's accuracy.

To be able to apply Algorithm 6 $\bar{x}$, $P_{\boldsymbol{x}}$ and $R[k]$ must be available. These are computed via sample statistics on scalar buffers holding the recent measurement history, made possible by the fact that the landing pad is assumed stationary.

Let us first describe how this is done for a generic scalar quantity z. First, a scalar measurement history buffer[15] $\mathcal{B}_{\hat{z}}$ of length M_{buffer} is formed. When an element is added to $\mathcal{B}_{\hat{z}}$ that would make the buffer size $> M_{\mathrm{buffer}}$, the oldest element is removed (via `std::deque`'s `push_back` and `pop_front` methods ISO [73], see Algorithm 9). Letting $\mathcal{B}_{\hat{z}}[j]$ denote the j-th element, the unbiased sample mean and variance are given by:

[15] `/general/alure_common/include/alure_common/univariate_sample_buffer.h`

$$\mu_{\hat{z}} = \frac{1}{|B_{\hat{z}}|} \sum_{j=1}^{|B_{\hat{z}}|} \mathcal{B}_{\hat{z}}[j] \qquad \text{Sample mean,} \tag{3.10}$$

$$s_{\hat{z}}^2 = \frac{1}{|B_{\hat{z}}| - 1} \sum_{j=1}^{|B_{\hat{z}}|} (\mathcal{B}_{\hat{z}}[j] - \mu_{\hat{z}})^2 \qquad \text{Sample variance,} \tag{3.11}$$

where $|B_{\hat{z}}|$ denotes the number of elements currently in the buffer. The finite-precision computer implementation of (3.11) is prone to loss of precision via catastrophic cancellation due to taking the difference between two large, similar, sums (the squared sum of $B_{\hat{z}}[j]$ and the sum of $B_{\hat{z}}[j]^2$). In severe cases when $s_{\hat{z}}^2 \ll \mu_{\hat{z}}^2$, a negative sample variance may be computed. To avoid this issue, Algorithm 7 is used as it is a much more numerically robust way of computing the mean and variance Welford [74], Knuth [75]. Note that while Algorithm 7 may be applied as a recursion when a new element is added to $\mathcal{B}_{\hat{z}}$, the current application is not recursive because the oldest element of $\mathcal{B}_{\hat{z}}$ is simultaneously removed[16]. Also note that Algorithm 7 naturally requires that $|\mathcal{B}_{\hat{z}}| > 1$.

Algorithm 8 then gives the bundle pose estimator itself. M_{init} denotes the number of initial bundle pose measurements required to initialize the RLS filter. The current implementation[17] uses $M_{\text{init}} = 5$ and $M_{\text{buffer}} = 50$, which has worked well in flight tests. It is noted that Algorithm 8 is not much more than a manager of the four measurement buffers for the four elements of $\boldsymbol{z}$ in (3.9) and a caller of the core RLS Algorithm 6. In addition, our implementation provides a RESETESTIMATOR() method which simply clears the four measurement buffers $\mathcal{B}_{\hat{x}}$, $\mathcal{B}_{\hat{y}}$, $\mathcal{B}_{\hat{z}}$ and $\mathcal{B}_{\hat{\psi}}$ and sets the state back to INITIALIZING. Results from using Algorithm 8 for landing are presented in Section 7.2.

Algorithm 7 Welford's method, a one–pass computation of mean and variance.
`/general/alure_common/src/univariate_sample_buffer.cpp:computeMeanVariance`

function COMPUTEMEANVARIANCE($\mathcal{B}_{\hat{z}}$)
  $n \leftarrow 0$ ▷ Initialize
  $\mu_{\hat{z}} \leftarrow 0$
  $s \leftarrow 0$
  for each $\hat{z} \in \mathcal{B}_{\hat{z}}$ **do** ▷ Iterate
    $n \leftarrow n + 1$
    $\delta \leftarrow \hat{z} - \mu_{\hat{z}}$
    $\mu_{\hat{z}} \leftarrow \mu_{\hat{z}} + \frac{\delta}{n}$
    $\delta_2 \leftarrow \hat{z} - \mu_{\hat{z}}$
    $s \leftarrow s + \delta\delta_2$
  end for
  $s_{\hat{z}}^2 \leftarrow \frac{s}{n-1}$ ▷ Assumes that $n > 1$
  return $\mu_{\hat{z}}$, $s_{\hat{z}}^2$
end function

[16] An alternative that would allow to apply recursion is to introduce an exponential forgetting factor into Welford's method Finch [76].

[17] `/autonomy_engine/alure_landing/config/landing_parameters.yaml`

Algorithm 8 Landing bundle pose estimator.
`/autonomy_engine/alure_landing/src/charging_pad_pose_estimator.cpp: stateMachineDefinition`

Initialize:
state ← INITIALIZING
Run:
while true do
  Wait for a new bundle pose detection $\hat{\boldsymbol{p}}^{\mathrm{l}}_{\mathrm{w}}$, $\hat{\boldsymbol{q}}^{\mathrm{l}}_{\mathrm{w}}$
  $\hat{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}} \leftarrow \sin(\frac{1}{2}\text{CONVERTQUATERNIONTOYAW}(\hat{\boldsymbol{q}}^{\mathrm{l}}_{\mathrm{w}}))$ ▷ Apply Algorithm 44
  $\mu_{\hat{x}}, s^2_{\hat{x}} \leftarrow$ ADDDATUM($\mathcal{B}_{\hat{x}}$, $\hat{p}^{\mathrm{l}}_{\mathrm{w,x}}$) ▷ Update buffers using Algorithm 9
  $\mu_{\hat{y}}, s^2_{\hat{y}} \leftarrow$ ADDDATUM($\mathcal{B}_{\hat{y}}$, $\hat{p}^{\mathrm{l}}_{\mathrm{w,y}}$)
  $\mu_{\hat{z}}, s^2_{\hat{z}} \leftarrow$ ADDDATUM($\mathcal{B}_{\hat{z}}$, $\hat{p}^{\mathrm{l}}_{\mathrm{w,z}}$)
  $\mu_{\hat{\psi}}, s^2_{\hat{\psi}} \leftarrow$ ADDDATUM($\mathcal{B}_{\hat{\psi}}$, $\hat{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}}$)
  switch state **do**
    case INITIALIZING
      if $|\mathcal{B}_x| \geq M_{\mathrm{init}}$ **then**
        $\bar{\boldsymbol{x}} \leftarrow (\mu_{\hat{x}}, \mu_{\hat{y}}, \mu_{\hat{z}}, \mu_{\hat{\psi}})$
        $P_{\boldsymbol{x}} \leftarrow \mathtt{diag}(s^2_{\hat{x}}, s^2_{\hat{y}}, s^2_{\hat{z}}, s^2_{\hat{\psi}})$
        Execute one step of Algorithm 6 ▷ Initialize the RLS filter
        state ← RLS_RUNNING
      end if
    case RLS_RUNNING
      $\hat{\boldsymbol{x}}[k] \leftarrow (\hat{p}^{\mathrm{l}}_{\mathrm{w,x}}, \hat{p}^{\mathrm{l}}_{\mathrm{w,y}}, \hat{p}^{\mathrm{l}}_{\mathrm{w,z}}, \hat{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}})$
      $R[k] \leftarrow \mathtt{diag}(s^2_{\hat{x}}, s^2_{\hat{y}}, s^2_{\hat{z}}, s^2_{\hat{\psi}})$
      Execute one step of Algorithm 6 ▷ Iterate RLS filter
      $\tilde{\boldsymbol{p}}^{\mathrm{l}}_{\mathrm{w}} \leftarrow (\tilde{\boldsymbol{x}}[k])[1:3]$ ▷ Estimated bundle position
      $\tilde{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}} \leftarrow (\tilde{\boldsymbol{x}}[k])[4]$
      $\tilde{\boldsymbol{q}}^{\mathrm{l,yaw}}_{\mathrm{w}} \leftarrow (\sqrt{1-(\tilde{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}})^2}, 0, 0, \tilde{q}^{\mathrm{l,yaw}}_{\mathrm{w,z}})$ ▷ Estimated bundle yaw
  end switch
end while

Algorithm 9 Function for adding an element to the measurement buffer.
`/general/alure_common/src/univariate_sample_buffer.cpp:addDatum`

function ADDDATUM($\mathcal{B}_{\hat{z}}$, $\hat{z}$)
  if $|\mathcal{B}_{\hat{z}}| \geq M$ **then**
    $\mathcal{B}_{\hat{z}}$`.pop_front()` ▷ Respect maximum history
  end if
  $\mathcal{B}_{\hat{z}}$`.push_back(`$\hat{z}$`)` ▷ Add the new measurement
  if $|\mathcal{B}_{\hat{z}}| > 1$ **then** ▷ Recompute sample mean and variance (Algorithm 7)
    return COMPUTEMEANVARIANCE($\mathcal{B}_{\hat{z}}$)
  else
    return 0, 0
  end if
end function

3.2 Quadrotor State Estimation

Quadrotor state estimation concerns the computation of the quadrotor state for use in e.g. guidance and control. For this purpose the SSF algorithm[18] Weiss et al. [77] is extended to work in an outdoor environment using an Inertial Measurement Unit (IMU), a GPS, a pressure sensor and a 3-axis magnetometer. This work was not the subject of this thesis but was done in parallel to the thesis by another project member. In the interest of completeness, this section briefly introduces SSF and the extensions that were made.

3.2.1 Introduction to SSF

SSF is a versatile and modular EKF-based algorithm which is able to fuse slow, noisy, delayed and possibly arbitrarily scaled measurements into a coherent state propagated at (up to) the IMU rate. The algorithm's theoretical foundation is described in Weiss and Siegwart [78] while its implementation details are given in Weiss et al. [77]. SSF uses the following state vector:

$$\boldsymbol{x}_{\text{state}} = (\underbrace{\boldsymbol{p}_{\text{w}}^{\text{b}}, \boldsymbol{v}_{\text{w}}^{\text{b}}, \boldsymbol{q}_{\text{w}}^{\text{b}}, \boldsymbol{b}_{\omega}, \boldsymbol{b}_{\text{a}}}_{\boldsymbol{x}_{\text{state,core}}}, (\lambda_{\text{s}}, \boldsymbol{p}_{\text{b}}^{\text{s}}, \boldsymbol{q}_{\text{b}}^{\text{s}}), \cdots), \tag{3.12}$$

where $\{\text{b}\}$ is the body (i.e. IMU) frame, $\boldsymbol{b}_{\omega} \in \mathbb{R}^3$ and $\boldsymbol{b}_{\text{a}} \in \mathbb{R}^3$ are the gyroscope and accelerometer biases and the $\cdots$ signify that the sensor calibration states $(\lambda_{\text{s}}, \boldsymbol{p}_{\text{b}}^{\text{s}}, \boldsymbol{q}_{\text{b}}^{\text{s}})$ are replicated for each new sensor used (e.g. camera, GPS, pressure, etc.). In particular, the calibration states are:

- The measurement scale factor λ_{s}. Included only when a scaling exists (e.g. for a camera);
- The sensor translation with respect to the IMU, $\boldsymbol{p}_{\text{b}}^{\text{s}}$[19]. Included only for sensors that measure position (e.g. GPS);
- The sensor rotation with respect to the IMU, $\boldsymbol{q}_{\text{b}}^{\text{s}}$. Included only for sensors that measure orientation (e.g. magnetometer).

The IMU is the "core" sensor for SSF which provides the measurements for core state $\boldsymbol{x}_{\text{state,core}}$ propagation. This is why the body frame coincides with the IMU frame. In practice, due to the presence of quaternions in (3.12), SSF uses the error state representation of (3.12) which increases numerical stability and handles the quaternion in its minimal representation Weiss and Siegwart [78], Trawny and Roumeliotis [69].
The original SSF algorithm uses the IMU together with one 6 DOF pose sensor (a camera) or one 3 DOF position sensor (e.g. a GPS or a laser tracker). Our implementation extends the framework to handle multi-sensor fusion as described for (3.12). The next section explains how we use this feature.

3.2.2 Extensions to SSF

The starting point for this project was an already modified SSF version from prior JPL work which improved the original algorithm from the functional and software stability perspectives. To meet S2 of Specification List 1, however, further extensions to SSF were required. This is motivated by the following reasons:

[18] `/navigation/jpl_sensor_fusion`

[19] The {s} sensor frame here is not to be confused with the spiral grid search trajectory frame in Chapter 4.

1. The long-term autonomy mission described in Chapter 6 involves hover and constant velocity flight periods throughout the mission. Yet without a magnetometer, yaw observability is only guaranteed when there is sufficient acceleration in least two IMU axes Weiss et al. [77]. In practice, the lack of magnetometer causes the quadrotor to describe a circular pattern in the horizontal plane after prolonged hovering;

    - **Solution**: include a magnetometer. We use the 3-axis magnetometer that comes with the AscTec AutoPilot was used Honeywell [25] (see Figure 2.3). Contrary to most implementations which use the magnetometer for yaw-only estimation, our implementation uses the normalized magnetic field vector $\boldsymbol{m}_{\mathrm{w}}$ (in the world frame) to update the full attitude quaternion $\boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}}$.

2. Our GPS exhibits multiple-meter drift in the $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}}$ axis, prompting the usage of other sensors to improve the height measurement;

    - **Solution**: include an additional sensor for height measurement. For this purpose, the pressure sensor bundled with the AscTec AutoPilot was used to measure the pressure height NXP [34]. In the future, sensors such as the TeraBee One [79] could potentially be used for greater accuracy[20].

3. The state needs to be reinitialized prior to each takeoff part of the state (e.g. accelerometer bias $\boldsymbol{b}_{\mathrm{a}}$) drifts while the quadrotor is stationary on the charging pad.

    - **Solution**: functionally extend SSF to allow hard re-initialization based on a ROS service call.

With the extra sensors (GPS, pressure and magnetometer), the SSF state becomes:

$$\boldsymbol{x}_{\mathrm{state}} = (\boldsymbol{p}_{\mathrm{w}}^{\mathrm{b}}, \boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}, \boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}}, \boldsymbol{b}_{\omega}, \boldsymbol{b}_{\mathrm{a}}, \boldsymbol{b}_{\mathrm{p}}, \boldsymbol{p}_{\mathrm{b}}^{\mathrm{g}}, \boldsymbol{m}_{\mathrm{w}}, \boldsymbol{q}_{\mathrm{b}}^{\mathrm{m}}, \boldsymbol{p}_{\mathrm{b}}^{\mathrm{p}}), \tag{3.13}$$

Note that for a better $\boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}$ estimate, we use the GPS velocity output as well for the measurement update step.
Originally, the state propagation part of SSF was located on the HLP. However, while the original SSF was designed to work with one sensor, we now have four sensors (GPS position, GPS velocity, pressure and magnetometer) which increases the measurement update rate. This increased rate exposed a bandwidth limitation issue in the serial link between the Odroid XU4 and the HLP. The symptoms of this limitation are small instabilities in the state estimate which sometimes grow unbounded. This was resolved by doing the state propagation on the Odroid XU4 at 200 Hz rather than directly on the HLP at 1 kHz. While this works, it likely introduces measurement delay into the control loop due to ROS network/serial communication delays. It is hypothesized that this occasionally leads to oscillatory position control (see Section 7.4.3). In a future SSF implementation for this project, HLP-side state propagation may be re-attempted.

[20] Range sensors have the drawback of being ground-referenced. If the ground is uneven (such as in a flight over bushes or shipping containers), a range sensor will measure a varying height even though the quadrotor is flying at a constant sea-level height.

Chapter 4

Guidance

The guidance subsystem is responsible for generating control reference signals with the aim of following a specific trajectory or arriving at a specific destination Grewal et al. [80]. Guidance design is subject to the design specifications given by Specification List 2. This report approaches the problem by splitting the guidance subsystem into:

- Trajectory *generators*, described in Section 4.1, which compute the desired trajectory;
- A trajectory *sequencer*, described in Section 4.2, which allows chaining together several trajectories with a well-defined set of mechanics;
- A trajectory *tracker*, described in Section 4.3, which uses the sequencer to send the controller references and implements additional robustness logic that leverages the trajectory sequencer's design.

S1	Robustly generate trajectories that are sufficient for the quadrotor to explore an open space and that are computational inexpensive such that they may be generated on-demand in real time.
S2	Provide an easy-to-use way of generating a "mission trajectory" which consists of point-to-point constant velocity segments and may contain hovering phases.
S3	Provide the ability to create a "search trajectory" in the event that, after returning home to land, the landing AprilTag bundle is not visible in the quadrotor's downfacing camera image..
S4	Provide an easy-to-use, high-level interface for chaining multiple trajectories together in real-time.
S5	Provide an easy-to-use, high-level interface to send the controller reference and to determine when the quadrotor has finished tracking the trajectory sequence.

Specification List 2: Guidance subsystem design specifications.

4.1 Trajectory Generators

4.1.1 Literature Review

Mellinger and Kumar [81] developed a snap-continuous trajectory generation methodology harnessing quadrotor differential flatness (see Appendix E). Quadrotors have actually been known to be differentially flat since the early 2000s. Murray et al. [82] show that every system that is dynamic feedback linearizable via endogenous feedback Martin [83] is differentially flat. In 2001, Mistler et al. [84] showed that a quadrotor is dynamic feedback linearizable via endogenous feedback. Since the popularization of using differential flatness for dynamically feasible multirotor trajectory generation by Mellinger and Kumar [81], the works of Richter et al. [85], Burri et al. [86], de Almeida and Akella [87] have improved the trajectory generation's numerical robustness. The method has also been expanded for better handling guidance in cluttered environments since the previous approaches do not guarantee a collision-free trajectory between waypoints Oleynikova et al. [88], Liu et al. [89], Campos-Macias et al. [90].
Mueller and D'Andrea [91], Mueller et al. [92], Hehn and D'Andrea [93] present a simplified but real-time trajectory generation framework that is nearly two orders of magnitude faster (on the order of tens of μs) than the above works (on the order to tens of ms). This makes it, unlike the previous works, suitable for use-cases where continuous trajectory re-planning or the evaluation of millions of trajectory candidates may be required (e.g. for Model Predictive Control (MPC)).
In our case of data acquisition, the focus of the above works on trajectory time optimality is less important than trajectory energy optimality. The quadrotor should fly for as long as possible and, when the battery is low and the quadrotor is returning home to land, trajectory generation should be prudent to not waste energy. Some quadrotor-specific energy efficient trajectory generation research is available Vicencio et al. [94], Morbidi et al. [95]. Even better, abundant literature is available on fuel-optimal trajectory generation via convex optimization for planetary landers Acikmese and Ploen [96], Blackmore et al. [97], Blackmore and Acikmese [98], Acikmese et al. [99], Szmuk et al. [100]. Given the similarities between quadrotors and rockets (the translation control of both depends on thrust vectoring), this mature research may be applicable to e.g. an energy-efficient return to home trajectory generation.

4.1.2 Polynomial Trajectory Generation

In light of Section 4.1.1 and S1 of Specification List 2, polynomial trajectory generation is chosen. However, unlike Mellinger and Kumar [81], Richter et al. [85], Burri et al. [86], the trajectory generators in this report use fully constrained polynomials. This means that there is no optimization involved and trajectory generation is extremely fast with matrix inversion (for matrices at most 30×30 in size) being the most time consuming part. Using polynomials for individual segments, all trajectories required for this project can be built. The current implementation[1] plans trajectories only in position while yaw can be set arbitrarily in the trajectory tracker (Section 4.3). The remainder of this section describes each trajectory generator in detail:

- Section 4.1.3 introduces basic polynomial theory which creates the foundation for working with polynomials that is used by all the trajectory generators;
- Section 4.1.4 defines what constitutes a valid trajectory;

[1] `/guidance/alure_trajectory`

- Section 4.1.5 describes the generation of a simple hover point;
- Section 4.1.6 describes mission trajectory generation;
- Section 4.1.7 describes transfer trajectory generation;
- Section 4.1.8 describes the spiral box search pattern generation.

4.1.3 Polynomial Building Blocks

Consider a generic temporal polynomial in $\mathbb{R}^n$ of order $N-1$:

$$\begin{aligned} \mathcal{P} : \mathbb{R} &\to \mathbb{R}^n \\ t &\mapsto \mathcal{P}(t) = \sum_{i=0}^{N-1} \boldsymbol{c}_i t^i, \end{aligned} \tag{4.1}$$

where $\boldsymbol{c}_{i=0,\dots,N-1} \in \mathbb{R}^n$ are the polynomial *coefficients*. Define the *coefficient vector* $\boldsymbol{c} := (\boldsymbol{c}_0, \boldsymbol{c}_1, \dots, \boldsymbol{c}_{N-1}) \in \mathbb{R}^{nN}$ which fully characterizes the polynomial. Trajectory generation algorithms construct polynomials by defining their coefficient vector. The j-th time derivative of the polynomial is given by:

$$\frac{\mathrm{d}^j \mathcal{P}(t)}{\mathrm{d}t^j} = \sum_{i=j}^{N-1} \frac{i!}{(i-j)!} \boldsymbol{c}_i t^{i-j} = \mathcal{D}_{\mathcal{P}}(t, j)\boldsymbol{c}, \tag{4.2}$$

where *polynomial derivative mapping function* $\mathcal{D}_{\mathcal{P}}$ is given by:

$$\begin{aligned} \mathcal{D}_{\mathcal{P}} : \mathbb{R} \times \mathbb{Z}_+ &\to \mathbb{R}^{n \times nN} \\ (t, j) &\mapsto \mathcal{D}_{\mathcal{P}}(t, j) = \begin{bmatrix} [\mathcal{D}_{\mathcal{P}}(t, j)]_0 & [\mathcal{D}_{\mathcal{P}}(t, j)]_1 & \cdots & [\mathcal{D}_{\mathcal{P}}(t, j)]_{N-1} \end{bmatrix}, \end{aligned} \tag{4.3}$$

where

$$[\mathcal{D}_{\mathcal{P}}(t, j)]_i := \begin{cases} \frac{i!}{(i-j)!} t^{i-j} I_n & i \geq j \\ 0_n & \text{otherwise} \end{cases}.$$

The $\mathcal{D}_{\mathcal{P}}$ function (4.3) is critical for constructing the start and end point constraints of the polynomial trajectory segments. In particular, a polynomial is generated by solving the following linear system called the *polynomial problem*[2]:

$$A_{\mathrm{c}} \boldsymbol{c} = \boldsymbol{d}_{\mathrm{c}}, \tag{4.4}$$

where $\boldsymbol{c} \in \mathbb{R}^{nN}$ is the above-defined coefficient vector while $A_{\mathrm{c}} \in \mathbb{R}^{nN \times nN}$ and $\boldsymbol{d}_{\mathrm{c}} \in \mathbb{R}^{nN}$ define the constraints on the polynomial derivatives. Note that A_{c} is square such that the polynomial is fully constrained – as aforementioned, no optimization takes place to construct the polynomials. Expanded, (4.4) looks like this:

$$\begin{bmatrix} \mathcal{D}_{\mathcal{P}}(t_1, j_1) \\ \mathcal{D}_{\mathcal{P}}(t_2, j_2) \\ \vdots \\ \mathcal{D}_{\mathcal{P}}(t_N, j_N) \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} \frac{\mathrm{d}^{j_1} \mathcal{P}(t_1)}{\mathrm{d}t^{j_1}} \\ \frac{\mathrm{d}^{j_2} \mathcal{P}(t_2)}{\mathrm{d}t^{j_2}} \\ \vdots \\ \frac{\mathrm{d}^{j_N} \mathcal{P}(t_N)}{\mathrm{d}t^{j_N}} \end{bmatrix}, \tag{4.5}$$

where it is noted that a polynomial of order $N-1$ supports at most N constraints on its derivatives. The values $t_{i=1,\cdots,N}$ and $j_{i=1,\cdots,N}$ specify the time and order of

[2] `/guidance/alure_trajectory/include/alure_trajectory/polynomial_problem.h`

the derivative constraint. The elements of $\boldsymbol{d}_\mathrm{c}$ fix the corresponding derivative value which must be satisfied by the polynomial.
Solving (4.4) is essentially a matrix inversion problem: $\boldsymbol{c} = A_\mathrm{c}^{-1}\boldsymbol{d}_\mathrm{c}$. However, A_c contains high powers of t (e.g. $[\mathcal{D}_\mathcal{P}(30,0)]_9 = 1.9683 \cdot 10^{13}$) which leads to poor numerical robustness. This implementation's trajectory generation, however, requires setting only *boundary* polynomial constraints for the segment start time $t = 0$ and end time $t = T$ (see Section 4.1.4). Therefore, (4.5) is specialized to the following form:

$$\begin{bmatrix} \mathcal{D}_\mathcal{P}(0,0) \\ \mathcal{D}_\mathcal{P}(0,1) \\ \vdots \\ \mathcal{D}_\mathcal{P}(0,N/2-1) \\ \mathcal{D}_\mathcal{P}(T,0) \\ \mathcal{D}_\mathcal{P}(T,1) \\ \vdots \\ \mathcal{D}_\mathcal{P}(T,N/2-1) \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} \mathcal{P}(0) \\ \frac{\mathrm{d}\mathcal{P}(0)}{\mathrm{d}t} \\ \vdots \\ \frac{\mathrm{d}^{N/2-1}\mathcal{P}(0)}{\mathrm{d}t^{N/2-1}} \\ \mathcal{P}(T) \\ \frac{\mathrm{d}\mathcal{P}(T)}{\mathrm{d}t} \\ \vdots \\ \frac{\mathrm{d}^{N/2-1}\mathcal{P}(T)}{\mathrm{d}t^{N/2-1}} \end{bmatrix}. \tag{4.6}$$

Using (4.3) in (4.6), it is noted that A_c then has a block lower triangular structure:

$$A_\mathrm{c} = \begin{bmatrix} \varSigma \in \mathbb{R}^{N/2\times N/2} & 0_{N/2} \\ \varGamma \in \mathbb{R}^{N/2\times N/2} & \varDelta^{N/2\times N/2} \end{bmatrix}, \tag{4.7}$$

which allows to invert A_c using its Schur complement Burri et al. [86]:

$$A_\mathrm{c}^{-1} = \begin{bmatrix} \varSigma^{-1} & 0_{N/2} \\ -\varDelta^{-1}\varGamma\varSigma^{-1} & \varDelta^{-1} \end{bmatrix}. \tag{4.8}$$

$\varSigma$ is triangular because it corresponds to the $t = 0$ derivatives whose non-zero elements are only the constant term of (4.2). Thus, $\varSigma$ is simple to invert. While $\varDelta$ contains high powers of t, it is of lower dimension and condition number than A_c thus it is easier to invert. The inverse given by (4.8) is therefore both more numerically stable and less computationally expensive to invert Burri et al. [86]. Algorithm 10 leverages (4.8) to create a standard way of building polynomials from provided A_c and $\boldsymbol{d}_\mathrm{c}$ matrices.

Algorithm 10 Coefficient vector computation for a fully constrained polynomial.
`/guidance/alure_trajectory/src/segment.cpp:createPolynomialFromConstraints`

function SOLVEPOLYNOMIAL(A_c, $\boldsymbol{d}_\mathrm{c}$)
  Let $A_\mathrm{c} \in \mathbb{R}^{M\times M}$ and $\boldsymbol{d}_\mathrm{c} \in \mathbb{R}^M$ ▷ $M = nN$
  assert M even
Decompose A_c and d_c:
  $\varSigma \leftarrow A_\mathrm{c}(1\!:\!M/2, 1\!:\!M/2)$ ▷ Diagonal matrix
  $\varGamma \leftarrow A_\mathrm{c}(M/2+1\!:\!M, 1\!:\!M/2)$
  $\varDelta \leftarrow A_\mathrm{c}(M/2+1\!:\!M, M/2+1\!:\!M)$
  $\boldsymbol{d}_{\mathrm{c},1} \leftarrow \boldsymbol{d}_\mathrm{c}(1\!:\!M/2)$
  $\boldsymbol{d}_{\mathrm{c},2} \leftarrow \boldsymbol{d}_\mathrm{c}(M/2+1\!:\!M)$
Use the Schur complement to invert A_c Burri et al. [86]:
  $\boldsymbol{c}_1 \leftarrow \varSigma^{-1}\boldsymbol{d}_{\mathrm{c},1}$
  $\boldsymbol{c}_2 \leftarrow \varDelta^{-1}(\boldsymbol{d}_{\mathrm{c},2} - \varGamma\boldsymbol{c}_1)$
  $\boldsymbol{c} \leftarrow (\boldsymbol{c}_1, \boldsymbol{c}_2)$
  return $\boldsymbol{c}$
end function

4.1.4 Trajectory Definition

Section 4.1.3 introduced a generic entity, the polynomial $\mathcal{P}$. A *segment*[3,4] $\mathcal{S}$ is now defined as an entity which consists of a function $\mathcal{S}_\mathcal{P}$ and a duration $\mathcal{S}_\mathrm{T}$. The segment's geometry is defined by $\mathcal{S}_\mathcal{P}$:

$$\begin{aligned} \mathcal{S}_\mathcal{P} : \mathbb{R} &\to \mathbb{R}^n \\ t &\mapsto \mathcal{S}_\mathcal{P}(t), \end{aligned}$$

where

$$\mathcal{S}_\mathcal{P}(t) = \begin{cases} \mathcal{P}(0) & \text{if } t < 0 \\ \mathcal{P}(t) & \text{if } t \in [0, \mathcal{S}_\mathrm{T}] \\ \mathcal{P}(\mathcal{S}_\mathrm{T}) & \text{otherwise,} \end{cases} \tag{4.9}$$

where $\mathcal{P}$ is the polynomial which is associated with the segment and which defines the segment's geometry. $\frac{\mathrm{d}^j \mathcal{S}_\mathcal{P}(t)}{\mathrm{d}t^j}$ is given by substituting (4.2) for $\mathcal{P}$ in (4.9). In the following algorithms, the notation $\mathcal{S} = \{T, \boldsymbol{c}\}$ shall *construct* a segment with duration T and a polynomial defined by the coefficient vector $\boldsymbol{c}$.
A *trajectory*[5] σ is a tuple of segments $\bar{\mathcal{S}} := \langle \mathcal{S}_1, \mathcal{S}_2, \ldots \rangle$. Let $\bar{\mathcal{S}}[i] = \mathcal{S}_i$. In the following algorithms, the notation $\sigma = \bar{\mathcal{S}}$ shall *construct* a trajectory with the segment tuple $\bar{\mathcal{S}}$. σ can be evaluated at a given time t via the function:

$$\begin{aligned} \sigma : \mathbb{R} &\to \mathbb{R}^n \\ t &\mapsto \sigma(t) = \textsc{evaluateTrajectory}(\sigma, t, 0), \end{aligned} \tag{4.10}$$

where Algorithm 11 defines the EVALUATETRAJECTORY() function. Note that $\frac{\mathrm{d}^j \sigma(t)}{\mathrm{d}t^j}$ is readily computed by replacing the 0 in (4.10) with j.

Algorithm 11 Trajectory evaluation function.
`/guidance/alure_trajectory/src/trajectory_base.cpp:evaluate.`

```
1:  function EVALUATETRAJECTORY(σ, τ, j)
2:      Let S̄ be the segment tuple associated with σ
    Select the segment corresponding to time τ:
3:      i ← 0
4:      while τ > (S̄[i])_T do
5:          if i = |S̄| then                          ▷ τ ≥ trajectory duration
6:              τ ← (S̄[i])_T                         ▷ Saturate at trajectory end
7:              break
8:          end if
9:          τ ← τ − (S̄[i])_T                         ▷ Move on to the next segment
10:         i ← i + 1
11:     end while
    Evaluate the selected segment:
12:     return d^j(S̄[i])_P(τ) / dt^j
13: end function
```

To generate a trajectory, one simply needs to populate $\bar{\mathcal{S}}$. The subsequent sections describe several trajectory generators that do this for specific trajectory types that are used by the autonomy engine (Chapter 6).

[3] `/guidance/alure_trajectory/include/alure_trajectory/segment_base.h`
[4] `/guidance/alure_trajectory/include/alure_trajectory/segment.h`
[5] `/guidance/alure_trajectory/include/alure_trajectory/trajectory_base.h`

4.1.5 Hover Point Trajectory

A *hover point* is defined as a hover location $\boldsymbol{p}_{\text{hover}} \in \mathbb{R}^3$ in space and a duration T_{hover} for which to stay at that location. A hover point can be considered as a trajectory which, according to Algorithm 11, will always evaluate to the same value, i.e. $\boldsymbol{p}_{\text{hover}}$. Algorithm 12 provides the trivial hover point trajectory generator[6].

Algorithm 12 Hover point trajectory generator.
`/guidance/alure_trajectory/src/hover_point.cpp:build`

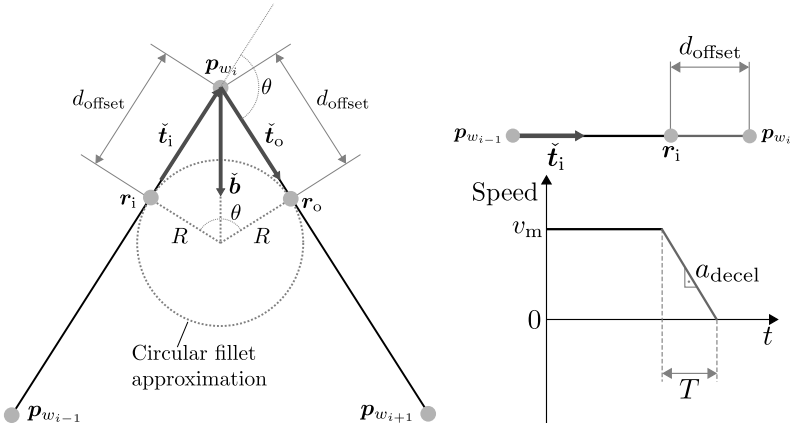
1: **function** CREATEHOVERPOINT($\boldsymbol{p}_{\text{hover}}$, T_{hover})
2: $\quad\boldsymbol{c} \leftarrow \boldsymbol{p}_{\text{hover}}$ ▷ Polynomial coefficient vector with only the constant term
3: $\quad\mathcal{S} \leftarrow \{T_{\text{hover}}, \boldsymbol{c}\}$
4: $\quad\bar{\mathcal{S}} \leftarrow \langle \mathcal{S} \rangle$
5: $\quad\sigma \leftarrow \bar{\mathcal{S}}$
6: **end function**

4.1.6 Waypoint Trajectory

The waypoint trajectory[7] defines the data acquisition path which the quadrotor flies between takeoff and landing. Its design is based on S2 of Specification List 2. Let $\mathcal{W}$ be the tuple of user-defined waypoints[8] with $|\mathcal{W}| > 0$ (at least one waypoint must be defined). Let $w_i \in \mathcal{W}$ denote the i-th waypoint, which has two properties: its position in the world frame, $\boldsymbol{p}_{w_i}$, and the duration for which the quadrotor is to hover at this waypoint, t_{w_i}.
The user also defines the *mission speed* v_{m} at which the inter-waypoint constant speed segments are to be flown and the *cornering radius* R. Corners formed by the constant velocity segments at each waypoint are filleted with an order 9 polynomial approximating a constant velocity circular fillet whose ideal cornering acceleration is $\approx v_{\text{m}}^2/R$. This *heuristic* allows the user to roughly control the maximum acceleration via R. As shown in Figure 4.2b, the achieved trajectory acceleration is larger. This is explained by the fact that acceleration is gradual at the start/end of the fillet (it is integrated up from snap), meaning that to achieve the same velocity change for a given time period the middle part of a fillet must have higher accelerations. A good rule of thumb is that the achieved maximum trajectory acceleration is $\approx 2v_{\text{m}}^2/R$. More advanced trajectory generation methods would be needed (e.g. iterating on the trajectory duration via the bisection method) in order to perfectly satisfy the maximum acceleration bound. For our purposes, this complexity was unnecessary.
The trajectory's segment tuple $\bar{\mathcal{S}}$ is populated by Algorithm 13. As shown in Figure 4.1a, when two segments meet at a corner (whose vertex is the waypoint w_i), a fillet is computed from boundary conditions at the incoming and outgoing vertices $\boldsymbol{r}_{\text{i}}$ and $\boldsymbol{r}_{\text{o}}$ that are determined by a circular fillet geometry. While a constant-velocity circle is strongly dynamically infeasible (it produces a step in acceleration from 0 during the constant velocity segment to v_{m}^2/R during the fillet), an order 9 polynomial fillet creates a snap-continuous transfer (i.e. dynamically feasible, with a caveat from Appendix E). A circular fillet is computed if and only if:

- The user did not require hovering at the waypoint ($t_{w_i} = 0$);
- The corner is not too tight (i.e. corner angle $> 20°$). Otherwise a circular fillet would typically initiate a turn far away from $\boldsymbol{p}_{w_i}$. Assuming that the user wants to fly reasonably close to each waypoint, this is undesirable.

[6] `/guidance/alure_trajectory/include/alure_trajectory/hover_point.h`
[7] `/guidance/alure_trajectory/include/alure_trajectory/waypoint_trajectory.h`
[8] `/autonomy_engine/alure_mission/config/mission_parameters.yaml`



(a) Circular-like fillet. (b) Deceleration-in fillet.

Figure 4.1: Visualization of fillet computation in Algorithm 13.

When these conditions are not passed (or it is the first or the last waypoint), a deceleration-in-acceleration-out fillet is computed instead. Because acceleration-out is the exact dual of deceleration-in, only the latter shall be explained. As shown in Figure 4.1b, the objective is for the quadrotor to come to a complete stop at $\boldsymbol{p}_{w_i}$. A constant deceleration transfer is assumed, for which the deceleration is computed to be $a_{\text{decel}} = v_{\text{m}}^2/R$. This conveniently lets the user continue to use R to control quadrotor acceleration during both these and the circular fillets. This leads to $T = v_{\text{m}}/a_{\text{decel}}$. The d_{offset} distance is then computed such that the quadrotor arrives at $\boldsymbol{p}_{w_i}$ with zero velocity. Therefore:

$$d_{\text{offset}} = v_{\text{m}}T - \frac{1}{2}a_{\text{decel}}T^2 \Rightarrow d_{\text{offset}} = \frac{v_{\text{m}}T}{2}.$$

Figure 4.2a shows a typical example of a generated waypoint trajectory, with the quadrotor body frame sampled along its length. Figure 4.2b shows the corresponding velocity and acceleration profile.

4.1.7 Transfer Trajectory

A *transfer trajectory*[9] is a single polynomial segment which smoothly transfers the first $N/2$ derivatives of a polynomial from a start to an end value in T seconds. Let us denote the start and end conditions as:

$$\boldsymbol{d}_0 \leftarrow \left(\mathcal{P}(0), \frac{\mathrm{d}^1\mathcal{P}(0)}{\mathrm{d}t^1}, \frac{\mathrm{d}^2\mathcal{P}(0)}{\mathrm{d}t^2}, \ldots, \frac{\mathrm{d}^{N/2-1}\mathcal{P}(0)}{\mathrm{d}t^{N/2-1}}\right) \in \mathbb{R}^{nN/2}, \tag{4.11}$$

$$\boldsymbol{d}_T \leftarrow \left(\mathcal{P}(T), \frac{\mathrm{d}^1\mathcal{P}(T)}{\mathrm{d}t^1}, \frac{\mathrm{d}^2\mathcal{P}(T)}{\mathrm{d}t^2}, \ldots, \frac{\mathrm{d}^{N/2-1}\mathcal{P}(T)}{\mathrm{d}t^{N/2-1}}\right) \in \mathbb{R}^{nN/2}, \tag{4.12}$$

where $n = 3$ since yaw is not planned. Together, (4.11) and (4.12) fully constrain a polynomial of order $N - 1$. Algorithm 14 solves for this polynomial and creates a trajectory out of it. Three types of transfer trajectories are used in the current implementation:

- A position transfer trajectory with $N = 10$ which is used by the landing autopilot for the final descent (Section 6.4);

[9] `/guidance/alure_trajectory/include/alure_trajectory/transfer_trajectory.h`

Algorithm 13 Waypoint trajectory generation.
`/guidance/alure_trajectory/src/waypoint_trajectory.cpp:build`

function createWaypointTrajectory($\mathcal{W}$, v_m, R)
  $M \leftarrow |\mathcal{W}|$ ▷ Total number of waypoints
  $\bar{\mathcal{S}} \leftarrow \varnothing$
Edge case of only one waypoint (simple hover):
  if $M = 1$ **then**
    if $t_{w_1} \neq 0$ **then**
      $T \leftarrow t_{w_1}$
    else
      $T \leftarrow \infty$ ▷ `std::numeric_limits<double>::max()` in C++
    end if
    $\bar{\mathcal{S}} \leftarrow$ addHoverSegment($\bar{\mathcal{S}}, \boldsymbol{p}_{w_1}, T$)
    $\sigma \leftarrow \bar{\mathcal{S}}$
    return σ
  end if
Nominal case of > 1 waypoints:
  for $i \leftarrow 1$ to M **do**
    if $i > 1$ **then** ▷ Compute incoming unit vector
      $\check{\boldsymbol{t}}_\mathrm{i} \leftarrow \frac{\boldsymbol{p}_{w_i} - \boldsymbol{p}_{w_{i-1}}}{\|\boldsymbol{p}_{w_i} - \boldsymbol{p}_{w_{i-1}}\|}$
    end if
    if $i < M$ **then** ▷ Compute outgoing unit vector
      $\check{\boldsymbol{t}}_\mathrm{o} \leftarrow \frac{\boldsymbol{p}_{w_{i+1}} - \boldsymbol{p}_{w_i}}{\|\boldsymbol{p}_{w_{i+1}} - \boldsymbol{p}_{w_i}\|}$
    end if
    circular $\leftarrow$ **false**
    intermediate_wp $\leftarrow$ ($i > 1$ **and** $i < M$)
    hover $\leftarrow$ ($t_{w_i} \neq 0$)
    if intermediate_wp **then**
      $\theta \leftarrow \arccos(\check{\boldsymbol{t}}_\mathrm{i} \cdot \check{\boldsymbol{t}}_\mathrm{o})$ ▷ Inbound and outbound segments' exterior angle
      circular $\leftarrow$ (**not** hover) **and** $\theta \leq 8\pi/9$)
    end if
    decel_in $\leftarrow$ $i = M$ **or** (intermediate_wp **and** (**not** circular))
    accel_out $\leftarrow$ $i = 1$ **or** (intermediate_wp **and** (**not** circular))
    if decel_in **or** accel_out **then**
      $a_\mathrm{decel} \leftarrow v_\mathrm{m}^2 / R$
      $T \leftarrow v_\mathrm{m} / a_\mathrm{decel}$
      $d_\mathrm{offset} \leftarrow v_\mathrm{m} T / 2$
    end if
Deceleration–in fillet at the last or to a hover waypoint:
    if decel_in **then**
      $\boldsymbol{r}_\mathrm{i} \leftarrow \boldsymbol{p}_{w_i} - d_\mathrm{offset} \check{\boldsymbol{t}}_\mathrm{i}$
      $\bar{\mathcal{S}} \leftarrow$addConstantVelocitySegment($\bar{\mathcal{S}}$, $\boldsymbol{r}_\mathrm{o}$, $\boldsymbol{r}_\mathrm{i}$, $\check{\boldsymbol{t}}_\mathrm{i}$)
      $\bar{\mathcal{S}} \leftarrow$addSegmentFromConstraints($\bar{\mathcal{S}}$, $\boldsymbol{r}_\mathrm{i}$, $v_\mathrm{m}\check{\boldsymbol{t}}_\mathrm{i}$, $\boldsymbol{p}_{w_i}$, $0_{3\times 1}$, T)
      if hover **then**
        $\bar{\mathcal{S}} \leftarrow$ addHoverSegment($\bar{\mathcal{S}}, \boldsymbol{p}_{w_i}, t_{w_i}$)
      end if
    end if
    (*Continued on next page*)

(*Started on previous page*)
Acceleration–out fillet at the first or from a hover waypoint:
if accel_out **then**
if hover **and** $i = 1$ **then** ▷ If $i \neq 1$, the hovering segment was already added by the deceleration–in fillet above
$\bar{\mathcal{S}} \leftarrow$ addHoverSegment$(\bar{\mathcal{S}}, \boldsymbol{p}_{w_i}, t_{w_i})$
end if
$\boldsymbol{r}_\mathrm{o} \leftarrow \boldsymbol{p}_{w_i} + d_\mathrm{offset}\check{\boldsymbol{t}}_\mathrm{o}$
$\bar{\mathcal{S}} \leftarrow$addSegmentFromConstraints$(\bar{\mathcal{S}}, \boldsymbol{p}_{w_i}, 0_{3\times1}, \boldsymbol{r}_\mathrm{o}, v_\mathrm{m}\check{\boldsymbol{t}}_\mathrm{o}, T)$
end if
Circular fillet:
if circular **then**
$\check{\boldsymbol{b}} \leftarrow \frac{\check{\boldsymbol{t}}_\mathrm{o} - \check{\boldsymbol{t}}_\mathrm{i}}{\|\check{\boldsymbol{t}}_\mathrm{o} - \check{\boldsymbol{t}}_\mathrm{i}\|}$
$T \leftarrow \frac{\theta R}{v_\mathrm{m}}$
$d_\mathrm{offset} \leftarrow \frac{R}{\|\check{\boldsymbol{b}}/(\check{\boldsymbol{b}}\cdot\check{\boldsymbol{t}}_\mathrm{o}) - \check{\boldsymbol{t}}_\mathrm{o}\|}$
$\boldsymbol{r}_\mathrm{i} \leftarrow \boldsymbol{p}_{w_i} - d_\mathrm{offset}\check{\boldsymbol{t}}_\mathrm{i}$
$\bar{\mathcal{S}} \leftarrow$addConstantVelocitySegment$(\bar{\mathcal{S}}, \boldsymbol{r}_\mathrm{o}, \boldsymbol{r}_\mathrm{i}, \check{\boldsymbol{t}}_\mathrm{i})$
$\boldsymbol{r}_\mathrm{o} \leftarrow \boldsymbol{p}_{w_i} + d_\mathrm{offset}\check{\boldsymbol{t}}_\mathrm{o}$
$\bar{\mathcal{S}} \leftarrow$addSegmentFromConstraints$(\bar{\mathcal{S}}, \boldsymbol{r}_\mathrm{i}, v_\mathrm{m}\check{\boldsymbol{t}}_\mathrm{i}, \boldsymbol{r}_\mathrm{o}, v_\mathrm{m}\check{\boldsymbol{t}}_\mathrm{o}, T)$
end if
end for
$\sigma \leftarrow \bar{\mathcal{S}}$
return σ
end function

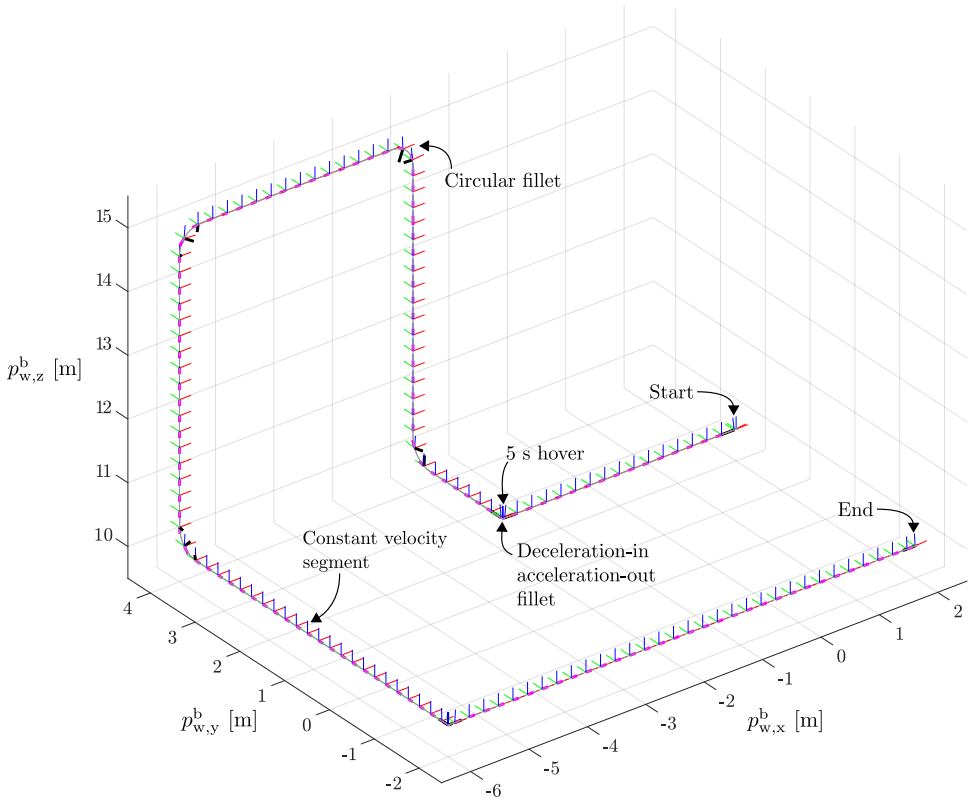
function addConstantVelocitySegment$(\bar{\mathcal{S}}, \boldsymbol{p}_\mathrm{i}, \boldsymbol{p}_\mathrm{f}, \check{\boldsymbol{t}})$
$\boldsymbol{u} \leftarrow \boldsymbol{p}_\mathrm{f} - \boldsymbol{p}_\mathrm{i}$
assert $\boldsymbol{u} \cdot \check{\boldsymbol{t}} \geq 0$ ▷ Checks that fillets are not too large for the segment between the original waypoints
$T \leftarrow \frac{\|\boldsymbol{u}\|}{v_\mathrm{m}}$ ▷ Time for a constant velocity transfer
if $T > 0$ **then** ▷ Skip if $T = 0$ (i.e. $\boldsymbol{u} = 0$)
$\bar{\mathcal{S}} \leftarrow$addSegmentFromConstraints$(\bar{\mathcal{S}}, \boldsymbol{p}_\mathrm{i}, v_\mathrm{m}\check{\boldsymbol{t}}, \boldsymbol{p}_\mathrm{f}, v_\mathrm{m}\check{\boldsymbol{t}}, T)$
end if
return $\bar{\mathcal{S}}$
end function

function addSegmentFromConstraints$(\boldsymbol{p}_\mathrm{i}, \boldsymbol{v}_\mathrm{i}, \boldsymbol{p}_\mathrm{f}, \boldsymbol{v}_\mathrm{f}, T)$
Solve the polynomial problem:

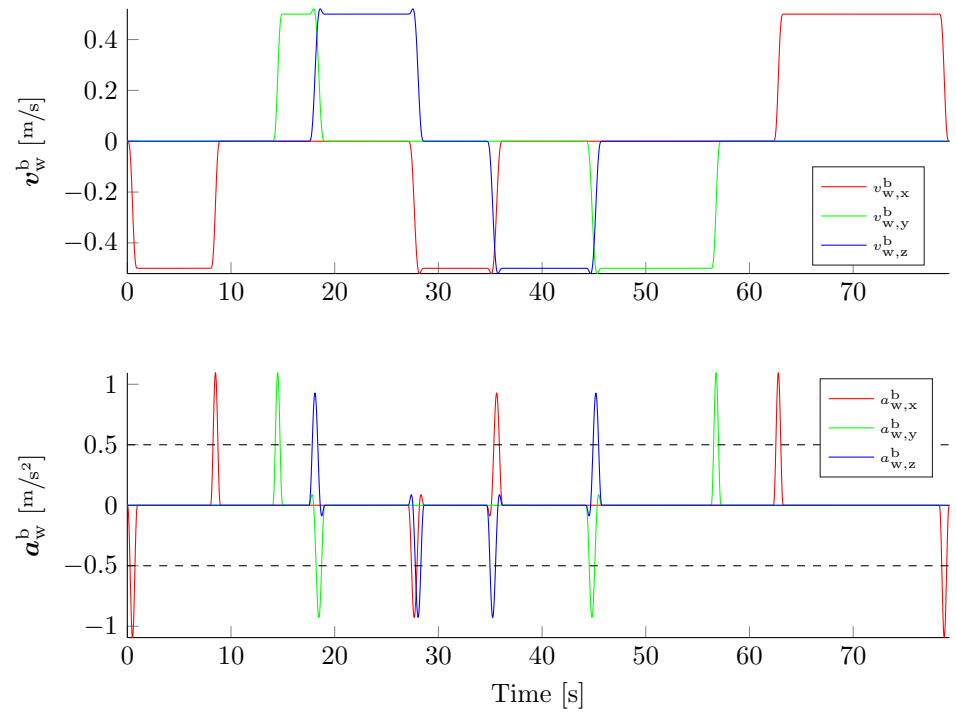
$$A_\mathrm{c} \leftarrow (\mathcal{D}_\mathcal{P}(0,0), \mathcal{D}_\mathcal{P}(0,1), \mathcal{D}_\mathcal{P}(0,2), \mathcal{D}_\mathcal{P}(0,3), \mathcal{D}_\mathcal{P}(0,4), \mathcal{D}_\mathcal{P}(T,0), \mathcal{D}_\mathcal{P}(T,1), \mathcal{D}_\mathcal{P}(T,2), \mathcal{D}_\mathcal{P}(T,3), \mathcal{D}_\mathcal{P}(T,4)) \in \mathbb{R}^{30\times30}$$

$\boldsymbol{d}_\mathrm{c} \leftarrow (\boldsymbol{p}_\mathrm{i}, \boldsymbol{v}_\mathrm{i}, 0_{9\times1}, \boldsymbol{p}_\mathrm{f}, \boldsymbol{v}_\mathrm{f}, 0_{9\times1})$
$\boldsymbol{c} \leftarrow$ solvePolynomial$(A_\mathrm{c}, \boldsymbol{d}_\mathrm{c})$ ▷ Apply Algorithm 10
$\mathcal{S} \leftarrow \{T, \boldsymbol{c}\}$
$\bar{\mathcal{S}} \leftarrow \langle \bar{\mathcal{S}}, \mathcal{S} \rangle$
return $\bar{\mathcal{S}}$
end function

function addHoverSegment$(\bar{\mathcal{S}}, \boldsymbol{p}, T)$ ▷ Proceed like Algorithm 12
$\mathcal{S} \leftarrow \{T, \boldsymbol{p}\}$
$\bar{\mathcal{S}} \leftarrow \langle \bar{\mathcal{S}}, \mathcal{S} \rangle$
return $\bar{\mathcal{S}}$
end function



(a) Generated waypoint trajectory. Corresponding quadrotor {b} frame, magenta $\boldsymbol{v}^{b}_{w}$ and black $\boldsymbol{a}^{b}_{w}$ are shown at 2 second time intervals. Body frame attitude is obtained via the flat output to state map (Appendix E).



(b) $\boldsymbol{v}^{b}_{w}$ and $\boldsymbol{a}^{b}_{w}$ corresponding to (a). The heuristic absolute acceleration bound v^{2}_{m}/R is shown by the dotted lines (in its most relaxed form, assuming it applies to each axis individually). Note that the trajectory acceleration surpasses v^{2}_{m}/R by about twice. This holds for many such generated trajectories.

Figure 4.2: Waypoint trajectory example.

- A velocity transfer trajectory with $N = 8$ which is used by the takeoff autopilot for the acceleration to/deceleration from takeoff velocity (Section 6.2);
- A "simple" position transfer trajectory with $N = 4$ which is used by the trajectory tracker to generate realignment trajectories (Section 4.3). Flight testing showed that satisfaction of only up to the velocity derivative yields better behaved (more predictable) trajectories than if it is required to also satisfy acceleration, jerk and snap continuity with the following segment. This creates a dynamic infeasibility at the polynomial segments' interface but is unnoticeable for the slow speeds ($< 2\,\mathrm{m/s}$) at which the quadrotor flies the data acquisition mission.

Algorithm 14 Transfer trajectory generation for the first $N/2$ derivatives.
`/guidance/alure_trajectory/src/transfer_trajectory.cpp:build`

function createTransferTrajectory($\boldsymbol{d}_0$, $\boldsymbol{d}_T$, T)
  assert $\boldsymbol{d}_0, \boldsymbol{d}_T \in \mathbb{R}^{3N/2}$
  Solve the polynomial problem:

$$A_\mathrm{c} \leftarrow (\mathcal{D}_\mathcal{P}(0,0), \mathcal{D}_\mathcal{P}(0,1), \mathcal{D}_\mathcal{P}(0,2), \ldots, \mathcal{D}_\mathcal{P}(0,N/2-1), \mathcal{D}_\mathcal{P}(T,0), \mathcal{D}_\mathcal{P}(T,1), \mathcal{D}_\mathcal{P}(T,2), \ldots, \mathcal{D}_\mathcal{P}(T,N/2-1)) \in \mathbb{R}^{3N\times 3N}$$

  $\boldsymbol{d}_\mathrm{c} \leftarrow (\boldsymbol{d}_0, \boldsymbol{d}_T) \in \mathbb{R}^{3N}$
  $\boldsymbol{c} \leftarrow$ solvePolynomial(A_c, $\boldsymbol{d}_\mathrm{c}$) ▷ Apply Algorithm 10
  $\mathcal{S} \leftarrow \{T, \boldsymbol{c}\}$
  $\bar{\mathcal{S}} \leftarrow \langle \mathcal{S} \rangle$
  $\sigma \leftarrow \bar{\mathcal{S}}$
  return σ
end function

4.1.8 Spiral Grid Search Trajectory

The spiral grid[10], also known as a *spiral box search*, is a documented underwater search pattern Wikipedia [101]. The objective of an underwater search is to find a known target object in a specific search area under water. This matches our use case: if the quadrotor returns to home and the downfacing camera does not see the landing pad, the quadrotor must find the pad with the assumption that it is somewhere close by. This would occur if e.g. the GPS drift is so large that the return to home location is vastly different from the original takeoff location.
Figure 4.3 visualizes the spiral grid. Its geometry is defined by the following parameters:

- The camera image physical width on the ground, $w_\mathrm{im,ground}$;
- The camera image physical height on the ground, $h_\mathrm{im,ground}$;
- Image advance fraction for adjacent spiral grid segments, $p_\mathrm{search} \in (0, 1]$ (e.g. $p_\mathrm{search} = 0.75$ denotes 25% image overlap in width and height on either side);
- The spiral height above the ground, h_spiral, corresponding to $p^\mathrm{s}_{\mathrm{w,z}}$;
- The only-yaw spiral attitude $\boldsymbol{q}^{\mathrm{s,yaw}}_\mathrm{w}$ with respect to the world frame.

[10] `/guidance/alure_trajectory/include/alure_trajectory/spiral_grid.h`

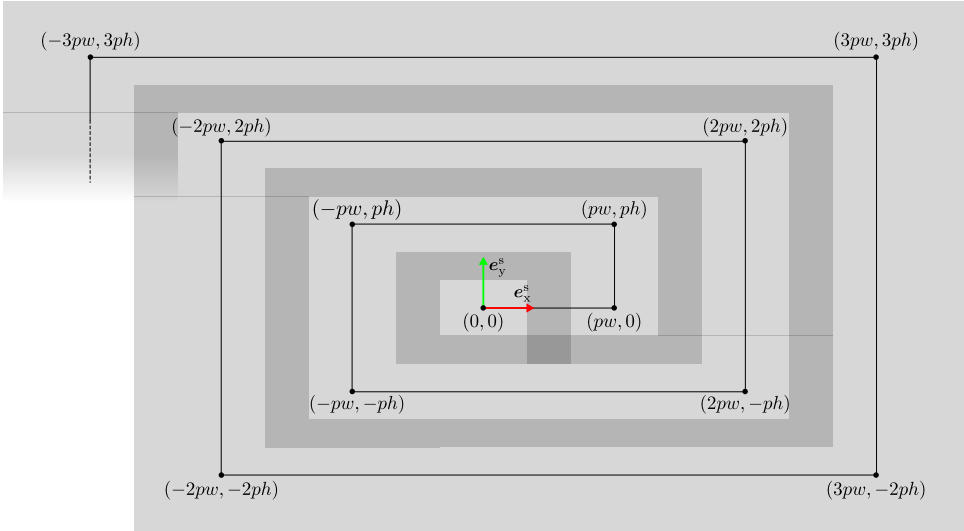


Figure 4.3: Spiral grid search trajectory. Let $p \equiv p_{\text{search}}$, $w \equiv w_{\text{im,ground}}$ and $h \equiv h_{\text{im,ground}}$. Gray regions represent the ground area covered by the downfacing camera image. Darker gray areas represent image overlap between adjacent paths (here shown for $p = 0.75$).

As shown in Figure 4.3, the spiral grid consists of straight segments describing a rectangular spiral. Each segment is generated via Algorithm 13 in the {s} frame, which makes dealing with image overlap trivial as yaw does not yet have to be accounted for. The overlap increases the likelihood of the ground being fully observed despite disturbances and imperfect tracking by the quadrotor.
The spiral grid trajectory generator is then in fact only responsible for generating the actual spiral vertices (black dots in Figure 4.3). This is done by recognizing that only the coefficients of $p_{\text{search}} w_{\text{im,ground}}$ and $p_{\text{seach}} h_{\text{im,ground}}$ need to be generated. In particular, these coefficients are (starting from the spiral center):

$$\begin{array}{c|cccccccccccc} \text{x} & 0 & 1 & 1 & -1 & -1 & 2 & 2 & -2 & -2 & 3 & 3 & -3 & \cdots \\ \text{y} & 0 & 0 & 1 & 1 & -1 & -1 & 2 & 2 & -2 & -2 & 3 & 3 & \cdots \end{array}$$

Noting that the y-coefficients are merely offset by one from the x-coefficients, the same pattern can be used to generate both. This pattern is developed in Table 4.1. Letting the j-th vertex ($j = 1, 2, 3, \ldots$) have associated with it the index $i = j + 2$, the vertex coordinates are given by:

$$\boldsymbol{w} = \begin{bmatrix} -\lfloor \frac{i}{4} \rfloor \text{sgn}(\lfloor \frac{i}{2} \rfloor \% 2 - 0.5) p_{\text{search}} w_{\text{im,ground}} \\ -\lfloor \frac{i-1}{4} \rfloor \text{sgn}(\lfloor \frac{i-1}{2} \rfloor \% 2 - 0.5) p_{\text{search}} h_{\text{im,ground}} \\ h_{\text{spiral}} \end{bmatrix}, \tag{4.13}$$

where h_{spiral} is the constant spiral height. The spiral grid search trajectory generator is given by Algorithm 15. The generator works by keeping in memory which segment it has last generated. When it is invoked, it generates the next segment. This way, the autonomy engine can keep extending the spiral by invoking GENERATENEXTSEGMENT() for as long as the landing pad has not been found (see Section 6.4). The spiral grid can also be reset back to start via a RESET() method which simply performs the initialization ($i \leftarrow 3$) of Algorithm 15.

x	0	1	1	−1	−1	2	2	−2	−2	3	3	−3	⋯
i	3	4	5	6	7	8	9	10	11	12	13	14	⋯
$\lfloor \frac{i}{4} \rfloor$	0	1	1	1	1	2	2	2	2	3	3	3	⋯
$\lfloor \frac{i}{2} \rfloor$	1	2	2	3	3	4	4	5	5	6	6	7	⋯
$\lfloor \frac{i}{2} \rfloor \% 2$	1	0	0	1	1	0	0	1	1	0	0	1	⋯
$-\text{sgn}(\lfloor \frac{i}{2} \rfloor \% 2 - 0.5)$	−1	1	1	−1	−1	1	1	−1	−1	1	1	−1	⋯
$-\lfloor \frac{i}{4} \rfloor \text{sgn}(\lfloor \frac{i}{2} \rfloor \% 2 - 0.5)$	0	1	1	−1	−1	2	2	−2	−2	3	3	−3	⋯

Table 4.1: Spiral vertex x-coefficient generator development. The first row shows the pattern to match. The following gray rows show the individual components that go into the overall pattern generator in the last row.

Algorithm 15 Spiral grid search trajectory generator. Generates the next spiral segment.

`/guidance/alure_trajectory/src/spiral_grid.cpp:generateNextSegment`

Initialization:

$i \leftarrow 3$

function GENERATENEXTSEGMENT()
  $\mathcal{W} \leftarrow \varnothing$
  $\mathcal{W} \leftarrow \langle \mathcal{W}, \text{GENERATESPIRALWAYPOINT}(i) \rangle$ ▷ Segment start vertex
  $i \leftarrow i + 1$
  $\mathcal{W} \leftarrow \langle \mathcal{W}, \text{GENERATESPIRALWAYPOINT}(i) \rangle$ ▷ Segment end vertex
  $a_{\text{ease,in,out}} \leftarrow 0.5$ ▷ Desired acceleration and deceleration at segment ends
  $R \leftarrow v_{\text{spiral}}^2 / a_{\text{ease,in,out}}$
  $\sigma \leftarrow \text{CREATEWAYPOINTTRAJECTORY}(\mathcal{W}, v_{\text{spiral}}, R)$ ▷ Generates segment
  return σ
end function

function GENERATESPIRALWAYPOINT(i)
  $\boldsymbol{w} \leftarrow$ evaluate (4.13)
  $\boldsymbol{w} \leftarrow C_{(\boldsymbol{q}_{\text{w}}^{\text{s}})} \boldsymbol{w}$ ▷ Transform waypoint into the world frame
  return $\boldsymbol{w}$
end function

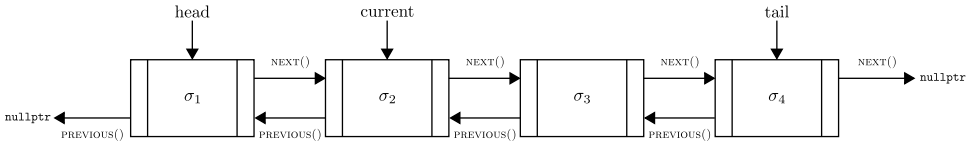


Figure 4.4: Trajectory sequencer implementation as a doubly-linked list of trajectory elements.

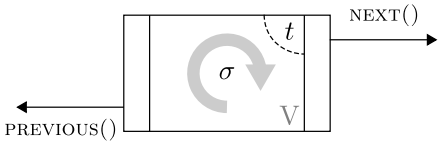


Figure 4.5: Trajectory element building block.

4.2 Trajectory Sequencer

The previous discussion allows the generation of a single trajectory σ which can be sampled in time via Algorithm 11. In practice, any mission would consist of multiple such trajectories *sequenced* or *chained* to be executed one after another (with, presumably but not necessarily, each trajectory beginning where the previous trajectory left off).
For this purpose, define a *trajectory list* or *sequencer*[11] $\Gamma_{\text{list}} = \langle \gamma_1, \gamma_2, \ldots \rangle$ where γ_i is a *trajectory list element*[12] which consists of an associated trajectory, σ_i, and several sequencing add-ons which shall be introduced throughout this section. For example, γ_1 may be an "align with landing pad" trajectory element, γ_2 may be a "descend toward landing pad" trajectory element, etc.
As shown in Figure 4.4, Γ_{list}'s implementation is based on an open doubly-linked list data structure which, as shall be explained below, provides an extensible, predictable, real time and thread safe way of evaluating $\Gamma_{\text{list}}(t)$ and of adding/removing/modifying trajectories.

4.2.1 Trajectory Element Mechanics

A trajectory element γ is the basic building block of a trajectory list. As partially shown in Figure 4.5, it has the following properties:

- The trajectory itself, σ;
- A time, t, storing the current time along σ and a time step Δt by which t is incremented when the element is evaluated;
- A playback type which may be either SINGLE, VOLATILE (indicated by the V) or CYCLIC (indicated by the ↻)[13];
- A pointer to the next trajectory element and returned by NEXT();
- A pointer to the previous trajectory element and returned by PREVIOUS();
- A total trajectory duration $\sum_{\mathcal{S} \in \bar{\mathcal{S}}} \mathcal{S}_{\text{T}}$, returned by DURATION();

[11] `/guidance/alure_trajectory/include/alure_trajectory/trajectory_list.h`
[12] `/guidance/alure_trajectory/include/alure_trajectory/trajectory_list_element.h`
[13] The CYCLIC playback type is useful for automatically repeating a trajectory. The special mechanics of the VOLATILE type shall be explained in Section 4.2.2 while its *raison d'être* will become apparent in Section 4.3.

- A FINISHED() method which returns a boolean $t \geq$ DURATION()[14];
- A SWAPTRAJECTORY(σ_{new}) method which replaces σ with σ_{new} while not changing t[15].

Let $\gamma = \{\sigma, \Delta t, \text{playback type}\}$ denote the construction of a trajectory element. A trajectory element operates in isolation and does only one thing – return a trajectory value when the trajectory list (Section 4.2.2) calls on it. The internal mechanics of this evaluation are defined in Algorithm 16 which allows to obtain up to the $j_{\max}$-th derivative of σ. Thanks to this algorithm, the user of the trajectory element need not worry about time – the only requirement is to evaluate the trajectory element at Δt^{-1} Hz in order for the trajectory derivatives to evolve in real (i.e. wall clock) time.

Algorithm 16 Trajectory element evaluation algorithm, which also executes internally the time propagation mechanics of the element.
`/guidance/alure_trajectory/src/trajectory_list_element.cpp:evaluate`

```
1: function EVALUATETRAJECTORYELEMENT(j_max)
2:     assert j_max ≥ 0
3:     derivatives ← ∅
4:     for j ← 0 to j_max do
5:         derivatives ← ⟨derivatives, EVALUATETRAJECTORY(σ, t, j)⟩ ▷ Algorithm 11
6:     end for
7:     t ← t + Δt
8:     if FINISHED() then
9:         if playback type SINGLE or VOLATILE then
10:            t ← DURATION()                          ▷ Saturate to total duration
11:        else
12:            t ← 0                                   ▷ Rewind
13:        end if
14:    end if
15:    return derivatives
16: end function
```

4.2.2 Trajectory List Mechanics

The trajectory list assembles multiple trajectory elements into an open doubly-linked list data structure[16] as shown in Figure 4.4. The data structure and all of its methods are implemented in thread safe fashion. Note that the head, current and tail elements in Figure 4.4 are `std::shared_ptr`s to the trajectory list elements. With the concept of pointers absent in mathematical notation, elements are "assigned" to head, current and tail in the pseudocode presented herein. In reality, these are pointer assignments.
Basic linked list operations are provided for modifying Γ_{list}[17]:

- PUSH_BACK(), PUSH_FRONT() and INSERT() for adding trajectory elements to Γ_{list}. Algorithm 17 explains the INSERT() method which is special in that

[14] Following from Algorithm 16, a CYCLIC trajectory *never* evaluates to FINISHED().
[15] The swapping method is useful for trajectories that are merely shifted in space, such as the landing descent trajectory described in Section 6.4.
[16] `/guidance/alure_trajectory/include/alure_trajectory/trajectory_list.h`
[17] PUSH_BACK and POP_BACK work in the same way as their `std::list` counterparts while INSERT inserts an element *after* the current one.

it extends the typical doubly-linked list with the VOLATILE playback type element. Note a very important mechanic: **at any given time there can be at most one** VOLATILE **element in the trajectory list**;

- POP_BACK(), POP_FRONT() and RESET() for removing some or all the trajectory elements from Γ_{list};
- EMPTY() for checking if Γ_{list} is empty.

These basic operations are provided for working with the data structure itself. Provided next are the actual "trajectory sequencing" methods:

- An EVALUATELIST($j_{\max}$) method which returns up to the $j_{\max}$-th derivative of the current trajectory at its current time, explained in Algorithm 18;
- FINISHED(), ABORT() and REWIND() methods which allow further control over Γ_{list} playback than the forward, linear playback offered by EVALUATELIST(). These are described in Algorithm 19. The REWIND() is trivial for the following reason: except when the current element is VOLATILE, **at most one element in the list is not at $t = 0$ at any given point in time**. When the current element *is* VOLATILE, moving the current pointer away from it automatically removes the VOLATILE element thanks to `std::shared_ptr`'s memory deallocation mechanics (only current owns the VOLATILE element object);
- A SWAPCURRENTTRAJECTORY(γ_{new}) method, described in Algorithm 20, which interfaces to the current element's SWAPTRAJECTORY() method.

The best way to describe the possible trajectory sequences produced by these methods is by example. Figure 4.6 provides four examples that cover all trajectory sequencing possibilities. Further to these, Section 4.3 and Chapter 6 make extensive use of the sequencer in the trajectory tracking and autonomy engine algorithms.

4.3 Trajectory Tracker

The trajectory sequencer of Section 4.2 enables one to seamlessly evaluate $\Gamma_{\text{list}}(t)$ via Algorithm 18. However, the sequencer is incapable of publishing a controller reference since its output is simply a tuple of raw trajectory derivatives. The *trajectory tracker*[18] provides this missing functionality. It is implemented as a CORESTATEMACHINE (see Appendix B) executed at $f_{\text{guidance}} = 20\,\text{Hz}$. Broadly speaking, it calls Algorithm 18 and transforms its output into a valid controller reference which gets published to the AscTec HLP interface in Figure 2.9. Because it is a CORESTATEMACHINE, it can be played and paused as explained in Appendix B. Algorithm 21 explains one iteration of the trajectory tracker. Two modes are provided, POSITIONTRACKING and VELOCITYTRACKING, that respectively work with the position and velocity modes of the translation controller in Section 5.8. Importantly, in the position control mode a VOLATILE trajectory is used in the emergency case of the position tracking error growing beyond a tolerated bound $\epsilon_{\text{tracking,thresh}}$, which is set to 1 m in the current implementation. This threshold may be violated e.g. due to a strong wing gust. The *realignment trajectory* transfers the quadrotor from its current position and velocity to those of the last trajectory reference prior to $\epsilon_{\text{tracking,thresh}}$ violation. Its evaluation follows exactly the example in Figure 4.6c. Namely, during the transfer the original trajectory element remains paused – a natural consequence of not evaluating Algorithm 16 for that element (i.e. that

[18] `/guidance/alure_trajectory/include/alure_trajectory/trajectory_tracker.h`

Algorithm 17 Insertion of a new element after the current one into the trajectory list.

`/guidance/alure_trajectory/src/trajectory_list.cpp:insert`

```
1: function INSERTELEMENT(γ_new)
2:     if EMPTY() then                                   ▷ Initialize the list
3:         head ← γ_new
4:         current ← γ_new
5:         tail ← γ_new
6:         return
7:     end if
8:     if γ_new is VOLATILE then
9:         if current is VOLATILE then    ▷ Remove the current VOLATILE element
10:            current ← current->NEXT()                  ▷ -> in the usual C++ sense
11:        end if
12:        γ_new->NEXT() ← current
13:        current ← γ_new
14:    else
15:        if current = tail then                  ▷ Inserting at the end of the list
16:            PUSH_BACK(γ_new)
17:        else              ▷ Inserting γ_new between current and current->NEXT()
18:            γ_new->NEXT() ← current->NEXT()
19:            γ_new->PREVIOUS() ← current
20:            current->NEXT()->PREVIOUS() ← γ_new
21:            current->NEXT() ← γ_new
22:        end if
23:    end if
24: end function
```

Algorithm 18 Evaluation of the current element up to the $j_{\max}$-th derivative and execution of element switching mechanics.

`/guidance/alure_trajectory/src/trajectory_list.cpp:evaluate`

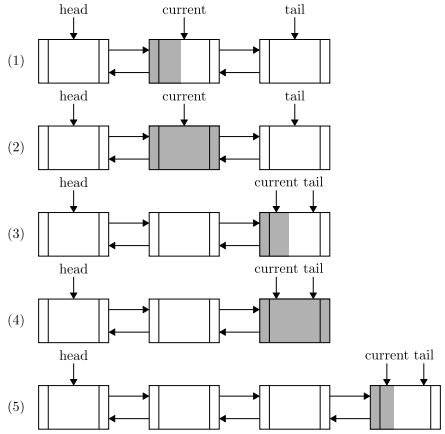
```
1: function EVALUATELIST(j_max)
2:     derivatives ← current->EVALUATETRAJECTORYELEMENT(j_max)
   Execute mechanics for moving across elements:
3:     if current->FINISHED() and current ≠ tail then
4:         current->REWIND()
5:         current ← current->NEXT()
6:     end if
7:     return derivatives
8: end function
```

Algorithm 19 Additional methods which allow greater control over Γ_{list} playback.
`/guidance/alure_trajectory/src/trajectory_list.cpp:finished`
`/guidance/alure_trajectory/src/trajectory_list.cpp:abort`
`/guidance/alure_trajectory/src/trajectory_list.cpp:rewind`

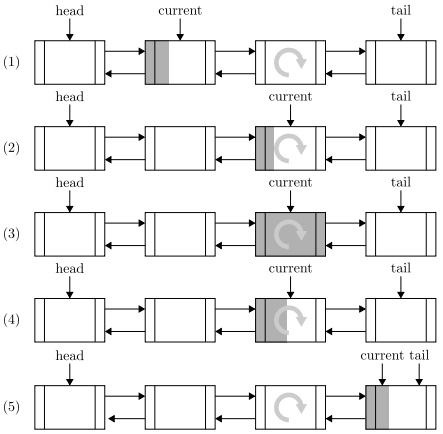
```
1: function FINISHED()
2:     return current = tail and current->FINISHED()
3: end function

4: function ABORT()
5:     assert current ≠ tail    ▷ The last element has no next element to abort to
6:     current->REWIND()
7:     current ← current->NEXT()
8: end function

9: function REWIND()
10:    current->REWIND()
11:    current ← head
12: end function
```

Algorithm 20 An interface to a trajectory element's SWAPTRAJECTORY(σ_{new}) method.
`/guidance/alure_trajectory/src/trajectory_list.cpp:swapCurrentTrajectory`

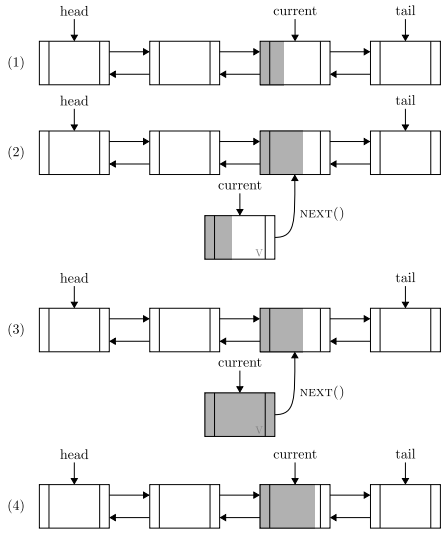
```
1: function SWAPCURRENTTRAJECTORY(σ_new)
2:     if current is VOLATILE then    ▷ Swapping a VOLATILE element's trajectory
       is not allowed. Swap instead the NEXT() element's trajectory, guaranteed to be
       non-VOLATILE.
3:         current->NEXT()->SWAPTRAJECTORY(σ_new)
4:     else
5:         current->SWAPTRAJECTORY(σ_new)
6:     end if
7: end function
```



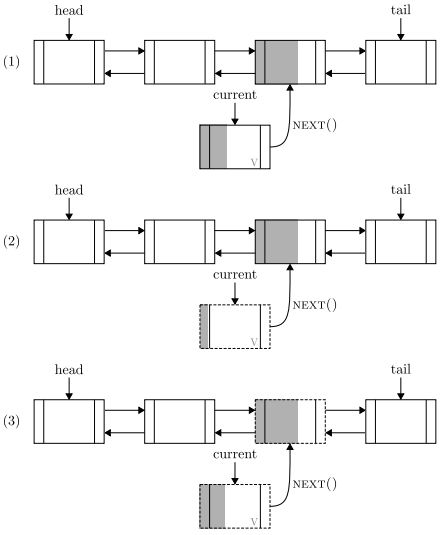
(a) A linear sequence example. As EVALUATELIST() gets called, time advances in (1) until the current element is FINISHED() in (2). EVALUATELIST() then REWIND()s the current element and advances it to the next element in (3). Eventually, the list is FINISHED() in (4). A new element is added via PUSH_BACK(), allowing the list to advance to it in (5).



(b) A sequence example with a CYCLIC element. From (1) to (2), calling EVALUATELIST() sets current to the cyclic element in the same way as (a). From (2) to (4), an arbitrary time may pass as each time that the cyclic element is FINISHED(), it gets rewound. Therefore, current never advances. In (5), current is ABORT()ed to move to the next element.



(c) An example with a VOLATILE element. Between (1) and (2), INSERTELEMENT() is used to add a VOLATILE element. This pauses the original current element while the VOLATILE element is traversed. In (3), the VOLATILE element is FINISHED(). The list transitions back to the original element while the VOLATILE element is removed. List evaluation continues in (4).



(d) An example of overriding a VOLATILE element and swapping an element. (1) represents the same situation as (2) in (c). In (2), INSERTELEMENT() is used to insert a VOLATILE element, which replaces the current one (and evaluation restarts from $t = 0$). In (3), SWAPCURRENTTRAJECTORY() replaces the trajectory of current->NEXT() while keeping its time the same.

Figure 4.6: Trajectory sequencing examples. Partial fill with a gray background is used to illustrate each element's time. Line style in (d) serves a purely visual purpose of highlighting that an element is different in some way (described by caption).

trajectory's t is not updated). The nominal trajectory sequence continues once the quadrotor is back on the original trajectory with a tolerated position tracking error. Section 7.3 shows results of the trajectory tracker in action.
Note that acceleration feed-forward, $\boldsymbol{a}_{\text{ff}}$, and body rate feed-forward, $\boldsymbol{\omega}_{\text{ff}}$ control terms described in Chapter 5 are not published. How to compute these, however, is described in Appendix E and is implemented[19]. Flight tests showed that the trajectory tracking lag introduced by reference pre-filtering (see Section 5.8) makes the consequently out-of-phase feed-forward term application detrimental to control accuracy. Future implementations that may venture to disable the pre-filter could benefit from these feed-forward terms.
The autonomy engine described in Chapter 6 needs to know when the trajectory list is finished. This serves as a state transition trigger. Evaluating the FINISHED() method of Algorithm 19, however, is not robust against the case of the quadrotor lagging behind the trajectory reference (as is currently the case due to Chapter 5's reference pre-filtering, e.g. see Figures 7.2d and 7.15). This would lead to a premature state transition. Therefore, the trajectory tracker provides a more advanced FINISHED() method in Algorithm 22. By conditioning also on a required position and yaw tracking accuracy, the quadrotor is insured to have indeed reached the end of $\mathit{\Gamma}_{\text{list}}$. Note that Algorithm 22 degenerates to the simple FINISHED() evaluation of Algorithm 19 for the velocity tracking mode. This mode is used only during takeoff (see Section 6.2) and it does not make sense to impose a velocity tracking error tolerance – using one may in fact *reduce* robustness in case of a consistently large velocity tracking error (e.g. due to strong wind) while the quadrotor continues to climb to undesirably (dangerously) high altitudes.

[19] `/general/alure_common/include/alure_common/state_representations.h`

Algorithm 21 One iteration of the trajectory tracker loop responsible for polling the trajectory sequencer and converting its output into a controller reference.
`/guidance/alure_trajectory/src/trajectory_tracker.cpp:stateMachineDefinition`

Initialization:
threshold_violated ← **false**
One iteration of trajectory tracking loop:
if PositionTracking **then**
  Get the quadrotor $\tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}}$ and $\tilde{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}}$
  derivatives ← evaluateList(4) ▷ Evaluate up to snap using Algorithm 18
Logic for triggering a realignment trajectory:
  if not threshold_violated **then** ▷ Use actual trajectory value
    $\boldsymbol{p}_{\mathrm{traj}} \leftarrow$ derivatives[1]
  else ▷ Use last value before error threshold violation
    $\boldsymbol{p}_{\mathrm{traj}} \leftarrow$ derivatives_last[1]
  end if
  violation ← $\|\tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}} - \boldsymbol{p}_{\mathrm{traj}}\| > \epsilon_{\mathrm{tracking,thresh}}$
  if violation **and** (**not** threshold_violated) **then**
    derivatives_last ← derivatives
  end if
  realignment_violation ← $\|\boldsymbol{p}_{\mathrm{w}}^{\mathrm{b}} - \text{derivatives}[1]\| > \epsilon_{\mathrm{tracking,thresh}}$
  if (violation **and** (**not** threshold_violated)) **or** (threshold_violated **and** realignment_violation) **then**
Compute transfer trajectory for the first 2 derivatives:
    $\boldsymbol{d}_{\mathrm{i}} \leftarrow (\tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}}, \tilde{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}})$
    $\boldsymbol{d}_{\mathrm{f}} \leftarrow (\boldsymbol{p}_{\mathrm{traj}}, \text{derivatives\_last}[2])$
    $T \leftarrow \frac{\|\boldsymbol{p}_{\mathrm{traj}} - \tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}}\|}{v_{\mathrm{transfer}}}$ ▷ Approximate trajectory duration using a constant transfer speed v_{transfer} assumption
    $\sigma_{\mathrm{realign}} \leftarrow$ createTransferTrajectory($\boldsymbol{d}_{\mathrm{i}}$, $\boldsymbol{d}_{\mathrm{f}}$, T)
    $\gamma_{\mathrm{realign}} \leftarrow \{\sigma_{\mathrm{realign}}, 1/f_{\mathrm{guidance}}, \text{VOLATILE}\}$
    insertElement($\gamma_{\mathrm{realign}}$) ▷ Into the Γ_{list} associated with the tracker
    derivatives ← evaluateList(4) ▷ Re-sample to get the realignment trajectory reference
  end if
  threshold_violated ← violation
else ▷ VelocityTracking
  derivatives ← evaluateList(0) ▷ Evaluate velocity only
end if
Publish the controller reference:
if PositionTracking **then**
  $\boldsymbol{p}_{\mathrm{ref}} \leftarrow$ derivatives[1]
  if ψ_{ref} set **then** ▷ Yaw reference may optionally be passed to tracker
    Publish $\boldsymbol{p}_{\mathrm{ref}}$ and ψ_{ref}
  else
    Publish $\boldsymbol{p}_{\mathrm{ref}}$ and tell controller to keep current yaw
  end if
else ▷ VelocityTracking
  $\boldsymbol{v}_{\mathrm{ref}} \leftarrow$ derivatives[1]
  if ψ_{ref} set **then** ▷ Yaw reference may optionally be passed to tracker
    Publish $\boldsymbol{v}_{\mathrm{ref}}$ and ψ_{ref}
  else
    Publish $\boldsymbol{v}_{\mathrm{ref}}$ and tell controller to keep current yaw
  end if
end if

Algorithm 22 Determine if the trajectory list is finished being tracked..
`/guidance/alure_trajectory/src/trajectory_tracker.cpp:finished`

```
1: function FINISHED()
   Velocity tracking mode case:
2:     if VELOCITYTRACKING then
3:         is_finished ← FINISHED()                                  ▷ Algorithm 19
4:         return is_finished
5:     end if
   Position tracking mode case:
6:     if not FINISHED() then                                        ▷ Algorithm 19
7:         return false
8:     end if
   Verify position tracking accuracy:
9:     p_traj ← (EVALUATELIST(0))[1]
10:    position_error ← ‖p̃_w^b − p_traj‖
11:    if position_error > position_error_tolerance then
12:        return false
13:    end if
14:    if ψ_ref not set then
15:        return true
16:    end if
   Verify yaw tracking accuracy (avoiding wrap-around issues via cos):
17:    q_des ← CONVERTYAWTOQUATERNION(ψ_ref)
18:    q_err ← (q̃_w^b)^{-1} ⊗ q_des                                  ▷ ≈ yaw error quaternion
19:    cos_yaw_error ← cos( CONVERTQUATERNIONTOYAW(q_err) )
20:    if cos_yaw_error < cos_yaw_error_tolerance then
21:        return false                                              ▷ Yaw error exceeds the tolerance
22:    end if
23:    return true                                                   ▷ All checks passed
24: end function
```

Chapter 5

Control

The control subsystem is responsible for manipulating the motor thrusts in order to execute the guidance commands while maintaining the airborne quadrotor's stability. The control subsystem is subject to the design specifications given by Specification List 3. S1 and S2 are robustness requirements. S3 is driven by a performance requirement for precision landing. S4 makes the design future-proof. Finally, S5 and S6 aim to minimize the learning curve for new users such that deployment and improvements to the current implementation can be done more easily.

S1	Stability must be maintained despite failure of the Odroid XU4 master computer.
S2	A fallback must be provided in the case of state estimation failure.
S3	Sufficient reference tracking accuracy must be maintained in the case of up to severe wind.
S4	The implementation should be autopilot platform-agnostic (i.e. be deployable on virtually any multirotor platform with only minor modifications).
S5	The implementation should be understandable and usable even by non-control engineers (e.g. computer vision/estimation specialists).
S6	Interaction with the control system should be identical to the `asctec_mav_framework` for the operator.

Specification List 3: Control subsystem design specifications.

5.1 Cascaded Control Overview

The current implementation is a cascaded control loop shown in Figure 5.1. It follows the conventional cascade control implementation in which the outputs of outer loop controllers become references for the inner loop controllers Skogestad and Postlethwaite [102]. Cascaded control is a natural choice for a multirotor, whose dynamics exhibit time scale separation. A well time scale separated multirotor has:

- Translation control loop bandwidth ≈ 1 to $2\,^{\text{rad}}/_{\text{s}}$;
- Attitude control loop bandwidth ≈ 5 to $10\,^{\text{rad}}/_{\text{s}}$;

- Body rate control loop bandwidth $\gtrapprox 40\,^{\mathrm{rad}}/_{\mathrm{s}}$;
- Motor bandwidth $\gtrapprox 100\,^{\mathrm{rad}}/_{\mathrm{s}}$.

A cascaded controller has the advantage that inner loops provide faster disturbance rejection and reduce the effect of nonlinearities [102], which is beneficial for accurate tracking as required by S3 in Specification List 3. Appendix D reveals that the AscTec Pelican motors have quite a low bandwidth of $\approx 30\,^{\mathrm{rad}}/_{\mathrm{s}}$ which limits the current implementation to a weakly time scale separated translation bandwidth of $\approx 1\,^{\mathrm{rad}}/_{\mathrm{s}}$, attitude bandwidth of $\approx 4\,^{\mathrm{rad}}/_{\mathrm{s}}$ and body rate bandwidth of $\approx 20\,^{\mathrm{rad}}/_{\mathrm{s}}$. Section 7.4 shows that this results in nevertheless good performance conditioned on a low feedback delay.

5.2 Literature Review

A rich body of literature exists on cascaded control architectures for multirotor control. Lupashin et al. [103] introduce a three-stage (position, attitude and body rate) quadrotor control scheme wherein individual loops are shaped to have first- or second-order complementary sensitivity functions. Position is a PD controller, attitude control considers directly the $\mathsf{SO}(3)$ rotation matrix and body rate control is a P controller with a feedback linearizing term. A simplified tuning based on the desired time constants and damping ratios followed by full model stability verification allows to design "high performance, yet tractable control laws". Faessler et al. [104] describes in slightly more detail their almost identical cascaded control loop. Both Lupashin et al. [103] and Faessler et al. [104] controllers do not contain integrators and derive their tracking accuracy from an extensive calibration procedure including the calibration of rotor thrusts, latency, sensor biases and measurement frames. Faessler et al. [105] use the same cascaded control loop but replace the reduced attitude controller with a nonlinear error quaternion-based controller introduced in Brescianini et al. [106]. They also introduce a new yaw controller which retains physical meaningfulness for any quadrotor attitude and present a simple method for switching from position to velocity control in the outer loop by simply setting the proportional term to zero. Highlighting the fact that the cascaded design is a suitable basic architecture for later expansion with more advanced algorithms (S4 and S5 in Specification List 3), Faessler et al. [107] improve the body rate controller by taking into account motor dynamics, a better thrust/torque model and a prioritized motor thrust saturation scheme which improves the quadrotor's stability in tight maneuvers.
Achtelik et al. [108, 109] simplify the three-stage cascaded loop to a two-stage loop. In both, the position controller is designed based on nonlinear dynamic inversion. In Achtelik et al. [108], only the outer loop translation controller is designed while the AscTec LLP's attitude controller is used for the inner loop. Achtelik et al. [108] is believed to be the theoretical foundation of the closed-source controller provided with `asctec_mav_framework` Achtelik et al. [36]. Achtelik et al. [109] use multicopters' differential flatness property (Appendix E) to design an LQR position controller which directly outputs target body rates and collective thrust, effectively bypassing the attitude control loop. Body rates are still controlled by a P controller with a feedback linearizing term. This design is argued to offer a higher closed-loop bandwidth and lower computational complexity.
Lee et al. [110] present a two-stage (translation and attitude) cascaded control loop wherein the controllers are designed via coordinate-free geometric control on the special Eucledian group $\mathsf{SE}(3)$. Three flight modes are introduced: position, velocity and attitude. Position and velocity are PD and P controllers, respectively, which are identical to Faessler et al. [105] in that they compute a desired acceleration

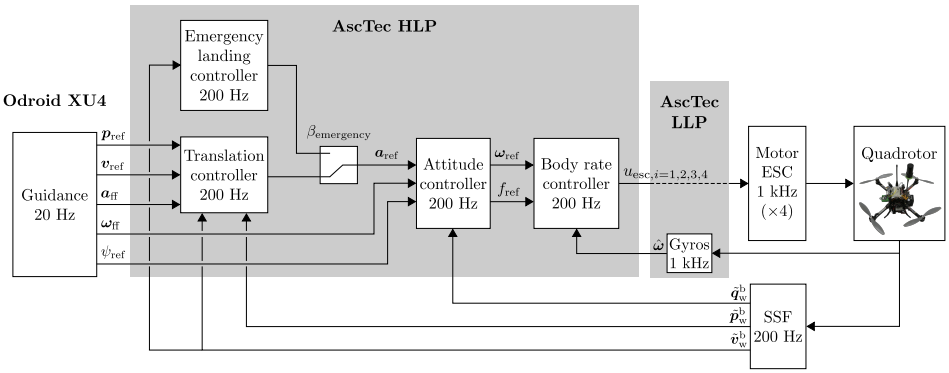


Figure 5.1: Cascaded control loop block diagram.

vector. This gets converted into an attitude which is passed to the nonlinear attitude controller. Stability during mode switching is retained thanks to the controllers' proven almost global asymptotic stability. This bypasses the need for a complicated reachability set analysis in order to establish switching stability. The attitude flight mode simply removes the outer loop (which can be replaced by a human pilot, for example). Lee et al. [111] complement this work by augmenting the position and attitude controllers with robustness terms and showing that the new controllers are robust to unstructured uncertainty in the translation and rotation dynamics, with pre-determined bounds. While neither work shows how the controller performs on a real multirotor, Mellinger and Kumar [81] present an identical cascaded architecture but with a simplified moment control law and show good tracking performance in highly dynamic maneuvers with an AscTec Hummingbird MAV Gurdan et al. [112]. In particular, the attitude controller of Mellinger and Kumar [81] omits the feedback linearizing and feedforward terms and allows diagonal gain matrices as opposed to the scalar gains in Lee et al. [110].

5.3 High-Level Description

The designed and implemented flight control loop[1] is shown in Figure 5.1. The implementation is wholly located on the HLP which makes control completely independent of the Odroid XU4 (if guidance fails, the quadrotor will simply hover at the most recent location). This satisfies S1 of Specification List 3. Although Section 5.2 cites several two-stage cascaded controllers which are claimed to have better performance, a three-stage architecture was chosen for the following reasons:

1. Brescianini et al. [106] prove that their attitude controller is globally asymptotically stable, whose discrete implementation is robust to measurement noise. Achtelik et al. [109] do not prove theoretical stability nor robustness to noise while Lee et al. [110] prove stability but do not prove robustness of the discrete time implementation to noise. Our implementation seeks the most robust possible control algorithm, therefore it uses the Brescianini et al. [106] attitude controller which yields a three-stage cascaded controller (this attitude controller outputs body rates, thus a third body rate control stage is required);

2. It is easy to incorporate features like a thrust vector cone constraint (see Section 5.7) when the translation controller outputs a desired acceleration

[1] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/control_system.h`

vector, as is the case for a three-stage architecture. This is not trivial for a two-stage architecture like that of Achtelik et al. [109];

3. A three-stage architecture is more compatible with existing autopilots (complements S4 of Specification List 3). For example, the attitude and body rate controllers can be entirely replaced by the AscTec standard attitude controller. When an existing body rate controller is available, engineers that use the implementation presented herein may also choose to use it instead of the body rate controller presented in Section 5.6.

Also note a non-flight advantage of implementing end-to-end the full control loop rather than using e.g. the AscTec LLP's attitude controller: **simulating the full GNC system becomes possible**. With the inaccessible LLP controller, only real flying or the more intricate Hardware In the Loop (HIL) simulation would be possible since the cascaded structure is otherwise incomplete. By implementing our own complete control pipeline, Software In the Loop (SIL) becomes possible which offers advantageous possibilities for rapid algorithm validation prior to HIL simulation or flying (see Section 7.1.1). In summary, the control loop presented herein is different from the original `asctec_mav_framework` control loop Achtelik et al. [108, 36] in the following ways:

- Our control loop is implemented fully on the HLP and is open source while the `asctec_mav_framework` controller is closed source and depends on the inaccessible LLP attitude controller;
- Our control loop can be implemented on any quadrotor platform and used in a SIL simulation (see Section 7.1.1);
- Our translation mode switching is bumpless (see Section 5.8.4) while the `asctec_mav_framework` controller cannot be switched between velocity or position and the acceleration modes mid-flight;
- We provide an emergency landing controller fallback for state estimation failure (Section 5.9) while the `asctec_mav_framework` controller does not;
- We provide a thrust calibration feature (see Section 5.6.4) while the `asctec_mav_framework` controller does not;
- We provide a more intuitive and documented ROS dynamic reconfigure interface to interact with the controller, which is a superset of the interface provided for the `asctec_mav_framework` controller.

The rest of this chapter provides a detailed theoretical and practical description of the design and implementation of the cascaded control loop:

- Section 5.4 presents the quadrotor Flight Dynamics Model (FDM);
- Section 5.5 lists the FDM parameters for the AscTec Pelican quadrotor;
- Section 5.6 presents the body rate controller;
- Section 5.7 presents the attitude controller;
- Section 5.8 presents the translation controller;
- Section 5.9 presents a fall-back safety mode for executing an emergency landing in case of state estimator failure.

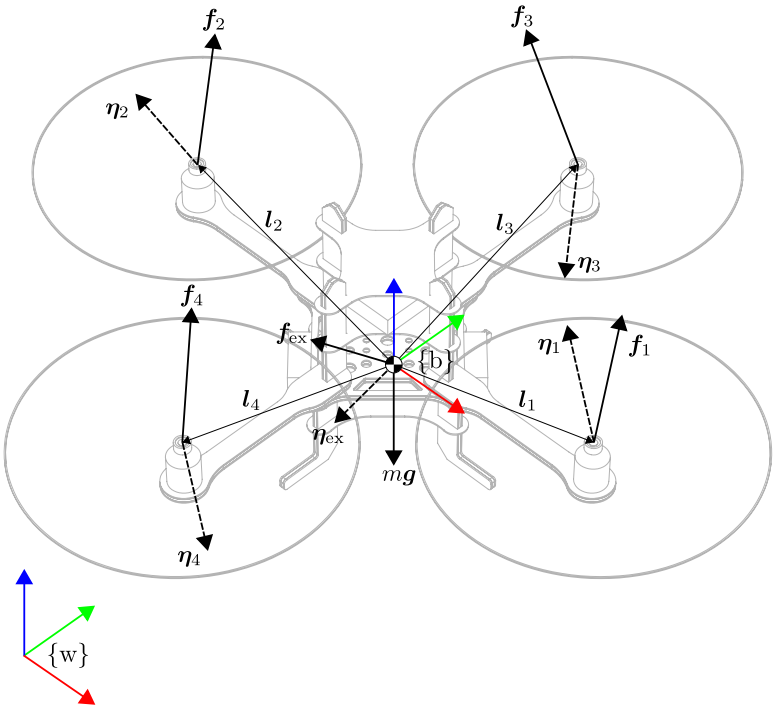


Figure 5.2: Quadrotor forces, torques and geometry. Dashed arrows are used for torque axis of rotation (via right hand rule). Propeller force and torque vector misalignments and off-rotation-axis components are exaggerated for visual clarity. The body frame {b} origin is the quadrotor's center of gravity. AscTec Pelican model by Lucena de Athayde Guimarães [113].

5.4 Quadrotor Flight Dynamics Model

An FDM is a critical design and simulation tool. Development of an FDM often brings new insights about the controlled system and precedes the control design. Although the quadrotor FDM is well known, it is presented here for completeness and will help the reader to better understand the control design choices. Two FDMs are presented:

- A high-fidelity FDM in Section 5.4.1 that is aimed at simulation and at helping to understand the full complexity of a quadrotor;
- A simplified FDM in Section 5.4.2 that is tractable for control design.

5.4.1 High-Fidelity FDM

Translation Dynamics

Consider the free-body diagram in Figure 5.2. The translation dynamics are given by kinematics and Newton's second law:

$$\dot{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}} = \boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}, \tag{5.1}$$

$$\dot{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}} = \frac{1}{m}\left(\boldsymbol{f}_{\mathrm{ex}} + \sum_{i=1}^{4} \boldsymbol{f}_i\right) + \boldsymbol{g}, \tag{5.2}$$

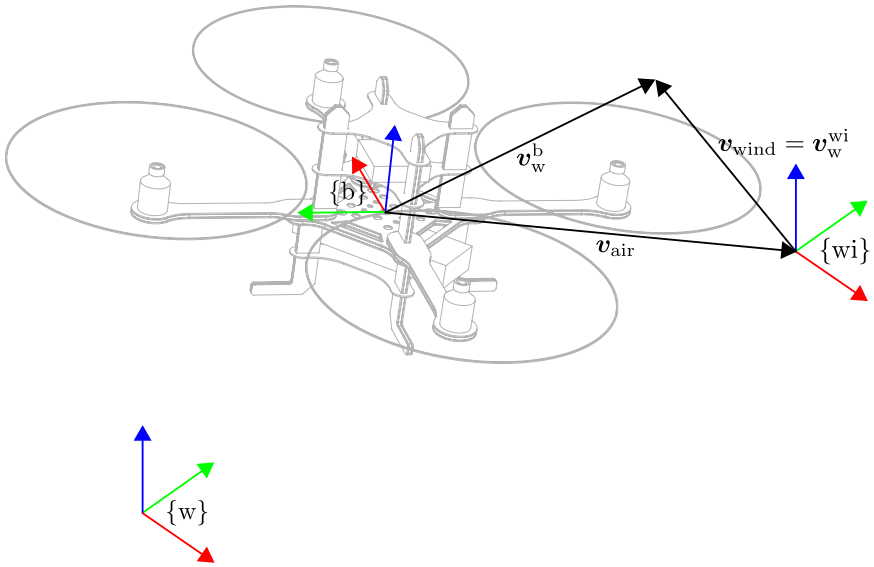


Figure 5.3: Quadrotor ground velocity ($\boldsymbol{v}_{\text{w}}^{\text{b}}$), air velocity ($\boldsymbol{v}_{\text{air}}$) and the wind velocity ($\boldsymbol{v}_{\text{wind}}$).

where:

- $\boldsymbol{f}_i$ is the net (aerodynamic) force generated by the i-th propeller reduced to the propeller geometric center;
- $\boldsymbol{f}_{\text{ex}}$ is the net effect of exogenous forces acting on the quadrotor body reduced to a force acting at the Center of Gravity (CoG);
- $\boldsymbol{g}$ is the gravity vector $(0, 0, -9.81)\,\text{m/s}^2$.

All quantities in (5.1, 5.2) are expressed in the world frame, which is assumed to be an inertial frame. Let us be very specific about three velocity quantities:

- $\boldsymbol{v}_{\text{w}}^{\text{b}}$ is the *ground velocity*[2] and represents the {b} frame motion with respect to the static {w} frame;
- $\boldsymbol{v}_{\text{wind}} = \boldsymbol{v}_{\text{w}}^{\text{wi}}$ is the *local wind velocity* and represents the wind frame {wi} motion with respect to the static {w} frame. The wind frame may be thought of as a non-rotating frame which translates along with the local air mass;
- $\boldsymbol{v}_{\text{air}}$ is the *air velocity*[3] and represents the {b} frame motion with respect to the wind frame. As shown in Figure 5.3:

$$\boldsymbol{v}_{\text{air}} = \boldsymbol{v}_{\text{w}}^{\text{b}} - \boldsymbol{v}_{\text{wind}}. \tag{5.3}$$

Dominated by aerodynamic drag, $\boldsymbol{f}_{\text{ex}}$ may be developed as:

$$\boldsymbol{f}_{\text{ex}} = \frac{1}{2}\rho c_{\text{D}} \boldsymbol{v}_{\text{air}} \|\boldsymbol{v}_{\text{air}}\| + \boldsymbol{f}_{\text{ex,other}}. \tag{5.4}$$

The first term is standard aerodynamic drag, where ρ is the air density and c_{D} is the conglomerated drag coefficient. The second term represents lesser exogenous effects and may further emulate model uncertainty and include e.g. a noise component.

[2] Whose magnitude is the *ground speed*.
[3] Whose magnitude is the *airspeed*.

Attitude Dynamics

The generic angular momentum equation is given by Phillips [114]:

$$\boldsymbol{\eta}_{\text{total}} = \frac{\mathrm{d}\boldsymbol{\zeta}_{\text{total}}}{\mathrm{d}t} + \boldsymbol{\omega} \times \boldsymbol{\zeta}_{\text{total}} \tag{5.5}$$

where $\boldsymbol{\eta}_{\text{total}}$ is the total torque acting on the system, $\boldsymbol{\zeta}_{\text{total}}$ is the system total angular momentum and $\boldsymbol{\omega}$ is the body angular velocity (i.e. "body rates"). All quantities are expressed in the body frame. Assuming that the quadrotor consists of one rigid body (the frame) and a set of spinning propellers, $\boldsymbol{\zeta}_{\text{total}}$ is given by:

$$\boldsymbol{\zeta}_{\text{total}} = J\boldsymbol{\omega} + \boldsymbol{\zeta}_{\text{rotor}}, \tag{5.6}$$

where $J \in \mathbb{R}^{3\times3}$ is the quadrotor total inertia tensor (including the motor rotors and propellers) and $\boldsymbol{\zeta}_{\text{rotor}}$ is the total angular momentum of the spinning motor rotors and propellers relative to the body-fixed frame:

$$\boldsymbol{\zeta}_{\text{rotor}} = \sum_{i=1}^{4} J_{\text{p},i}\omega_{\text{p},i}\boldsymbol{k}_{\text{p},i}, \tag{5.7}$$

where (for the i-th propeller) $J_{\text{p},i} \in \mathbb{R}^{3\times3}$ is the inertia tensor (in the body frame) of the propeller and rotor assembly, $\omega_{\text{p},i}$ is the angular speed and $\boldsymbol{k}_{\text{p},i}$ is the propeller plane normal which points up (i.e. $\boldsymbol{k}_{\text{p},i}\cdot\boldsymbol{e}_{\text{z}}^{\text{b}} > 0$) for anticlockwise rotation and down for clockwise rotation. The rate of change of total angular momentum is given by:

$$\frac{\mathrm{d}\boldsymbol{\zeta}_{\text{total}}}{\mathrm{d}t} = J\dot{\boldsymbol{\omega}} + \sum_{i=1}^{4} J_{\text{p},i}\dot{\omega}_{\text{p},i}\boldsymbol{k}_{\text{p},i}. \tag{5.8}$$

Using (5.6), (5.7) and (5.8) in (5.5), one obtains:

$$\boldsymbol{\eta}_{\text{total}} = J\dot{\boldsymbol{\omega}} + \sum_{i=1}^{4} J_{\text{p},i}\dot{\omega}_{\text{p},i}\boldsymbol{k}_{\text{p},i} + \boldsymbol{\omega} \times \left(J\boldsymbol{\omega} + \sum_{i=1}^{4} J_{\text{p},i}\omega_{\text{p},i}\boldsymbol{k}_{\text{p},i} \right). \tag{5.9}$$

The total torque $\boldsymbol{\eta}_{\text{total}}$, expressed in the body frame, is given by:

$$\boldsymbol{\eta}_{\text{total}} = \boldsymbol{\eta}_{\text{ex}} + \sum_{i=1}^{4} \left(\boldsymbol{\eta}_i + \boldsymbol{l}_i \times \boldsymbol{f}_i \right), \tag{5.10}$$

where $\boldsymbol{\eta}_{\text{ex}}$ is the net exogenous torque (e.g. from integrated effect of wind pressure i.e. drag) while $\boldsymbol{\eta}_i$ is the (aerodynamic) torque generated by the i-th propeller reduced to the propeller geometric center. Using quaternion rate of change kinematics and rearranging (5.9), the attitude dynamics are obtained:

$$\dot{\boldsymbol{q}}_{\text{w}}^{\text{b}} = \frac{1}{2}\begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \otimes \boldsymbol{q}_{\text{w}}^{\text{b}}, \tag{5.11}$$

$$\dot{\boldsymbol{\omega}} = J^{-1}\Bigg(\boldsymbol{\eta}_{\text{ex}} + \sum_{i=1}^{4} \left(\boldsymbol{\eta}_i + \boldsymbol{l}_i \times \boldsymbol{f}_i - J_{\text{p},i}\dot{\omega}_{\text{p},i}\boldsymbol{k}_{\text{p},i} \right) - \boldsymbol{\omega} \times \left(J\boldsymbol{\omega} + \sum_{i=1}^{4} J_{\text{p},i}\omega_{\text{p},i}\boldsymbol{k}_{\text{p},i} \right) \Bigg), \tag{5.12}$$

where $\otimes$ represents quaternion multiplication (see Trawny and Roumeliotis [69], Solà [70] for an introduction to quaternions).

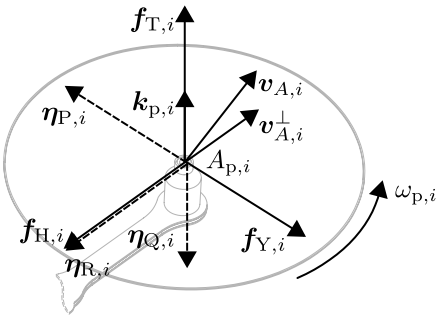


Figure 5.4: Dominant aerodynamic forces and torques acting on a single propeller in forward flight. The propeller is shown spinning counter-clockwise.

Propeller Aerodynamics in Forward Flight

This section models the aerodynamic force $\boldsymbol{f}_i$ and torque $\boldsymbol{\eta}_i$ produced by the i-th propeller. Figure 5.4 shows their decomposition into dominant components as given by classical blade element theory Martin and Salaun [115]. Let $A_{\mathrm{p},i}$ be the propeller geometric center and $\boldsymbol{v}_{\mathrm{A},i}$ the air velocity of $A_{\mathrm{p},i}$. The dominant components of $\boldsymbol{f}_i$ are:

- The *thrust force*,
$$\boldsymbol{f}_{\mathrm{T},i} = \mathcal{M}_{\boldsymbol{T}_i}(\omega_{\mathrm{p},i})\boldsymbol{k}'_{\mathrm{p},i}, \tag{5.13}$$
where $\mathcal{M}_{\boldsymbol{T}_i}$ is some map from the propeller angular speed to the thrust and $\boldsymbol{k}'_{\mathrm{p},i} \parallel \boldsymbol{k}_{\mathrm{p},i}$ but always points up (i.e. $\boldsymbol{k}'_{\mathrm{p},i} \cdot \boldsymbol{e}^{\mathrm{b}}_{\mathrm{z}} > 0$). The $\mathcal{M}_{\boldsymbol{T}_i}$ map shall be specified for the simplified FDM in Section 5.4.2.

- The *drag force* (also called the *H-force*),
$$\boldsymbol{f}_{\mathrm{H},i} = -\omega_{\mathrm{p},i} c_{\mathrm{H},i} \boldsymbol{v}^{\perp}_{\mathrm{A},i}, \tag{5.14}$$
where $\boldsymbol{v}^{\perp}_{\mathrm{A},i} = \boldsymbol{v}_{\mathrm{A},i} - (\boldsymbol{v}_{\mathrm{A},i} \cdot \boldsymbol{k}_{\mathrm{p},i})\boldsymbol{k}_{\mathrm{p},i}$ is the propeller air velocity's projection onto the propeller plane and $c_{\mathrm{H},i}$ is the propeller drag force coefficient;

- The *side force*,
$$\boldsymbol{f}_{\mathrm{Y},i} = \omega_{\mathrm{p},i} c_{\mathrm{Y},i} \boldsymbol{v}_{\mathrm{A},i} \times \boldsymbol{k}_{\mathrm{p},i}, \tag{5.15}$$
where $c_{\mathrm{Y},i}$ is the propeller side force coefficient.

Composing (5.13) to (5.15):

$$\boldsymbol{f}_i = \boldsymbol{f}_{\mathrm{T},i} + \boldsymbol{f}_{\mathrm{H},i} + \boldsymbol{f}_{\mathrm{Y},i} + \boldsymbol{f}_{\mathrm{other},i}, \tag{5.16}$$

where $\boldsymbol{f}_{\mathrm{other},i}$ captures unmodelled effects. The dominant components of $\boldsymbol{\eta}_i$ are:

- The *drag torque*,
$$\boldsymbol{\eta}_{\mathrm{Q},i} = -\mathcal{M}_{\boldsymbol{Q}_i}(\omega_{\mathrm{p},i})\boldsymbol{k}_{\mathrm{p},i}, \tag{5.17}$$
where $\mathcal{M}_{\boldsymbol{Q}_i}$ is some map from the propeller angular speed to (positive) drag torque. It shall be specified for the simplified FDM in Section 5.4.2;

- The rolling torque,
$$\boldsymbol{\eta}_{\mathrm{R},i} = -\omega_{\mathrm{p},i} c_{\mathrm{R},i} \boldsymbol{v}^{\perp}_{\mathrm{A},i}, \tag{5.18}$$
where $c_{\mathrm{R},i}$ is the propeller rolling torque coefficient;

- The pitching torque,

$$\boldsymbol{\eta}_{\mathrm{P},i} = -\omega_{\mathrm{p},i} c_{\mathrm{P},i} \boldsymbol{v}_{\mathrm{A},i} \times \boldsymbol{k}_{\mathrm{p},i}, \tag{5.19}$$

  where $c_{\mathrm{P},i}$ is the propeller pitching torque coefficient.

Composing (5.17) to (5.19):

$$\boldsymbol{\eta}_i = \boldsymbol{\eta}_{\mathrm{Q},i} + \boldsymbol{\eta}_{\mathrm{R},i} + \boldsymbol{\eta}_{\mathrm{P},i} + \boldsymbol{\eta}_{\mathrm{other},i}, \tag{5.20}$$

where $\boldsymbol{\eta}_{\mathrm{other},i}$ captures unmodelled effects. Some of these effects that are lumped into $\boldsymbol{f}_{\mathrm{other},i}$ and $\boldsymbol{\eta}_{\mathrm{other},i}$ are known but are negligible:

- Contributions due to linear and angular accelerations of the propeller have been omitted due to the propeller's much smaller mass when compared to the quadrotor ($\approx 12\,\mathrm{g}$ vs. $\approx 1.96\,\mathrm{kg}$ in our case);
- Higher-order terms related to $\boldsymbol{v}_{\mathrm{A},i}$ are omitted, which is valid when $\boldsymbol{v}_{\mathrm{A},i}$ is small with respect to the propeller tip speed. This is certainly true for us since the AscTec Pelican, which does not exceed a $2\,\mathrm{m/s}$ ground speed for the full cycle autonomy task, has a propeller tip speed of approximately $77\,\mathrm{m/s}$ at hover;
- Contributions due to the angular velocity $\boldsymbol{\omega}$ of the quadrotor structure itself is omitted, although it is included in Martin and Salaun [115]. This is valid when $\|\boldsymbol{\omega}\|$ is small, which it is in our case (the quadrotor operates close to hover);
- The force and torque coefficients in (5.14), (5.15), (5.18) and (5.19) are assumed constant, but are known to vary with angle of attack Mueller and D'Andrea [116], Hoffmann et al. [117]. This is valid when the angle of attack is small, which is true in our case (the quadrotor operates close to hover).

BLDC Motor Dynamics

The "input variables" driving the quadrotor motion are the propeller speeds $\omega_{\mathrm{p},i}$ and enter the FDM via (5.13) to (5.15) and (5.17) to (5.19). In reality, however, the fastest yet still reasonable dynamic element to consider are the motor dynamics.
The AscTec Pelican uses four BLDC motors (see Section 2.1). Datasheets for these motors are not available. We assume that they are the 3-phase, *wye*-connected BLDC motors common in the Radio Control (RC) community. The aim of this section is to determine the motor dynamics' *order* which serves as a basis for the simplified motor model used in Section 5.4.2. We do not seek to delve into the details of BLDC operation since there is no need for it in the present control design. The interested reader may find such details in works like Melkote et al. [118], Kapun et al. [119], Khorrami et al. [120].
The operating principle of a BLDC motor is that, at any time instance, two phases are active while the third one is off. The result is that on average and for a constant rotor speed ω_{m}, the supplied voltage and current to the motor are constant. As discussed in Bangura [121], a simplified BLDC model is in fact that of a DC motor as shown in Figure 5.5. The well known DC motor dynamics are readily computed in state space form[4]:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} i_{\mathrm{m}} \\ \omega_{\mathrm{m}} \end{bmatrix} = \begin{bmatrix} -\frac{R_{\mathrm{m}}}{L_{\mathrm{m}}} & -\frac{k_{\mathrm{m}}}{L_{\mathrm{m}}} \\ \frac{k_{\mathrm{m}}}{J_{\mathrm{m}}} & -\frac{b_{\mathrm{m}}}{J_{\mathrm{m}}} \end{bmatrix} \begin{bmatrix} i_{\mathrm{m}} \\ \omega_{\mathrm{m}} \end{bmatrix} + \begin{bmatrix} \frac{1}{L_{\mathrm{m}}} \\ 0 \end{bmatrix} v_{\mathrm{m}} + \begin{bmatrix} 0 \\ -\frac{1}{J_{\mathrm{m}}} \end{bmatrix} \tau_{\mathrm{l}}, \tag{5.21}$$

where:

[4] Static friction torque is negligible for small BLDC motors Bangura [121].

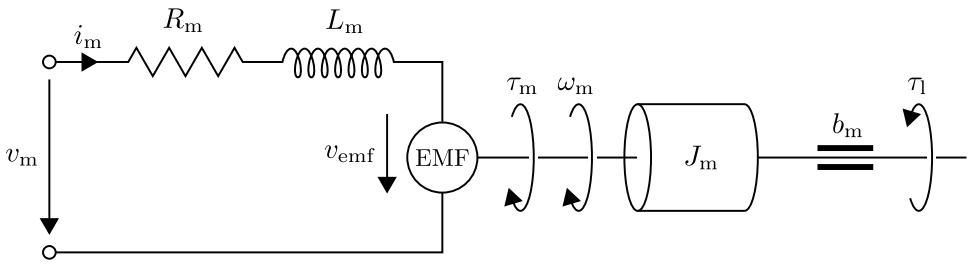


Figure 5.5: DC motor model used as an approximation of BLDC operation. Motor torque $\tau_\mathrm{m} = k_\mathrm{m} i_\mathrm{m}$ and back-EMF $v_\mathrm{emf} = k_\mathrm{m}\omega_\mathrm{m}$. R_m, L_m and J_m are scalar.

- v_m is the motor input voltage;
- i_m is the current drawn by the motor;
- ω_m is the motor angular speed;
- τ_l is the load torque e.g. the drag torque generated by a propeller;
- J_m is the rotor inertia including any attached components e.g. a propeller;
- k_m is the back-Electromotive Force (EMF) constant, equal to the motor torque constant[5];
- R_m and L_m are the phase resistance and inductance respectively.

In coarse terms, an ESC controls the motor by manipulating v_m in order to achieve a desired ω_m, which is specified via an ESC command. One sees that the dynamics from input v_m to output ω_m are **second-order**.

Summary

The quadrotor translation dynamics are given by (5.1) and (5.2), where the propellers' aerodynamic force is given by (5.16) and the exogenous force is given by (5.4). The attitude dynamics are given by (5.11) and (5.12), where the propellers' aerodynamic torque is given by (5.20). The propulsion subsystem is determined to have second-order dynamics from input voltage to output rotor angular speed.
By itself, this high-fidelity model may serve as a useful starting point for simulation. However, we use RotorS (see Section 7.1.1) and its existing, similar multirotor FDM for simulation Furrer et al. [122]. Instead, we shall simplify this FDM in the next section in order to create a model from which control laws may be developed. Furthermore, the high-fidelity treatise presented above will elucidate many control design choices (e.g. the first-order motor dynamics assumption as shall be seen next).

5.4.2 Simplified Model

The model in Section 5.4.1 is now simplified to a suitable level of complexity for control design. Figure 5.6 illustrates the free-body diagram corresponding to this simpler model. The following assumptions are made:

- The $A_{\mathrm{p},i}$'s are distributed radially symmetrically and the $\{\mathrm{b}\}$ frame origin is placed in their geometric center. An offset along $\boldsymbol{e}_\mathrm{z}^\mathrm{b}$ is irrelevant for the dynamics, given the next assumption;

[5] In SI units.

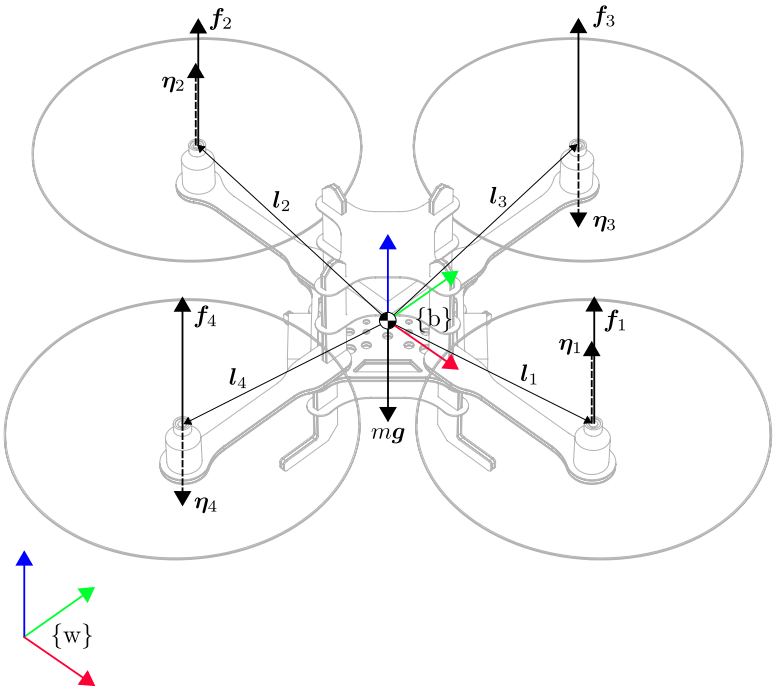


Figure 5.6: Simplified version of Figure 5.2 that is suitable for control design purposes.

- The normals of the propeller planes, $\boldsymbol{k}_{\mathrm{p},i}$, are (anti-)parallel to $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}$;
- The CoG is at the body frame origin;
- The exogenous forces $\boldsymbol{f}_{\mathrm{ex}}$ and torques $\boldsymbol{\eta}_{\mathrm{ex}}$ are disturbance forces that shall be rejected by the controller. They shall not be modeled explicitly (thus, aerodynamic drag is not made explicit in the design);
- $\boldsymbol{f}_i = \boldsymbol{f}_{\mathrm{T},i}$ and $\boldsymbol{\eta}_i = \boldsymbol{\eta}_{\mathrm{Q},i}$. Let us from here on denote $f_i := \|\boldsymbol{f}_i\|$ and $\eta_i = \|\boldsymbol{\eta}_i\|$;
- $\boldsymbol{J}_{\mathrm{p},i} \approx 0$ such that propeller gyroscopic effects in (5.12) can be neglected;
- The motor dynamics are approximated as first-order in thrust, which has been done previously for quadrotor control Faessler et al. [107], Lupashin et al. [103]. Note that this is similar to neglecting the fast dynamics of i_{m} in (5.21) – but the theoretical reasoning is non-trivial because thrust and angular speed dynamics are not readily relatable as their relationship is not linear (see (5.25)). Nevertheless, the very good model fit in Figure D.9 validates this approach which also significantly simplifies reasoning about time-scale separation for the body rate controller (Section 5.6).

The high-fidelity FDM is readily simplified using these assumptions to yield the following model:

$$\begin{cases} \dot{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}} = \boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}} \\ \dot{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}} = \boldsymbol{g} + \dfrac{\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}{m} \displaystyle\sum_{i=1}^{4} f_i & (5.22) \\ \dot{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{b}} = \dfrac{1}{2} \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \otimes \boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}} \\ \dot{\boldsymbol{\omega}} = J^{-1} \left(\displaystyle\sum_{i=1}^{4} \big((-1)^j \eta_i \boldsymbol{e}_3 + \boldsymbol{l}_i \times (\boldsymbol{e}_3 f_i) \big) - \boldsymbol{\omega} \times (J\boldsymbol{\omega}) \right) & (5.23) \\ \dot{f}_i = \begin{cases} \tau_{\mathrm{m,up}}^{-1} (f_{i,\mathrm{ref}} - f_i) & \text{if } f_{i,\mathrm{ref}} \geq f_i \\ \tau_{\mathrm{m,dn}}^{-1} (f_{i,\mathrm{ref}} - f_i) & \text{if } f_{i,\mathrm{ref}} < f_i \end{cases} & (5.24) \end{cases}$$

where $j = 1, 1, -1, -1$ for $i = 1, 2, 3, 4$. We use the following full-quadratic thrust and torque maps Faessler et al. [107]:

$$f_{i,\mathrm{ref}} = \mathcal{M}_{\mathrm{f}}(u_{\mathrm{esc},i}) := k_{\mathrm{f},2} u_{\mathrm{esc},i}^2 + k_{\mathrm{f},1} u_{\mathrm{esc},i} + k_{\mathrm{f},0}, \tag{5.25}$$

$$\eta_{i,\mathrm{ref}} = \mathcal{M}_{\eta}(u_{\mathrm{esc},i}) := k_{\eta,2} u_{\mathrm{esc},i}^2 + k_{\eta,1} u_{\mathrm{esc},i} + k_{\eta,0}. \tag{5.26}$$

Note that these maps are from the ESC command rather than the angular speed. Although angular speed is the direct variable influencing aerodynamic force and torque, our motor control is open-loop whose input is the ESC command. This is how the AscTec AutoPilot works. Implementing closed-loop motor control was not judged to be a priority and thus was not implemented (see Section 8.2). Therefore, like Faessler et al. [107], we simplify the model to better suit the control design by considering indirectly the i-th motor ESC command, $u_{\mathrm{esc},i}$, instead. The actual propeller torque is given via (5.24), (5.25) and (5.26):

$$\eta_i = \kappa(f_i) := \mathcal{M}_{\eta}(\mathcal{M}_{\mathrm{f}}^{-1}(f_i)). \tag{5.27}$$

We further evaluate the sum in (5.23) to define the (simplified) net torque called the *body torque*:

$$\boldsymbol{\xi} = \begin{bmatrix} l_{\mathrm{arm}}(f_3 - f_4) \\ l_{\mathrm{arm}}(f_2 - f_1) \\ \eta_1 + \eta_2 - \eta_3 - \eta_4 \end{bmatrix}, \tag{5.28}$$

where l_{arm} denotes the propeller geometric centers' radial distance away from the body frame origin, i.e. the *arm length*. It is also noted for (5.22) that $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} = C_{(\boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}})} \boldsymbol{e}_3$. Finally, the *collective thrust* is defined as:

$$f := \sum_{i=1}^{4} f_i. \tag{5.29}$$

The complete simplified FDM is given by:

Parameter	Value
$\hat{m}$	$1.956\,\mathrm{kg}$
l_{arm}	$0.21\,\mathrm{m}$
$\tilde{J}$	$\begin{bmatrix} 1.37 \cdot 10^{-2} & 4.35 \cdot 10^{-4} & 0 \\ 4.35 \cdot 10^{-4} & 1.37 \cdot 10^{-2} & 0 \\ 0 & 0 & 1.64 \cdot 10^{-2} \end{bmatrix} \mathrm{kg} \cdot \mathrm{m}^2$
$\tau_{\mathrm{m,up}}$	$53\,\mathrm{ms}$
$\tau_{\mathrm{m,dn}}$	$24\,\mathrm{ms}$
$\{k_{\mathrm{f},2}, k_{\mathrm{f},1}, k_{\mathrm{f},0}\}$	$\{2.4547 \cdot 10^{-4}, 5.6132 \cdot 10^{-3}, 2.2335 \cdot 10^{-1}\}$
$\{k_{\eta,2}, k_{\eta,1}, k_{\eta,0}\}$	$\{4.8680 \cdot 10^{-6}, 6.6934 \cdot 10^{-5}, 4.7642 \cdot 10^{-3}\}$

Table 5.1: AscTec Pelican simplified FDM (5.30) parameters.

$$
\begin{cases}
\dot{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}} = \boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}} \\
\dot{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}} = \boldsymbol{g} + C_{(\boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}})} \boldsymbol{e}_3 \dfrac{f}{m} \\
\dot{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{b}} = \dfrac{1}{2} \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \otimes \boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}} \\
\dot{\boldsymbol{\omega}} = J^{-1}(\boldsymbol{\xi} - \boldsymbol{\omega} \times (J\boldsymbol{\omega})) \\
\dot{f}_i = \begin{cases} \tau_{\mathrm{m,up}}^{-1}(f_{i,\mathrm{ref}} - f_i) & \text{if } f_{i,\mathrm{ref}} \geq f_i \\ \tau_{\mathrm{m,dn}}^{-1}(f_{i,\mathrm{ref}} - f_i) & \text{if } f_{i,\mathrm{ref}} < f_i \end{cases} \\
f_{i,\mathrm{ref}} = \mathcal{M}_{\mathrm{f}}(u_{\mathrm{esc},i}) = k_{\mathrm{f},2} u_{\mathrm{esc},i}^2 + k_{\mathrm{f},1} u_{\mathrm{esc},i} + k_{\mathrm{f},0} \\
\mathcal{M}_{\eta}(u_{\mathrm{esc},i}) = k_{\eta,2} u_{\mathrm{esc},i}^2 + k_{\eta,1} u_{\mathrm{esc},i} + k_{\eta,0} \\
\eta_i = \mathcal{M}_{\eta}(\mathcal{M}_{\mathrm{f}}^{-1}(f_i)) \\
f = \sum_{i=1}^{4} f_i \\
\boldsymbol{\xi} = \begin{bmatrix} l_{\mathrm{arm}}(f_3 - f_4) \\ l_{\mathrm{arm}}(f_2 - f_1) \\ \eta_1 + \eta_2 - \eta_3 - \eta_4 \end{bmatrix}
\end{cases}
\tag{5.30}
$$

5.5 System Identification

The simplified FDM (5.30) parameters are m, l_{arm}, J, $\tau_{\mathrm{m,up}}$, $\tau_{\mathrm{m,dn}}$, $\{k_{\mathrm{f},2}, k_{\mathrm{f},1}, k_{\mathrm{f},0}\}$ and $\{k_{\eta,2}, k_{\eta,1}, k_{\eta,0}\}$. Table 5.1 lists these parameters' values for the AscTec Pelican quadrotor AscTec [123]. Note that, unlike m and J, the other parameters are approximations of reality that are not physical/fundamental properties of the quadrotor. They are not distinguished as measured or estimated via a hat or a tilde, since no corresponding "true value" exists. Appendix D details the complete procedure for how these parameters were obtained.

5.6 Body Rate Controller

Figure 5.1 shows that the *body rate controller* forms the innermost loop of the cascaded flight control system. The body rate controller is responsible for tracking $\boldsymbol{\omega}_{\text{ref}}$, in other words for keeping $(\boldsymbol{\omega}_{\text{ref}} - \boldsymbol{\omega})$ small. The body rate controller implementation consists of a handwritten C code interface[6] and ARM 7 compatible auto-generated C code from a Simulink model[7] of the body rate control law in Section 5.6.1. This section describes the body rate controller as follows:

- Section 5.6.1 develops the core feedback control law which determines the reference body torques;
- Section 5.6.3 introduces a prioritized thrust saturation scheme, which converts body torques to feasible thrusts while prioritizing stability;
- Section 5.6.4 describes a calibration procedure for compensating small differences in each motor's thrust and torque curves.

5.6.1 Body Rate Control Law

The control law consists of a linear part and a nonlinear feedback linearizing part Lupashin et al. [103]:

$$\boldsymbol{\xi}_{\text{ref}} = \boldsymbol{\xi}_{\text{linear}} + \hat{\boldsymbol{\omega}} \times (\tilde{J}\hat{\boldsymbol{\omega}}), \tag{5.31}$$

where $\hat{\boldsymbol{\omega}}$ denotes bias-compensated gyro measurements. Assuming that $\boldsymbol{\xi} \approx \boldsymbol{\xi}_{\text{ref}}$, we plug (5.31) into the fourth equation of (5.30) to obtain:

$$\dot{\boldsymbol{\omega}} = J^{-1}(\boldsymbol{\xi}_{\text{linear}} + \underbrace{\hat{\boldsymbol{\omega}} \times (\tilde{J}\hat{\boldsymbol{\omega}}) - \boldsymbol{\omega} \times (J\boldsymbol{\omega})}_{\approx 0}) = J^{-1}\boldsymbol{\xi}_{\text{linear}}, \tag{5.32}$$

where we assume that $\hat{\boldsymbol{\omega}} \approx \boldsymbol{\omega}$ (i.e. the gyro measurements are accurate) and $\tilde{J} \approx J$ (i.e. the identified inertia tensor is accurate). In practice, the coupling term is only relevant for high-performance applications (which ours isn't) since generally $\boldsymbol{\omega} \times (J\boldsymbol{\omega}) \ll \boldsymbol{\xi}$.

Single Axis Body Rate Plant

Given that the off-diagonal terms of $\tilde{J}$ in Table 5.1 are 2 orders of magnitude smaller than the diagonal terms, assume that $\tilde{J}$ is diagonal. Then the body rate plant[8] is given by:

$$\dot{\boldsymbol{\omega}} = \begin{bmatrix} \tilde{J}_{\text{xx}} & 0 & 0 \\ 0 & \tilde{J}_{\text{yy}} & 0 \\ 0 & 0 & \tilde{J}_{\text{zz}} \end{bmatrix}^{-1} \boldsymbol{\xi}_{\text{linear}} = \tilde{J}_{\text{diag}}^{-1}\boldsymbol{\xi}_{\text{linear}}. \tag{5.33}$$

This is a decoupled problem in the body axes, thanks to the diagonal approximation of $\tilde{J}$. Consequently, we consider first a generic body rate control problem in one dimension (i.e. one axis of (5.33)) with the plant:

$$\dot{\omega} = \frac{\xi}{J}, \tag{5.34}$$

[6] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/body_rate_controller.h`

[7] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/body_rate_controller_core.slx`

[8] Control design often uses the term *plant* to denote the dynamical model to be controlled.

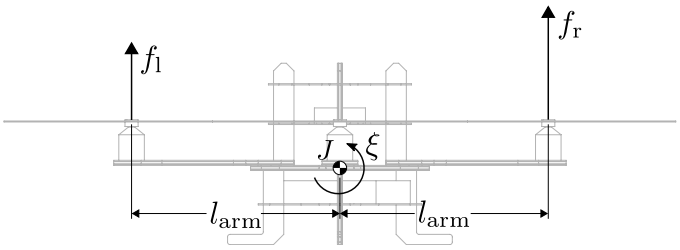


Figure 5.7: Simplified dynamics for a single axis used for decoupled body rate linear controller design. J is the moment of inertia about the axis under consideration.

where the usage of a generic ξ and J is for notational simplicity in the following control design and is not to be confused with $\boldsymbol{\xi} \in \mathbb{R}^3$, the body torque, and J the total inertia tensor. Here, J represents the moment of inertia associated with a particular axis for which the problem may be instantiated. Let us consider that it is either the $\boldsymbol{e}_\mathrm{x}^\mathrm{b}$ or the $\boldsymbol{e}_\mathrm{y}^\mathrm{b}$ axis. In this case, Figure 5.7 shows that propeller differential thrust is responsible for generating ξ. In particular, (5.34) is developed to:

$$\dot{\omega} = \frac{l_\mathrm{arm}}{J}(f_\mathrm{r} - f_\mathrm{l}) \quad \underset{\text{transform}}{\overset{\text{Laplace}}{\Rightarrow}} \quad s\omega = \frac{l_\mathrm{arm}}{J}(f_\mathrm{r} - f_\mathrm{l}), \tag{5.35}$$

where s is the complex variable associated Laplace transform. Note that our notation leaves the (s) in $\omega(s)$ and $f_*(s)$ implicit[9]. The thrust dynamics of (5.30) are simplified to a single time constant $\tau_\mathrm{m} = (\tau_\mathrm{m,dn} + \tau_\mathrm{m,up})/2$. Faessler et al. [107] find this to be a good approximation since generally for changing body torque, one motor spins up while another spins down. The thrust dynamics' transfer function is then obtained as:

$$\begin{aligned} \dot{f}_* &= \frac{1}{\tau_\mathrm{m}}(f_{*,\mathrm{ref}} - f_*) \\ \Rightarrow \quad s f_* &= \frac{1}{\tau_\mathrm{m}}(f_{*,\mathrm{ref}} - f_*) \\ \Rightarrow \frac{f_*}{f_{*,\mathrm{ref}}} &= \frac{1}{\tau_\mathrm{m} s + 1}. \end{aligned} \tag{5.36}$$

Substituting (5.36) into (5.35):

$$s\omega = \frac{l_\mathrm{arm}(f_\mathrm{r,ref} - f_\mathrm{l,ref})}{J(\tau_\mathrm{m} s + 1)}. \tag{5.37}$$

Noting that the reference body torque for the considered axis $\xi = l_\mathrm{arm}(f_\mathrm{r,ref} - f_\mathrm{l,ref})$, we finally obtain a plant usable for control design and which accounts for motor dynamics:

$$G_\omega(s) := \frac{\omega}{\xi} = \frac{1}{Js(\tau_\mathrm{m} s + 1)}. \tag{5.38}$$

We note that $G_\omega(s)$ is simply an integrator (for the body rate dynamics) in series with a low-pass filter (for the motor dynamics). Although $G_\omega(s)$ was developed for the $\boldsymbol{e}_\mathrm{x}^\mathrm{b}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{b}$ axes, the exact same reasoning shows that it also applies for the $\boldsymbol{e}_\mathrm{z}^\mathrm{b}$ axis under the assumption that $\eta_i = \kappa(f_i) = \kappa f_i$, i.e. κ is constant. This is

[9]The $*$ denotes a "catch all" wildcard.

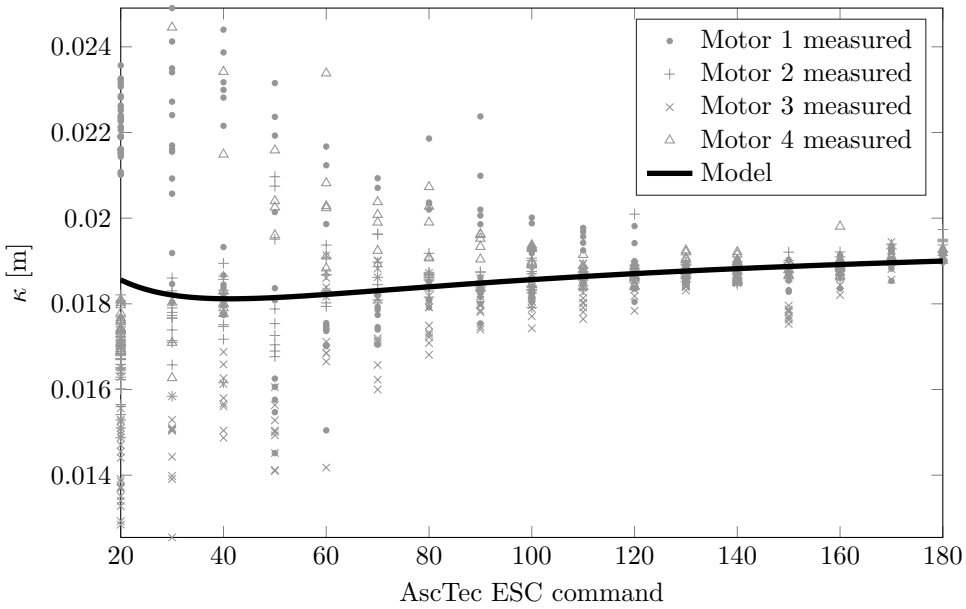


Figure 5.8: Measured and identified κ in $\eta_i = \kappa(f_i)$. The model is computed using (5.27) and the identified coefficients in Table 5.1.

the case for purely quadratic $\mathcal{M}_\mathrm{f}$ and $\mathcal{M}_\eta$ maps. Figure 5.8 shows that it is still approximately true in our case of full quadratic maps (5.25) and (5.26).

Body Rate Core Linear Feedback Control

This section designs a control law which produces $\boldsymbol{\xi}_\mathrm{linear}$ to track $\boldsymbol{\omega}_\mathrm{ref}$. Similarly to Lupashin et al. [103] and Faessler et al. [105], a proportional control law is designed:

$$\xi_\mathrm{linear} = kJ(\omega_\mathrm{ref} - \omega). \tag{5.39}$$

Figure 5.9 shows the resulting control loop. The proportional gain k is scaled by J. This allows to specify k for a "normalized" plant whose $J = 1\,\mathrm{kg}\cdot\mathrm{m}^2$ and which consequently does not need to be changed when the identified J changes[10]. Furthermore, the following delays are modeled in the control loop:

- $e^{-T_\mathrm{d}s/2}$ models the delays induced by the Zero Order Hold (ZOH) associated with the discrete implementation of the controller (see Appendix G);
- $e^{-T_\mathrm{m}s}$ models body rate measurement delay;
- $e^{-T_\mathrm{a}s}$ models delays in the communication path from the controller to the motors.

The AscTec LLP provides gyroscope measurements at 1 kHz so we set $T_\mathrm{m} = 1\,\mathrm{ms}$. Furthermore, there is 1 kHz communication link from the HLP to the LLP and from the LLP to the motor ESCs (see Figure 2.3). This motivates using $T_\mathrm{a} = 2\,\mathrm{ms}$. A common rule for choosing the discretization frequency f_sample is such that the sensitivity function $|S(j\omega)| \approx 1\ \forall\omega > 2\pi f_\mathrm{sample}$. We choose to discretize at $f_\mathrm{sample} = 200\,\mathrm{Hz}$. The ZOH-induced delay is then $T_\mathrm{d}/2 = f_\mathrm{sample}^{-1}/2 = 2.5\,\mathrm{ms}$. The total loop delay is consequently:

[10] Large changes of J may, however, trigger motor saturation or interference with the motor dynamics' time scale separation.

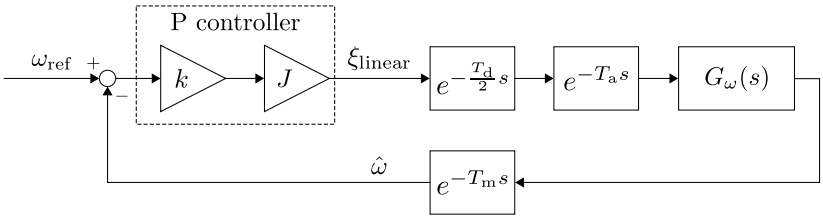


Figure 5.9: Body rate control loop for a single axis.

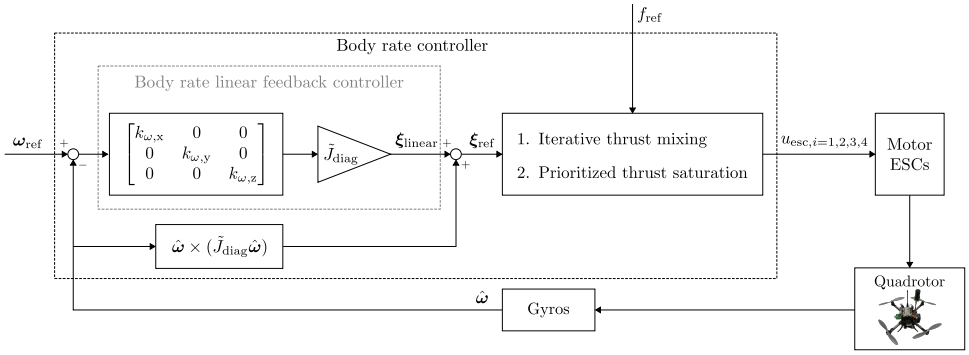


Figure 5.10: Body rate controller overall structure.

$$T_{\text{tot}} \approx T_\text{m} + T_\text{a} + \frac{T_\text{d}}{2} = 5.5\,\text{ms}. \tag{5.40}$$

The open-loop transfer function corresponding to Figure 5.9 is:

$$L_\omega(s) = kJe^{-T_{\text{tot}}s}G_\omega(s), \tag{5.41}$$

where k is the only tuning parameter. We now return to the problem of tuning (5.39) for the three body axes of the AscTec Pelican quadrotor. This boils down to selecting the proportional gains $k_{\omega,\text{x}}$, $k_{\omega,\text{y}}$ and $k_{\omega,\text{z}}$ for axes $\boldsymbol{e}_\text{x}^\text{w}$, $\boldsymbol{e}_\text{y}^\text{w}$ and $\boldsymbol{e}_\text{z}^\text{w}$ respectively. Because the quadrotor is radially quasi-symmetric, we choose $k_{\omega,\text{x}} = k_{\omega,\text{y}}$. Tuned by suspending the quadrotor via tight string along one body axis at a time, Table 5.2 summarizes the results that are supported with properties from the theoretically modeled loop in Figure 5.9. Figure 5.11 provides the Bode plots of $G_\omega(s)$, $L_\omega(s)$ with their gain margins and finally the closed-loop transfer functions. Combining (5.39) for the three axes and substituting it into the overall body rate control law (5.31), the body rate control law is given by:

$$\boldsymbol{\xi}_{\text{ref}} = \tilde{J}_{\text{diag}} \begin{bmatrix} k_{\omega,\text{x}} & 0 & 0 \\ 0 & k_{\omega,\text{y}} & 0 \\ 0 & 0 & k_{\omega,\text{z}} \end{bmatrix} (\boldsymbol{\omega}_{\text{ref}} - \hat{\boldsymbol{\omega}}) + \hat{\boldsymbol{\omega}} \times (\tilde{J}_{\text{diag}}\hat{\boldsymbol{\omega}}). \tag{5.42}$$

The remaining task is to actually reproduce $\boldsymbol{\xi}_{\text{ref}}$ by selecting the individual motors thrusts via the ESC commands $u_{\text{esc},i}$, while also satisfying f_{ref} from the attitude controller (see Figure 5.1). This is a non-trivial task for the following reasons:

1. The $\boldsymbol{\xi}_{\text{ref}}$ and f_{ref} combination may be infeasible for the motors to produce (e.g. too large requested torque);

2. The full-quadratic models (5.25) and (5.26) do not yield a closed-form allocation map from $\boldsymbol{\xi}$ and f to the f_i's due to algebraic-loop type dependence of κ on the f_i's (see Figure 5.8) Faessler et al. [107].

Variable	Value
$k_{\omega,\mathrm{x}}$	45
Bandwidth $G_{\omega,\mathrm{x}}$	39.8 rad/s
Bandwidth $L_{\omega,\mathrm{x}}$	22.6 rad/s
Bandwidth $S_{\omega,\mathrm{x}}$	18.5 rad/s
Phase margin	42°
Gain margin	16 dB

(a) ω_x, ω_y control loops (identical).

Variable	Value
$k_{\omega,\mathrm{z}}$	18.75
Bandwidth $G_{\omega,\mathrm{z}}$	35.8 rad/s
Bandwidth $L_{\omega,\mathrm{z}}$	16 rad/s
Bandwidth $S_{\omega,\mathrm{z}}$	14.6 rad/s
Phase margin	53°
Gain margin	20 dB

(b) ω_z control loop.

Table 5.2: Properties of the tuned body rate control loop in Figure 5.9. See Figure 5.11 for corresponding Bode plots.

The following sections tackle this challenge via iterative thrust allocation followed by a prioritized thrust saturation step. The overall body rate control structure from the $\boldsymbol{\omega}_\mathrm{ref}$ and f_ref inputs to the ESC command outputs (Figure 5.1) is given by Figure 5.10.

5.6.2 Iterative Thrust Mixing

The goal of iterative *thrust mixing* (or *thrust allocation*) is to convert $\boldsymbol{\xi}_\mathrm{ref}$ and T_ref into desired individual motor thrusts $T_{i=1,2,3,4}$. For this task, a crucial relationship for the i-th motor is defined Lupashin et al. [103], Faessler et al. [107]:

$$f_i = \gamma_i \bar{f}_i, \tag{5.43}$$

where f_i is the actual thrust, $\bar{f}_i$ is the thrust identified with the load-cell[11] (Appendix D) and γ_i is the *thrust factor*[12]. Assume the saturation constraint $\bar{f}_i \in [\bar{f}_\mathrm{min}, \bar{f}_\mathrm{max}]$ determined by the physical motor thrust limits and, even more strictly, the thrust range for which the thrust/torque models are valid (see Appendix D). $\bar{f}_i$ is what (5.25) and (5.26) require in order to produce valid results since these are the values that the models were constructed from while f_i is **the thrust that we actually want the i-th motor to produce**. Ideally, $\gamma_i = 1\ \forall i$ but in practice electrical and mechanical variations in the motors will lead to a γ_i only in the vicinity of 1. Thrust factor determination is termed *motor thrust calibration* and is detailed in Section 5.6.4. A *thrust allocation problem* is constructed from (5.27), (5.28) and (5.29):

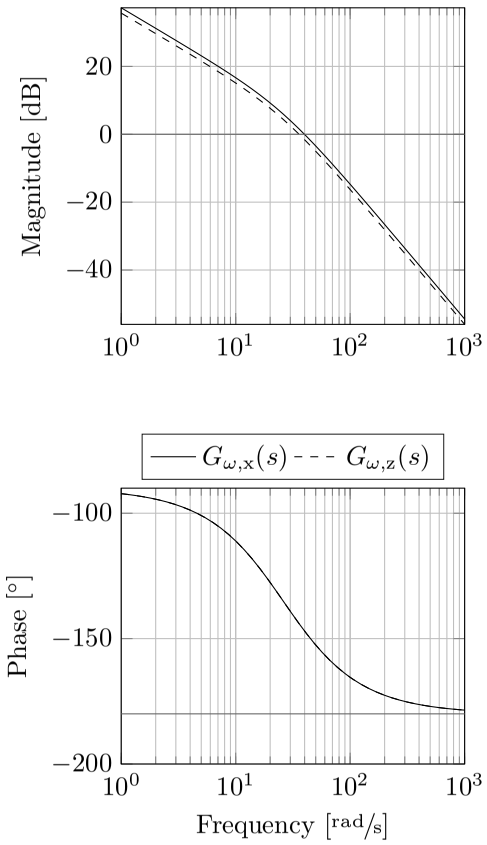
$$\begin{bmatrix} \boldsymbol{\xi}_\mathrm{ref} \\ f_\mathrm{ref} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0 & l_\mathrm{arm} & -l_\mathrm{arm} \\ -l_\mathrm{arm} & l_\mathrm{arm} & 0 & 0 \\ \kappa(\bar{f}_1) & \kappa(\bar{f}_2) & -\kappa(\bar{f}_3) & -\kappa(\bar{f}_4) \\ 1 & 1 & 1 & 1 \end{bmatrix}}_{\text{Allocation matrix } \mathcal{A}} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}. \tag{5.44}$$

Note that unlike (5.27), the $\kappa(\cdot)$ function is taken with respect to $\bar{f}_i$ the load-cell thrust. While (5.27) works with what it assumed to be the "true" thrust and torque maps, in practice only the "load-cell identified" maps are available – hence this difference. Using (5.25), (5.26) and the fact that $u_\mathrm{esc,i}$ is always positive, the $\kappa(\cdot)$ function is readily defined[13]:

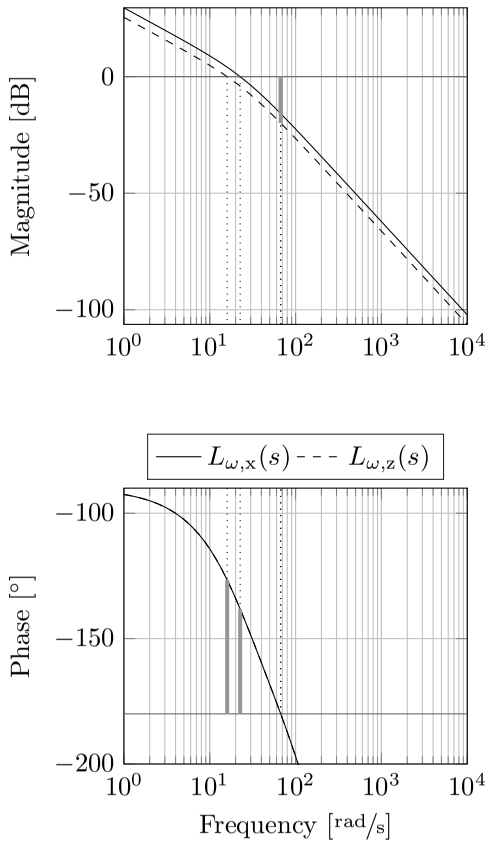
[11] Called from here on the *load-cell thrust*.

[12] Be careful to not confuse γ_i the thrust factor of the i-th motor with γ_i the i-th trajectory list element in Chapter 4. The distinction is clear from context.

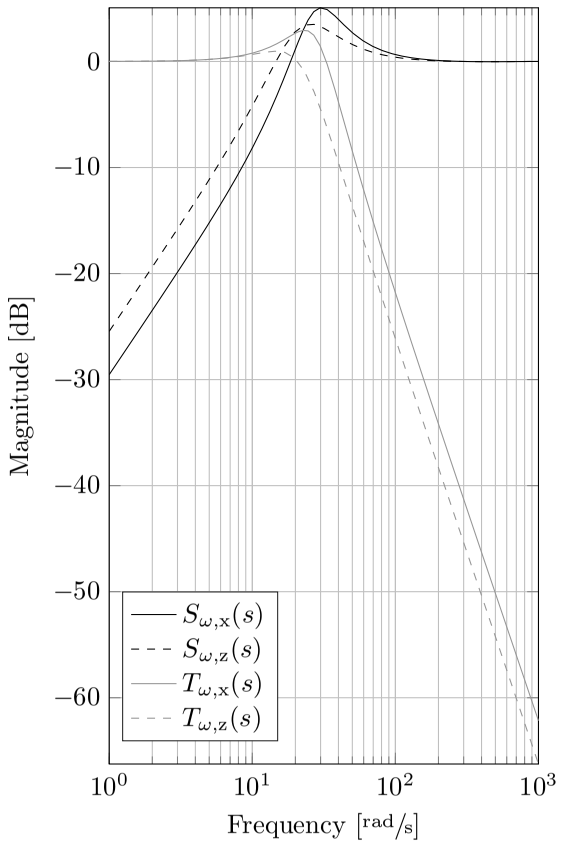
[13] Note that with respect to (5.27), this definition includes a division by f_i to cancel out the multiplication by f_i in (5.44).



(a) Bode plots of the body rate plant (5.38).



(b) Bode plots of the body rate control loops' open-loop transfer function (5.41).



(c) Bode plots of the body rate control loops' sensitivity and complementary sensitivity transfer functions.

Figure 5.11: Bode plots from the body rate controller design procedure. See Table 5.2 for numerical values.

$$\kappa(\bar{f}_i) := \frac{\mathcal{M}_\eta\left(\frac{-k_{\mathrm{f},1}+\sqrt{k_{\mathrm{f},1}^2-4k_{\mathrm{f},2}(k_{\mathrm{f},0}-\bar{f}_i)}}{2k_{\mathrm{f},2}}\right)}{f_i}. \tag{5.45}$$

Solving the thrust allocation problem (5.44) is what is known as *thrust allocation* or *thrust mixing*. When $\kappa(\bar{f}_i)$ is constant, this is readily done via inversion since $\mathcal{A}$ is guaranteed to be invertible. Although this is nearly the case for us (see Figure 5.8), in general it is not the case Faessler et al. [107]. To keep the implementation generally applicable to whatever shape the $\kappa(\cdot)$ function may take (which complements S4 of Specification List 3), we employ the iterative solution scheme proposed by Faessler et al. [107] and detailed in Algorithm 23.

Algorithm 23 Iterative thrust mixing.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/body_rate_controller_core.slx:stepBodyRateController`

$\mathcal{I} \leftarrow \{1,2,3,4\}$
$\bar{\kappa} \leftarrow 0$
$f_i \leftarrow 0.25 \cdot f_{\mathrm{ref}} \quad \forall i \in \mathcal{I}$ ▷ Initial solution
for $k \leftarrow 1$ to 3 **do** ▷ Faessler et al. [107] show that ≤ 3 iterations are needed
  $\bar{f}_i \leftarrow \gamma_i^{-1} f_i \quad \forall i \in \mathcal{I}$ ▷ Compute load-cell thrusts
  Saturate $\bar{f}_i$ to $[\bar{f}_{\min}, \bar{f}_{\max}]$
  if $k = 1$ **then**
    $\bar{f}_{i,\bar{\kappa}} \leftarrow \bar{f}_i \quad \forall i \in \mathcal{I}$
  end if
  $\kappa_i \leftarrow \kappa(\bar{f}_i) \quad \forall i \in \mathcal{I}$ ▷ Evaluate (5.45)
  Update the f_i's by solving the allocation problem:

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 0 & l_{\mathrm{arm}} & -l_{\mathrm{arm}} \\ -l_{\mathrm{arm}} & l_{\mathrm{arm}} & 0 & 0 \\ \kappa_1 & \kappa_2 & -\kappa_3 & -\kappa_4 \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{\xi}_{\mathrm{ref}} \\ f_{\mathrm{ref}} \end{bmatrix} \tag{5.46}$$

  if $\exists \gamma_i^{-1} f_i \notin [\bar{f}_{\min}, \bar{f}_{\max}]$ **then** ▷ Prioritized thrust saturation will alter yaw torque anyway
    $\bar{\kappa} \leftarrow 0.25 \cdot \sum_{i=1}^{4} \kappa(\bar{f}_{i,\bar{\kappa}})$ ▷ Compute the mean κ
    Solve (5.46) for the f_i's with $\kappa_i = \bar{\kappa} \; \forall i \in \mathcal{I}$
    break
  end if
end for

5.6.3 Prioritized Thrust Saturation

With $f_{i=1,2,3,4}$ available as outputs of the iterative thrust mixing Algorithm 23, the goal of *prioritized thrust saturation* is to ensure that $\gamma_i^{-1} f_i \in [\bar{f}_{\min}, \bar{f}_{\max}] \; \forall i = 1,2,3,4$. The simplest approach would be to clamp each f_i individually. The shortcoming of this method is that it does not guarantee that any $\boldsymbol{\xi}_{\mathrm{ref}}$ or f_{ref} is kept. As a result, violent saturation may dangerously compromise the quadrotor's stability when the $\xi_{\mathrm{ref,x}}$ or $\xi_{\mathrm{ref,y}}$ torques are largely affected. Faessler et al. [107] introduced a prioritized thrust saturation scheme to overcome this shortcoming with the following priorities for satisfying the original reference:

Highest priority $\xi_{\text{ref,x}}$ and $\xi_{\text{ref,y}}$ which are the most important since they stabilize the quadrotor and are responsible for thrust vector pointing.

Intermediate priority Collective thrust f.

Lowest priority $\xi_{\text{ref,z}}$, the least important since rotation about the thrust vector is irrelevant for the quadrotor's translational motion.

Algorithm 24 performs the prioritized thrust saturation. It follows the same steps as Faessler et al. [107] but provides more details, in particular with respect to the appropriate usage of f_i and $\bar{f}_i$ of (5.43). With the saturated f_i's output by Algorithm 24, the ESC commands are readily computed by inverting (5.25):

$$u_{\text{esc},i} = \frac{-k_{\text{f,1}} + \sqrt{k_{\text{f,1}}^2 - 4k_{\text{f,2}}(k_{\text{f,0}} - f_i/\gamma_i)}}{2k_{\text{f,2}}} \quad i = 1, 2, 3, 4. \tag{5.47}$$

The body rate controller's performance is evaluated in Section 7.4.1.

Algorithm 24 Prioritized thrust saturation algorithm.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/body_rate_controller_core.slx:stepBodyRateController`

$\mathcal{I} \leftarrow \{1, 2, 3, 4\}$
Yaw torque saturation:
j, dir, amount $\leftarrow$ checkViolation() ▷ Algorithm 25
if $j \neq 0$ **then**
  if $|\xi_{\text{ref,z}}| > \xi_{\text{z,assured}}$ **then**
    $f_j \leftarrow f_j - \text{dir} \cdot \text{amount}$ ▷ Saturates the largest violator
    Solve (5.44) with $\xi_{\text{ref,z}}$ and f_j removed (directly invertible)
    if $\bar{\kappa} = 0$ **then** ▷ Value from Algorithm 23
      $\bar{\kappa} \leftarrow 0.25 \cdot \sum_{i=1}^{4} \kappa(\hat{f}_{i,\bar{\kappa}})$ ▷ Compute the mean κ
    end if
    $\xi_{\text{ref,z}} \leftarrow \bar{\kappa}(f_1 + f_2 - f_3 - f_4)$
    if $\text{sign}(\xi_{\text{ref,z}}) \cdot \xi_{\text{z}} < \xi_{\text{z,assured}}$ **then** ▷ Torque goes in wrong direction
      $\xi_{\text{ref,z}} \leftarrow \text{sign}(\xi_{\text{ref,z}}) \cdot \xi_{\text{z,assured}}$
      Solve (5.46) for the f_i's with $\kappa_i = \bar{\kappa}\ \forall i \in \mathcal{I}$
    end if
  end if
Collective thrust saturation:
  j, dir, amount, upper_and_lower $\leftarrow$ checkViolation()
  if idx $\neq 0$ **then**
    if not upper_and_lower **then** ▷ Otherwise cannot vary collective thrust without influencing $\xi_{\text{ref,x}}$ or $\xi_{\text{ref,y}}$
      $f_i \leftarrow f_i - \text{dir} \cdot \text{amount} \quad \forall i \in \mathcal{I}$ ▷ Shifts all thrusts equally s.t. f_j is set to its closest thrust limit
    end if
  end if
end if
Perform thrust clamping: ▷ Last resort
Set $f_i \in [\gamma_i \bar{f}_{\min}, \gamma_i \bar{f}_{\max}] \quad \forall i \in \mathcal{I}$

5.6.4 Thrust Calibration

Differences in air density, propeller quality, etc. lead to the motors having slightly different thrust and torque curves from the ones identified with a load-cell in Ap-

Algorithm 25 Compute properties of the most infeasible T_i.

`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/body_rate_controller_core.slx:stepBodyRateController`

```
1: function checkViolation()
2:     idx ← 0                          ▷ Index i corresponding to the most infeasible f_i
3:     dir ← 0          ▷ Infeasibility direction indicator (violation below or above)
4:     amount ← 0                                                  ▷ Violation amount
5:     upper ← false                          ▷ true if ∃f_i s.t. f_i > γ_i f̄_max
6:     lower ← false                          ▷ true if ∃f_i s.t. f_i < γ_i f̄_min
7:     for i ← 1 to 4 do
   Check maximum thrust violation:
8:         v ← f_i − γ_i f̄_max
9:         if v > 0 then
10:            upper ← true
11:            if v > amount then           ▷ Check if this f_i is more infeasible
12:                amount ← v
13:                dir ← 1
14:                idx ← i
15:            end if
16:        end if
   Check minimum thrust violation:
17:        v ← γ_i f̄_min − f_i
18:        if v > 0 then
19:            lower ← true
20:            if v > amount then           ▷ Check if this f_i is more infeasible
21:                amount ← v
22:                dir ← −1
23:                idx ← i
24:            end if
25:        end if
26:    end for
27:    upper_and_lower ← upper and lower
28:    return idx, dir, amount, upper_and_lower
29: end function
```

pendix D. As explained in (5.43), these variations are modeled by the *thrust factors* $\gamma_{i=1,2,3,4}$. The concept of thrust factors was previously introduced in Lupashin et al. [103], Faessler et al. [104, 107]. *Thrust calibration* refers to the process of determining the γ_i's. When the quadrotor is in a perfect hover, (5.44) reduces to:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ m\|\boldsymbol{g}\| \end{bmatrix} = \begin{bmatrix} 0 & 0 & l_{\text{arm}} & -l_{\text{arm}} \\ -l_{\text{arm}} & l_{\text{arm}} & 0 & 0 \\ \kappa(\bar{f}_1) & \kappa(\bar{f}_2) & -\kappa(\bar{f}_3) & -\kappa(\bar{f}_4) \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \gamma_1 \bar{f}_1 \\ \gamma_2 \bar{f}_2 \\ \gamma_3 \bar{f}_3 \\ \gamma_4 \bar{f}_4 \end{bmatrix}, \tag{5.48}$$

which is readily solved for the γ_i's by inverting $\mathcal{A}$ (the $\bar{f}_i$'s are known via (5.25) since we of course know the $u_{\text{esc},i}$'s that we apply). However, perfect hover is not achievable and the quadrotor will always have some small motion. Consequently, instantaneous knowledge of the $\bar{f}_i$'s is not sufficient for an accurate γ_i estimate. Instead, the $\bar{f}_i$'s are recorded over a period (e.g. 20 s) and their median value is taken, denoted $\texttt{median}(\bar{f}_i)$, for $i = 1, 2, 3, 4$. Substituting the $\bar{f}_i$'s in (5.48) with $\texttt{median}(\bar{f}T_i)$, the γ_i's are readily obtained. Note that if a certain γ_i is very different from 1 (and it is not due to a bad calibration e.g. from a bad hover), it is a useful indicator of a faulty motor Lupashin et al. [103].
This computation is implemented as an offline routine[14] which the user may run as a once-in-a-while thrust calibration for the quadrotor. An online implementation as part of the autonomy engine (Chapter 6) was attempted. However, in outdoor flight the quadrotor is not still enough in hover to reliably compute the γ_i's and statistical/threshold-based tests all are either too conservative (reject valid hovers) or must be made dangerously accepting (accept bad hovers). Note that **using a bad set of γ_i's is extremely dangerous and may easily crash the quadrotor**. Wishing to keep the implementation as robust as possible, it was deemed safer to keep thrust calibration an offline-only procedure.

5.7 Attitude Controller

Figure 5.1 shows that the *attitude controller*[15] forms an intermediate loop between the translation and body rate controllers. The attitude controller is responsible for tracking the reference acceleration $\boldsymbol{a}_{\text{ref}}$ and yaw ψ_{ref} by generating a body rate reference $\boldsymbol{\omega}_{\text{ref}}$ and collective thrust reference f_{ref} for the body rate controller (Section 5.6).
The attitude controller is based on the globally asymptotically stable quaternion-based controller introduced by Brescianini et al. [106]. Our implementation, however, uses a novel alternative quaternion-only yaw control method which has the same advantages as the approach of Faessler et al. [105] (i.e. it is physically meaningful for all attitudes) while being more computationally efficient.

5.7.1 Control Algorithm Description

The attitude controller decomposes into a clean set of steps which are described in the following sections. The controller makes heavy use of frame transformations and the reader is suggested to refer to Figure 5.12 for a summary of the utilized frames and the quaternions between them. A background in quaternions is useful for understanding the following section, which may be obtained from e.g. Trawny and Roumeliotis [69], Solà [70].

[14] `/thrust_calibration/thrust_calibration.m`

[15] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/attitude_controller.h`

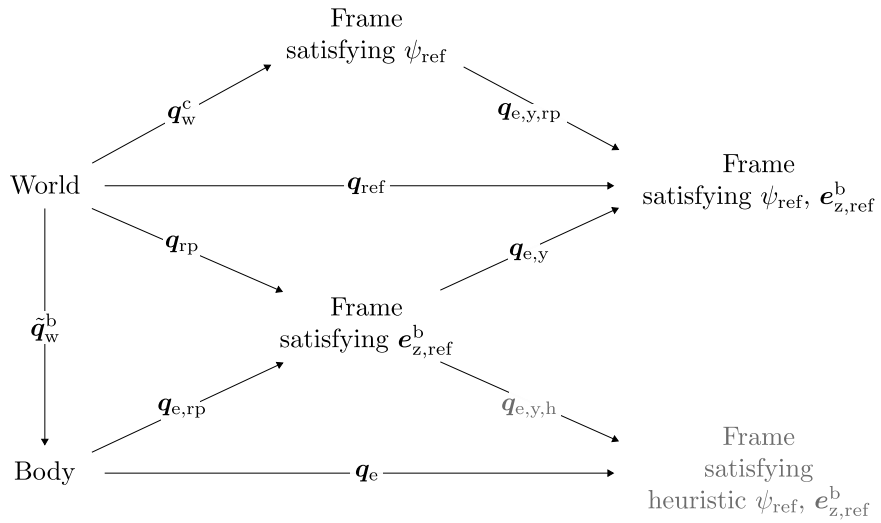


Figure 5.12: Frames and quaternions involved in the attitude controller computations.

Step 1: Thrust Vector Determination

The first step is to convert $\boldsymbol{a}_{\text{ref}}$ into a reference thrust (unit) vector $\boldsymbol{e}_{\text{z,ref}}^{\text{b}}$. Algorithm 26 performs this, handling the $\boldsymbol{a}_{\text{ref}} = 0_{3\times 1}$ singularity by retaining the previous iteration's $\boldsymbol{e}_{\text{z,ref}}^{\text{b}}$ value. Since the quadrotor must support its weight, nominally $\|\boldsymbol{a}_{\text{ref}}\| \approx m\|\boldsymbol{g}\|$. Therefore, if a singularity is ever to occur, it is expected to be a trivial-time event during which Algorithm 26 prevents undefined values from entering the rest of the algorithm.

Algorithm 26 Thrust pointing direction determination.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/attitude_controller.c:stepAttitudeController`

Initialization:
$\boldsymbol{e}_{\text{z,ref}}^{\text{b}} \leftarrow \boldsymbol{e}_3$ ▷ Pointing up
Iteration k**:**
if $\|\boldsymbol{a}_{\text{ref}}\| > 10^{-15}$ **then** ▷ Division by 0 protection
  $\boldsymbol{e}_{\text{z,ref}}^{\text{b}} \leftarrow \frac{\boldsymbol{a}_{\text{ref}}}{\|\boldsymbol{a}_{\text{ref}}\|}$
end if

Step 2: Tilt Angle Saturation

Define the *thrust vector reference tilt angle* $\alpha_{\text{tilt}} = \arccos(\boldsymbol{e}_{\text{z}}^{\text{w}} \cdot \boldsymbol{e}_{\text{z,ref}}^{\text{b}})$. Thrust vector tilt angle saturation allows one to limit the tilt angle such that $\alpha_{\text{tilt}} \leq \alpha_{\text{tilt, max}}$. Figure 5.13 shows that this effectively constraints the thrust vector to stay within a cone with an opening angle $2\alpha_{\text{tilt, max}}$. Algorithm 27 performs the tilt angle saturation.

Flight tests show that tilt angle saturation has a highly beneficial robustness side effect. As long as the attitude and body rate controllers are stable, a degraded (e.g. unstable) translation controller will not crash the quadrotor. The craft will at most "wobble" because the cone constraint will not allow $\boldsymbol{e}_{\text{z}}^{\text{b}}$ to exit an upright cone (assuming a reasonably small α_{tilt} e.g. $< 20\,^{\circ}$). In other words, **tilt angle saturation maintains an upright orientation in case of translation control**

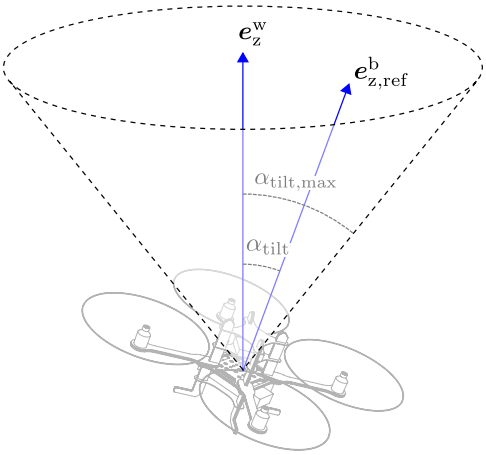


Figure 5.13: Tilt angle constraint.

instability. Note that translation control instability along e_z^w is still vulnerable but, as Section 7.4.3 shows, it occurs after instability along e_x^w and e_y^w.

Algorithm 27 Tilt angle saturation algorithm, inspired by Brescianini et al. [106].
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/attitude_controller.c:stepAttitudeController`

Initialization:

$v_{tilt} \leftarrow e_1$ ▷ Thrust vector reference tilt axis

Iteration k**:**

if $e_{z,ref}^b \cdot e_z^w < \cos(\alpha_{tilt,max})$ **then** ▷ Saturation required

Determine the normal vector to e_z^w and $e_{z,ref}^w$, i.e. the *tilt axis*:

  norm $\leftarrow \|e_z^w \times e_{z,ref}^b\|$

  if norm $\neq 0$ **then** ▷ If parallel, skip to keep last tilt axis v_{tilt}

    $v_{tilt} \leftarrow \frac{e_z^w \times e_{z,ref}^b}{\text{norm}}$

  end if

  $q_{tilt} \leftarrow \big(\cos(0.5 \cdot \alpha_{tilt,max}), v_{tilt} \sin(0.5 \cdot \alpha_{tilt,max})\big)$

  $e_{z,ref}^b \leftarrow C_{q_{tilt}} e_3$ ▷ Saturated thrust vector, in the world frame

  $a_{ref} \leftarrow (a_{ref} \cdot e_{z,ref}^b) e_{z,ref}^b$ ▷ Project a_{ref} onto saturated thrust vector

end if

Step 3: Desired Collective Thrust Computation

With the thrust vector determined, it is possible to compute the first attitude controller output, the collective thrust f_{ref}. The unsaturated collective thrust is computed by projecting a_{ref} onto the current thrust pointing direction estimate, $\tilde{e}_z^b$:

$$f_{ref,unsat} = m\|a_{ref} \cdot \tilde{e}_z^b\|. \tag{5.49}$$

$f_{ref,unsat}$ is subsequently saturated such that $f_{ref} \in [f_{min}, f_{max}]$ the user defined minimum and maximum collective thrusts[16]:

[16] The current implementation uses $f_{min} = 2\,\text{N}$ and $f_{max} = 36\,\text{N}$.

$$f_{\text{ref}} = \begin{cases} f_{\min} & \text{if } f_{\text{ref,unsat}} < f_{\min} \\ f_{\text{ref,unsat}} & \text{if } f_{\text{ref,unsat}} \in [f_{\min}, f_{\max}] \\ f_{\max} & \text{if } f_{\text{ref,unsat}} > f_{\max} \end{cases} \tag{5.50}$$

Step 4: Roll-Pitch Error Quaternion Computation

This step computes, in the body frame, the *roll-pitch error quaternion* (also known as the *reduced error quaternion*) $\boldsymbol{q}_{\text{e,rp}}$ from the current to the desired thrust vector. Because the tilt axis from $\boldsymbol{e}_3$ to any other vector is always contained in the $(\boldsymbol{e}_1, \boldsymbol{e}_2)$ plane, $q_{\text{e,rp,z}} = 0$ from which this error quaternion derives its name. This is straight forward by using Algorithm 46:

$$\boldsymbol{q}_{\text{e,rp}} = \text{computeErrorQuaternion}(\boldsymbol{e}_3, C^{-1}_{\tilde{\boldsymbol{q}}^{\text{b}}_{\text{w}}} \boldsymbol{e}^{\text{b}}_{\text{z,ref}}). \tag{5.51}$$

Step 5: Yaw Error Quaternion Computation

The yaw error quaternion is the fundamental component of yaw control and computes the *yaw error quaternion* $\boldsymbol{q}_{\text{e,y}}$. We present a novel improvement[17] over the yaw controller presented in Faessler et al. [105]. The latter has the advantage over Brescianini et al. [106] of being meaningful for any orientation of the quadrotor. However, it requires computing a quaternion from a rotation matrix, which is an involved process with several conditions Martin John Baker [71] and therefore should be avoided if possible. Our approach shows that it is in fact possible to forgo this complexity and deal only with quaternions.
Given a reference yaw ψ_{ref}, the quaternion rotating the world frame by ψ_{ref} into the *yawed world frame* $\{\text{c}\}$ is readily given by $\boldsymbol{q}^{\text{c}}_{\text{w}} = (\cos(\psi_{\text{ref}}/2), 0, 0, \sin(\psi_{\text{ref}}/2))$. Similarly to (5.51), the roll-pitch error quaternion is computed in the $\{\text{c}\}$ frame:

$$\boldsymbol{q}_{\text{e,y,rp}} = \text{computeErrorQuaternion}(\boldsymbol{e}_3, C^{-1}_{(\boldsymbol{q}^{\text{c}}_{\text{w}})} \boldsymbol{e}^{\text{b}}_{\text{z,ref}}). \tag{5.52}$$

The full reference attitude is then given by:

$$\boldsymbol{q}_{\text{ref}} = \boldsymbol{q}_{\text{e,y,rp}} \otimes \boldsymbol{q}^{\text{c}}_{\text{w}}. \tag{5.53}$$

No rotation-matrix-to-quaternion conversion has been performed to obtain $\boldsymbol{q}_{\text{ref}}$, which is our novelty over Faessler et al. [105] where $\boldsymbol{q}_{\text{ref}}$ computation involves such a conversion. The yaw error quaternion is then readily obtained:

$$\boldsymbol{q}_{\text{rp}} = \boldsymbol{q}_{\text{e,rp}} \otimes \tilde{\boldsymbol{q}}^{\text{b}}_{\text{w}} \tag{5.54}$$

$$\boldsymbol{q}_{\text{e,y}} = \boldsymbol{q}_{\text{ref}} \otimes \boldsymbol{q}^{-1}_{\text{rp}}. \tag{5.55}$$

Step 6: Apply Yaw Mixing and Yaw Direction Heuristic

The error quaternion $\boldsymbol{q}_{\text{e,y}}$ achieves a minimum-angle rotation. Two practical issues exist, however, with using $\boldsymbol{q}_{\text{e,y}}$ for control:

- Treating yaw error with the same weight as the roll-pitch error is not natural with respect to a quadrotor's available control authority. As explained in Section 5.6, roll and pitch angles are much more important (they are responsible for maintaining stability) and are easier to achieve (via motor differential

[17] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/attitude_controller.c:stepAttitudeController`

thrust). Yaw, meanwhile, has no influence on stability[18] and is much harder to achieve (via the more limited motor differential torque). A large yaw error will yield a disproportionately large $\omega_{\mathrm{ref,z}}$ and may compromise stability if prioritized thrust saturation protection of Section 5.6 is not used. Therefore, it is generally desirable to purposefully discount the yaw error;

- Minimum-angle does not imply minimum-time. Suppose that the yaw error is close to 180°. Then, for large $|\omega_{\mathrm{z}}|$ it is likely faster to achieve the desired yaw by continuing to yaw in the same direction than it is to accelerate back to the minimum-angle direction Brescianini et al. [106].

Algorithm 28 addresses both issues by computing the *heuristic yaw-error quaternion* $\boldsymbol{q}_{\mathrm{e,y,h}}$ based on Brescianini et al. [106]. To discount yaw error, a p_{att}-fraction of it is taken. To achieve the yaw in minimum time, a heuristic[19] is used based on Figure 5.14. For small ψ_{err}, a large yaw rate is required to not take the minimum-angle route while for large ψ_{err} the existing yaw rate requirement diminishes. Note that for very small ψ_{err} ($< 10\,^\circ$), this yaw heuristic is not applied.
Once the yaw direction is switched, the heuristic must ensure that it does not get switched back again. Given the attitude control law (5.58), a sufficient condition for this is:

$$\omega_{\mathrm{z,min}} < \frac{2}{\tau_{\mathrm{att}}} \sin(\frac{p_{\mathrm{att}}\pi}{2}). \tag{5.56}$$

Algorithm 28 Yaw mixing to discount the yaw error and to heuristically choose the minimum-time yaw direction.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/attitude_controller.c:stepAttitudeController`

sign_qw $\leftarrow -1$ if $q_{\mathrm{e,y,w}} < 0$, 1 otherwise
sign_qz $\leftarrow -1$ if $q_{\mathrm{e,y,z}} < 0$, 1 otherwise
$k_3 \leftarrow$ sign_qw $\cdot$ sign_qz
$\psi_{\mathrm{err}} \leftarrow 2\arccos(\text{sign\_qw} \cdot q_{\mathrm{e,y,w}})$ ▷ $\psi_{\mathrm{err}} \in [0, \pi]$ by construction
Pick yaw direction from a heuristic:
if $\psi_{\mathrm{err}} > 10\,^\circ$ **then**
  threshold $\leftarrow \frac{\pi - \psi_{\mathrm{err}}}{\pi} \omega_{\mathrm{z,min}}$ ▷ See Figure 5.14
  if $k_3\hat{\omega}_{\mathrm{z}} < 0$ **and** $-k_3\hat{\omega}_{\mathrm{z}} >$ threshold **then**
    $k_3 \leftarrow -k_3$ ▷ Reverse yaw direction
    $\psi_{\mathrm{err}} \leftarrow \pi$ ▷ Continue to yaw with maximum error, to minimize time
  end if
end if
Reduce the weight of $q_{\mathrm{e,y}}$ by taking a p_{att}-fraction of yaw error:
$\boldsymbol{q}_{\mathrm{e,y,h}} \leftarrow (\cos(\frac{p_{\mathrm{att}}\psi_{\mathrm{err}}}{2}), 0, 0, k_3 \sin(\frac{p_{\mathrm{att}}\psi_{\mathrm{err}}}{2}))$

Step 7: Compute Reference Body Rates

From the previous steps, the roll-pitch error quaternion $\boldsymbol{q}_{\mathrm{e,rp}}$ and the heuristic yaw-error quaternion $\boldsymbol{q}_{\mathrm{e,y,h}}$ are available. Define the *total error quaternion* as their composition:

$$\boldsymbol{q}_{\mathrm{e}} = \boldsymbol{q}_{\mathrm{e,y,h}} \otimes \boldsymbol{q}_{\mathrm{e,rp}}. \tag{5.57}$$

[18] This is a theoretical statement. In practice, extreme yaw rates will compromise stability if the attitude controller is not sampled fast enough to accordingly re-allocate $\boldsymbol{\omega}_{\mathrm{ref}}$.
[19] Being a heuristic, no actual minimum-time guarantee is given.

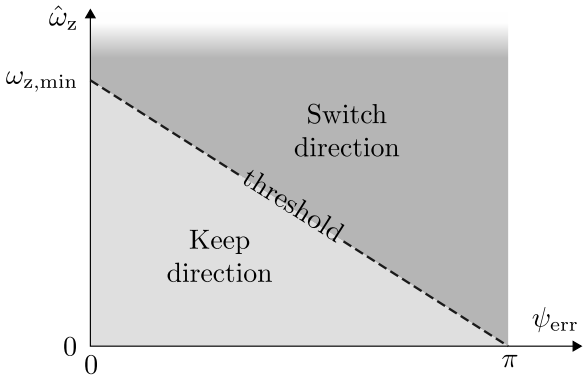


Figure 5.14: Yaw direction heuristic for minimizing time to reach the reference yaw.

The following attitude control law is applied Brescianini et al. [106]:

$$\boldsymbol{\omega}_{\text{ref}} = \frac{2}{\tau_{\text{att}}} \text{sgn}(q_{\text{e,w}}) \begin{bmatrix} q_{\text{e,x}} \\ q_{\text{e,y}} \\ q_{\text{e,z}} \end{bmatrix} + \boldsymbol{\omega}_{\text{ff}}, \quad \text{sgn}(q_{\text{e,w}}) := \begin{cases} 1 & q_{\text{e,w}} \geq 0 \\ -1 & q_{\text{e,w}} < 0 \end{cases}, \tag{5.58}$$

where $\boldsymbol{\omega}_{\text{ff}}$ is a feed-forward term and τ_{att} defines a first-order system time constant for the closed loop attitude dynamics.

Proof (that τ_{att} is the first-order attitude time constant).
Begin by substituting (5.58) into the open loop attitude dynamics (third equation of 5.30) to yield the closed loop attitude dynamics:

$$\dot{\boldsymbol{q}}_{\text{w}}^{\text{b}} = \frac{1}{2} \begin{bmatrix} 0 \\ \frac{2}{\tau_{\text{att}}} \text{sgn}(q_{\text{e,w}}) \begin{bmatrix} q_{\text{e,x}} \\ q_{\text{e,y}} \\ q_{\text{e,z}} \end{bmatrix} \end{bmatrix} \otimes \boldsymbol{q}_{\text{w}}^{\text{b}}. \tag{5.59}$$

Without loss of generality, let $\boldsymbol{q}_{\text{ref}} = \boldsymbol{q}_{\text{I}}$ Brescianini et al. [106] and let $p_{\text{att}} = 1$ such that $\boldsymbol{q}_{\text{e,y,h}} = \boldsymbol{q}_{\text{e,y}}$. It then follows from Figure 5.12 that $\boldsymbol{q}_{\text{e}} = \boldsymbol{q}_{\text{ref}} \otimes (\boldsymbol{q}_{\text{w}}^{\text{b}})^{-1} = \boldsymbol{q}_{\text{I}} \otimes (\boldsymbol{q}_{\text{w}}^{\text{b}})^{-1} = (\boldsymbol{q}_{\text{w}}^{\text{b}})^{-1} = (q_{\text{w,w}}^{\text{b}}, -q_{\text{w,x}}^{\text{b}}, -q_{\text{w,y}}^{\text{b}}, -q_{\text{w,z}}^{\text{b}})$. Substituting into (5.59):

$$\dot{\boldsymbol{q}}_{\text{w}}^{\text{b}} = \frac{1}{\tau_{\text{att}}} \text{sgn}(q_{\text{w,w}}^{\text{b}}) \begin{bmatrix} (q_{\text{w,x}}^{\text{b}})^2 + (q_{\text{w,y}}^{\text{b}})^2 + (q_{\text{w,z}}^{\text{b}})^2 \\ -q_{\text{w,w}}^{\text{b}} q_{\text{w,x}}^{\text{b}} \\ -q_{\text{w,w}}^{\text{b}} q_{\text{w,y}}^{\text{b}} \\ -q_{\text{w,w}}^{\text{b}} q_{\text{w,z}}^{\text{b}} \end{bmatrix}. \tag{5.60}$$

For small attitude tracking errors of $\boldsymbol{q}_{\text{ref}} = \boldsymbol{q}_{\text{I}}$, $\boldsymbol{q}_{\text{w}}^{\text{b}} \approx (1, 0, 0, 0)$ this reduces to a decoupled first-order system with time constant τ_{att}:

$$\dot{\boldsymbol{q}}_{\text{w}}^{\text{b}} = \frac{1}{\tau_{\text{att}}} \begin{bmatrix} 0 \\ -q_{\text{w,x}}^{\text{b}} \\ -q_{\text{w,y}}^{\text{b}} \\ -q_{\text{w,z}}^{\text{b}} \end{bmatrix}. \quad \square \tag{5.61}$$

We can then work with the approximation that τ_{att}^{-1} is the bandwidth for thrust vector tracking and, given the yaw heuristic in Algorithm 28, a $p_{\text{att}}/\tau_{\text{att}}$ bandwidth for yaw tracking. Given that attitude tracking is indeed precise as demonstrated in Section 7.4.2, this is a valid assumption.

Parameters	Value
τ_{att}	$0.25\,\text{s}$
p_{att}	0.1
$\omega_{\text{z,min}}$	$50\,^{\circ}/\text{s}$

Table 5.3: Attitude controller tuned parameters for the AscTec Pelican.

The control law (5.58) has the following beneficial properties Brescianini et al. [106]:

- The $\text{sgn}(\cdot)$ operation ensures that the quadrotor is always rotated in the direction minimizing the rotation angle necessary to achieve the desired attitude;
- $\boldsymbol{q}_{\text{e}} = 0$ is a globally asymptotically stable equilibrium point. In other words, $\forall \boldsymbol{q}_{\text{e}} \in \mathbb{S}^3$, $\lim_{t\to\infty} \|\boldsymbol{q}_{\text{e}}\| = 0$;
- The discrete implementation of (5.58) is robust to measurement noise due to the hysteresis effect created by the non-changing $\boldsymbol{\omega}_{\text{ref}}$ output between the discrete controller implementation's iterations.

Summary

Algorithm 29 combines the above steps into a single attitude control algorithm. The overall algorithm may be understood simply as composed of step 1) compute $\boldsymbol{q}_{\text{e}}$ and step 2) apply (5.58).

5.7.2 Parameter Selection

The attitude controller has three parameters: τ_{att}, p_{att} and $\omega_{\text{z,min}}$. These were tuned directly following the body rate controller, with the quadrotor still suspended via tight string along one body axis at a time. Steps in $\boldsymbol{a}_{\text{ref}}$ were input to the attitude controller and τ_{att} was chosen just smaller than the value beginning to show an oscillatory attitude response due to time scale interference with the body rate inner loop. The less critical p_{att} and $\omega_{\text{z,min}}$ were set directly in test flights to values that cause reasonably fast yaw response while still significantly discounting yaw error. Table 5.3 shows the tuned parameter values.
Note that $\tau_{\text{att}} = 0.25\,\text{s}$ yields a $\approx 4\,^{\text{rad}}/\text{s}$ bandwidth for thrust vector tracking. This is only a weak time scale separation with the $18.5\,^{\text{rad}}/\text{s}$ bandwidth of the ω_{x} and ω_{y} control (see Table 5.2). The AscTec Pelican's slow motor dynamics (Table 5.1) make it a difficult to control system when reasonably fast translation control is desired. This is because the low motor bandwidth compresses the bandwidths of the three cascade stages such that they exhibit only a weak time scale separation between each other. Whether the control difficulty is a result of the three-stage architecture choice is a potential further work avenue (e.g. by comparing performance with a two-stage cascaded architecture, see Section 8.2). The attitude controller's nevertheless good performance is presented in Section 7.4.2.

5.8 Translation Control

Figure 5.1 shows that the *translation controller* forms the outermost control loop of the cascaded flight control system. The translation controller is responsible for computing the reference acceleration $\boldsymbol{a}_{\text{ref}}$, in the world frame, based on inputs either from guidance or a human pilot. Because $\boldsymbol{a}_{\text{ref}}$ defines a reference *thrust vector* (see Section 5.7), the translation controller may be thought of as a thrust vector calculator such that the quadrotor achieves the desired translation in space. Figure 5.15

Algorithm 29 Attitude controller summary.

Initialization:

$\boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}} \leftarrow \boldsymbol{e}_3$ ▷ Pointing up

$\boldsymbol{v}_{\mathrm{tilt}} \leftarrow \boldsymbol{e}_1$ ▷ Thrust vector reference tilt axis

Iteration k:

Step 1 (Thrust Vector Determination):

if $\|\boldsymbol{a}_{\mathrm{ref}}\| > 10^{-15}$ **then** ▷ Division by 0 protection

  $\boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}} \leftarrow \frac{\boldsymbol{a}_{\mathrm{ref}}}{\|\boldsymbol{a}_{\mathrm{ref}}\|}$

end if

Step 2 (Tilt Angle Saturation):

if $\boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}} \cdot \boldsymbol{e}^{\mathrm{w}}_{\mathrm{z}} < \cos(\alpha_{\mathrm{tilt,max}})$ **then** ▷ Saturation required

Determine the normal vector to $\boldsymbol{e}^{\mathrm{w}}_{\mathrm{z}}$ and $\boldsymbol{e}^{\mathrm{w}}_{\mathrm{z,ref}}$, i.e. the *tilt axis*:

  $\mathrm{norm} \leftarrow \|\boldsymbol{e}^{\mathrm{w}}_{\mathrm{z}} \times \boldsymbol{e}^{\mathrm{w}}_{\mathrm{z,ref}}\|$

  if $\mathrm{norm} \neq 0$ **then** ▷ If parallel, skip to keep last tilt axis $\boldsymbol{v}_{\mathrm{tilt}}$

    $\boldsymbol{v}_{\mathrm{tilt}} \leftarrow \frac{\boldsymbol{e}^{\mathrm{w}}_{\mathrm{z}} \times \boldsymbol{e}^{\mathrm{w}}_{\mathrm{z,ref}}}{\mathrm{norm}}$

  end if

  $\boldsymbol{q}_{\mathrm{tilt}} \leftarrow \big(\cos(0.5 \cdot \alpha_{\mathrm{tilt,max}}), \boldsymbol{v}_{\mathrm{tilt}} \sin(0.5 \cdot \alpha_{\mathrm{tilt,max}})\big)$

  $\boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}} \leftarrow C_{\boldsymbol{q}_{\mathrm{tilt}}} \boldsymbol{e}_3$ ▷ Saturated thrust vector, in the world frame

  $\boldsymbol{a}_{\mathrm{ref}} \leftarrow (\boldsymbol{a}_{\mathrm{ref}} \cdot \boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}}) \boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}}$ ▷ Project $\boldsymbol{a}_{\mathrm{ref}}$ onto saturated thrust vector

end if

Step 3 (Desired Collective Thrust Computation):

$f_{\mathrm{ref,unsat}} \leftarrow m \|\boldsymbol{a}_{\mathrm{des}} \cdot \tilde{\boldsymbol{e}}^{\mathrm{b}}_{\mathrm{z}}\|$ ▷ (5.49)

$f_{\mathrm{ref}} \leftarrow$ apply (5.50)

Step 4 (Roll-Pitch Error Quaternion Computation):

$\boldsymbol{q}_{\mathrm{e,rp}} \leftarrow \textsc{computeErrorQuaternion}(\boldsymbol{e}_3, C^{-1}_{\tilde{\boldsymbol{q}}^{\mathrm{b}}_{\mathrm{w}}} \boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}})$ ▷ (5.51)

Step 5 (Yaw Error Quaternion Computation):

$\boldsymbol{q}_{\mathrm{e,y,rp}} \leftarrow \textsc{computeErrorQuaternion}(\boldsymbol{e}_3, C^{-1}_{(\boldsymbol{q}^{\mathrm{c}}_{\mathrm{w}})} \boldsymbol{e}^{\mathrm{b}}_{\mathrm{z,ref}}$ ▷ (5.52)

$\boldsymbol{q}_{\mathrm{ref}} \leftarrow \boldsymbol{q}_{\mathrm{e,y,rp}} \otimes \boldsymbol{q}^{\mathrm{c}}_{\mathrm{w}}$ ▷ (5.53)

$\boldsymbol{q}_{\mathrm{rp}} \leftarrow \boldsymbol{q}_{\mathrm{e,rp}} \otimes \tilde{\boldsymbol{q}}^{\mathrm{b}}_{\mathrm{w}}$ ▷ (5.54)

$\boldsymbol{q}_{\mathrm{e,y}} \leftarrow \boldsymbol{q}_{\mathrm{ref}} \otimes \boldsymbol{q}^{-1}_{\mathrm{rp}}$ ▷ (5.55)

Step 6 (Apply Yaw Mixing and Yaw Direction Heuristic):

$\mathrm{sign\_qw} \leftarrow -1$ if $q_{\mathrm{e,y,w}} < 0$, 1 otherwise

$\mathrm{sign\_qz} \leftarrow -1$ if $q_{\mathrm{e,y,z}} < 0$, 1 otherwise

$k_3 \leftarrow \mathrm{sign\_qw} \cdot \mathrm{sign\_qz}$

$\psi_{\mathrm{err}} \leftarrow 2 \arccos(\mathrm{sign\_qw} \cdot q_{\mathrm{e,y,w}})$ ▷ $\psi_{\mathrm{err}} \in [0, \pi]$ by construction

Pick yaw direction from a heuristic:

if $\psi_{\mathrm{err}} > 10\,^\circ$ **then**

  $\mathrm{threshold} \leftarrow \frac{\pi - \psi_{\mathrm{err}}}{\pi} \omega_{\mathrm{z,min}}$ ▷ See Figure 5.14

  if $k_3 \hat{\omega}_{\mathrm{z}} < 0$ **and** $-k_3 \hat{\omega}_{\mathrm{z}} > \mathrm{threshold}$ **then**

    $k_3 \leftarrow -k_3$ ▷ Reverse yaw direction

    $\psi_{\mathrm{err}} \leftarrow \pi$ ▷ Continue to yaw with maximum error, to minimize time

  end if

end if

Reduce the weight of $\boldsymbol{q}_{\mathrm{e,y}}$ by taking a p_{att}-fraction of yaw error:

$\boldsymbol{q}_{\mathrm{e,y,h}} \leftarrow (\cos(\frac{p_{\mathrm{att}} \psi_{\mathrm{err}}}{2}), 0, 0, k_3 \sin(\frac{p_{\mathrm{att}} \psi_{\mathrm{err}}}{2}))$

Step 7 (Compute Reference Body Rates):

$\boldsymbol{q}_{\mathrm{e}} \leftarrow \boldsymbol{q}_{\mathrm{e,y,h}} \otimes \boldsymbol{q}_{\mathrm{e,rp}}$ ▷ (5.57)

Apply (5.58):

$$\boldsymbol{\omega}_{\mathrm{ref}} = \frac{2}{\tau_{\mathrm{att}}} \mathrm{sgn}(q_{\mathrm{e,w}}) \begin{bmatrix} q_{\mathrm{e,x}} \\ q_{\mathrm{e,y}} \\ q_{\mathrm{e,z}} \end{bmatrix}, \quad \mathrm{sgn}(q_{\mathrm{e,w}}) := \begin{cases} 1 & q_{\mathrm{e,w}} \geq 0 \\ -1 & q_{\mathrm{e,w}} < 0 \end{cases},$$

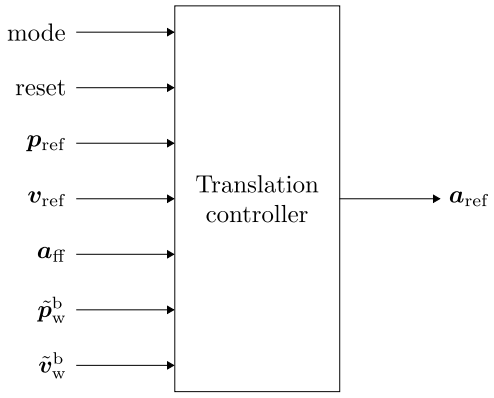


Figure 5.15: Input-output view of the translation controller.

shows a high-level view of the translation controller architecture as an input-output black box. Three modes are available as specified by the "mode" input:

1. Acceleration mode, which does open-loop acceleration $\boldsymbol{a}_{\text{ff}}$ feed-forwarding, is presented in Section 5.8.1;
2. Position mode, which tracks $\boldsymbol{p}_{\text{ref}}$, is presented in Section 5.8.2;
3. Velocity mode, which tracks $\boldsymbol{v}_{\text{ref}}$, is presented in Section 5.8.3.

Exactly one of these modes is active at any point during flight. The translation controller implementation consists of a handwritten C code interface[20] and ARM 7 compatible auto-generated C code from a Simulink model[21] of the position and velocity controllers of Sections 5.8.2 and 5.8.3.
This chapter describes the algorithms that drive each mode. Robustness analysis and tracking performance are presented in Section 7.4.3.

5.8.1 Acceleration Mode

Given an input feed-forward acceleration $\boldsymbol{a}_{\text{ff}}$, the acceleration mode computes a gravity-compensated reference acceleration in an open-loop fashion:

$$\boldsymbol{a}_{\text{ref}} = \boldsymbol{a}_{\text{ff}} - \boldsymbol{g}. \tag{5.62}$$

This mode is aimed at the following use-cases:

- Attitude and body rate inner loops' combined performance evaluation, since the translation controller is bypassed;
- External translation controller implementation (e.g. MPC running on the Odroid XU4 and sending $\boldsymbol{a}_{\text{ref}}$ over Universal Asynchronous Receiver-Transmitter (UART) directly to the attitude controller);
- Manual flight with an RC transmitter and a human pilot, in particular to stress other algorithms on the quadrotor as this mode allows more aggressive flying than the position and velocity modes.

[20] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/translation_controller.h`

[21] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/translation_controller_core.slx`

5.8.2 Position Mode

The position mode produces $\boldsymbol{a}_{\text{ref}}$ with the objective of keeping the tracking error $(\boldsymbol{p}_{\text{ref}} - \tilde{\boldsymbol{p}}_{\text{w}}^{\text{b}})$ small. Since $\boldsymbol{a}_{\text{ref}} \in \mathbb{R}^3$, only three control inputs are available and therefore at most three plant outputs may be independently controlled Skogestad and Postlethwaite [102]. The position mode chooses these outputs to be $\tilde{\boldsymbol{p}}_{\text{w}}^{\text{b}} \in \mathbb{R}^3$.

Position Dynamics Plant

Consider the first two equations of (5.30):

$$\begin{cases} \dot{\boldsymbol{p}}_{\text{w}}^{\text{b}} = \boldsymbol{v}_{\text{w}}^{\text{b}} \\ \dot{\boldsymbol{v}}_{\text{w}}^{\text{b}} = \boldsymbol{g} + C_{(\boldsymbol{q}_{\text{w}}^{\text{b}})} \boldsymbol{e}_3 \dfrac{f}{m}. \end{cases} \tag{5.63}$$

If the attitude controller perfectly tracks the desired thrust vector, $C_{(\boldsymbol{q}_{\text{w}}^{\text{b}})} \boldsymbol{e}_3 \frac{T}{m} \approx \boldsymbol{a}_{\text{ref}}$. This assumption is valid for sufficient time scale separation from the attitude controller. Define the *gravity-compensated reference acceleration* $\bar{\boldsymbol{a}}_{\text{ref}}$:

$$\bar{\boldsymbol{a}}_{\text{ref}} = \boldsymbol{a}_{\text{ref}} + \boldsymbol{g}. \tag{5.64}$$

The system (5.63) then reduces to a double integrator:

$$\ddot{\boldsymbol{p}}_{\text{w}}^{\text{b}} = \bar{\boldsymbol{a}}_{\text{ref}}. \tag{5.65}$$

This is a 3-input ($\bar{\boldsymbol{a}}_{\text{ref}}$) 3-output ($\boldsymbol{p}_{\text{w}}^{\text{b}}$) Multi-Input Multi-Output (MIMO) plant to control. However, it is decoupled in all three $\boldsymbol{e}_{\text{x}}^{\text{w}}$, $\boldsymbol{e}_{\text{y}}^{\text{w}}$ and $\boldsymbol{e}_{\text{z}}^{\text{w}}$ axes which prompts a decoupled control approach. In particular, we simplify the control problem to that of a Single-Input Single-Output (SISO) plant corresponding to any one of the axes:

$$\ddot{p} = \bar{a}, \tag{5.66}$$

where p and $\bar{a}$ are the components along the axis of $\boldsymbol{p}_{\text{w}}^{\text{b}}$ and $\bar{\boldsymbol{a}}_{\text{ref}}$ respectively. The Laplace transform gives a simple double integrator plant:

$$G_{\text{p}}(s) := \frac{1}{s^2}. \tag{5.67}$$

Note that the attitude inner loop dynamics are ignored, which is valid for sufficient time scale separation. They shall be re-introduced in Section 7.4.3 for controller robustness analysis.

Control Approach

We use Proportional Integral Derivative (PID) type controllers to control the plant (5.67). Identical gains shall be used for the symmetric dynamics in the $\boldsymbol{e}_{\text{x}}^{\text{w}}$ and $\boldsymbol{e}_{\text{y}}^{\text{w}}$ axes while a more aggressive set of gains shall be used for the $\boldsymbol{e}_{\text{z}}^{\text{w}}$ axis. Position along the latter axis can be controlled faster by simply changing the motor thrusts rather than requiring to tilt the thrust vector. Two control architectures are designed:

- A 1 DOF parallel PID architecture which allows faster and more precise position tracking but requires $\boldsymbol{p}_{\text{ref}}$ evolution to be dynamically feasible in order to avoid overshoots;
- A 2 DOF parallel PID architecture which pre-filters the reference to remove overshoot for any $\boldsymbol{p}_{\text{ref}}$, but introduces a tracking lag.

Note that we do not venture into more advanced control approaches like MPC due to S1 of Specification List 3. For robustness' sake, the controller is best kept on the highly reliable AscTec HLP. MPC implementation on the computationally constrained ARM7 processor would be non-trivial. Projects requiring more advanced position control can implement MPC or other such approaches via the acceleration mode (Section 5.8.1).

1 DOF Parallel PID Control Architecture

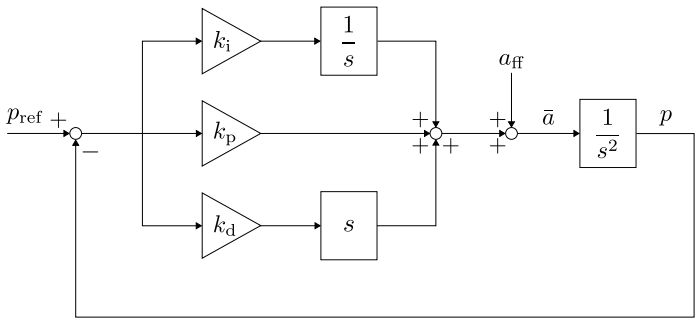
Figure 5.16 develops the 1 DOF parallel PID architecture. The final architecture is shown in Figure 5.16d. Note that the tilde notation for the output estimate is not used as we do not model the state estimator in the control design. The actual position controller implementation uses the estimate $\tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}}$.
A parallel form PID controller is used, whose transfer function is given by:

$$K_{\mathrm{p}}(s) = k_{\mathrm{p}} + \frac{k_{\mathrm{i}}}{s} + k_{\mathrm{d}}s, \tag{5.68}$$

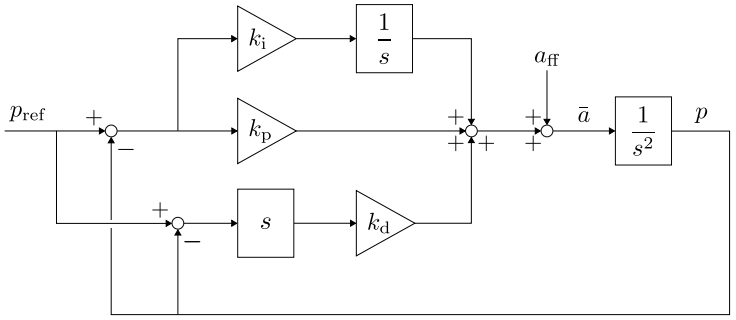
where k_{p}, k_{i} and k_{d} are the proportional, integral and derivative gains respectively. The primary challenge with implementing (5.68) is the derivative s. Because numerical differentiation amplifies noise, s is often placed in series with a low-pass filter such that $s/(s/\omega_{\mathrm{d}} + 1)$ is used instead where ω_{d} is the low-pass filter's cutoff frequency. In the present case of position control, however, we can do better by using directly the velocity estimate $\tilde{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}}$ and the known $\boldsymbol{p}_{\mathrm{ref}}$ derivative (i.e. $\boldsymbol{v}_{\mathrm{ref}}$) from guidance. As a result, Figure 5.16 shows a series of equivalence steps to arrive at an architecture where the s block is implicit.
The 1 DOF architecture is advantageous for quadrotors flying aggressive dynamically feasible trajectories because it does not pre-filter the reference, thus has less tracking lag. When the quadrotor is well calibrated, the integral term may even be removed ($k_{\mathrm{i}} = 0$). However, the position response is prone to overshoot for a dynamically infeasible reference signal (similar to Figure 5.18a). Flight tests have shown a 2 DOF architecture to work better for leisure or generic (calm) flying, which matches our use case of a data acquisition mission. This architecture is described next.

2 DOF Parallel PID Control Architecture

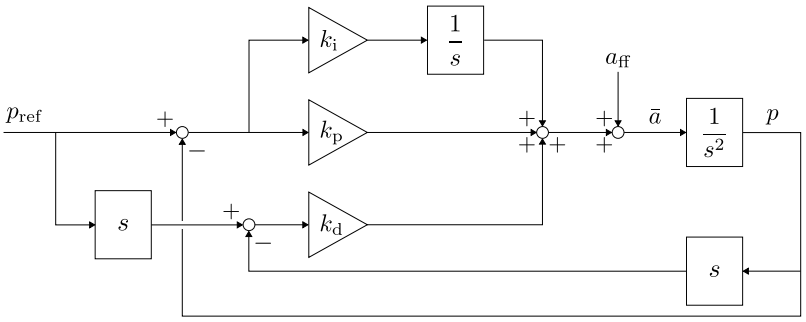
The 2 DOF architecture consists of a parallel PID and a *reference pre-filter*. The pre-filter allows to match an ideal complementary sensitivity transfer function.
The original architecture is shown in Figure 5.17a. With respect to Figure 5.16a, a p_{ref} pre-filter $F_{\mathrm{p}}(s)$ is added and a_{ff} is removed. The latter decision is based on flight tests which showed that due to the additional negative phase introduced by the pre-filter, a feed-forward acceleration term is actually detrimental to performance when following a trajectory.
The architecture in Figure 5.17a, however, suffers from the same problem of s block presence. By applying a similar set of steps as in Figure 5.16, Figure 5.17b shows an equivalent architecture with the s block made implicit.
We make a crucial design choice to remove an explicit velocity reference v_{ref}. Instead, the kinematic relationship $v_{\mathrm{ref}} = \dot{p}_{\mathrm{ref}}$ is enforced via a derivative in the *reference rate of change pre-filter* $F_{\dot{\mathrm{p}}}(s) := F_{\mathrm{p}}(s)s$. This choice is motivated by the fact that it is robust to a non-differentiable p_{ref}. Although $F_{\mathrm{p}}(s)s \equiv sF_{\mathrm{p}}(s)$ and $sp_{\mathrm{ref}} \equiv v_{\mathrm{ref}}$, this theoretical result breaks down in practice when p_{ref} is e.g. a step. In this case, the *derivative-first* approach yields $F_{\mathrm{p}}(s)(sp_{\mathrm{ref}}) = F_{\mathrm{p}}(s)\delta$ (a pre-filtered unit impulse) while the *derivative-second* approach yields $s(F_{\mathrm{p}}(s)p_{\mathrm{ref}})$ (a



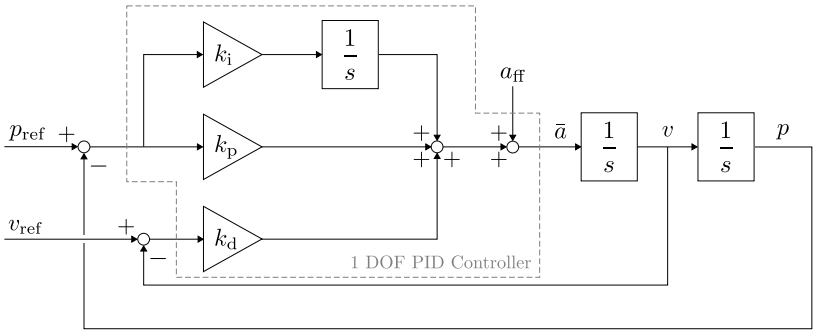
(a) The original architecture where the parallel-form PID is clearly visible.



(b) Equivalent architecture with k_d and s swapped and the summer duplicated.

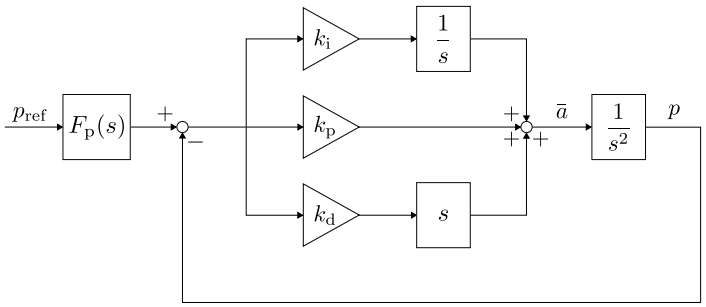


(c) Equivalent architecture where the s block is moved to before the summer.

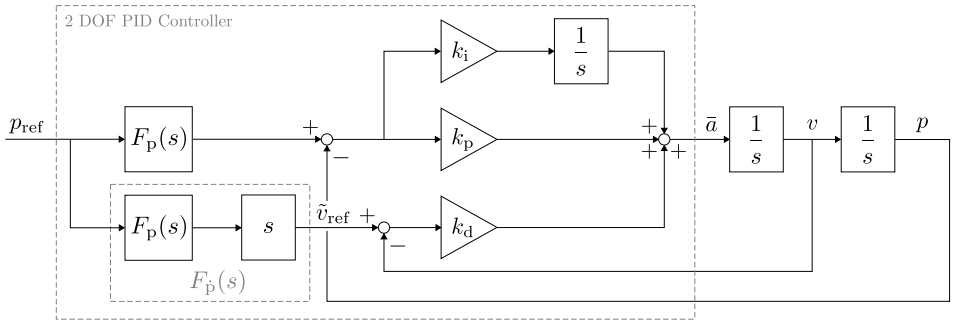


(d) Final architecture which assumes $\dot{p}_\mathrm{ref} = v_\mathrm{ref}$ and exploits the dynamics $\dot{p} = v$. The s block is now implicit.

Figure 5.16: Equivalence steps for transforming a parallel form PID controller into an architecture where the D term is implicit.



(a) The original 2 DOF architecture.



(b) Equivalent final architecture where the s block is now implicit.

Figure 5.17: Original and implementation-suitable versions of the 2 DOF parallel PID with a reference pre-filter.

smooth signal with a small jump at the start). In other words, taking the derivative *after* the pre-filter is practically achievable even for a non-differentiable reference. The advantage of the derivative-second approach is apparent in Figure 5.18 which simulates the unit step response of Figure 5.17b for the derivative-first and derivative-second arrangements. An arrangement using v_{ref} directly is also shown, where $v_{\text{ref}} = 0$ as it would be in practice for a step (e.g. imagine commanding the quadrotor to a hover point at some distant location). As seen in Figure 5.18a, only the derivative-second architecture matches the ideal response.

We now turn to the subject of determining $F_{\text{p}}(s)$ and $F_{\dot{\text{p}}}(s)$. Since Figures 5.17a and 5.17b are equivalent, we shall consider Figure 5.17a from which it is more straightforward to derive transfer functions. The *loop transfer function* (also known as the *return ratio*) of Figure 5.17a is composed of all the elements in the loop path:

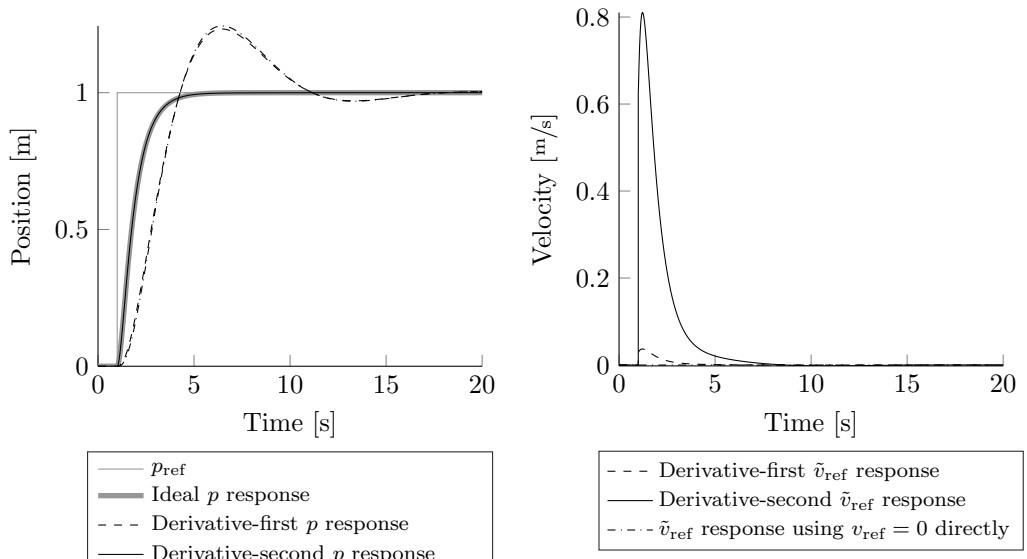
$$L_{\text{p}}(s) := K_{\text{p}}(s)G_{\text{p}}(s) = \left(k_{\text{p}} + \frac{k_{\text{i}}}{s} + k_{\text{d}}s\right)\frac{1}{s^2}. \tag{5.69}$$

The complementary sensitivity is given by:

$$T_{\text{p}}(s) = \frac{p}{p_{\text{ref}}} = \frac{L_{\text{p}}(s)F_{\text{p}}(s)}{1 + L_{\text{p}}(s)}. \tag{5.70}$$

We wish to match an ideal complementary sensitivity corresponding to a second-order system, represented by its canonical unity steady-state gain form:

$$T_{\text{p,ideal}}(s) := \frac{\omega_{\text{n}}^2}{s^2 + 2\zeta\omega_{\text{n}}s + \omega_n^2}, \tag{5.71}$$



(a) p responses to p_{ref} unit step.

(b) $\tilde{v}_{\text{ref}}$ responses to p_{ref} unit step.

Figure 5.18: Unit step responses of p and $\tilde{v}_{\text{ref}}$ for variations of the 2 DOF PID control architecture in Figure 5.17b. Using control gains in Table 5.4.

where ω_{n} is the natural frequency and ζ is the dimensionless damping ratio. $\zeta < 1$ gives an underdamped response, $\zeta = 1$ gives a critically damped response and $\zeta > 1$ gives an overdamped response. Increasing ω_{n} increases the bandwidth while increasing ζ decreases it (but for $\zeta \geq 1$, the step response has no oscillations). We compute $F_{\text{p}}(s)$ via model matching by making $T_{\text{p}}(s) = T_{\text{p,ideal}}(s)$:

$$T_{\text{p}}(s) = \frac{L_{\text{p}}(s)F_{\text{p}}(s)}{1 + L_{\text{p}}(s)} = T_{\text{p,ideal}}(s) \;\Rightarrow\; F_{\text{p}}(s) = \frac{(1 + L_{\text{p}}(s))T_{\text{p,ideal}}(s)}{L_{\text{p}}(s)}. \tag{5.72}$$

The *relative degree* of a transfer function is its number of poles minus its number of zeros, which is also equal to the denominator order minus the numerator order. A transfer function is *proper*, and therefore is *realizable* (i.e. can be practically implemented on-line), if the relative degree ≥ 0. A *semi-proper* transfer function has relative degree 0. In the present case, $T_{\text{p,ideal}}(s)$ has relative degree 2 and $L_{\text{p}}(s)$ has relative degree 1, making $F_{\text{p}}(s)$ relative degree 1. Therefore, $F_{\text{p}}(s)$ is realizable which means that we can match $T_{\text{p,ideal}}(s)$ exactly[22]. Substituting (5.69) and (5.71) into (5.72) and simplifying, we obtain the direct expression for $F_{\text{p}}(s)$:

$$F_{\text{p}}(s) = \frac{\frac{\omega_{\text{n}}^2}{k_{\text{d}}}s^3 + \omega_{\text{n}}^2 s^2 + \frac{\omega_{\text{n}}^2 k_{\text{p}}}{k_{\text{d}}}s + \frac{\omega_{\text{n}}^2 k_{\text{i}}}{k_{\text{d}}}}{s^4 + (\frac{k_{\text{p}}}{k_{\text{d}}} + 2\zeta\omega_{\text{n}})s^3 + (\frac{k_{\text{i}} + 2\zeta\omega_{\text{n}}k_{\text{p}}}{k_{\text{d}}} + \omega_{\text{n}}^2)s^2 + \frac{2\zeta\omega_{\text{n}}k_{\text{i}} + \omega_{\text{n}}^2 k_{\text{p}}}{k_{\text{d}}}s + \frac{\omega_{\text{n}}^2 k_{\text{i}}}{k_{\text{d}}}}. \tag{5.73}$$

The velocity pre-filter is then given by:

$$F_{\dot{\text{p}}}(s) = F_{\text{p}}(s)s, \tag{5.74}$$

[22]This is a theoretical statement relying on a perfect plant and a continuous-time implementation. The complementary sensitivity of the real system, implemented in discrete-time with uncertainties in the quadrotor dynamics (notwithstanding the ignored inner loop dynamics), will not exactly match $T_{\text{p,ideal}}(s)$. Nevertheless, Section 7.4.3 shows that this is close to being the case.

Parameter	Value
k_p	6
k_i	3
k_d	10
ω_n	2.5
ζ	1.2
Phase Margin	86.6 °
Gain Margin	−26 dB

(a) Tuned parameters for position control along the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ axes.

Parameter	Value
k_p	12
k_i	3
k_d	9
ω_n	8
ζ	1.2
Phase Margin	81.6 °
Gain Margin	−31.1 dB

(b) Tuned parameters for position control along the $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ axis.

Table 5.4: Position control tuned parameters.

which is semi-proper and therefore is still realizable. Semi-proper transfer functions have a direct input feedthrough component which means that the output of $F_{\dot{\mathrm{p}}}(s)$ will contain an initial jump for a non-zero input, which is visible at $t = 1\,\mathrm{s}$ in Figure 5.18b.
Summarizing, the theoretical block-diagram version of the 2 DOF PID control architecture is given by Figure 5.17b where $F_\mathrm{p}(s)$ is given by (5.73) and $F_{\dot{\mathrm{p}}}(s)$ is given by (5.74).

Parameter Selection

The 1 DOF PID architecture parameters are k_p, k_i and k_d. The 2 DOF PID architecture additionally has the ω_n and ζ parameters. Our approach is to use the same k_p, k_i and k_d values for both architectures and, on top of these, to set the ω_n and ζ values for the 2 DOF architecture. The tuning procedure is explained in Appendix F. Table 5.4 lists the resulting tuned parameters for the AscTec Pelican. Table 5.4 shows that the tuned loop provides adequate gain and phase margins. $G_\mathrm{p}(s)$ being an unstable system, however, a Nyquist plot is a more illustrative robustness indicator. Figure 5.19 plots a logarithmic Nyquist plot[23] which has the advantage of being fully contained in a circle of radius 2 Zanasi and Grossi [124]. To achieve this, the magnitude M of the plotted transfer function is remapped to

$$L_2(M) := \begin{cases} M^{\log_{10}(2)} & \text{for } M \leq 1 \\ 2 - M^{-\log_{10}(2)} & \text{for } M > 1. \end{cases} \tag{5.75}$$

It can be seen in 5.19 that the Nyquist plot stays well away from -1 and that the closed-loop system is stable (via the Nyquist criterion, since $L_\mathrm{p}(s)$ has no open Right-Half Plane (RHP) poles and the Nyquist plot encircles -1 a net zero number of times). Robustness will be further investigated in Section 7.4.3 where it is shown that despite the good gain and phase margins, position control becomes oscillatory as soon as any measurement delay is introduced and becomes unstable for measurement delays on the order of 0.1 s. This shall be linked to the variable performance observed during flight tests and a suggestion for a rigorous latency analysis will be suggested in Section 8.2.

5.8.3 Velocity Mode

The velocity mode produces $\boldsymbol{a}_\mathrm{ref}$ with the objective of keeping the tracking error $(\boldsymbol{v}_\mathrm{ref} - \tilde{\boldsymbol{v}}_\mathrm{w}^\mathrm{b})$ small. Like for the position mode, since $\boldsymbol{a}_\mathrm{ref} \in \mathbb{R}^3$, only three control

[23] All other Nyquist plots in this report also use this representation.

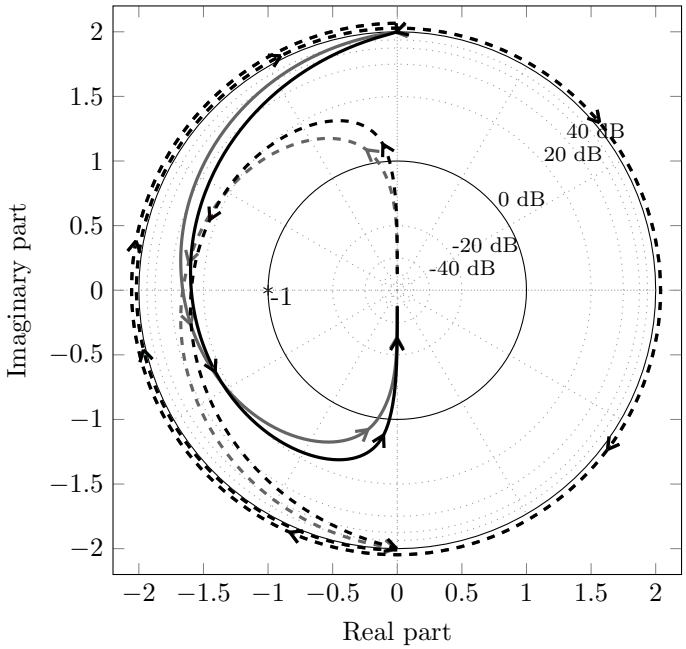


Figure 5.19: $L_{\mathrm{p}}(s)$ Nyquist plot for the $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{w}}$, $\boldsymbol{e}_{\mathrm{y}}^{\mathrm{w}}$ (black) and $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}}$ (gray). Dotted lines show negative frequencies.

inputs are available and therefore at most three outputs may be independently controlled Skogestad and Postlethwaite [102]. The velocity mode chooses these outputs to be $\tilde{\boldsymbol{v}}_{\mathrm{w}}^{\mathrm{b}} \in \mathbb{R}^3$. The reader shall note that the control design is very similar to the position mode, thus the following description shall read very similarly to Section 5.8.2.

Velocity Dynamics Plant

The same decoupled control architecture of Section 5.8.2 is used. Namely, three SISO controllers shall be designed for the $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{w}}$, $\boldsymbol{e}_{\mathrm{y}}^{\mathrm{w}}$ and $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}}$ axes. Let v be the velocity along a given axis, then $\dot{p} = v$ using the notation of (5.66). Consequently, the SISO differential equation to be controlled is:

$$\dot{v} = \bar{a}, \tag{5.76}$$

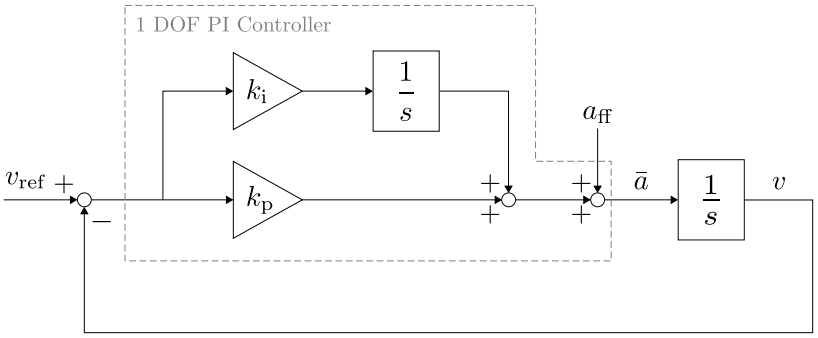
where v and $\bar{a}$ are components along a given axis of respectively the world-frame velocity, $\boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}$, and the gravity-compensated reference acceleration, $\bar{\boldsymbol{a}}_{\mathrm{ref}}$. The Laplace transform gives a simple integrator plant:

$$G_{\mathrm{v}}(s) := \frac{1}{s}. \tag{5.77}$$

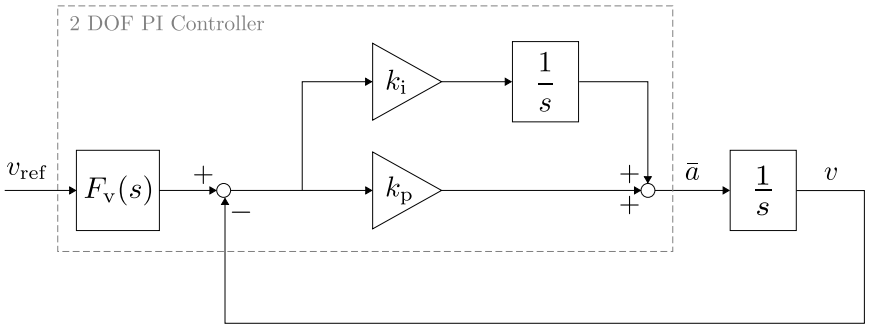
As for the position mode, the attitude inner loop dynamics are ignored, which is valid for sufficient time scale separation. They shall be re-introduced in Section 7.4.3 for controller robustness analysis.

Control Approach

Proportional Integral (PI) type controllers are used to control the plant (5.77). The derivative term is omitted because making the s block implicit as in Section 5.8.2 would require using the very noisy accelerometer signal. Identical gains shall be



(a) 1 DOF PI controller architecture for velocity tracking.



(b) 2 DOF PI controller architecture for velocity tracking.

Figure 5.20: 1 DOF and 2 DOF control architectures for velocity tracking.

used for the symmetric dynamics in the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ axes while a more aggressive set of gains shall be used for the $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ axis. Velocity along the latter axis can be controlled faster by simply changing the motor thrusts rather than requiring to tilt the thrust vector. Two control architectures (Figure 5.20) are designed:

- A 1 DOF PI architecture which allows faster and more precise velocity tracking but requires $\boldsymbol{v}_\mathrm{ref}$ evolution to be dynamically feasible in order to avoid overshoots;
- A 2 DOF PI architecture which pre-filters the reference to remove overshoot for any $\boldsymbol{v}_\mathrm{ref}$, but introduces a tracking lag.

1 DOF PI Control Architecture

The 1 DOF PI control architecture's theoretical block diagram is shown in Figure 5.20a. Note that the tilde notation for the output estimate is not used as we do not model the state estimator in the control design. The actual velocity controller uses the estimate $\tilde{\boldsymbol{v}}_\mathrm{w}^\mathrm{b}$.
The PI controller transfer function is nothing but (5.68) with $k_\mathrm{d} = 0$:

$$K_\mathrm{v}(s) = k_\mathrm{p} + \frac{k_\mathrm{i}}{s}, \tag{5.78}$$

where k_p and k_i are the proportional and integral gains respectively. Since there is no derivative term, implementing (5.78) is straight forward but the integral term requires anti-windup protection as explained in Section 5.8.4.
Like for the position mode, the 1 DOF architecture is advantageous for quadrotors flying aggressive dynamically feasible trajectories because it does not pre-filter the reference, thus has less tracking lag. When the quadrotor is well calibrated, the integral term may even be removed ($k_\mathrm{i} = 0$). However, the velocity response is prone to overshoot for a dynamically infeasible reference signal. Flight tests have shown a 2 DOF architecture to work better for leisure or generic (calm) flying, which matches our use case of a data acquisition mission. This architecture is described next.

2 DOF PI Control Architecture

The 2 DOF architecture, shown in Figure 5.20b, consists of a PI controller and a reference pre-filter. With respect to Figure 5.20a, a v_ref pre-filter $F_\mathrm{v}(s)$ is added and a_ff is removed (for the same reason as for the position mode, it degrades performance due to being out of phase with the pre-filtered reference due to the negative phase of $F_\mathrm{v}(s)$).
We now determine $F_\mathrm{v}(s)$. The loop transfer function of Figure 5.20b is:

$$L_\mathrm{v}(s) := K_\mathrm{v}(s)G_\mathrm{v}(s) = \left(k_\mathrm{p} + \frac{k_\mathrm{i}}{s}\right)\frac{1}{s}. \tag{5.79}$$

The complementary sensitivity is given by:

$$T_\mathrm{v}(s) = \frac{v}{v_\mathrm{ref}} = \frac{L_\mathrm{v}(s)F_\mathrm{v}(s)}{1 + L_\mathrm{v}(s)}. \tag{5.80}$$

We wish to match an ideal complementary sensitivity corresponding to a first-order system with unity DC gain:

$$T_\mathrm{v,ideal}(s) := \frac{1}{\tau s + 1}, \tag{5.81}$$

where τ is the time constant of the ideal response. We compute $F_\mathrm{v}(s)$ via model matching by making $T_\mathrm{v}(s) = T_\mathrm{v,ideal}(s)$:

$$T_\mathrm{v}(s) = \frac{L_\mathrm{v}(s)F_\mathrm{v}(s)}{1 + L_\mathrm{v}(s)} = T_\mathrm{v,ideal}(s) \;\Rightarrow\; F_\mathrm{v}(s) = \frac{(1 + L_\mathrm{v}(s))T_\mathrm{v,ideal}(s)}{L_\mathrm{v}(s)}. \tag{5.82}$$

Both $T_\mathrm{v,ideal}(s)$ and $L_\mathrm{v}(s)$ have relative degree 1, making $F_\mathrm{v}(s)$ relative degree 0. Therefore, $F_\mathrm{v}(s)$ is realizable which means that we can match $T_\mathrm{v,ideal}(s)$ exactly[24] Substituting (5.79) and (5.81) into (5.82) and simplifying, we obtain the direct expression for $F_\mathrm{v}(s)$:

$$F_\mathrm{v}(s) = \frac{\frac{1}{\tau k_\mathrm{p}}s^2 + \frac{1}{\tau}s + \frac{k_\mathrm{i}}{\tau k_\mathrm{p}}}{s^2 + (\frac{1}{\tau} + \frac{k_\mathrm{i}}{k_\mathrm{p}})s + \frac{k_\mathrm{i}}{\tau k_\mathrm{p}}}. \tag{5.83}$$

Summarizing, the theoretical block-diagram version of the 2 DOF PI control architecture is given by Figure 5.20b where $F_\mathrm{v}(s)$ is given by (5.83).

[24] This is a theoretical statement and the same caveats apply as for the position controller.

Parameter	Value
k_p	6
k_i	2
τ	0.7
Phase Margin	86.8 °
Gain Margin	− inf dB

(a) Tuned parameters for velocity control along the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ axes.

Parameter	Value
k_p	7
k_i	2
τ	0.2
Phase Margin	87.7 °
Gain Margin	− inf dB

(b) Tuned parameters for velocity control along the $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ axis.

Table 5.5: Velocity control tuned parameters.

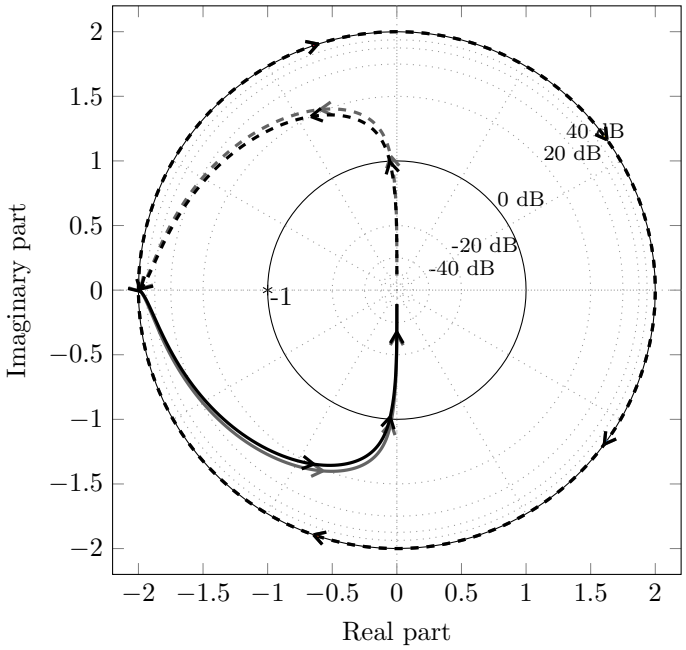


Figure 5.21: $L_\mathrm{v}(s)$ Nyquist plot for the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$, $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ (black) and $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ (gray). Dotted lines show negative frequencies.

Parameter Selection

The 1 DOF PI architecture parameters are k_p and k_i. The 2 DOF PI architecture additionally has the τ parameter. Our approach is to use the same k_p and k_i values for both architectures and, on top of these, to set the τ values for the 2 DOF architecture. The tuning procedure is explained in Appendix F. Table 5.5 lists the resulting tuned parameters for the AscTec Pelican.

Table 5.5 shows that the tuned loop provides good gain and phase margins. The robustness is verified by the Nyquist plot in Figure 5.21, which stays well away from -1 and is closed-loop stable (via the Nyquist criterion, since $L_\mathrm{v}(s)$ has no open RHP poles and the Nyquist plot does not encircle -1). Robustness will be further investigated in Section 7.4.3 where it is shown that velocity tracking is not oscillatory for up to $\approx 40\,\mathrm{ms}$ of measurement delay, leading to almost no observations of velocity oscillations in real flight tests.

5.8.4 Implementation Details

A good translation controller implementation has the following features:

1. The user can seamlessly switch between the 1 DOF and 2 DOF control architectures mid-flight;
2. The user can seamlessly modify control gains mid-flight;
3. Switching between acceleration, position and velocity control modes is *bumpless*, meaning that the control output derivatives must coincide at the time of switching Åström and Hägglund [125].

The first and second features relate to real-time control gain tuning during flight tests. It is desirable that the user may be able to safely change control parameters during a flight test in order to facilitate the gain tuning process. The third feature is crucial for the autonomy mission (Chapter 6) since the quadrotor changes between the velocity and position modes post-takeoff. The following sections describe implementation details which enable these features.

Resettable and Real Time Modifiable Pre-Filters

The 2 DOF architectures presented in Sections 5.8.2 and 5.8.3 both make use of reference pre-filters. Three challenges are associated with their implementation:

- The pre-filters were developed in continuous time yet the implementation is discrete time;
- The pre-filter parameters should be modifiable mid-flight to facilitate the gain tuning process;
- Switching between the 1-DOF and 2-DOF control architectures must be bumpless, for which a major challenge is proper handling of pre-filter toggling.

Let us consider a generic pre-filter $F(s)$, which may be any of the pre-filters in Sections 5.8.2 and 5.8.3. All three challenges are addressed by rewriting $F(s)$ in state space form and discretizing via the Tustin transform (which amounts to trapezoidal integration). A discrete state space representation is more numerically stable than a discrete transfer function Chen [126]. This is particularly important for us because of our use of single-precision accuracy (i.e. `float`) for computational efficiency. Indeed, a single-precision discretization of $F(s)$ given parameters in Table 5.6 diverges while the discrete state space form is stable.
Let $F(s)$ be given by the generic transfer function:

$$F(s) = \frac{b_{n-1}s^{n-1} + b_{n-2}s^{n-2} + \cdots + b_1 s + b_0}{a_n s^n + a_{n-1}s^{n-1} + a_{n-2}s^{n-2} + \cdots + a_1 s + a_0} + D, \tag{5.84}$$

where the constant D leaves open the possibility of direct input feed-through in case $F(s)$ is semi-proper. Practically, D can be extracted from a semi-proper transfer function via partial fraction decomposition (e.g. MATLAB's `partfrac`). Recall the generic state space formulation:

$$\begin{aligned} \dot{\boldsymbol{x}}(t) &= A\boldsymbol{x}(t) + B\boldsymbol{u}(t) \\ \boldsymbol{y}(t) &= C\boldsymbol{x}(t) + D\boldsymbol{u}(t), \end{aligned} \tag{5.85}$$

where $\boldsymbol{x}(t) \in \mathbb{R}^n$ is the state, $\boldsymbol{u}(t) \in \mathbb{R}^m$ is the input and $\boldsymbol{y}(t) \in \mathbb{R}^p$ is the output. A particular state space structure is known as the observer canonical form of $F(s)$ which is written as Skogestad and Postlethwaite [102]:

Coefficient	Expression
b_3	$\frac{\omega_\mathrm{n}^2}{k_\mathrm{d}}$
b_2	ω_n^2
b_1	$\frac{\omega_\mathrm{n}^2 k_\mathrm{p}}{k_\mathrm{d}}$
b_0	$\frac{\omega_\mathrm{n}^2 k_\mathrm{i}}{k_\mathrm{d}}$
a_3	$\frac{k_\mathrm{p}}{k_\mathrm{d}} + 2\zeta\omega_\mathrm{n}$
a_2	$\frac{k_\mathrm{i}+2\zeta\omega_\mathrm{n}k_\mathrm{p}}{k_\mathrm{d}} + \omega_\mathrm{n}^2$
a_1	$\frac{2\zeta\omega_\mathrm{n}k_\mathrm{i}+\omega_\mathrm{n}^2 k_\mathrm{p}}{k_\mathrm{d}}$
a_0	$\frac{\omega_\mathrm{n}^2 k_\mathrm{i}}{k_\mathrm{d}}$
D	0

(a) Coefficients for $F_\mathrm{p}(s)$ (5.73).

Coefficient	Expression
b_3	$\omega_\mathrm{n}^2 - \left(\frac{k_\mathrm{p}}{k_\mathrm{d}} + 2\zeta\omega_\mathrm{n}\right)\frac{\omega_\mathrm{n}^2}{k_\mathrm{d}}$
b_2	$\frac{\omega_\mathrm{n}^2 k_\mathrm{p}}{k_\mathrm{d}} - \left(\frac{k_\mathrm{i}+2\zeta\omega_\mathrm{n}k_\mathrm{p}}{k_\mathrm{d}} + \omega_\mathrm{n}^2\right)\frac{\omega_\mathrm{n}^2}{k_\mathrm{d}}$
b_1	$\frac{\omega_\mathrm{n}^2 k_\mathrm{i}}{k_\mathrm{d}} - \frac{\omega_\mathrm{n}^2(2\zeta\omega_\mathrm{n}k_\mathrm{i}+\omega_\mathrm{n}^2 k_\mathrm{p})}{k_\mathrm{d}^2}$
b_0	$-\frac{\omega_\mathrm{n}^4 k_\mathrm{i}}{k_\mathrm{d}^2}$
a_3	$\frac{k_\mathrm{p}}{k_\mathrm{d}} + 2\zeta\omega_\mathrm{n}$
a_2	$\frac{k_\mathrm{i}+2\zeta\omega_\mathrm{n}k_\mathrm{p}}{k_\mathrm{d}} + \omega_\mathrm{n}^2$
a_1	$\frac{2\zeta\omega_\mathrm{n}k_\mathrm{i}+\omega_\mathrm{n}^2 k_\mathrm{p}}{k_\mathrm{d}}$
a_0	$\frac{\omega_\mathrm{n}^2 k_\mathrm{i}}{k_\mathrm{d}}$
D	$\frac{\omega_\mathrm{n}^2}{k_\mathrm{d}}$

(b) Coefficients for $F_{\dot{\mathrm{p}}}(s)$ (5.74).

Coefficient	Expression
b_1	$\frac{1}{\tau} - (\frac{1}{\tau} + \frac{k_\mathrm{i}}{k_\mathrm{p}})\frac{1}{\tau k_\mathrm{p}}$
b_0	$\frac{k_\mathrm{i}(\tau k_\mathrm{p}-1)}{\tau k_\mathrm{p}}$
a_1	$\frac{1}{\tau} + \frac{k_\mathrm{i}}{k_\mathrm{p}}$
a_0	$\frac{k_\mathrm{i}}{\tau k_\mathrm{p}}$
D	$\frac{1}{\tau k_\mathrm{p}}$

(c) Coefficients for $F_\mathrm{v}(s)$ (5.83).

Table 5.6: Observer canonical form (5.86) coefficients of the $F(s)$, $F_{\dot{\mathrm{p}}}(s)$ and $F_\mathrm{v}(s)$ pre-filters.

$$A = \begin{bmatrix} -a_{n-1} & 1 & 0 & \cdots & 0 & 0 \\ -a_{n-2} & 0 & 1 & & 0 & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ -a_2 & 0 & 0 & & 1 & 0 \\ -a_1 & 0 & 0 & \cdots & 0 & 1 \\ -a_0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_2 \\ b_1 \\ b_0 \end{bmatrix}, \tag{5.86}$$
$$C = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}, \qquad D = D.$$

By identifying the pre-filters in Sections 5.8.2 and 5.8.3 with the generic form (5.84), their observer canonical form is readily determined. Table 5.6 lists the observer canonical form coefficients for each pre-filter.

Using Table 5.6, the pre-filter dynamics can be updated whenever a parameter changes. However, the state space formulation (5.85) is still continuous time. This is discretized with sampling time T_s via the Tustin transform Megretski [127]:

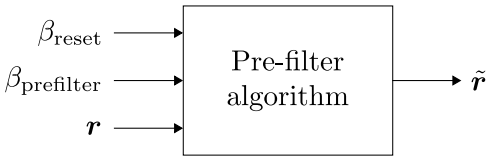


Figure 5.22: Pre-filter implementation input-output view. $\boldsymbol{r}$ is the raw reference and $\tilde{\boldsymbol{r}}$ is the pre-filtered reference.

$$\begin{aligned} A_{\text{d}} &= (I_n + A\frac{T_{\text{s}}}{2})(I_n - A\frac{T_{\text{s}}}{2})^{-1} \\ B_{\text{d}} &= T_{\text{s}}(I_n - A\frac{T_{\text{s}}}{2})^{-1}B \\ C_{\text{d}} &= C(I_n - A\frac{T_{\text{s}}}{2})^{-1} \\ D_{\text{d}} &= D + \frac{T_{\text{s}}}{2}C(I_n - A\frac{T_{\text{s}}}{2})^{-1}B, \end{aligned} \tag{5.87}$$

such that the discrete-time state space is written as:

$$\begin{aligned} \dot{\boldsymbol{x}}[k+1] &= A_{\text{d}}\boldsymbol{x}[k] + B_{\text{d}}\boldsymbol{u}[k] \\ \boldsymbol{y}[k] &= C_{\text{d}}\boldsymbol{x}[k] + D_{\text{d}}\boldsymbol{u}[k]. \end{aligned} \tag{5.88}$$

Note that (5.87) returns the same matrices as `c2d(ss(A,B,C,D),T,'tustin')` in MATLAB. Given Table 5.6 and (5.87), a pre-filter is readily implemented in discrete time and updated whenever a parameter changes. Lastly, bumpless on/off switching of a pre-filter is achieved by implementing it using Algorithm 30 as an input/output block illustrated by Figure 5.22. For the switch to be bumpless, it is required that at the switch time instance:

- β_{reset}=**true**;
- The reference signal's operating point is reset (see next section on reference operating point resetting).

Define the following Boolean variables:

- $\beta_{\text{reset,ext}}$ which is triggered (i.e. set to **true**) externally to the controller – in practice, during control mode switching;
- $\beta_{\text{reset,int}}$ which is triggered by the internal control implementation during parameter changes;
- $\beta_{\text{prefilter}}$ which is **true** to turn the pre-filter on (i.e. use the 2 DOF architecture) and **false** to turn the pre-filter off (i.e. use the 1 DOF architecture);
- β_{axis} which is **true** to enable the particular control axis in the decoupled implementation (**false** sets the acceleration mode for this axis).

The reset Boolean is then defined as:

$$\beta_{\text{reset}} := \beta_{\text{reset,ext}} \textbf{ or } \beta_{\text{reset,int}} \textbf{ or } (\beta_{\text{axis}} \text{ rising edge}) \textbf{ or } (\beta_{\text{prefilter}} \text{ change}). \tag{5.89}$$

Figure 5.23 illustrates the reset Boolean triggering.

Algorithm 30 Pre-filter algorithm describing the internal workings of the block in Figure 5.22.

`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/common_blocks.slx:prefilterReference`

Initialization:

$\boldsymbol{x} \leftarrow 0$

Iteration k**:**

if β_{reset} **then**
    $\boldsymbol{x} \leftarrow 0$
end if
if $\beta_{\text{prefilter}}$ **then**
    $\tilde{\boldsymbol{r}} \leftarrow C_{\text{d}}\boldsymbol{x} + D_{\text{d}}\boldsymbol{r}$
    $\boldsymbol{x} \leftarrow A_{\text{d}}\boldsymbol{x} + B_{\text{d}}\boldsymbol{r}$
else
    $\tilde{\boldsymbol{r}} \leftarrow \boldsymbol{r}$
end if

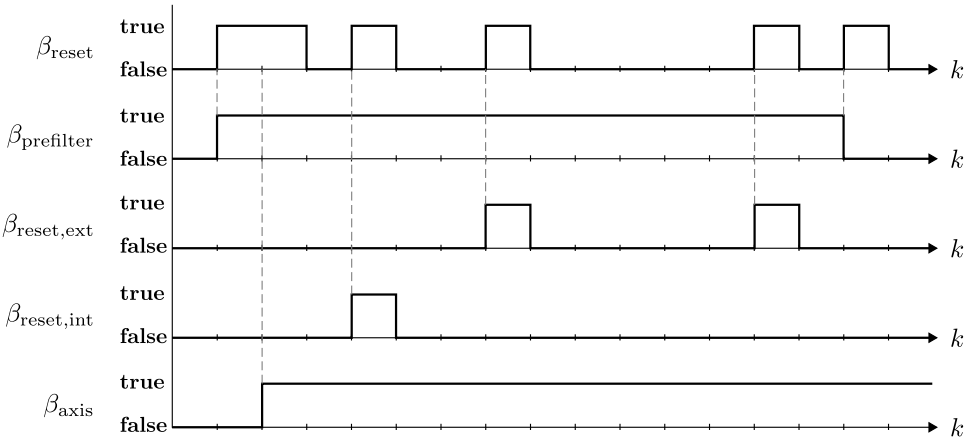


Figure 5.23: β_{reset} triggering as per (5.89). Horizontal axis shows iteration count. Vertical dashed lines indicate which variable was responsible for triggering β_{reset}.

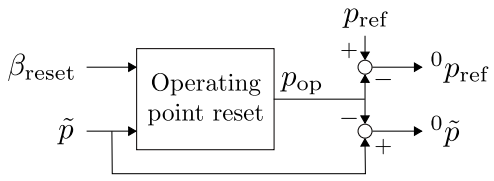


Figure 5.24: Reference opearting point setting block diagram.

Resettable Reference Operating Point

Let r be a reference, r_{op} the *reference operating point* and $^0r := r - r_{\text{op}}$ the *tared reference*. To achieve bumpless mode switching, Algorithm 30 requires updating r_{op} when changing control mode.
The reason is easy to visualize. Imagine a situation where the quadrotor has flown up to a 10 m altitude using e.g. the velocity mode. Pre-filters relating to the position mode did not participate in this altitude change thus their values are arbitrarily "floating". Now the quadrotor switches into the position mode and, according to Algorithm 30, the $p_{\text{ref,z}}$ pre-filter gets reset to zero but the reference is still 10 m. The pre-filter will thus "filter" what it sees as a reference step from 0 m to 10 m. The result is that the quadrotor will first drop in altitude and then rise along with the pre-filtered reference back up to 10 m.
To avoid such "bumps", the position reference point needs to be reset whenever $\beta_{\text{reset}} = \textbf{true}$. Note that the $\boldsymbol{v}_{\text{ref}}$ operating point is assumed to always be zero, thus reference point resetting is only implemented for $\boldsymbol{p}_{\text{ref}}$. In particular, considering a single control axis in the context of the decoupled design of Section 5.8.2, Algorithm 31 updates the position reference point p_{op} and Figure 5.24 computes the *tared* position reference $^0p_{\text{ref}}$ and estimate $^0\tilde{p}$. These variables are then used by the position pre-filter $F_{\text{p}}(s)$ and the controller $K_{\text{p}}(s)$ in place of their non-tared p_{ref} and $\tilde{p}$ counterparts.

Algorithm 31 Reference operating point updating algorithm.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/common_blocks.slx:opReset`

Initialization:
$p_{\text{op}} \leftarrow \tilde{p}$
Iteration k**:**
if β_{reset} **then**
  $p_{\text{op}} \leftarrow \tilde{p}$
end if

Resettable Discrete Integrator With Anti-Windup

In practice, the thrust vector cone saturation of Section 5.7.1 and the collective thrust feasible range constrain $a_{\text{ref}} \in [a_{\text{ref,min}}, a_{\text{ref,max}}]$. Outside of this range, a_{ref} is saturated to the upper or lower bound. Integrator windup is therefore an issue for the $1/s$ blocks of the PID controllers in Sections 5.8.2 and 5.8.3.
As explained by Algorithm 32, we implement the $1/s$ block as a resettable discrete-time trapezoidal (i.e. Tustin) integrator with digital anti-windup protection. The concept is further illustrated by Figure 5.25, which highlights that while we allow a_{ref} to cross the saturation boundaries (due to the P and D terms), in these regions the integrator is frozen unless it acts in the opposite direction to the saturation.
Note that the current implementation completely resets the integrator when $\beta_{\text{reset}} = \textbf{true}$. This is not bumpless, since resetting "wipes" the integrator's memory. This is

most noticeable as a very small dip in altitude if e.g. the quadrotor's mass knowledge is imperfect. The integrator has to "do the work again" to remove steady state error. A future implementation could be fully bumpless by e.g. setting the integrator to exactly equal a_{ref} at the switch time instance Åström and Hägglund [125]. Because extensive flight testing was done without this feature, it was opted to not implement it in order to minimize the risk of introducing adverse behavior.

Algorithm 32 Generic resettable discrete-time integrator with anti-windup protection, with gain k_{i}, sampling period T_{s} and output u_{i}.
`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab/simulink_models/common_blocks.slx:integrator`

Initialization:
$u_{\text{prev}} \leftarrow 0$
$y_{\text{prev}} \leftarrow 0$
Iteration k**:**
Anti-windup logic:
$e \leftarrow {}^{0}p_{\text{ref}} - {}^{0}\tilde{p}$ ▷ Control error
$u \leftarrow k_{\text{i}} e$
if $a_{\text{ref}} < a_{\text{ref,min}}$ **then**
    $u \leftarrow \begin{cases} 0 & u < 0 \\ u & \text{otherwise} \end{cases}$
else if $a_{\text{ref}} > a_{\text{ref,max}}$ **then**
    $u \leftarrow \begin{cases} 0 & u > 0 \\ u & \text{otherwise} \end{cases}$
end if
$y \leftarrow y_{\text{prev}} + \frac{T_{\text{s}}}{2}(u + u_{\text{prev}})$ ▷ Trapezoidal integration
if reset **then** ▷ Integrator resetting functionality
    $y \leftarrow 0$
end if
$u_{\text{prev}} \leftarrow u$
$y_{\text{prev}} \leftarrow y$
$u_{\text{i}} \leftarrow y$ ▷ Integral term output (I contribution)

5.9 Emergency Landing Control

An *emergency landing controller* is a translation controller which is responsible for mitigating the consequences of a crash following a state estimate outage. This may occur due to state estimator failure (because of software or the utilized sensors) or due to communication link failure between the HLP and the Odroid XU4 (which runs the state estimator). Because attitude control can be performed using IMU only, the loss of a state estimate compromises only the position and velocity modes of the translation controller in Section 5.8.

5.9.1 Literature Review

A body of literature exists on quadrotor emergency controllers in the event of state estimate loss Faessler et al. [105], Mueller and D'Andrea [116], Lupashin et al. [103]. These software-based solutions all aim to prevent a crash or, if a crash in inevitable (i.e. due to prolonged state estimate loss), to minimize its consequences. Faessler et al. [105] use an IMU and a range sensor Terabee [79] and achieve a recovery rate of 85%. The downside of their approach is the requirement to carry a range sensor for

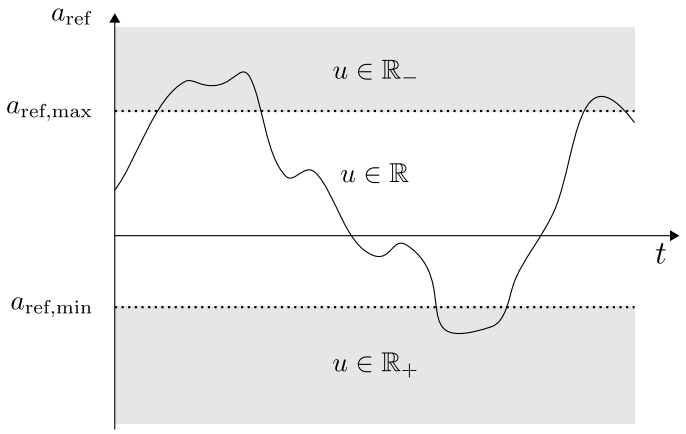


Figure 5.25: Illustration of integrated signal u constraint as function of a_{ref} saturation.

what is (hopefully) a very rare emergency event. The range sensor occupies space and increases the weight and system cost. Mueller and D'Andrea [116], Lupashin et al. [103] use an emergency controller which only uses the on-board gyroscope body rate measurement, making it an advantageous software-only solution since any quadrotor carries an IMU.

5.9.2 Control Description

Our emergency landing controller[25] performs an open-loop downward acceleration maneuver. Once activated, it does not use any sensors and is thus even simpler than the controller in Mueller and D'Andrea [116], Lupashin et al. [103].
Far from being the focus of this thesis, the solution proposed herein represents a preliminary effort to reduce the damage from a state estimator failure-associated crash or, better, to reduce the translational divergence which ensues from controlling on a bad estimate. This helps a human safety pilot to take over during test flights. The objective of the emergency controller being to **land quickly**, introducing feedback (via gyros and accelerometers) is welcome but not immediately necessary (see Section 8 for a further discussion).
As shown in Figure 5.1, the emergency landing controller overrides the translation controller when $\beta_{\text{emergency}} = \textbf{true}$. This Boolean is set as follows:

- If the translation controller is in acceleration mode with the control reference coming from an RC transmitter (i.e. a human pilot), $\beta_{\text{emergency}}$ is always **false**. Because attitude control is not compromised by state estimator failure and a human pilot's capacity to command a correct $\boldsymbol{a}_{\text{ref}}$ persists, there is no need to trigger the emergency landing controller[26];

- Otherwise, $\beta_{\text{emergency}} = \textbf{true}$ when the state estimate has not been updated for 100 ms **or** the standard deviation of $\tilde{\boldsymbol{p}}^{\text{b}}_{\text{w}}$ exceeds 1 m along any axis.

Let $\mathcal{B}_{\tilde{v}^{\text{b}}_{\text{w,z}}}$ denote a t_{em} second First-In First-Out (FIFO) buffer of $\tilde{v}^{\text{b}}_{\text{w,z}}$. As long as $\beta_{\text{emergency}} = \textbf{false}$, $\mathcal{B}_{\tilde{v}^{\text{b}}_{\text{w,z}}}$ gets populated with new velocity estimates while old ones

[25] `/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/emergency_lander.h`

[26] The current implementation does, however, seamlessly switch from the state estimator's attitude estimate $\tilde{\boldsymbol{q}}^{\text{b}}_{\text{w}}$ to the on-board gyro integrated one provided by the AscTec LLP.

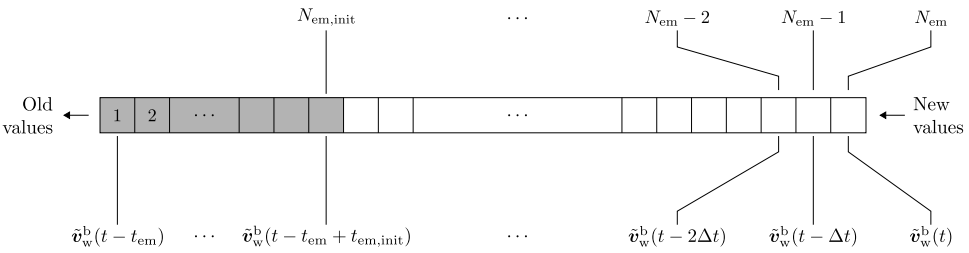


Figure 5.26: Illustration of $\mathcal{B}_{\tilde{v}^{\text{b}}_{\text{w,z}}}$. The buffer sampling time $\Delta t \equiv f^{-1}_{\text{em,buffer}}$. Gray elements are used for $v_{\text{em,0,z}}$ computation in (5.92).

are removed[27] as shown in Figure 5.26. When an emergency landing is triggered (i.e. $\beta_{\text{emergency}} = \textbf{true}$), it is likely that the most recent entries of $\mathcal{B}_{\tilde{v}^{\text{b}}_{\text{w,z}}}$ are polluted with good-going-bad estimates since the state estimate standard deviation takes a non-trivial time to grow to 1 m once e.g. a global sensor (GPS or VICON) is lost. It is therefore assumed that only the oldest $t_{\text{em,init}}$ seconds contain valid estimates. In the current implementation, $t_{\text{em}} = 3\,\text{s}$ and $t_{\text{em,init}} = 1\,\text{s}$. The reason for the long $t_{\text{em,init}}$ duration is motivated by the fact that the data acquisition trajectory (Section 4.1.6) is dominated by constant velocity trajectory segments.
$\mathcal{B}_{\tilde{v}^{\text{b}}_{\text{w,z}}}$ is sampled at a rate of $f_{\text{em,buffer}}$ (currently 20 Hz). Then, in terms of element indexing:

$$N_{\text{em}} := t_{\text{em}} f^{-1}_{\text{em,buffer}}, \tag{5.90}$$

$$N_{\text{em,init}} := t_{\text{em,init}} f^{-1}_{\text{em,buffer}}, \tag{5.91}$$

represent the buffer total and "valid estimate" number of elements respectively (as shown in Figure 5.26). Each time that an emergency landing is triggered, Algorithm 33 (i.e. the emergency landing controller itself) is executed (with initialization always being done first). This algorithm performs an open-loop constant vertical acceleration $a_{\text{em,z}}$ maneuver to bring the quadrotor to a desired descent velocity $v^{\text{b}}_{\text{w,z}} = v_{\text{em,z}}$. Currently, $a_{\text{em,z}} = 2\,\text{m/s}^2$ and $v_{\text{em,z}} = -2\,\text{m}s$. We do not perform the simple constant-acceleration-down maneuver of Mueller and D'Andrea [116] because our quadrotor is expected to fly high (e.g. a $-1\,\text{m/s}^2$ vertical acceleration starting from rest at 30 m altitude would lead to a violent $\approx 30\,\text{km/h}$ touchdown). Bleeding off the horizontal velocity components is not performed as it is assumed that the quadrotor flies above an open field with abundant available space. If $\beta_{\text{emergency}}$ returns to **false** at any point, control is passed back to the nominal translation controller (Section 5.8).

[27] Unpopulated elements are zero.

Algorithm 33 Emergency landing controller which performs an open-loop acceleration to $v^{\mathrm{b}}_{\mathrm{w,z}} = v_{\mathrm{em,z}}$.

`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/emergency_lander.c:setEmergencyLandingMode`

`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/emergency_lander.c:decelerateOpenLoopSingleAxis`

Initialization:

Estimate of the current (latest valid) velocity:

$$v_{\mathrm{em,0,z}} \leftarrow \frac{1}{N_{\mathrm{em,init}}} \sum_{i=1}^{N_{\mathrm{em,init}}} \mathcal{B}_{\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}}[i] \tag{5.92}$$

$v_{\mathrm{z}} \leftarrow v_{\mathrm{em,0,z}}$ ▷ Initial open-loop integrated vertical velocity

$\mathcal{B}_{\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}}[i] \leftarrow 0 \ \forall i = 1, \ldots, N_{\mathrm{em}}$ ▷ For future emergencies

$\beta_{\mathrm{done}} \leftarrow$ **false** ▷ Acceleration maneuver end indicator

Iteration at 200 Hz: ▷ Figure 5.1

$\boldsymbol{a}_{\mathrm{ref}} \leftarrow 0_{3\times 1}$

if not β_{done} **then**

sign_before $\leftarrow v_{\mathrm{z}} > v_{\mathrm{em,z}}$

$$a_{\mathrm{ref,z}} \leftarrow \begin{cases} -a_{\mathrm{em,z}} & \text{sign\_before} \\ a_{\mathrm{em,z}} & \textbf{not } \text{sign\_before} \end{cases}$$

$v_{\mathrm{z}} \leftarrow v_{\mathrm{z}} + 0.005 \cdot a_{\mathrm{ref,z}}$ ▷ Euler integration at 200 Hz

sign_now $\leftarrow v_{\mathrm{z}} > v_{\mathrm{em,z}}$

if sign_now $\neq$ sign_before **then**

$\beta_{\mathrm{done}} \leftarrow$ **true**

end if

end if

$\boldsymbol{a}_{\mathrm{ref}} \leftarrow \boldsymbol{a}_{\mathrm{ref}} - \boldsymbol{g}$ ▷ Gravity compensation

Chapter 6

Autonomy Engine

The autonomy engine subsystem implements the logic for executing a full-cycle autonomous data acquisition mission. It may be thought of as the high-level "brain" of the quadrotor.

6.1 Autonomy Engine Overview

Figure 6.1 illustrates a high-level overview of the (nominal) long-term data acquisition cycle, composed of the charging, takeoff, mission and landing *phases*. To initialize, the user places the quadrotor on the charging pad and starts the autonomy engine. The quadrotor subsequently waits until the battery is charged, then takes off and executes the pre-defined data acquisition mission. Finally, when the mission is complete, it returns to the charging pad and lands. The flying part of the cycle is referred to as a *sortie*. The cycle repeats indefinitely and with no human interaction until it is stopped by the operator. Robust behavior for cases like low battery, critical battery or failed motors are not shown in Figure 6.1 and shall be presented later in this chapter.
Each phase is implemented as a state machine. A general design pattern has been to first define the logic via a flowchart and then convert the flowchart into a UML-inspired state machine. Appendix B provides recommended background on our state machine implementation and the UML state machine diagrams used herein. Unless otherwise specified, all state machines are executed at the guidance frequency $f_{\text{guidance}} = 20\,\text{Hz}$ (the same one as introduced in Chapter 4).
The state machines for each phase are stand-alone subsystems that execute from start to end the logic associated with each phase based on an external call to action.

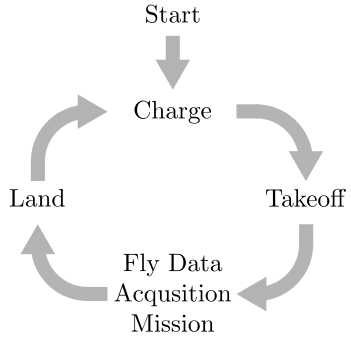


Figure 6.1: Nominal cycle for full-cycle autonomous data acquisition.

These phase-specific state machines are called *autopilots*. An additional "master" logic is needed to issue the action calls in an appropriate sequence. This is achieved via a high-level *master* state machine. The relationship between the master and phase-specific state machines is illustrated in Figure 6.4.

6.1.1 Chapter Organization

Appendix B introduces the state machine implementation of this thesis. As a consequence of this implementation and its features, this chapter adopts a strict structure shown in Figure 6.3 in order to deliver a consistent description. The absence of a particular structural element indicates its absence in the state machine. The remainder of this chapter describes the state machines that comprise the autonomy engine:

- Section 6.2 describes the takeoff autopilot;
- Section 6.3 describes the mission autopilot;
- Section 6.4 describes the landing autopilot;
- Section 6.5 describes the emergency landing autopilot;
- Section 6.6 describes the master state machine.

Results of using the autonomy engine are presented in Section 7.5.

6.2 Takeoff Autopilot

The takeoff autopilot[1] implements the logic associated with the takeoff phase of Figure 6.1. The takeoff autopilot logic in Figure 6.5 is implemented as the state machine shown in Figure 6.6.

6.2.1 Initialization

Initial State CHECK_PERFORMANCE

Loaded Parameters

Parameter	*Description*	*Units*
h_{takeoff}	Target takeoff height above the landing pad	m
$t_{\text{takeoff,max}}$	Timeout for climbing to h_{takeoff}	s
v_{takeoff}	Target takeoff velocity	$^{\text{m}}/_{\text{s}}$
a_{takeoff}	Target acceleration for acceleration/deceleration to/from v_{takeoff}	$^{\text{m}}/_{\text{s}^2}$
$u_{\text{esc,motor,check}}$	Motor ESC command for checking motor nominal performance	-
$r_{\text{motor,check}}$	Motor Rotations Per Minute (RPM) tolerance for validating motor nominal performance	$^{1}/_{\text{min}}$
$t_{\text{motor,check}}$	Motor nominal performance check duration	s
$n_{\text{motor,check}}$	Maximum number of motor nominal performance check trials	-

[1] `/autonomy_engine/alure_takeoff/include/alure_takeoff/takeoff_autopilot.h`

1.n State Machine Name

Short description of what this state machine does and an obligatory UML state machine diagram in Figure 6.2

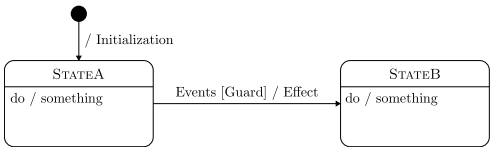


Figure 6.2: Generic state machine diagram.

1.n.1 Initialization

Specifies effect of transition from initial pseudostate to initial state

Initial State StateA *specifies the initial state*

Loaded Parameters

List of parameters loaded during initialization

Other Actions

Other one-time actions performed during initialization

1.n.2 Runtime

State StateA

Short description of what this state does

Do Action

Do action as described in Appendix B

Transition to StateB *specifies the target state*
When no events or guards are specified, the transition happens immediately as described in Appendix B

Events

Guard

Effect

Transition to ··· *Other transitions can be specified*

State StateB

··· *Same structure as for* StateA

Figure 6.3: Structural layout for the n-th state machine description as per the implementation in Appendix B. StateA and StateB are used as placeholders. There may be multiple "Transition" sections for each state and multiple "State" sections for each state machine.

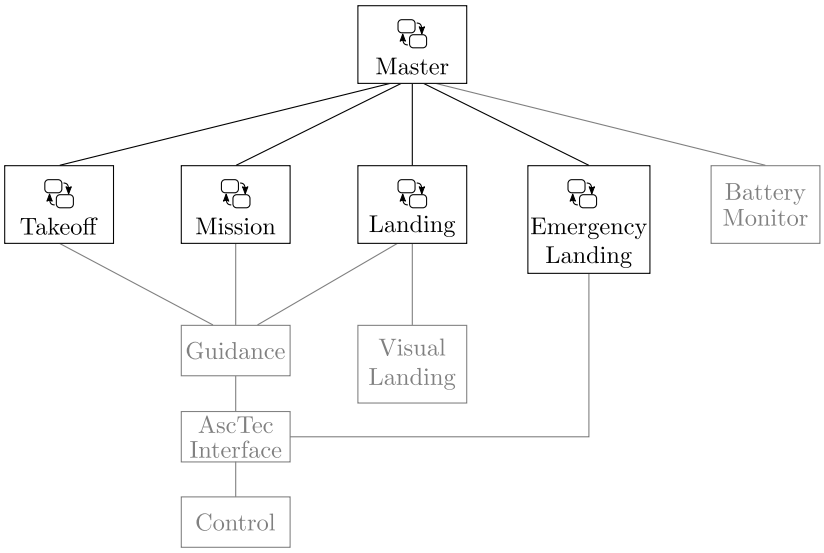


Figure 6.4: Master-slave hierarchy between the master and phase-specific state machines. Subsystems composing specifically the autonomy engine are in black.

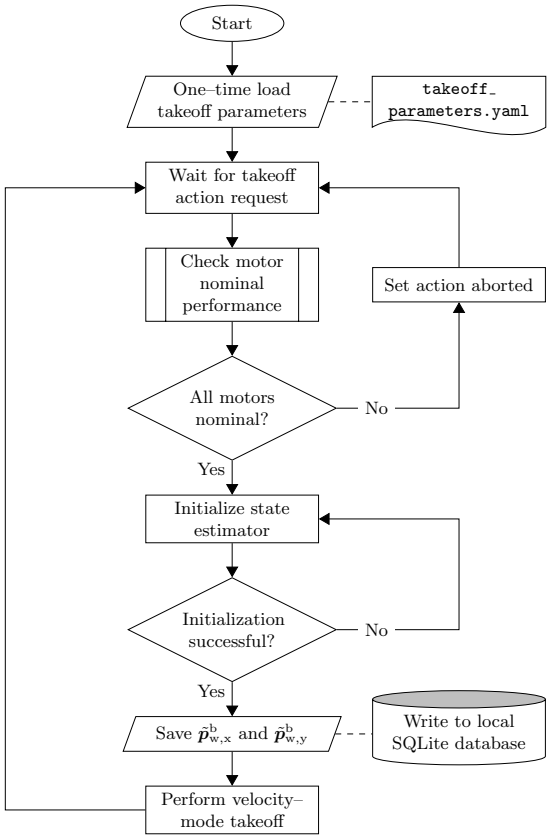


Figure 6.5: Takeoff algorithm flowchart.

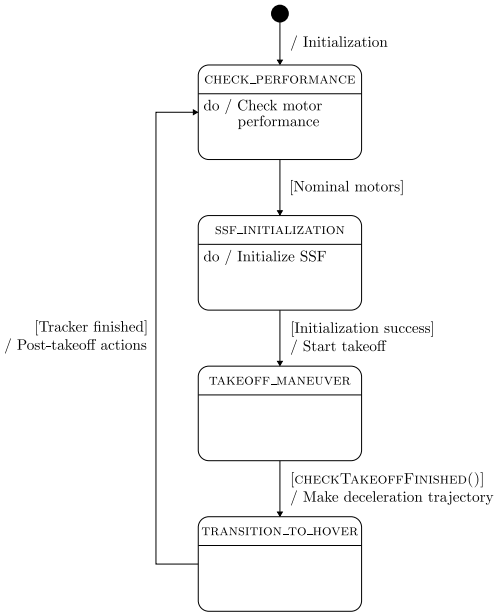


Figure 6.6: Takeoff state machine.

Other Actions

The takeoff autopilot owns a trajectory list $\Gamma_{\text{list,takeoff}}$ and associates it with a VelocityTracking trajectory tracker, as explained in Chapter 4. Initially empty, $\Gamma_{\text{list,takeoff}}$ gets populated at initialization with a velocity transfer trajectory from rest (quadrotor on charging pad) to takeoff velocity $\boldsymbol{v}_{\text{takeoff}} = (0, 0, v_{\text{takeoff}})$. This is done by calling makeVelocityTransferTrajectory$(0_{3\times1},\ \boldsymbol{v}_{\text{takeoff}})$ as described in Algorithm 34.

Algorithm 34 Velocity transfer trajectory generation.
`/autonomy_engine/alure_takeoff/src/takeoff_autopilot.cpp: makeVelocityTransferTrajectory`

1: **function** makeVelocityTransferTrajectory($\boldsymbol{v}_0$, $\boldsymbol{v}_{\text{f}}$)
2: Construct velocity trajectory ($N = 8$) endpoint derivatives:

$$\boldsymbol{d}_0 \leftarrow (\boldsymbol{v}_0, 0_{9\times1})$$
$$\boldsymbol{d}_T \leftarrow (\boldsymbol{v}_{\text{f}}, 0_{9\times1})$$

3: $T \leftarrow \|\boldsymbol{v}_{\text{f}} - \boldsymbol{v}_0\|/a_{\text{takeoff}}$ ▷ Constant acceleration velocity transfer duration
4: $\sigma \leftarrow$createTransferTrajectory($\boldsymbol{d}_0$, $\boldsymbol{d}_T$, T) ▷ Algorithm 14
5: $\gamma \leftarrow \{\sigma, f_{\text{guidance}}^{-1}, \text{SINGLE}\}$
6: $\Gamma_{\text{list,takeoff}}$->push_back($\gamma$)
7: **end function**

6.2.2 Runtime

State check_performance

It is prudent to verify motor performance before committing to takeoff. Indeed, the AscTec Pelican motors often "jitter" on start-up. On several rare occasions, a

dislodged ESC Inter-Integrated Circuit (I^2C) bus during ground handling led to the complete non-functioning of a motor.

Do Action

Validate motor nominal performance by executing Algorithm 35 which runs for $t_{\text{motor,check}}$ seconds and, in case of failed validation, reruns the validation $n_{\text{motor,check}}$ times. Algorithm 35 works by commanding a $u_{\text{esc,motor,check}}$ ESC command to all motors and reading the RPM feedback $\boldsymbol{r}_{\text{motor}} \in \mathbb{R}^4$. As long as all RPM measurements received during the $t_{\text{motor,check}}$ seconds are within the tolerated RPM band, nominal motor performance is confirmed. In the case of even a single RPM outlier, nominal motor performance is failed and another trial is performed. If the maximum number of trials is reached, motor performance is declared abnormal. In this case, the ROS action client is notified that takeoff has been aborted and the takeoff autopilot sleeps (selfPause()).

Transition to SSF_INITIALIZATION

Guard Motor nominal performance check has passed (Algorithm 35 returned **true**).

State SSF_INITIALIZATION

As explained in Section 3.2, SSF must be re-initialized prior to takeoff after the prolonged charging phase.

Do Action

Call SSF hard-initialization service which completely re-initializes the state, wiping all memory. This decouples individual sorties in state, making sure that potential adverse events in e.g. the previous landing do not compromise the state estimate quality for the current takeoff.

Transition to TAKEOFF_MANEUVER

Guard SSF initialization ROS service returned **true**.

Effect First, the current horizontal location $\boldsymbol{p}_{\text{rth}} := (\tilde{p}^{\text{b}}_{\text{w,x}}, \tilde{p}^{\text{b}}_{\text{w,y}})$ is memorized in the Odroid XU4 non-volatile storage in a local SQLite database[2] Hipp et al. [128]. Next, the translation controller of Section 5.8 is set to the velocity mode and is given control over the motor speeds. Lastly, the trajectory tracker is played. The quadrotor begins tracking the acceleration trajectory created during initialization (the first element of $\Gamma_{\text{list,takeoff}}$). Lastly, the current time is memorized as $t_{0,\text{takeoff}}$ (the *takeoff start time*).

[2]Keeping the crucial $\boldsymbol{p}_{\text{rth}}$ in Random Access Memory (RAM) is undesirable – if the autonomy engine crashes, the quadrotor will forget where is home. Although the functionality has not been implemented, a future implementation could auto-restart the autonomy engine which can continue to use $\boldsymbol{p}_{\text{rth}}$, which persists in the SQLite database.

Algorithm 35 Nominal motor performance validation routine.
`/autonomy_engine/alure_takeoff/src/takeoff_autopilot.cpp:checkMotorNominalPerformance`
`/autonomy_engine/alure_takeoff/src/takeoff_autopilot.cpp:validateMotors`

function CHECKMOTORNOMINALPERFORMANCE()
    Enable direct motor control ▷ To pass direct ESC commands
    $n \leftarrow 0$ ▷ Trial counter
    while $n < n_{\text{motor,check}}$ **do**
        Turn on all motors with $u_{\text{esc,motor,check}}$ ESC command
        Sleep 5 seconds ▷ To give the motors time to spin up
        rate $\leftarrow 10$ ▷ 10 Hz RPM check rate
        for $i \leftarrow 1, 2, \ldots, \text{rate} \cdot t_{\text{motor,check}}$ **do**
            SUCCESS $\leftarrow$ VALIDATEMOTORS()
            if not SUCCESS **then**
                Turn off all motors
                $n \leftarrow n + 1$ ▷ Increment trial counter
                break
            end if
            Sleep rate^{-1} seconds ▷ Maintain rate Hz checking frequency
        end for
        if not SUCCESS **then**
            Sleep 2 seconds ▷ Short wait before next trial
            continue
        end if
        return true ▷ Nominal performance check passed
    end while
    return false ▷ All trials failed nominal performance check
end function

function VALIDATEMOTORS()
    $r_{\text{nominal}} \leftarrow \left(25 + \frac{175 u_{\text{esc,motor,check}}}{200}\right) 43$ ▷ AscTec nominal ESC to RPM map
    $r_{\max} \leftarrow r_{\text{nominal}} + r_{\text{motor,check}}$
    $r_{\min} \leftarrow r_{\text{nominal}} - r_{\text{motor,check}}$
    for $i \leftarrow 1$ to 4 **do**
        if $r_{\text{motor}}[i] < r_{\min}$ **or** $r_{\text{motor}}[i] > r_{\max}$ **then**
            return false ▷ Motor i fails nominal performance check
        end if
    end for
    return true
end function

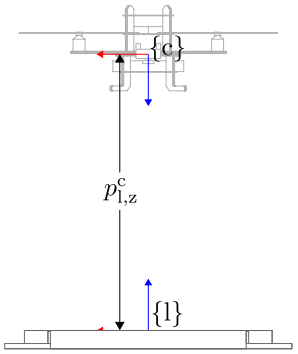


Figure 6.7: Landing bundle detection during takeoff provides a height-based way of finishing the takeoff.

State TAKEOFF_MANEUVER

This is a passive state in which the takeoff autopilot waits for the takeoff finish condition to be satisfied. Note that the trajectory tracker, running in a parallel thread, naturally brings the quadrotor to $\boldsymbol{v}_{\text{takeoff}}$ velocity without polluting the autonomy engine source code. This is the advantage of our hierarchical implementation – low-level functionality is encapsulated in separate classes, leaving the autonomy engine to deal with the high-level logic.

Transition to TRANSITION_TO_HOVER

Guard Algorithm 36 evaluates to **true**. As shown in Figure 6.7, this nominally happens via the AprilTag bundle measurement of the camera height. If for any reason (e.g. significant horizontal velocity component) the AprilTag-based condition is never satisfied, a timeout finishes the takeoff.

Effect Add a deceleration to hover velocity transfer trajectory to $\Gamma_{\text{list,takeoff}}$ via MAKEVELOCITYTRANSFERTRAJECTORY($\tilde{\boldsymbol{v}}^{b}_{w}$, $0_{3\times1}$).

Algorithm 36 Takeoff finish trigger.
`/autonomy_engine/alure_takeoff/src/takeoff_autopilot.cpp:checkTakeoffFinished`

```
1: function CHECKTAKEOFFFINISHED()
2:     if t − t_{0,takeoff} > t_{takeoff,max} then
3:         return true                                  ▷ Takeoff timeout
4:     end if
5:     if p̂^c_{l,z} > h_takeoff then
6:         return true                                  ▷ Takeoff nominal finish
7:     end if
8:     return false
9: end function
```

State TRANSITION_TO_HOVER

This state waits for the tracker to finish, then leaves the quadrotor in a hovering position mode.

Transition to CHECK_PERFORMANCE

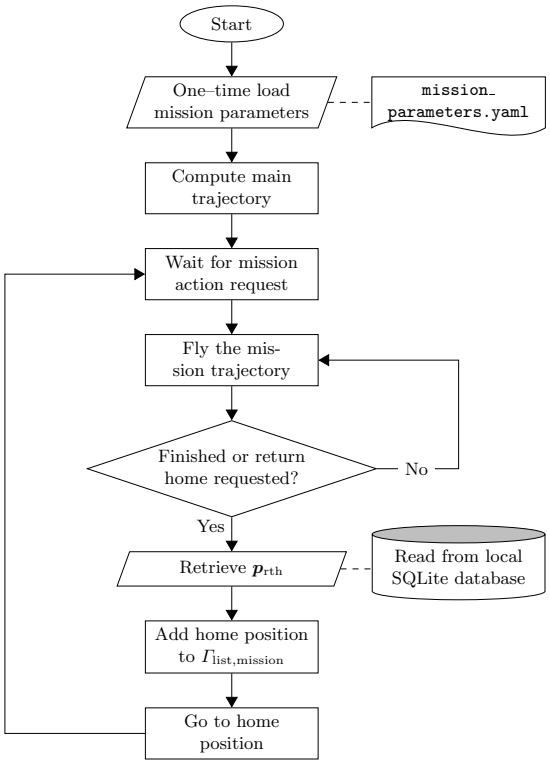


Figure 6.8: Mission algorithm flowchart.

Guard The tracker finishes (Algorithm 22). Since the tracker uses the VELOCITYTRACKING mode, this occurs as soon as the deceleration trajectory is finished playing (regardless to the quadrotor's actual current velocity).

Effect The tracker is paused and the deceleration trajectory is removed by calling $\Gamma_{\text{list,takeoff}}$->POP_BACK(). The translation controller is switched to the position mode (Section 5.8) which leaves the quadrotor hovering ("holding") its current position. The ROS action client (i.e. the master state machine) is notified that takeoff is finished and the takeoff autopilot sleeps (SELFPAUSE()).

6.3 Mission Autopilot

The mission autopilot[3] implements the logic associated with the mission phase of Figure 6.1. The mission autopilot logic in Figure 6.8 is implemented as the state machine shown in Figure 6.9.

6.3.1 Initialization

Initial State BEGIN_MISSION

[3] /autonomy_engine/alure_mission/include/alure_mission/mission_autopilot.h

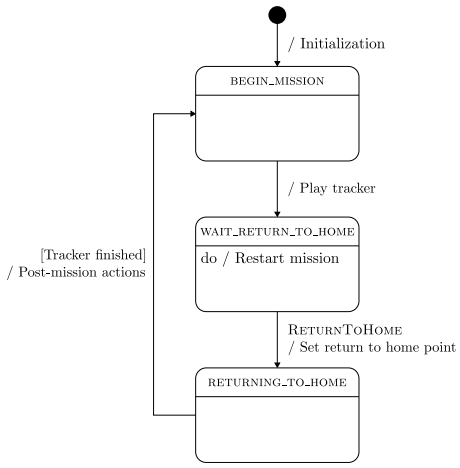


Figure 6.9: Mission state machine.

Loaded Parameters

Parameter	*Description*	*Units*
$\mathcal{W}_{\text{mission}}$	Waypoints list defining the mission trajectory	-
n_{mission}	Number of times to re-fly the mission trajectory	-
v_{mission}	Target speed between mission trajectory waypoints	$^{\text{m}}/_{\text{s}}$
R_{mission}	Cornering radius at pass-through waypoints	m
h_{rth}	Return to home target height (and therefore the starting height for landing)	m

Other Actions

The mission autopilot owns a trajectory list $\Gamma_{\text{list,mission}}$ and associates it with a PositionTracking trajectory tracker, as explained in Chapter 4. Initially empty, $\Gamma_{\text{list,mission}}$ gets populated at initialization with the mission trajectory by calling createWaypointTrajectory($\mathcal{W}_{\text{mission}}$, v_{mission}, R_{mission}) with sampling time f_{guidance}^{-1}. The playback type is set to cyclic when $n_{\text{mission}} = 0$, otherwise it is set to single.
Additionally, a ReturnToHome event is defined. As shall be explained in Section 6.3.2, it can be internally or externally invoked and has the consequence of aborting the mission and returning the quadrotor home.

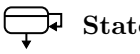
6.3.2 Runtime

State BEGIN_MISSION

This is a passive state in which the autopilot is paused. When the autopilot is woken by a ROS action call, the transition effect from this state is used to play the trajectory tracker.

Transition to WAIT_RETURN_TO_HOME

Effect Play the trajectory tracker. Thanks to its realignment trajectory generation feature (Section 4.3), the tracker internally handles taking the quadrotor from its post-takeoff hover position to the mission trajectory start.



State WAIT_RETURN_TO_HOME

In this state, the quadrotor is flying the mission (data acquisition) trajectory defined by $\mathcal{W}_{\text{mission}}$. When this is finished, or an external return to home request is received via a ROS service, the transition effect commands the quadrotor to return to the charging pad.

Do Action

Make the quadrotor repeat the mission n_{mission} times as explained in Algorithm 37. When $n_{\text{mission}} = 0$, the mission trajectory element's playback type is CYCLIC (Section 6.3.1) so the tracker's FINISHED() method never evaluates to **true**. In this case, the transition to the RETURNING_TO_HOME state happens when the battery voltage becomes low (see Section 6.6).

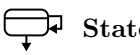
Algorithm 37 Mission repeating logic.
`/autonomy_engine/alure_mission/src/mission_autopilot.cpp:stateMachineDefinition`

Initialization:
1: $n_{\text{mission,count}} \leftarrow 1$
Runtime:
2: **if** tracker is FINISHED() **then** ▷ Algorithm 22
3: **if** $n_{\text{mission,count}} < n_{\text{mission}}$ **then**
4: $n_{\text{mission,count}} \leftarrow n_{\text{mission,count}} + 1$
5: $\Gamma_{\text{list,mission}}$->REWIND() ▷ Algorithm 19
6: **else**
7: Invoke RETURNTOHOME event
8: **end if**
9: **end if**

Transition to RETURNING_TO_HOME

Events RETURNTOHOME, which can get invoked by Algorithm 37 or externally. The external invocation happens when the master state machine (Section 6.6) observes that battery voltage has become low.

Effect $\boldsymbol{p}_{\text{rth}}$ is read from the local SQLite database and the return to home location is added to $\Gamma_{\text{list,mission}}$ via CREATEHOVERPOINT($(\boldsymbol{p}_{\text{rth}}, h_{\text{rth}})$,0) with f^{-1}_{guidance} sampling period and SINGLE playback type. Then, the mission trajectory is aborted via ABORT() (Algorithm 19). Thanks to the trajectory tracker's automatic realignment trajectory generation, the quadrotor is smoothly guided back to the charging pad.



State RETURNING_TO_HOME

This is a passive state which waits for the quadrotor to arrive at $(\boldsymbol{p}_{\text{rth}}, h_{\text{rth}})$ which due to GPS drift is likely close, but not exactly at, the original takeoff location. When this happens, the ROS action client (i.e. master state machine) is notified that the quadrotor has returned home.

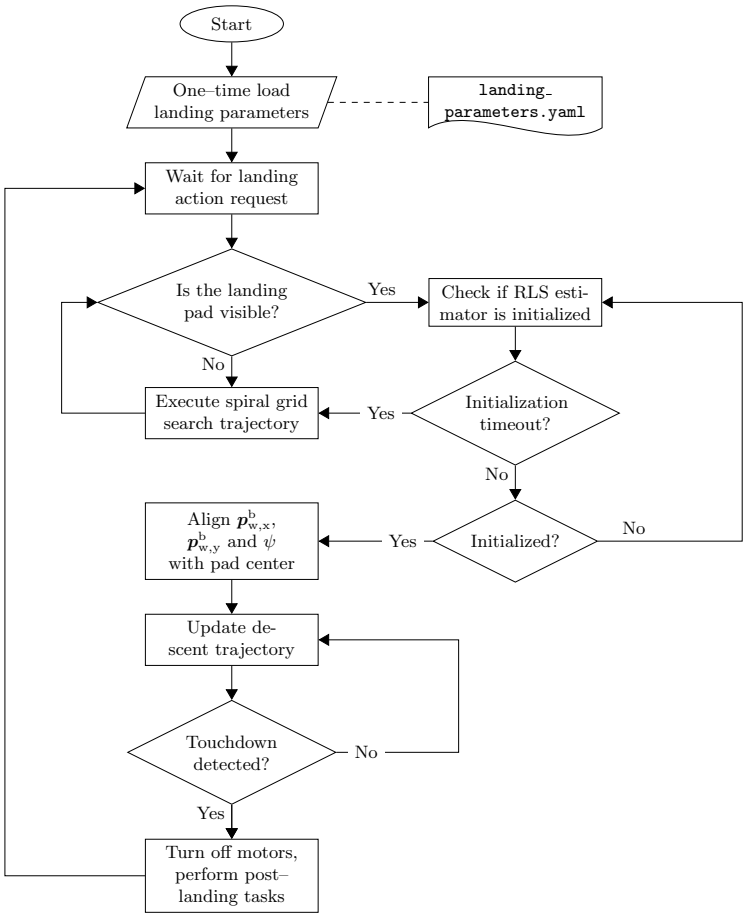


Figure 6.10: Landing algorithm flowchart.

Transition to BEGIN_MISSION

Guard The tracker finishes (Algorithm 22). Since the tracker uses the POSITION-TRACKING mode, this occurs when the quadrotor arrives with sufficient accuracy[4] at $(\boldsymbol{p}_{\mathrm{rth}}, h_{\mathrm{rth}})$.

Effect The tracker is paused and the return to home location is removed from $\Gamma_{\mathrm{list,mission}}$ (via POP_BACK()). The ROS action client (i.e. the master state machine) is notified that the mission is finished and the mission autopilot sleeps (SELFPAUSE()).

6.4 Landing Autopilot

The landing autopilot[5] implements the logic associated with the landing phase of Figure 6.1. The landing autopilot logic in Figure 6.10 is implemented as the state machine shown in Figure 6.11.

[4] The current implementation uses a 0.15 m position error and a 20 ° yaw error tolerance for Algorithm 22.

[5] `/autonomy_engine/alure_landing/include/alure_landing/landing_autopilot.h`

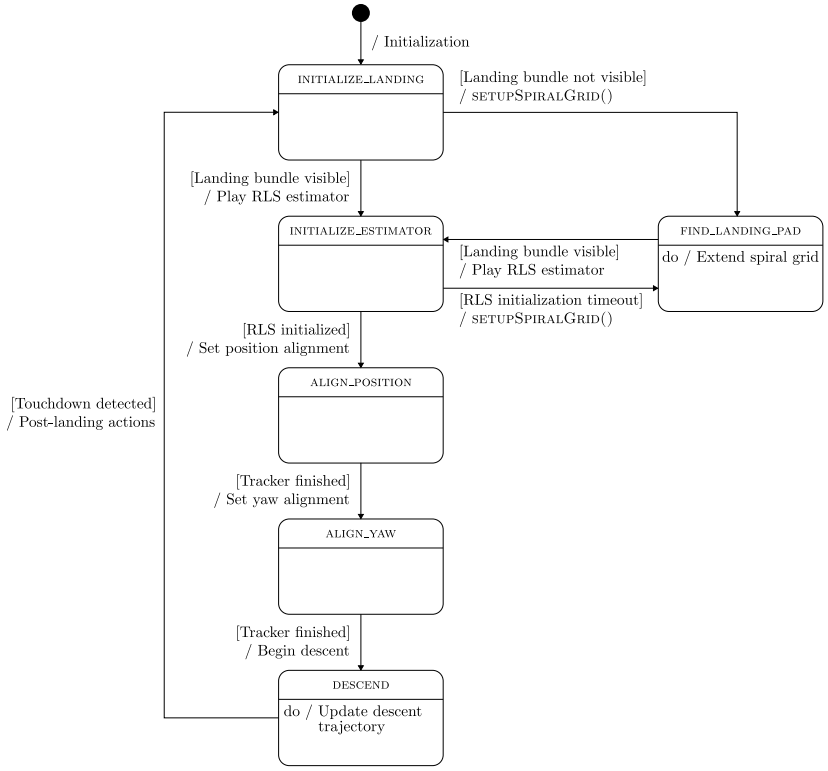


Figure 6.11: Landing state machine.

6.4.1 Initialization

Initial State INITIALIZE_LANDING

Loaded Parameters

Parameter	*Description*	*Units*
v_{descend}	Final descent speed	$^{\text{m}}/_{\text{s}}$
v_{search}	Speed of the spiral grid search segments	$^{\text{m}}/_{\text{s}}$
v_{align}	Speed of the position alignment trajectory with the charging pad origin	$^{\text{m}}/_{\text{s}}$
$v_{\text{touchdown}}$	Touchdown detection velocity threshold	$^{\text{m}}/_{\text{s}}$
$h_{\text{touchdown}}$	Touchdown detection height threshold	m
$t_{\text{rls,init,max}}$	Timeout for initializing the AprilTag RLS estimator in Figure 3.1	s
p_{search}	Horizontal and vertical advance fraction of the image ground size for the spiral grid search trajectory	-

Other Actions

The mission autopilot owns a trajectory list $\Gamma_{\text{list,landing}}$ and associates it with a PositionTracking trajectory tracker, as explained in Chapter 4.

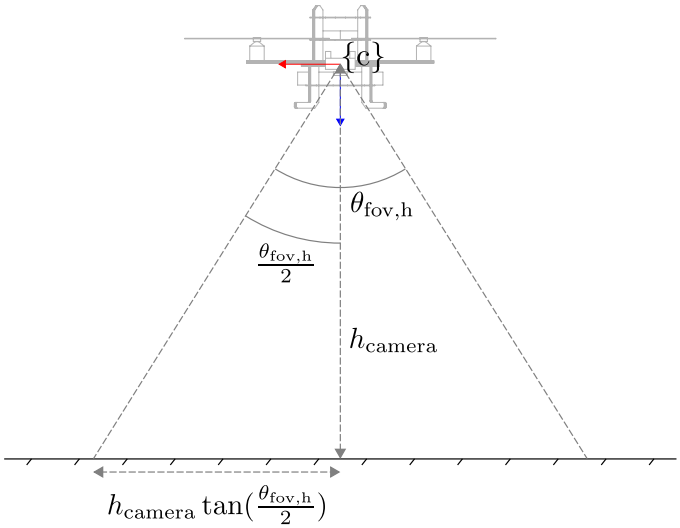


Figure 6.12: Image physical width as a function of camera height.

6.4.2 Runtime

State INITIALIZE_LANDING

This is a passive state which checks whether the landing AprilTag bundle is visible in the downfacing camera image. If yes, the AprilTag bundle pose RLS estimator[6] is played. Otherwise, a spiral grid search trajectory is begun.

Transition to INITIALIZE_ESTIMATOR

Guard The landing bundle is detected by the bundle pose measurement node[7] in Section 3.1.3 (the check is run for 5 seconds).

Effect The RLS estimator is started (via PLAY(), since it is implemented as a CORESTATEMACHINE as explained in Appendix B).

Transition to FIND_LANDING_PAD

Guard The landing bundle is not detected (i.e. the negation of the guard for the transition to the INITIALIZE_ESTIMATOR state).

Effect Define the spiral grid geometry and start flying the spiral via Algorithm 38. (6.1) is obtained via the geometry illustrated in Figure 6.12, with an identical geometry applying for (6.2).

State FIND_LANDING_PAD

This state extends the spiral grid search trajectory as long as the downfacing camera does not detect the landing bundle.

[6] `/autonomy_engine/alure_landing/include/alure_landing/charging_pad_pose_estimator.h`

[7] `/sensing/apriltags2_ros/apriltags2_ros/include/apriltags2_ros/continuous_detector.h`

Algorithm 38 Spiral grid geometry definition as required by Section 4.1.8.
`/autonomy_engine/alure_landing/src/landing_autopilot.cpp:setupSpiralGrid`
`/autonomy_engine/alure_landing/src/landing_autopilot.cpp:addNextSpiralSegmentTrajectory`

Define the spiral grid:
Call spiral grid RESET() ▷ See Section 4.1.8
Image physical size on the ground:
$h_{\text{camera}} \leftarrow \tilde{p}^{\text{c}}_{\text{w,z}}$
Let $\theta_{\text{fov,h}}$ and $\theta_{\text{fov,v}}$ be the camera horizontal and vertical FOV angles

$$w_{\text{im,ground}} \leftarrow 2h_{\text{camera}} \tan(\theta_{\text{fov,h}}/2) \tag{6.1}$$
$$h_{\text{im,ground}} \leftarrow 2h_{\text{camera}} \tan(\theta_{\text{fov,v}}/2) \tag{6.2}$$

Transform from spiral to world frame:
$h_{\text{spiral}} \leftarrow \tilde{p}^{\text{b}}_{\text{w,z}}$ ▷ Spiral at the IMU level
$\boldsymbol{q}^{\text{s}}_{\text{w}} \leftarrow$ convertYawToQuaternion(convertQuaternionToYaw($\tilde{\boldsymbol{q}}^{\text{c}}_{\text{w}}$))
Start tracking the spiral:
addNextSpiralGridSegment()
Play tracker

function addNextSpiralGridSegment()
    $\gamma \leftarrow \{\text{generateNextSegment}(), f^{-1}_{\text{guidance}}, \text{SINGLE}\}$ ▷ Extend spiral (Algorithm 15)
    $\Gamma_{\text{list,landing}}$->PUSH_BACK($\gamma$)
end function

Do Action

When the trajectory tracker is FINISHED() (Algorithm 22), addNextSpiralGridSegment() is called (Algorithm 38).

Transition to INITIALIZE_ESTIMATOR

Guard The landing bundle is detected by the bundle pose measurement node (same as for transition from INITIALIZE_LANDING to INITIALIZE_ESTIMATOR, but with an immediate timeout).

Effect The trajectory tracker is paused and $\Gamma_{\text{list,landing}}$'s RESET() method is called (see Section 4.2.2). This is done to wipe all spiral grid segment elements from the trajectory list. The RLS estimator is then played.

State INITIALIZE_ESTIMATOR

This is a passive state in which the landing autopilot waits until the RLS estimator is initialized.

Transition to FIND_LANDING_PAD

Guard $t_{\text{rls,init,max}}$ seconds have passed in this state (i.e. the RLS estimator is taking too long to initialize).

Effect It is assumed that the reason for the long initialization period is that the landing bundle is not visible, so no bundle pose measurements come in. This may be caused e.g. by position drift (due to GPS) or a "lucky" initial bundle pose measurement of a barely visible landing pad. SETUPSPIRALGRID() is called to start flying a spiral grid search trajectory.

Transition to ALIGN_POSITION

Guard The RLS is initialized (i.e. RLS_RUNNING state in Algorithm 8).

Effect Call Algorithm 39 to compute the pad-aligned body frame pose as illustrated in Figure 6.13. The frames $\{\mathrm{b_{des}}\}$ and $\{\mathrm{c_{des}}\}$ describe the pad-aligned body and camera frame poses respectively. These are computed such that the $\{\mathrm{l}\}$ frame's $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{l}}$ and $\boldsymbol{e}_{\mathrm{y}}^{\mathrm{l}}$ axes appear upright in the downfacing camera image, hence $\boldsymbol{q}_{\mathrm{c_{des}}}^{\mathrm{l}}$ corresponds to a 180 ° rotation about $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c_{des}}}$ (see Figure 6.14). The body frame origin is computed to be directly above the $\{\mathrm{l}\}$ frame origin. Finally, the trajectory tracker is played and the quadrotor begins tracking a realignment trajectory to it.

Algorithm 39 Computation of pad-aligned body pose, $\boldsymbol{p}_{\mathrm{align}}$ and $\boldsymbol{q}_{\mathrm{align}}$ and addition of a position alignment hover point to $\Gamma_{\mathrm{list,landing}}$.
`/autonomy_engine/alure_landing/src/landing_autopilot.cpp:setAlignmentLocation`
`/autonomy_engine/alure_landing/src/landing_autopilot.cpp:computeAlignedPositionAndYaw`

function SETALIGNMENTLOCATION()
  COMPUTEALIGNEDPOSITIONANDYAW($\tilde{p}_{\mathrm{w,z}}^{\mathrm{b}}$)
  $\sigma \leftarrow$CREATEHOVERPOINT($\boldsymbol{p}_{\mathrm{align}}$, 0)
  $\gamma \leftarrow \{\sigma, f_{\mathrm{guidance}}^{-1}, \mathrm{SINGLE}\}$ ▷ Trajectory list element
  $\Gamma_{\mathrm{list,landing}}$->PUSH_BACK($\gamma$)
end function

function COMPUTEALIGNEDPOSITIONANDYAW(h_{align})
Pad-aligned body frame attitude:
  $\boldsymbol{q}_{\mathrm{align}} \leftarrow (\boldsymbol{q}_{\mathrm{c_{des}}}^{\mathrm{l}} \otimes \tilde{\boldsymbol{q}}_{\mathrm{b}}^{\mathrm{c}})^{-1} \otimes \tilde{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{l,yaw}}$ ▷ $\tilde{q}_{\mathrm{b}}^{\mathrm{c}}$ from camera-IMU calibration
Pad-aligned body frame position:
  $\boldsymbol{p}_{\mathrm{align}} \leftarrow \tilde{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{l}} + (h_{\mathrm{align}} - \tilde{p}_{\mathrm{w,z}}^{\mathrm{l}})\boldsymbol{e}_3$
end function

State ALIGN_POSITION

This is a passive state during which the quadrotor aligns itself in position with the landing bundle origin (i.e. the $\{\mathrm{l}\}$ frame).

Transition to ALIGN_YAW

Guard Trajectory tracker's FINISHED() method evaluates to **true**.

Effect Set the tracker reference yaw ψ_{ref} (Algorithm 21) to CONVERTQUATERNIONTOYAW($\boldsymbol{q}_{\mathrm{align}}$).

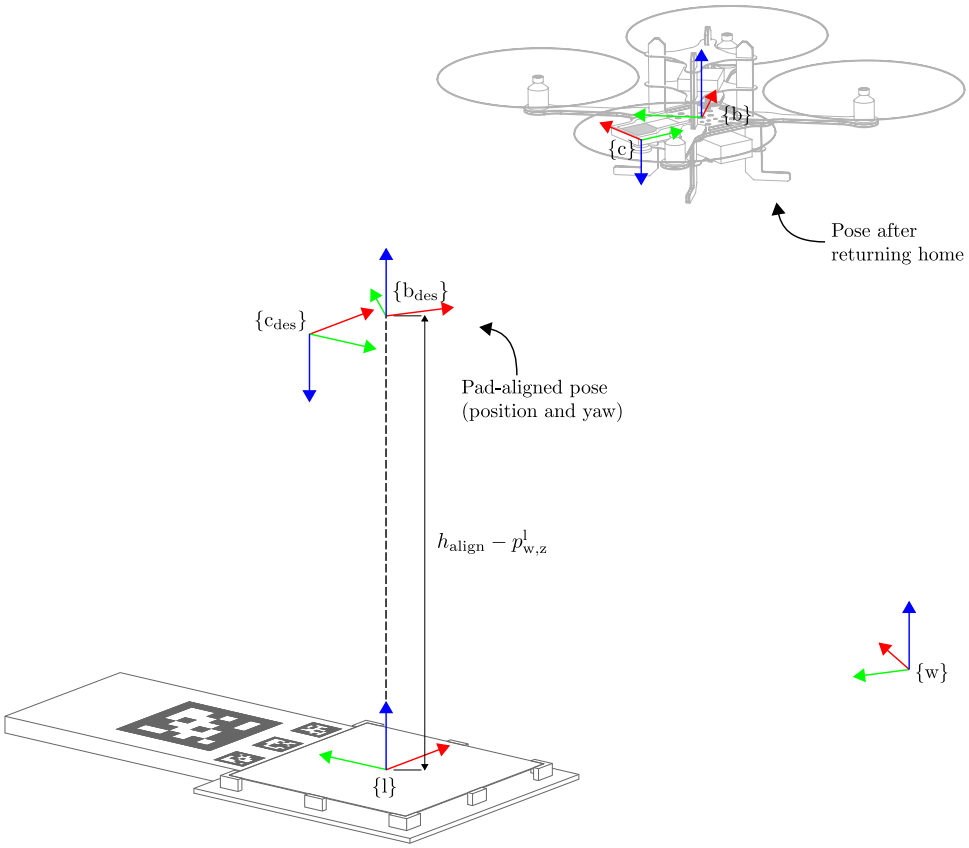


Figure 6.13: Frame setup for pad-aligned body frame pose computation.

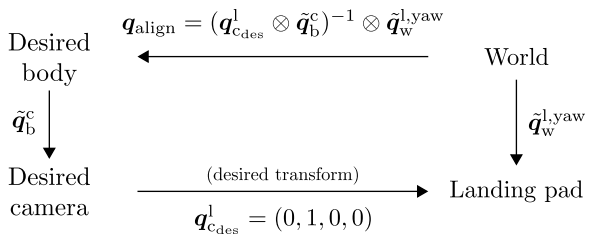


Figure 6.14: Quaternion map for computing $\boldsymbol{q}_{\text{align}}$.

State ALIGN_YAW

This is a passive state during which the quadrotor aligns itself in yaw with the landing bundle origin (i.e. the {l} frame) while holding $\boldsymbol{p}_{\text{align}}$.

Transition to DESCEND

Guard Trajectory tracker's FINISHED() method evaluates to **true**.

Effect Add the descent trajectory to $\Gamma_{\text{list,landing}}$ via UPDATEDESCENTTRAJECTORY(**false**) (Algorithm 40).

Algorithm 40 Landing final descent trajectory computation.
`/autonomy_engine/alure_landing/src/landing_autopilot.cpp:updateDescentTrajectory`

function UPDATEDESCENTTRAJECTORY(swap)
  COMPUTEALIGNEDPOSITIONANDYAW($\tilde{p}^{\text{b}}_{\text{w,z}}$) ▷ Update alignment with latest RLS estimate
  Compute alignment trajectory:
  $\boldsymbol{p}_0 \leftarrow \boldsymbol{p}_{\text{align}}$
  $\boldsymbol{p}_{\text{f}} \leftarrow (p_{\text{align,x}}, p_{\text{align,y}}, -1)$ ▷ Trajectory end 1 m underneath the landing pad
  $T \leftarrow \|\boldsymbol{p}_{\text{f}} - \boldsymbol{p}_0\| / v_{\text{descend}}$
  $\check{\boldsymbol{u}} \leftarrow (\boldsymbol{p}_{\text{f}} - \boldsymbol{p}_0) / \|\boldsymbol{p}_{\text{f}} - \boldsymbol{p}_0\|$
  Construct position transfer trajectory ($N = 10$) endpoint derivatives:

$$\boldsymbol{d}_0 \leftarrow (\boldsymbol{p}_0, v_{\text{descend}}\check{\boldsymbol{u}}, 0_{9\times 1})$$
$$\boldsymbol{d}_{\text{T}} \leftarrow (\boldsymbol{p}_{\text{f}}, v_{\text{descend}}\check{\boldsymbol{u}}, 0_{9\times 1})$$

  $\sigma \leftarrow$ CREATETRANSFERTRAJECTORY($\boldsymbol{d}_0$, $\boldsymbol{d}_{\text{T}}$, T)
  if swap **then**
    $\Gamma_{\text{list,landing}}$->SWAPCURRENTTRAJECTORY($\sigma$) ▷ Algorithm 20
  else
    $\gamma \leftarrow \{\sigma, f^{-1}_{\text{guidance}}, \text{SINGLE}\}$
    $\Gamma_{\text{list,landing}}$->PUSH_BACK($\gamma$)
  end if
end function

State DESCEND

In this state the quadrotor is vertically descending while maintaining $p_{\text{align,x}}$, $p_{\text{align,y}}$ and $\boldsymbol{q}_{\text{align}}$. In other words, the quadrotor is reducing altitude while maintaining horizontal and yaw alignment with the landing bundle origin (i.e. center of the charging pad).

Do Action

Call UPDATEDDESCENTTRAJECTORY(**true**) (Algorithm 40) in order to update the trajectory with the latest landing bundle pose estimate.

Transition to INITIALIZE_LANDING

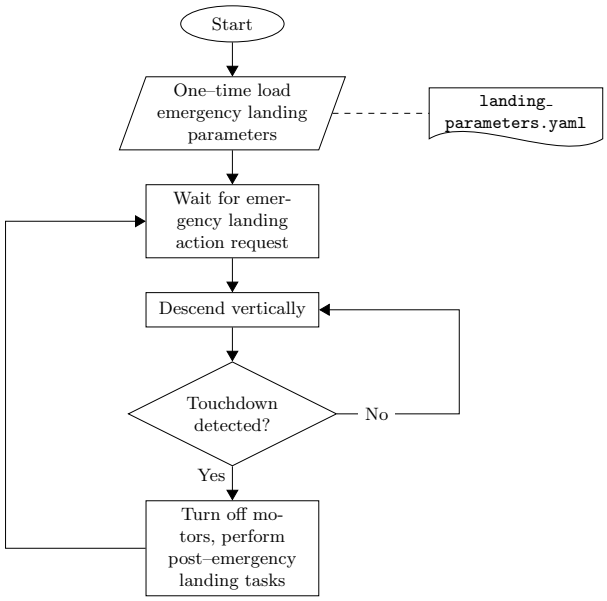


Figure 6.15: Emergency lander algorithm flowchart.

Guard Touchdown detection when the following condition evaluates to **true**:

$$(\tilde{p}^{\mathrm{b}}_{\mathrm{w,z}} - \tilde{p}^{\mathrm{l}}_{\mathrm{w,z}}) < h_{\mathrm{touchdown}} \textbf{ and } |\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}| < v_{\mathrm{touchdown}}, \tag{6.3}$$

where the small-velocity condition $|\tilde{v}^{\mathrm{b}}_{\mathrm{w,z}}| < v_{\mathrm{touchdown}}$ is in practice crucial for a robust touchdown detection. With this condition, touchdown is detected immediately on ground contact rather than at some non-zero height above the landing pad.

Effect The motors are turned off, the trajectory tracker is paused and $\Gamma_{\mathrm{list,landing}}$ is RESET(). The RLS estimator is also paused and RESETESTIMATOR() is called (Section 3.1.5). The ROS action client (i.e. the master state machine) is notified that landing is finished and the landing autopilot sleeps (SELFPAUSE()).

6.5 Emergency Lander

The emergency lander[8] implements the logic associated with the emergency landing functionality which brings the quadrotor to a soft touchdown at its current location. This is not to be confused with the on-board (i.e. on-HLP) emergency landing controller of Section 5.9. While the latter is a fallback algorithm for landing the quadrotor without a state estimate, the emergency lander described herein is off-board (i.e. on-Odroid XU4) and simply lands the quadrotor at the current location (with the state estimate available). The emergency lander logic in Figure 6.15 is implemented as the state machine shown in Figure 6.16.

6.5.1 Initialization

Initial State WAIT_EMERGENCY

[8] `/autonomy_engine/alure_landing/include/alure_landing/emergency_lander.h`

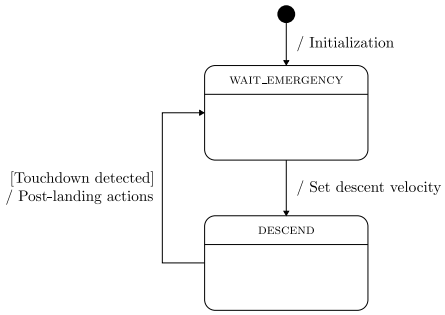


Figure 6.16: Emergency lander state machine.

Loaded Parameters

The emergency lander reuses v_{descend} and $v_{\text{touchdown}}$ parameters from the landing autopilot in Section 6.4.

6.5.2 Runtime

State WAIT_EMERGENCY

This is a passive state in which the emergency lander is paused.

Transition to DESCEND

Effect The translation controller is set to the velocity mode (see Section 5.8) and a descent velocity $(0, 0, -v_{\text{landing}})$ is commanded. Note the simplicity – no trajectory gets generated, the quadrotor is simply told to descend vertically (until it eventually hits the ground). Lastly, the emergency lander sleeps for 1 second in order to give the quadrotor time to reach the descent velocity before checking (6.4) in the DESCEND state.

State DESCEND

The emergency lander waits in this state for the quadrotor to hit the ground.

Transition to WAIT_EMERGENCY

Guard Touchdown is detected when the following condition evaluates to **true**:

$$\|\tilde{\boldsymbol{v}}_{\text{w}}^{\text{b}}\| < v_{\text{touchdown}}. \tag{6.4}$$

Note that (6.4) does not check the height because $\tilde{p}_{\text{w,z}}^{\text{b}}$ is not perfectly ground-referenced, thus does not provide a robust touchdown condition. Checking height is possible for the landing pad because the AprilTag bundle pose estimate provides a reliable height estimate.

Effect The motors are turned off and the descent velocity reference publisher is paused. The ROS action client (i.e. the master state machine) is notified that emergency landing is finished and the emergency lander sleeps (selfPause()).

6.6 Master State Machine

The master state machine[9] coordinates calls to the takeoff, mission, landing and emergency landing autopilots in order to execute the full-cycle autonomous data acquisition cycle in Figure 6.1. It is a special state machine in that it does no computations itself. It only makes state transitions based on events and off-loads computations to the phase-specific autopilots. A flowchart was not a good design tool for the master state machine, so a state machine was developed directly as shown in Figure 6.17.

6.6.1 Initialization

Initial State CHARGING

Loaded Parameters

Parameter	*Description*	*Units*
V_{charged}	Battery charged voltage	V
V_{rth}	Battery low voltage	V
V_{critical}	Battery critical voltage	V
$t_{\text{sync,max}}$	Timeout for data synchronization	s

Other Actions

The following events are defined (see Appendix B for a discussion on deferred):

Event	*Invocation*	*Description*
PAUSE	User	Pause the autonomy engine (if flying, lands the quadrotor on charging pad first).
FORCELAND	User	Return to charging pad (but do not pause state autonomy engine after landing!). Useful for testing.
FORCETAKEOFF	User	Take off even if battery is not charged. Useful for testing.
EMERGENCYLAND	User	Execute emergency landing. Useful for landing.
BATTERYCHARGED	$V_{\text{bat}} > V_{\text{charged}}$	Battery is charged
BATTERYLOW	$V_{\text{bat}} < V_{\text{rth}}$	Battery charge is low
BATTERYCRITICAL	$V_{\text{bat}} < V_{\text{critical}}$	Battery charge is critically low

Note that the "User" events are invoked by the user via a ROS service call.

6.6.2 Runtime

State CHARGING

In this state the quadrotor is sitting (with motors off) on the charging pad while the battery charges. At the same time, mission data is uploaded to a ground computer.

[9] `/autonomy_engine/alure_main_sm/include/alure_main_sm/main_state_machine.h`

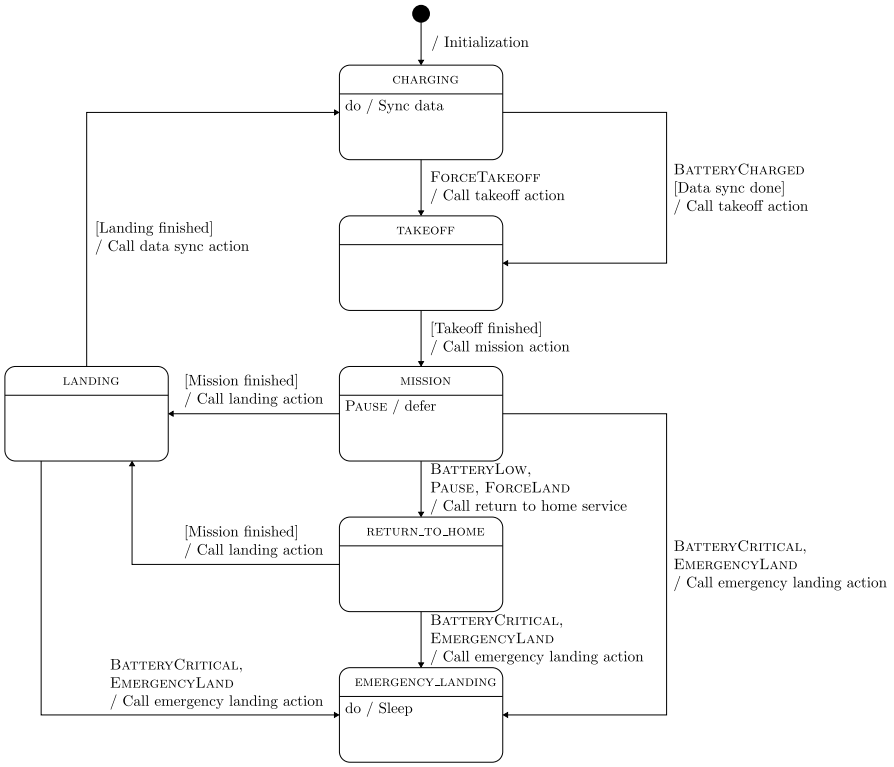


Figure 6.17: Master state machine.

Do Action

The data synchronizer[10] is monitored for finishing (its action returns successfully). If more than $t_{\text{sync,max}}$ time passes, the synchronization is timed out. If the Pause event is invoked, the state machine is put to sleep (selfPause()).

Transition to TAKEOFF

Events ForceTakeoff.

Effect The takeoff autopilot is called to action.

Transition to TAKEOFF

Events BatteryCharged.

Guard Data from the previous mission has either been uploaded to the ground computer (the ROS action returns successfully) or a timeout for it has been reached (currently set to 1 hour).

Effect The takeoff autopilot is called to action.

[10]Data synchronization is provided by `rsync_ros` McClung [129].

State TAKEOFF

The takeoff autopilot performs a takeoff during this state.

Transition to MISSION

Guard The takeoff action returns successfully.

Effect The mission autopilot is called to action.

State MISSION

The mission autopilot flies the data acquisition mission during this state.

Transition to LANDING

Guard The mission action returns successfully (i.e. the data acquisition mission is finished and the quadrotor has returned to home, hovering somewhere close by to the charging pad).

Effect The landing autopilot is called to action.

Transition to RETURN_TO_HOME

Events BatteryLow, Pause **or** ForceLand.

Effect Externally invoke the ReturnToHome event of the mission autopilot (see Section 6.3.2).

Transition to EMERGENCY_LANDING

Events BatteryCritical **or** EmergencyLand.

Effect Abort the mission autopilot (see Appendix B preempt note) and call the emergency lander to action.

State RETURN_TO_HOME

In this state the quadrotor is flying back to $(\boldsymbol{p}_{\text{rth}}, h_{\text{rth}})$ (see Section 6.3) from the point where the data acquisition trajectory was aborted.

Transition to EMERGENCY_LANDING

Events BatteryCritical **or** EmergencyLand.

Effect Abort the mission autopilot (see Appendix B preempt note) and call the emergency lander to action.

Transition to LANDING

Guard The mission action returns successfully (i.e. the data acquisition mission is finished and the quadrotor has returned to home, hovering somewhere close by to the charging pad).

Effect The landing autopilot is called to action.

 **State** LANDING

The landing autopilot lands the quadrotor on the charging pad during this state.

Transition to EMERGENCY_LANDING

Events BATTERYCRITICAL **or** EMERGENCYLAND.

Effect Abort the landing autopilot (see Appendix B preempt note) and call the emergency lander to action.

Transition to CHARGING

Guard The landing action returns successfully (the quadrotor is now safely sitting back on the charging pad).

Effect The data synchronizer is called to action, which begins uploading the mission data to the ground computer.

 **State** EMERGENCY_LANDING

This is "terminal" state in which the master state machine remains paused forever. Once in this state, no further interaction with the master state machine is possible. Manual operator intervention is required post-emergency landing since, by design, the quadrotor battery charge is dangerously low if this state is entered.

Do Action

Pause the state machine (only run once, since thereafter the state machine is paused).

Chapter 7

Results

This chapter presents the properties and the results of the GNC and autonomy engine implementations.

7.1 Overview

Data has been assembled from three sources – simulation, indoor and outdoor flights – as described below. In the following sections, "+" and "–" list items shall represent advantages and disadvantages respectively.

7.1.1 RotorS Simulation

The RotorS open-source MAV simulator Furrer et al. [122] consists of a set of worlds, models, plugins and ROS interfaces for the Gazebo simulation environment Koenig and Howard [130]. RotorS is a mature project with several key benefits:

+ It seamlessly integrates with the ROS environment and its design aims to make the simulation-to-hardware transition as easy as replacing the Gazebo simulator interface with a ROS node that does hardware communication;

+ It uses the fully-featured, stable, mature and widely used Open Dynamics Engine (ODE) for physics simulation Russell Smith [131]. This makes for a high-fidelity simulation;

+ Gazebo uses OGRE [132] for 3D graphics rendering which enables running computer vision algorithms such as the landing navigation pipeline.

These aspects make RotorS a natural choice for virtually testing the GNC system and autonomy engine in a higher fidelity and more expansive environment than what a MATLAB/Simulink simulation can offer. Indeed, RotorS[1,2] was expanded and partially modified to run the actual complete system presented herein, except for SSF (ground truth state is used instead). This creates a SIL simulation of the full system and future work could expand to even create a HIL simulation. Results gathered with RotorS have the following trade-offs:

+ The simulation can be made as "perfect" as the engineer wishes. For instance, ground truth state can be provided and motor dynamics can be removed (such

[1] /simulation_packages/rotors_simulator

[2] /simulation_packages/rotors_simulator/rotors_control/include/rotors_control/jpl_multirotor_control_interface_node.h

that the motors respond instantly to ESC commands). By stripping away complexity, the engineer can expose underlying algorithmic or programming flaws which cause adverse behavior even in a perfect world;

+ The simulation removes the logistic complexity of testing a drone. The engineer can let the drone fly as long and far as is wished without worrying about battery life or Federal Aviation Administration (FAA) permits. Naturally, a simulation offers orders of magnitude faster iterations than flight tests in the early to middle stages of algorithm design;

+ The simulation enables testing extreme conditions such as recovery from an inverted orientation (Section 7.4.2) or flight in severe wind (Section 7.6.2). These conditions are either dangerous or simply difficult to reproduce in reality but serve as useful indicators of system performance limits;

+ The simulation is 100% true-to-reality for the autonomy engine and guidance pipelines. The correct functioning of these non-vision, non-physics related subsystems quasi-guarantees their correct operation in real life;

+ The simulation is a highly beneficial training tool for new engineers working with the system, enabling them to learn operating procedures in a risk-free environment;

– The simulation inevitably uses simplified quadrotor dynamics. This begins with negligible simplifications within ODE[3] and extends through significant simplifications such as non-misaligned motor thrust vectors, simplified propeller aerodynamics, no ground effect, simplified motors dynamics, simplified wind interactions, an assumption that the quadrotor is a rigid body, etc. Some of these simplifications can potentially be removed via additional Gazebo plugins. These simplifications affect the control subsystem in that real life performance is expected to be worse (in the sense that the simulation shows a best-case performance);

– The simulation uses ground truth (i.e. perfect) state rather than SSF and its array of sensors. Including SSF in the simulation is entirely possible (see Section 8.2) but was not done due to being out of this thesis' scope;

– Gazebo's default usage of OGRE does not provide photo-realistic rendering and therefore computer vision algorithms also operate with best-case performance.

Throughout this chapter, simulation data shall be referred to as *RotorS data*.

7.1.2 Test Flights Using VICON Pose Measurements

The next step up in complexity from simulation are indoor test flights using SSF to fuse IMU data with the highly accurate and precise pose measurements from a VICON Bonita [133] matrix of infrared cameras. Results gathered indoors have the following trade-offs:

+ Indoor test flights can be performed more rapidly because they are not encumbered by the logistics of flight permits, equipment setup, weather, etc. associated with outdoor flight. This makes for a faster design feedback cycle;

[3] E.g. the friction cone is approximated as a pyramid for computational efficiency.

+ VICON offers a best-case state estimate and there are no wind disturbances indoors. This is an advantage during the mid-stages of design because additional complexities of limited sensing and external disturbances do not have to be dealt with while more basic issues are being worked out;

– The above base-case state estimate and no wind can also be disadvantages because limited sensing and wind are inherent qualities of outdoor flight to which the system must be robust, but which cannot readily be tested in this setup;

– The usable VICON volume is approximately a $3 \times 3 \times 3\,\text{m}$ cube. This is too small to test the system's full operating range in terms of speed, initial height above charging pad for landing, etc.

Throughout this chapter, data collected from indoor flight tests shall be referred to as *indoor data*.

7.1.3 Outdoor Flight Tests

Outdoor flight tests correspond to complete system tests in an outdoor environment. Data collected in this manner has the following trade-offs:

+ These are truly *complete* system tests and therefore expose all the strengths and weaknesses of the implementation;

– Good data is hard to collect. Outdoor flights not requiring a license need to be tethered. In combination with limited flying space and building obstructions on the JPL campus, the system cannot be flown at high altitude while low-altitude outdoor flights suffer from worst-case state estimate conditions due to GPS multipath effects, etc. Non-tethered flights, meanwhile, require an FAA license and an open space (e.g. the Mars Yard) and are thus much more logistically demanding. Consequently, outdoor flight testing is time consuming and leads to slow design iterations.

Throughout this chapter, data collected from indoor flight tests shall be referred to as *outdoor data*.

7.1.4 Chapter Organization

The remainder of this chapter presents results that aim to highlight important/novel aspects of the system. The presentation is structured as follows:

- Section 7.2 compares the landing bundle pose RLS estimator output to raw bundle pose measurements;
- Section 7.3 presents guidance realignment trajectory generation;
- Section 7.4 evaluates the robustness and performance of the cascaded control system;
- Section 7.5 demonstrates the autonomy engine in action for nominal flight as well as several robustness edge cases;
- Section 7.6 provides statistics on landing precision, which is the most complex flight phase and is one of the most important contributions of this thesis;
- Section 7.7 shows a battery voltage curve which demonstrates autonomous recharging over an 11 hour full-cycle autonomous flight test.

7.2 Navigation

Figure 7.1 compares the landing bundle pose measurements $\hat{\boldsymbol{p}}^{\mathrm{l}}_{\mathrm{w}}$ and $\hat{\psi}_{\mathrm{l}}$ (Algorithm 2 output) to the pose estimates $\tilde{\boldsymbol{p}}^{\mathrm{l}}_{\mathrm{w}}$ and $\tilde{\psi}_{\mathrm{l}}$ (Algorithm 8 output). The graphs show that the RLS estimator is both more precise (has lower variance) and more accurate (is closer to the ground truth). This comparison is representative of all performed landings. Rather than evaluate statistical properties of the RLS estimator as a standalone system, we follow a more utilitarian approach and evaluate directly the full-system precision landing performance in Section 7.6.

7.3 Guidance

The main contribution of the guidance subsystem is the trajectory sequencer, which implements precise sequencing mechanics to allow high-level real-time sequencing of trajectories, and the trajectory tracker's realignment trajectory feature which employs the VOLATILE trajectory list element.

Figure 7.2 presents both features in action for a case scenario of the landing phase where large disturbances are introduced by manually flying the quadrotor away from the desired trajectory. In Figure 7.2a, the quadrotor (top left frame) has returned home and the downfacing camera detects the landing bundle (bottom right frame, showing also the {w} frame origin). In Figure 7.2b, the landing autopilot is triggered and a position alignment hover point is created above the landing bundle position measurement. The quadrotor begins tracking a VOLATILE realignment trajectory towards it, since the alignment hover point is $> 1\,\mathrm{m}$ away from the quadrotor's current location. However, a disturbance in Figure 7.2c, once it becomes large enough, triggers in Figure 7.2d the re-generation of a realignment trajectory based on the quadrotor's current position and velocity. The original realignment trajectory is removed as per the mechanics in Section 4.2 (see Figure 4.6d). This happens again in Figure 7.2e. Eventually, the quadrotor arrives at the pad-aligned position, aligns itself in yaw and begins the final descent in Figure 7.2f. A yet another disturbance in Figure 7.2g triggers a realignment trajectory. This time, it brings the quadrotor back to the original (playback type SINGLE) descent trajectory with a matching position and velocity in Figure 7.2h. The quadrotor comes to a soft, precise landing in Figure 7.2i.

Section 7.5 shall present robustness cases that use other guidance features, namely the spiral grid search trajectory in Figure 7.22.

7.4 Control

The controllers presented in Chapter 5 shall now be evaluated for their performance and, where it was not already done in Chapter 5, their robustness.

7.4.1 Body Rate Control

This section evaluates the body rate controller developed in Section 5.6.

Tracking Performance

Recall that for a generic variable x and its control reference x_{ref}, the Root Mean Square Error (RMSE) tracking error is defined as:

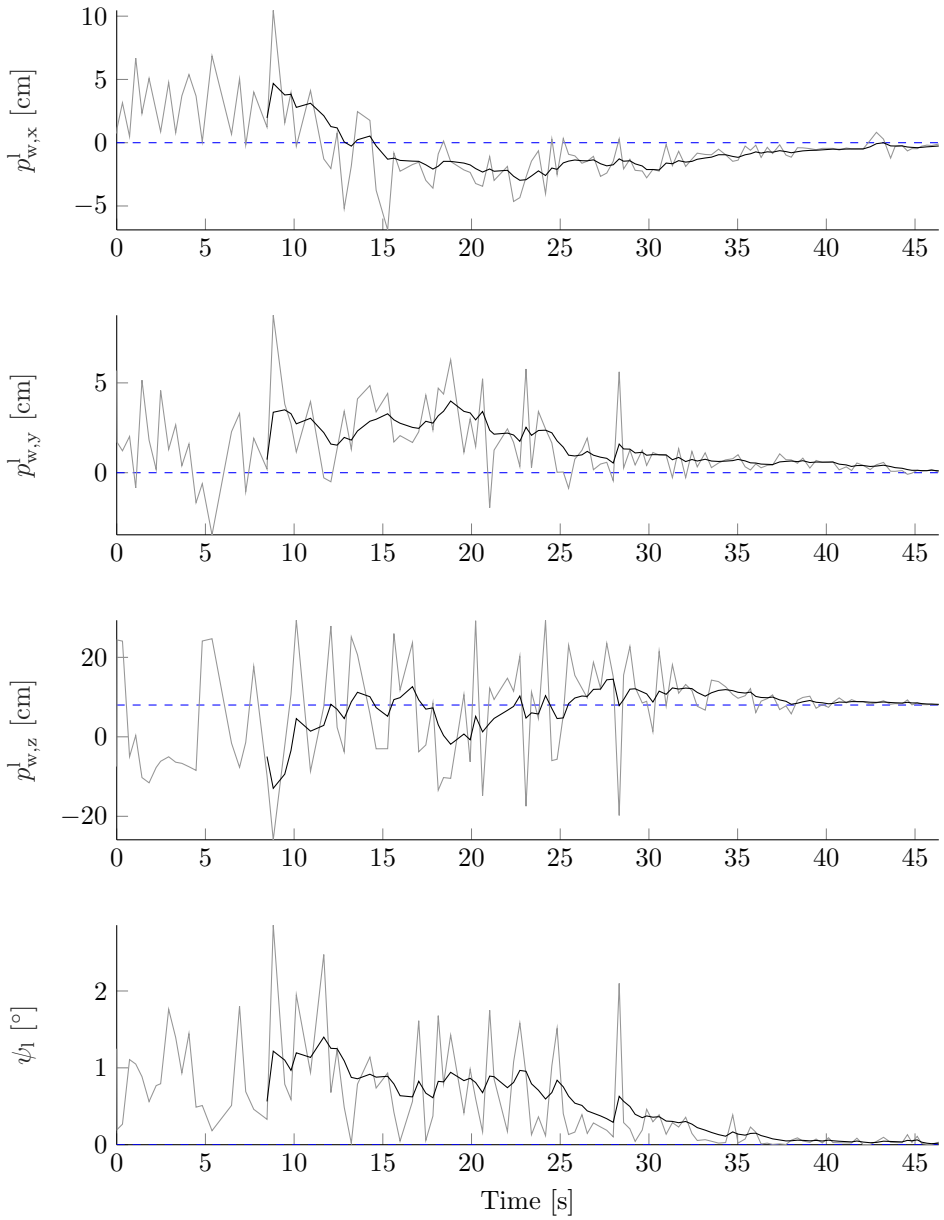


Figure 7.1: Landing bundle pose estimate compared to raw measurements during a representative landing. The blue dashed line shows ground truth, the gray line shows AprilTag 2 measurements and the black line shows the RLS estimate mean. *RotorS data.*

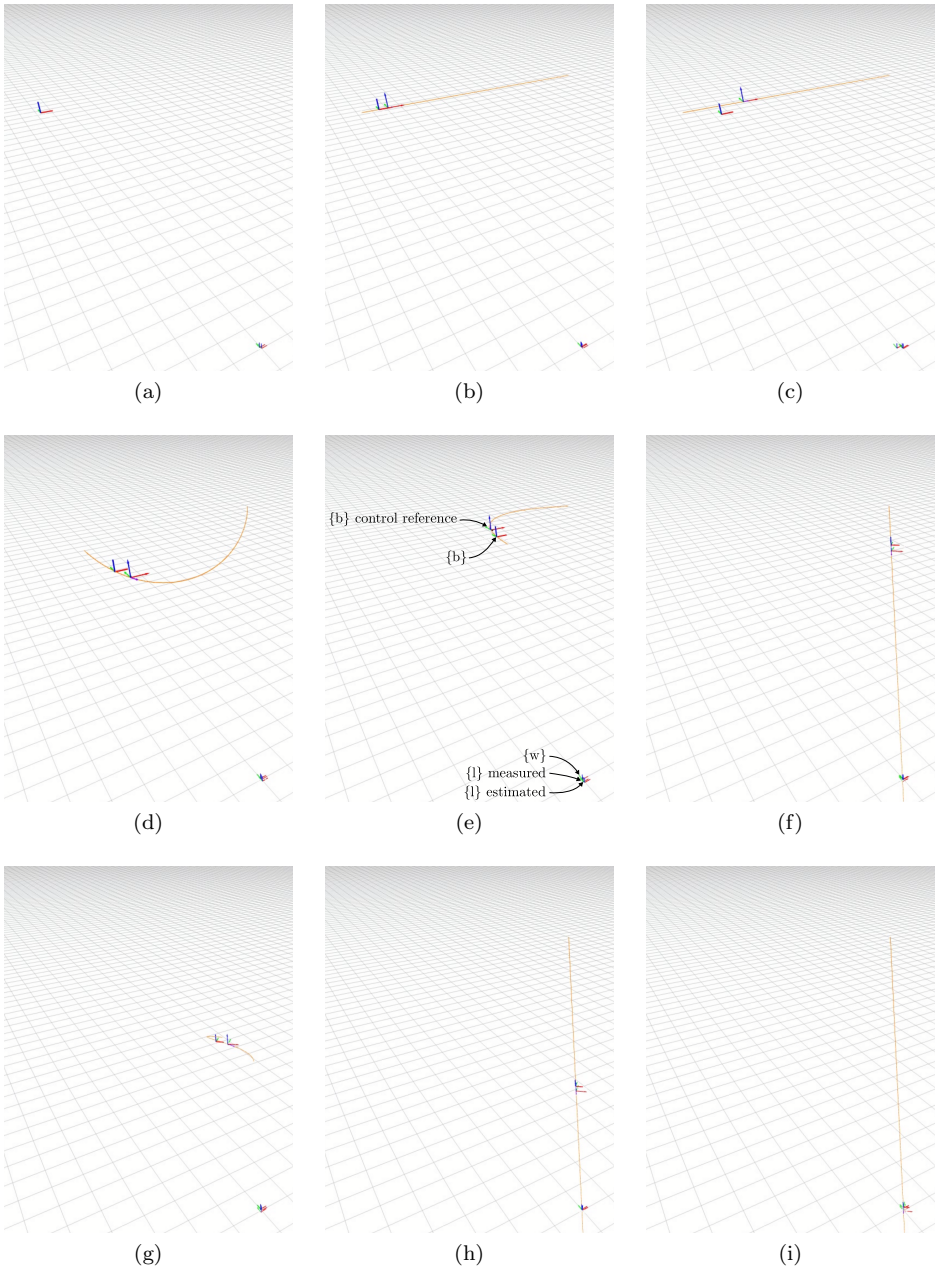


Figure 7.2: Realignment trajectory illustration during the landing phase. (e) annotates the shown frames. Note that the {l} measured and estimated frames correspond to the Figure 3.1 RLS estimator block input and output respectively. *RotorS data.*

Quantity	RMSE [°]	$\text{NRMSE}_{\text{range}}$ [-]
Indoor ω_{x}	9	0.182
Indoor ω_{y}	9	0.079
Indoor ω_{z}	4	1.074
RotorS ω_{x}	16	0.078
RotorS ω_{y}	14	0.092
RotorS ω_{z}	7	0.072

Table 7.1: Body rate tracking RMSE and $\text{NRMSE}_{\text{range}}$.

$$\text{RMSE} := \sqrt{\frac{1}{N}\sum_{k=1}^{N}(\hat{x}[k] - x_{\text{ref}}[k])^2}, \tag{7.1}$$

where N is the sample size. Recall also the Normalized RMSE (NRMSE), for which two alternative definitions exist:

$$\text{NRMSE}_{\text{range}} := \frac{\text{RMSE}}{\max_{k=1,\dots,N} x_{\text{ref}}[k] - \min_{k=1,\dots,N} x_{\text{ref}}[k]}, \tag{7.2}$$

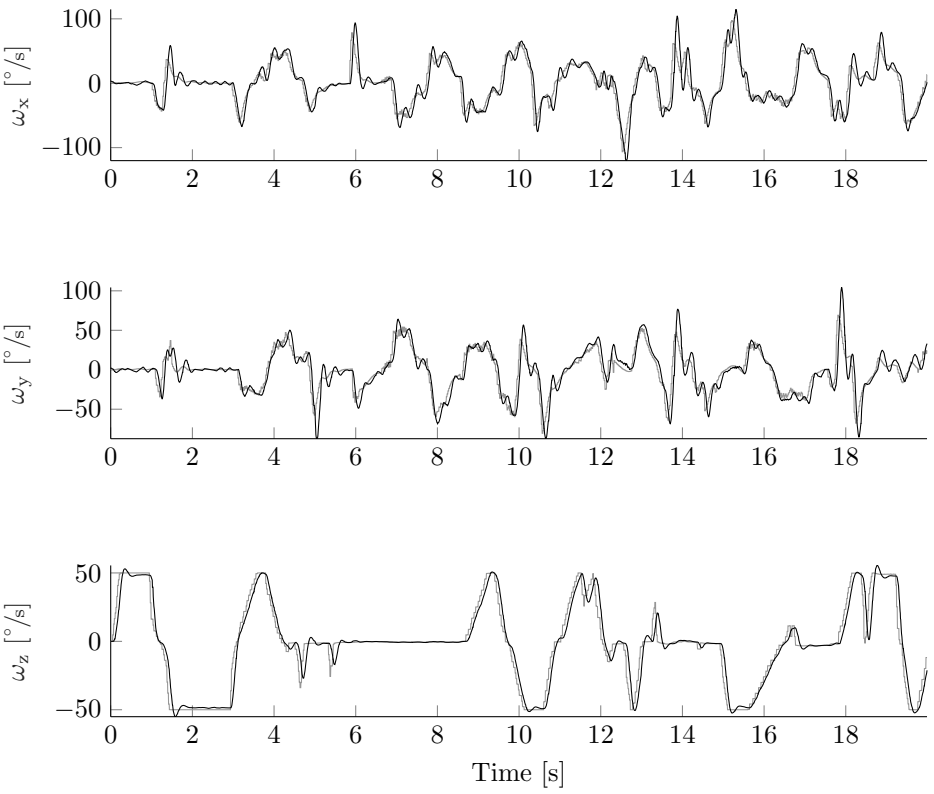
$$\text{NRMSE}_{\text{mean}} := \frac{\text{RMSE}}{\frac{1}{N}\sum_{k=1}^{N} x_{\text{ref}}[k]}, \tag{7.3}$$

While the RMSE is a useful absolute measure of tracking accuracy, it cannot be readily used to compare two different experiments where e.g. controller reference ranges are different. A controller required to track a reference that sweeps a greater range will naturally achieve lower accuracy due to nonlinearities like saturation. NRMSE, meanwhile, enables such a comparison via normalization. $\text{NRMSE}_{\text{range}}$ is particularly useful when the mean of x_{ref} evaluates close to zero. Since the quadrotor hovers on average, this is the case for the components of $\boldsymbol{\omega}_{\text{ref}}$, thus $\text{NRMSE}_{\text{range}}$ is used for the body rate controller performance comparison.

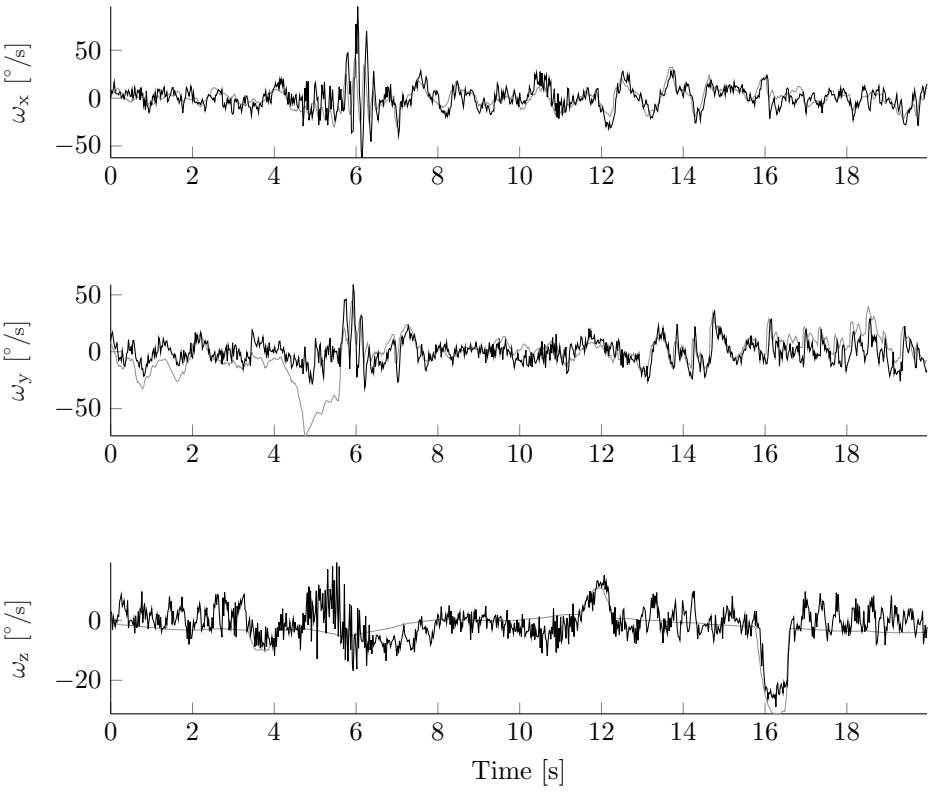
Figure 7.3 shows simulated and real body rate tracking. Based on this data, Table 7.1 quantifies the $\text{NRMSE}_{\text{range}}$ with the objective of comparing simulated to real body rate tracking performance. As expected, using the noisy gyro measurements the real body rate controller performs worse than its simulated counterpart (which uses ground-truth $\boldsymbol{\omega}$). We note that, as expected, tracking accuracy is symmetric in ω_{x} and ω_{y} due to the quadrotor's radial symmetry. The $\text{NRMSE}_{\text{range}}$ is misleading for the indoor data ω_{z} tracking, whose range is quasi-zero. One also sees small oscillations in the simulated body rates which are due to the weak time scale separation between the body rate controller and the relatively slow AscTec BLDC motor dynamics (see Appendix D).

Prioritized Saturation

A tether attached to the quadrotor became taught in the $t \in [4, 6]$ second period in Figure 7.3. This resulted in significantly worse tracking during this time period. Figure 7.4 shows the body torques corresponding to Figure 7.3. One sees that during this time period the body rate controller gave up almost all ξ_{z} and some collective thrust f while not compromising ξ_{x} and ξ_{y} as the quadrotor attempts to recover from the disturbance.



(a) *RotorS data.*



(b) *Indoor data.*

Figure 7.3: Body rate tracking. Gray and black lines show reference and measured body rates, respectively.

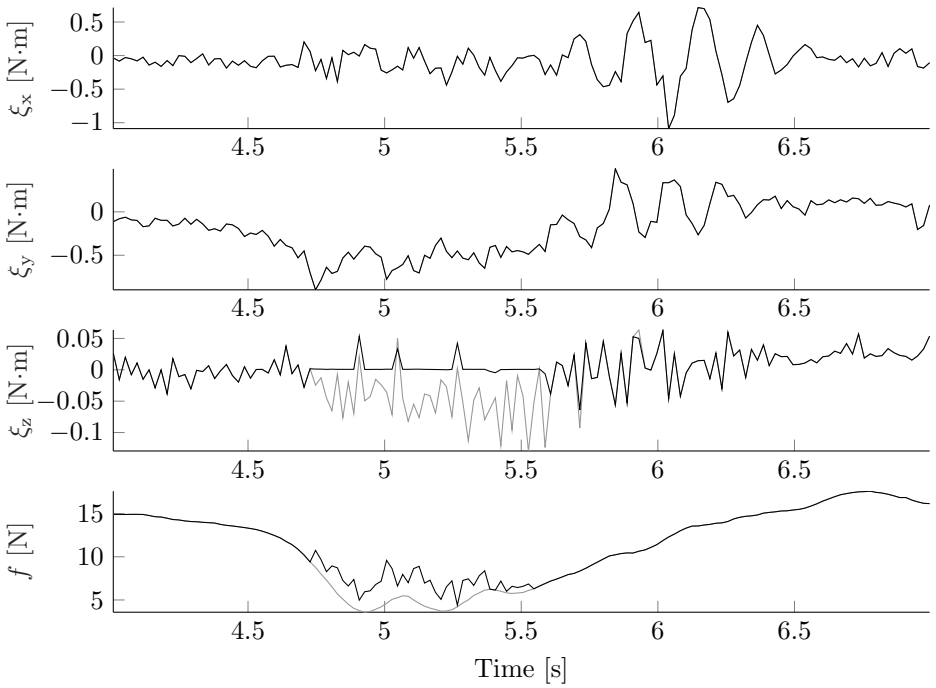


Figure 7.4: Prioritized thrust saturation gives up z-torque ξ_z, then collective thrust f, when the safety tether tightens and produces an exogenous disturbance. *Indoor data.*

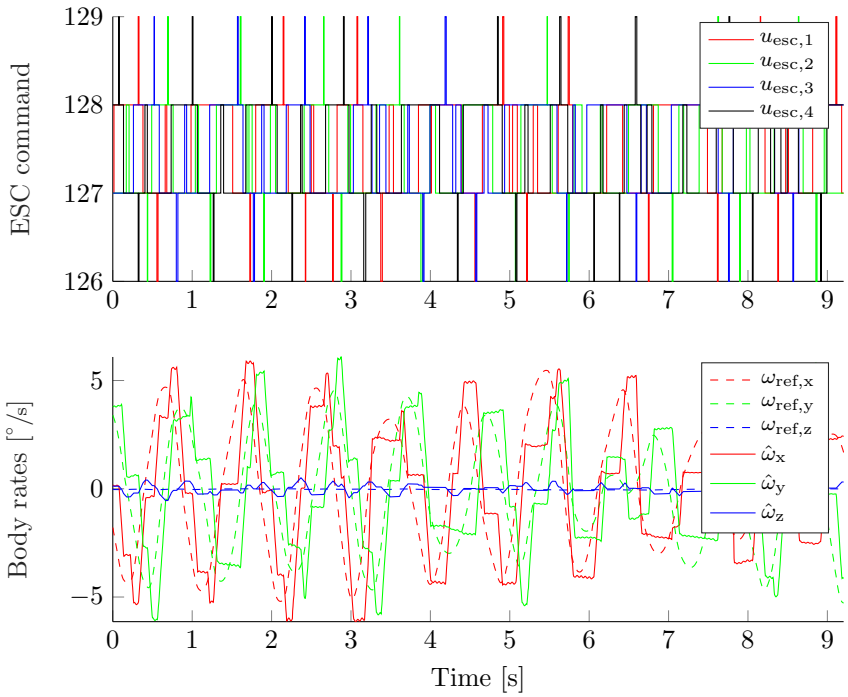


Figure 7.5: ESC discretization from 0 to 200 induces small body rate oscillations at hover. *RotorS data.*

ESC Discretization Note

The AscTec ESC command is discretized in the 0 to 200 range. Consequently, only a discrete set of body torques is achievable. The result in practice is a constant body torque overshoot to bring the quadrotor to a perfectly level position, leading to small body rate oscillations. As shown in Figure 7.5, these oscillations are clearly visible in simulation where, thanks to using ground truth measurement, perfect hover really is achievable should the ESC command have been continuous. In reality, the discretization effect is hard to distinguish from many other factors (e.g. state estimate error, model uncertainty) that prevent perfect hover but is expected to have a naturally adverse effect.

7.4.2 Attitude Controller

This section evaluates the attitude controller developed in Section 5.7.

Step Responses

The RotorS simulator is used to determine the step response in the $\boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}}$ components.

Roll-pitch responses. Step responses in $q_{\mathrm{w,x}}^{\mathrm{b}}$ and $q_{\mathrm{w,y}}^{\mathrm{b}}$ are achieved by, starting from hover, commanding $\boldsymbol{a}_{\mathrm{ref}}$ to $(10, 0, 0)$ and to $(0, -10, 0)$ respectively. This creates a positive step of $\approx 45\,^\circ$ in attitude about the respective axis, given that the quadrotor does not accelerate in any other axis and accounting for gravity.
The results are shown in Figures 7.6a and 7.6b. Unlike the theoretical first-order response demonstrated in Section 5.7.1, the actual step response exhibits higher-order dynamics due to the weak time-scale separation with the body rate control. Nevertheless, in both cases the response reaches 63.2 % of the reference in $\approx 0.2\,\mathrm{s}$ which is close to the expected 0.25 s given Table 5.3[4].

Yaw response. A yaw step response is achieved by, starting from hover with $\psi = 0\,^\circ$, commanding a $\psi_{\mathrm{ref}} = 180\,^\circ$. Except for the initial peak in ω_{z} (again due to motor dynamics), the step response is close to that of a first-order system. However, the time constant in this case is $\approx 1.5\,\mathrm{s}$ unlike the expected 2.5 s as according to Table 5.3. This may be explained by the large yaw error of $180\,^\circ$ while the first-order system approximation applies only for small attitude errors.

Yaw Heuristic

The yaw heuristic is tested as follows. The quadrotor is initially commanded to rotate with different values of ω_{z}. When ψ reaches $90\,^\circ$ with an accuracy of $1\,^\circ$, $\psi_{\mathrm{ref}} = 0\,^\circ$ is set. Figure 7.7 shows the results. Interestingly, the heuristic increases the time to return to $\psi = 0\,^\circ$. Brescianini et al. [106] show, meanwhile, that in their case the heuristic is beneficial. It appears that the heuristics may only have a positive effect for high $\omega_{\mathrm{z,min}}$ values ($\approx 230\,^{\mathrm{deg}}/_{\mathrm{s}}$ in Brescianini et al. [106] while only $50\,^{\mathrm{deg}}/_{\mathrm{s}}$ for us). The heuristic in the current implementation may nevertheless be beneficial for other multirotors on which the controller may be deployed.

General Attitude Tracking

Recall that the angle associated with a quaternion $\boldsymbol{q}$ is given by $2\arccos(q_{\mathrm{w}})$. Figure 7.8 plots this angle for the $\boldsymbol{q}_{\mathrm{e,rp}}$ and $\boldsymbol{q}_{\mathrm{e,y}}$ quaternions from Chapter 5.7 for

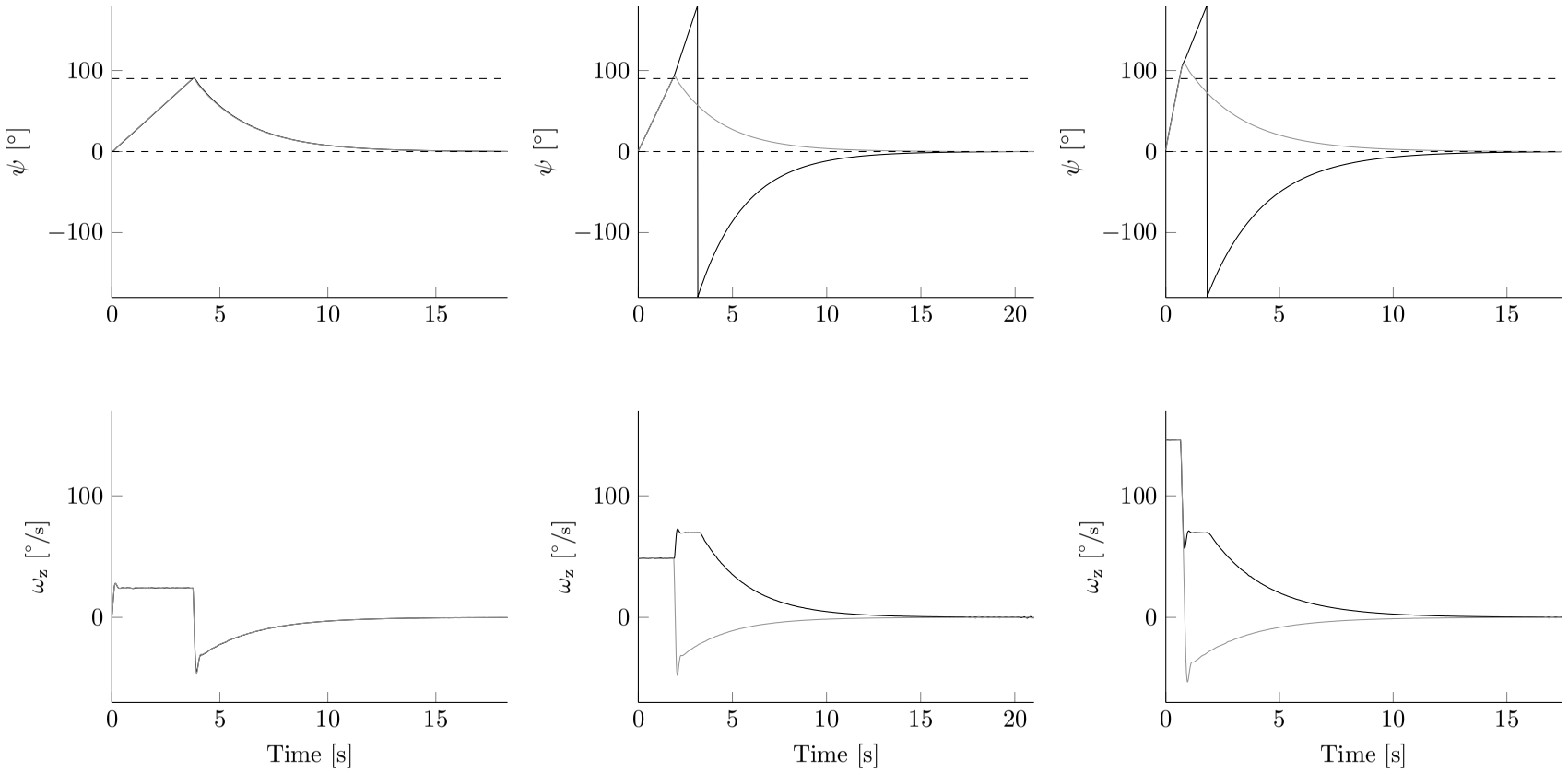
[4] When motor dynamics are removed in the simulation, then the time to reach 63.2 % of the reference becomes 0.24 s which is even closer to the expected 0.25 s.

(a) Response for a step input of 45° in roll.

(b) Response for a step input of 45° in pitch.

(c) Response for a step input of 180° in ψ_{ref}.

Figure 7.6: Step response of $\boldsymbol{q}^{\text{b}}_{\text{w}}$ components. Gray and black lines show the reference and the measured values respectively. *RotorS data*.



(a) $\omega_z = 25$ °/s. (b) $\omega_z = 50$ °/s. (c) $\omega_z = 150$ °/s.

Figure 7.7: Yaw heuristic test. The quadrotor is set to rotate at a given initial value of ω_z. Once the yaw angle reaches 90° with a precision of 1°, the quadrotor is commanded to return to $\psi = 0°$. *RotorS data.*

real and for simulated data. As expected, the results show slightly better tracking performance for the simulation where the quadrotor model is perfectly known and ground truth state is used. However, real tracking performance is very similar and, in both cases, the tracking error is less than 15 °.

Inverted Attitude Recovery

The attitude control law (5.58) is globally asymptotically stable. As shown in Figure 7.9, this is verified in simulation where the quadrotor recovers from an initial upside-down orientation with the body rate $\boldsymbol{\omega} \approx (0, 0, -600)\,^{\circ}/\mathrm{s}$.

7.4.3 Translation Control

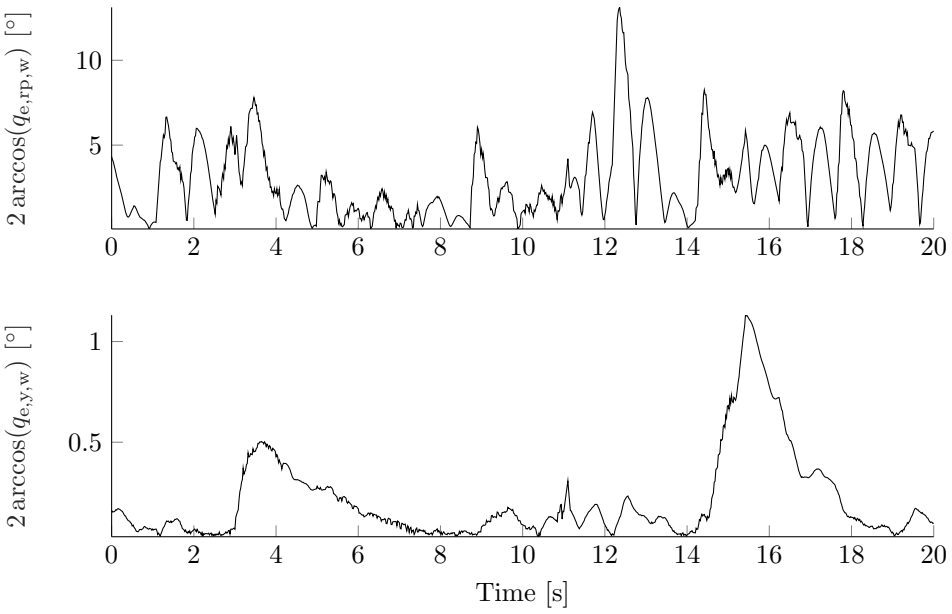
This section evaluates the performance and robustness of the position and velocity control modes developed in Section 5.8. Crucial insight is obtained into the controllers' robustness to measurement time delay.

Position Controller Robustness

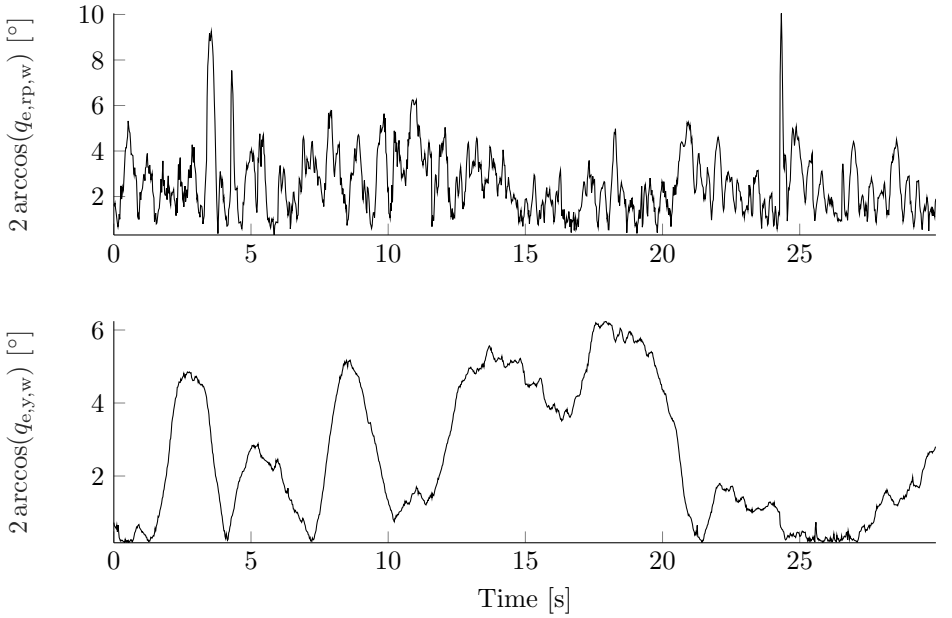
Position controller robustness is analyzed with respect to the following uncertainties:

- Motor time constant $\tau_\mathrm{m} \in [0.005, 0.1]\,\mathrm{s}$ (nominally 0.038 s according to Table 5.1);
- Inner loops' combined time constant $\tau_\mathrm{inner} \in [0.1, 0.4]$ (nominally $\tau_\mathrm{inner} = \tau_\mathrm{att} = 0.25\,\mathrm{s}$ according to Table 5.3);
- A non-zero *feedback delay* θ_feedback (e.g. due to delays in the sensors, in data communication and in the state estimator itself).

The motivation for analyzing these uncertainties in particular comes from the fact that they influence the closed-loop phase. Increasing τ_m, τ_inner or θ_feedback reduces the phase margin and thereby compromises the closed-loop stability. In particular, the lack of oscillatory position response in the RotorS simulation (where $\theta_\mathrm{feedback} \approx 0$) and its intermittent presence in real test flights (where $\theta_\mathrm{feedback} > 0$ and potentially is too large) motivates us to determine the limiting factors of the position controller's stability.
Figure 7.10a shows the loop structure used for robustness analysis. Position control along $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ is assumed to be identical, therefore we shall in this section refer to *xy-* and *z-position controller* robustness. Note that analyzing the 2 DOF control architecture is not necessary because the reference pre-filter is external to the feedback loop thus has no influence on system response to load and output disturbances, measurement noise, etc. A simplifying assumption is furthermore made that the effect of τ_m is decoupled from that of τ_inner. This is clearly false for large τ_m which compromise time scale separation between the body rate controller and the motor dynamics.
Algorithm 41 describes how robustness is analyzed. Let τ_actuator represent the *actuator time constant* (i.e. τ_inner for xy-position control and τ_m for z-position control). In essence, we sample a 1 dimensional grid of τ_actuator in the ranges cited above and, for each sample, compute the destabilizing $\theta_\mathrm{feedback,max}$ as that which nullifies the phase margin (knowing that a θ_feedback delay adds $-\theta_\mathrm{feedback}\omega_\mathrm{c}$ rad of phase at the crossover frequency $\omega_\mathrm{c}\,{}^{\mathrm{rad}}/\mathrm{s}$).
The results of Algorithm 41 are plotted in Figure 7.11, which also visualizes the changing phase margin (φ_PM in Algorithm 41) as a function of τ_actuator and θ_feedback. Including the inner loop dynamics $1/(\tau_\mathrm{actuator}s + 1)$ in the control loop decreases the phase margin, even without feedback delay, as shown in Figure 7.12. This is

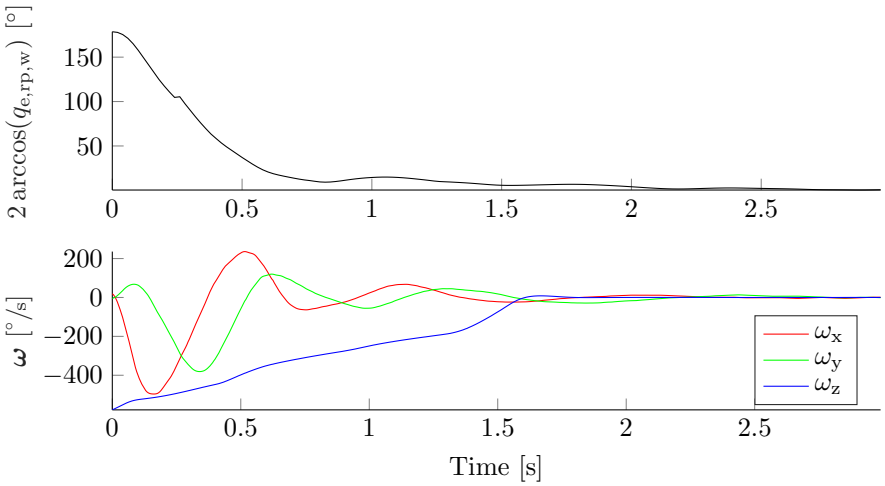


(a) *RotorS data.*

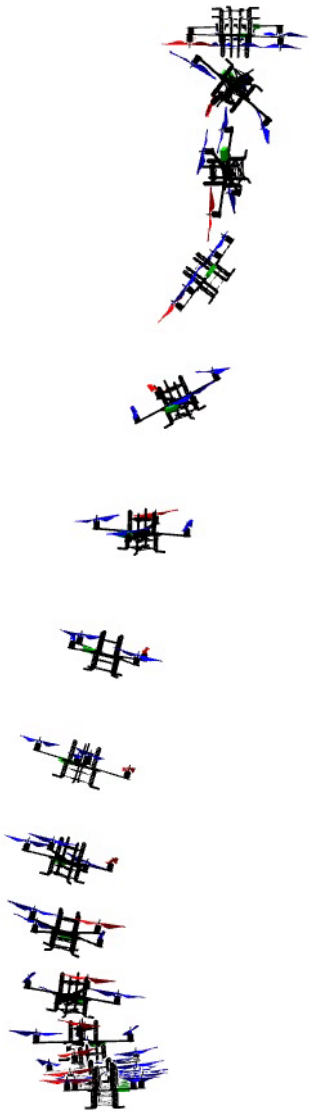


(b) *Outdoor data.*

Figure 7.8: Attitude tracking accuracy, shown as the angle of the error quaternions $\boldsymbol{q}_{\mathrm{e,rp}}$ and $\boldsymbol{q}_{\mathrm{e,y}}$. Due to yaw error discounting in Section 5.7, the yaw error dynamics are noticeably slower.

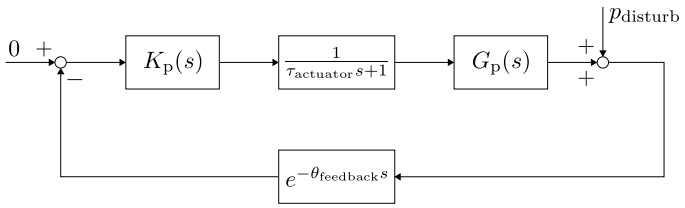


(a) Roll-pitch error quaternion angle and body rates during the recovery maneuver.

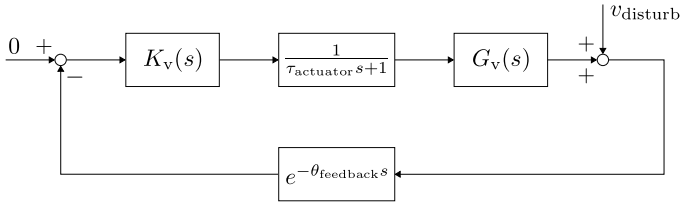


(b) Recovery action shot.

Figure 7.9: Attitude recovery from an initial upside-down orientation with the body rate $\boldsymbol{\omega} \approx (0, 0, -600)\,^\circ/\mathrm{s}$. The quadrotor drops 4.7 m in the process. Translation control is in velocity mode. *RotorS data.*



(a) Control loop for position controller (Section 5.8.2) robustness analysis. p_disturb is an output disturbance.

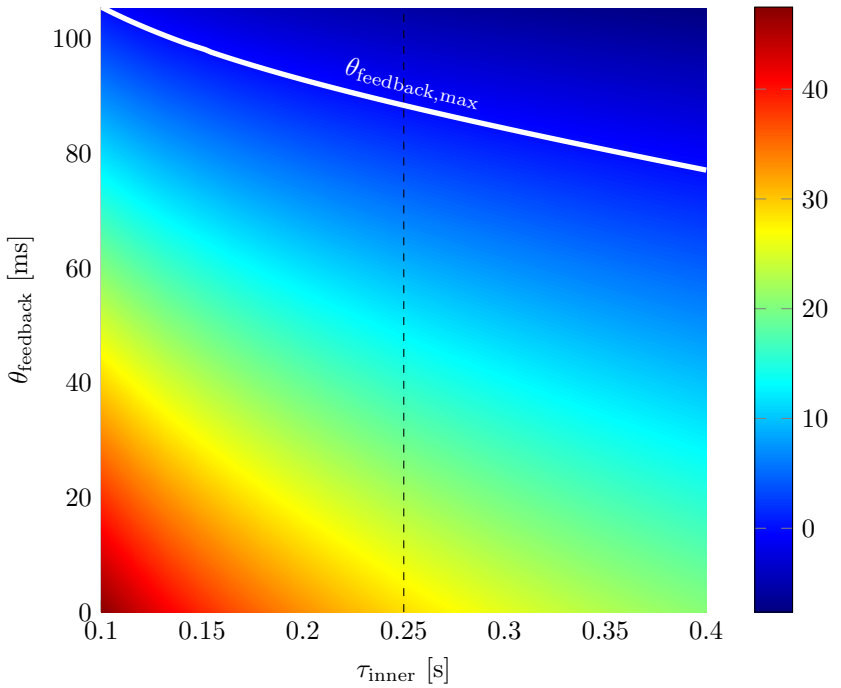


(b) Control loop for velocity controller (Section 5.8.3) robustness analysis. v_disturb is an output disturbance.

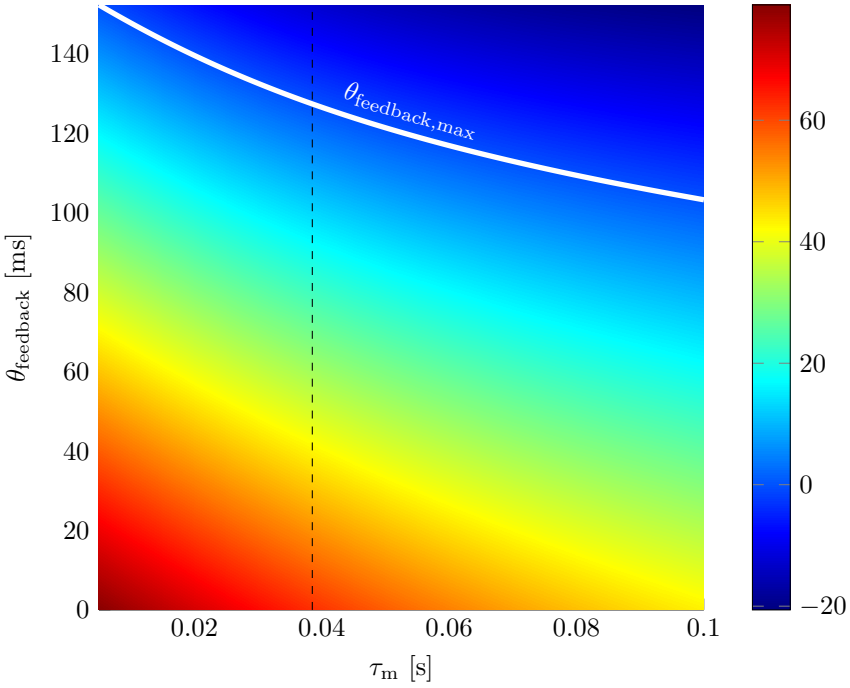
Figure 7.10: Control loop structures used for translation control robustness analysis.

Algorithm 41 Computation of destabilizing feedback delay θ_feedback as a function of the actuator time constant τ_actuator.

Let $\tau_\mathrm{actuator} \in [\tau_\mathrm{min}, \tau_\mathrm{max}]$ ▷ Actuator time constant uncertainty
$N \leftarrow 100$ ▷ Number of τ_actuator samples
for $i \leftarrow 1$ to N **do**
  $\tau_i \leftarrow (\tau_\mathrm{max} - \tau_\mathrm{min})\frac{i-1}{N-1} + \tau_\mathrm{min}$
  $L(s) \leftarrow \frac{K_\mathrm{p}(s)G_\mathrm{p}(s)}{\tau_i s+1}$ ▷ Open-loop transfer function using (5.68), (5.67)
  Compute φ_PM rad phase margin and ω_c $^\mathrm{rad}/_\mathrm{s}$ crossover frequency of $L(s)$ ▷ MATLAB `margin()`
  $\theta_\mathrm{feedback,max}[i] \leftarrow \frac{\varphi_\mathrm{PM}}{\omega_\mathrm{c}}$
end for

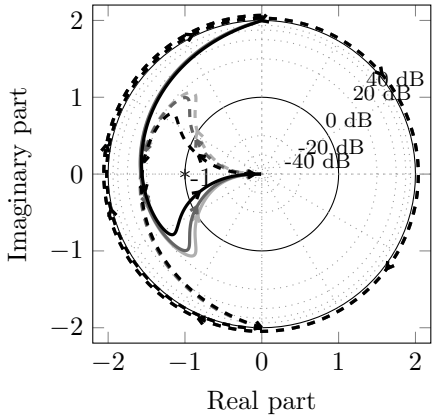


(a) Phase margin [°] color map and $\theta_{\text{feedback,max}}$ for xy-position control.

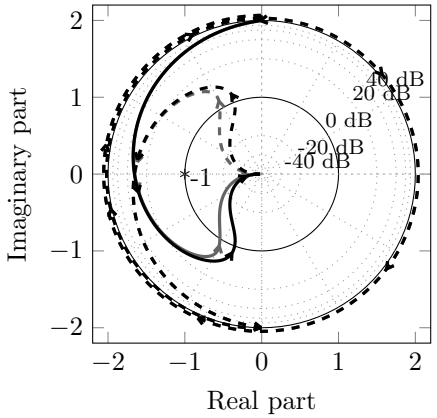


(b) Phase margin [°] color map and $\theta_{\text{feedback,max}}$ for z-position control.

Figure 7.11: Position controller phase margin and destabilizing feedback delay as a function of an uncertain τ_{actuator}. Black dashed line shows the nominal τ_{actuator} value.



(a) Nyquist plots of xy-position controller with $\tau_{\text{inner}} = 0.2, 0.25$ and $0.4\,\text{s}$. Compared to Figure 5.19, the phase margin is reduced.



(b) Nyquist plots of z-position controller with $\tau_{\text{m}} = 24$ and $53\,\text{ms}$. Compared to Figure 5.21, the phase margin is reduced.

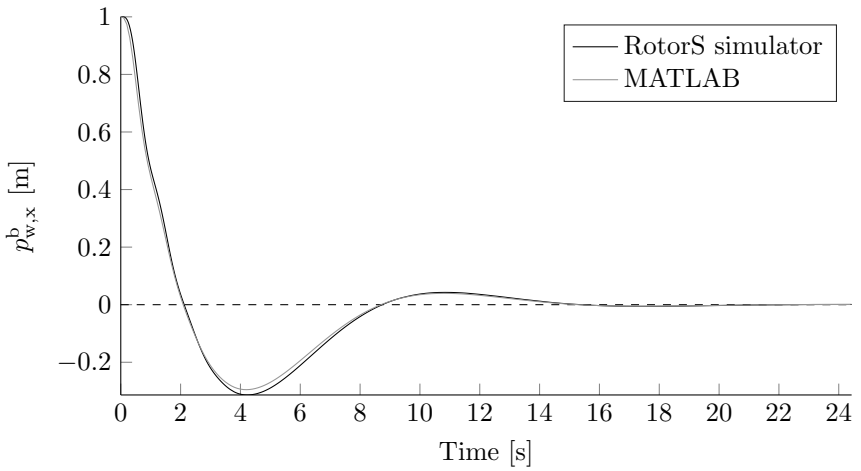
Figure 7.12: Comparison of how the inclusion of "actuator dynamics" (whether these are of the attitude controller or the motors) reduces the phase margin by introducing negative phase and making the Nyquist plot pass closer to the -1 point. This effects grows with decreasing time scale separation.

expected and is the direct consequence of weak time scale separation between the cascaded loops, causing the negative phase in the inner loops to interfere with that of the outer (position control) loop. Increasing θ_{feedback} further reduces the phase margin, eventually driving the system unstable (negative phase margin).

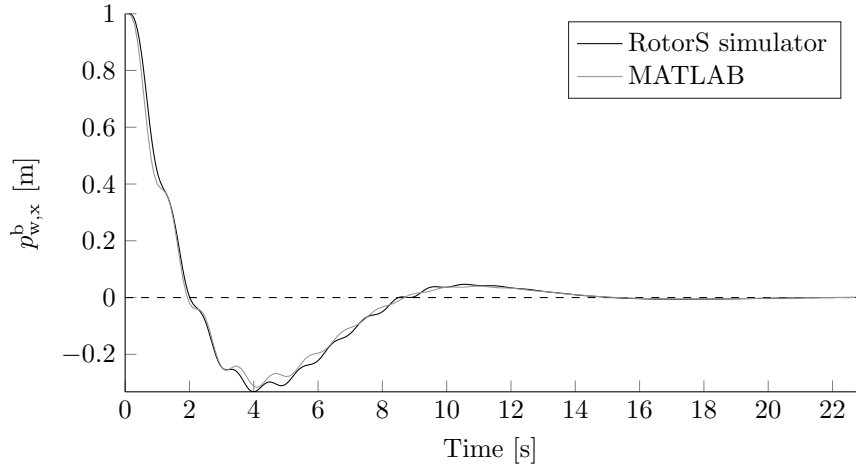
Analyzing Figure 7.11, one sees that xy-position control destabilizes first as it relies on the lower-bandwidth attitude controller for thrust vectoring (as opposed to z-position control, which relies on the faster motor dynamics to simply increase/decrease thrust). Taking the worst-case $\tau_{\text{inner}} = 0.4\,\text{s}$, xy-position control tolerates up to $\approx 80\,\text{ms}$ of feedback delay. However, to retain a reasonable phase margin of $\approx 30\,^\circ$ against other unmodeled system uncertainties, at the nominal $\tau_{\text{inner}} = 0.25\,\text{s}$ (according to Table 5.3) virtually no delay is tolerated. A non-zero θ_{feedback} is expected to induce an increasingly oscillatory position response.

These results are validated by comparing the response of the simplified model in Figure 7.10a to the high-fidelity RotorS simulation. As shown in Figure 7.13, the full controller implementation exhibits a very similar response to the simplified model used for robustness analysis, validating that the simple model correctly captures the dominant control loop aspects. However, the RotorS simulation's disturbance response matches that of the simplified model for delay values of $\approx 15\,\text{ms}$ less which suggests that the real implementation has by default more negative phase than our model. This is expected, as the one-parameter actuator model $1/(\tau_{\text{actuator}} s + 1)$ is naturally too simple to model the complex inner loop interactions with weak time scale separation.

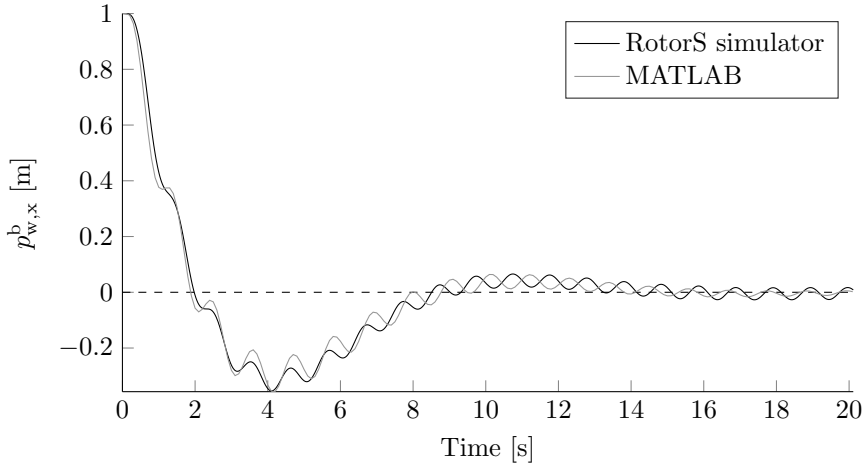
In conclusion, the position controller is *robustly stable* for up to $\approx 80\,\text{ms}$ of feedback delay. However, *robust performance* quickly begins to suffer for non-zero feedback delay as the phase margin is quick to erode to $< 30\,^\circ$ where an oscillatory position response will be observed. In real flight tests, it is hypothesized that θ_{feedback} was intermittently significantly non-zero, leading to a variably oscillatory position response. A remedy is discussed in Chapter 8.



(a) $\theta_{\mathrm{feedback}} = 0\,\mathrm{s}$.



(b) $\theta_{\mathrm{feedback}} = 40\,\mathrm{ms}$ in RotorS and 65 ms for the simplified model.



(c) $\theta_{\mathrm{feedback}} = 55\,\mathrm{ms}$ in RotorS and 80 ms in MATLAB.

Figure 7.13: Comparison of $p^{\mathrm{b}}_{\mathrm{w,x}}$ response to a 1 m step in p_{disturb} in Figure 7.10a for various $\theta_{\mathrm{feedback}}$ values. The simplified model is a MATLAB implementation of Figure 7.10a while RotorS uses the actual full control system implementation.

Tracked Variable	Outdoor RMSE [cm]	RotorS RMSE [cm]
Raw $p_{\mathrm{ref,x}}$	22.6	21.4
Pre-filtered $p_{\mathrm{ref,x}}$	5.6	1.3
Raw $p_{\mathrm{ref,y}}$	23.3	19.7
Pre-filtered $p_{\mathrm{ref,y}}$	7.6	1.3
Raw $p_{\mathrm{ref,z}}$	1.84	0.6
Pre-filtered $p_{\mathrm{ref,z}}$	1.7	0.1

Table 7.2: Position tracking RMSE for an outdoor and a RotorS test flight using the same reference (corresponding to Figure 7.15). In the first column, raw and pre-filtered values correspond respectively to p_{ref} and to the output of $F_{\mathrm{p}}(s)$ in Figure 5.17b.

Position Controller Tracking Performance

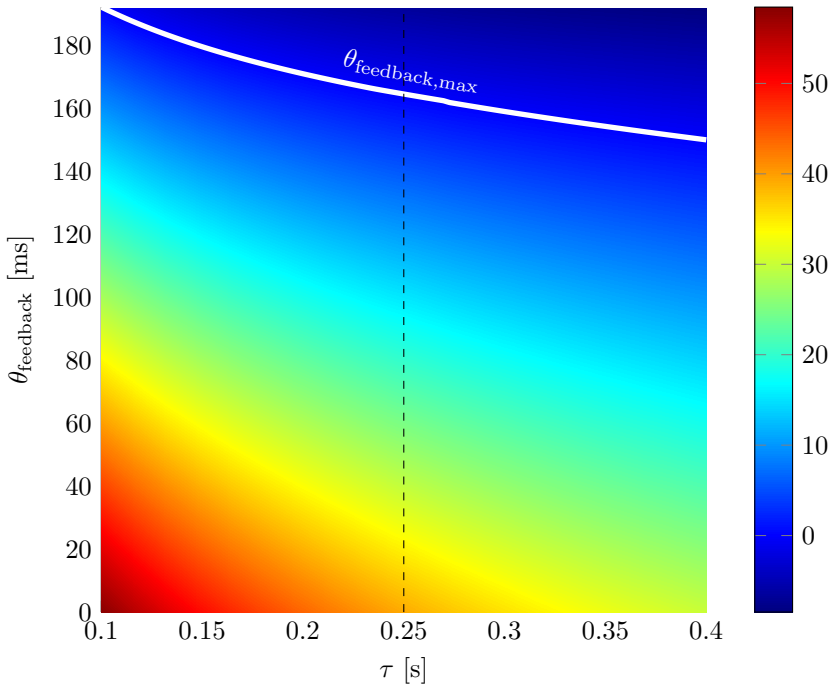
As mentioned in Section 5.8, the 2 DOF control architecture is used for position tracking. This follows from experience of this architecture causing a more predictable performance with almost no overshoot of the reference. Outdoor (with RTK GPS) position tracking performance is compared to RotorS simulation in Figure 7.15 by feeding RotorS the reference signal recorded during an outdoor flight. Figure 7.16 displays the corresponding tracking error and Table 7.2 computes the tracking error RMSE. As expected, the RotorS simulation has more accurate tracking (lower RMSE) because it uses ground-truth state for control, the quadrotor dynamic model parameters are known exactly and no disturbances (e.g. wind) act on the quadrotor. As can be seen from Figure 7.15, the pre-filtered reference lags behind the raw reference due to the negative phase introduced by $F_{\mathrm{p}}(s)$. This is the price to be paid with a pre-filter: while overshoots are removed and any arbitrary reference may be passed, the pre-filter induces a tracking lag.

Velocity Controller Robustness

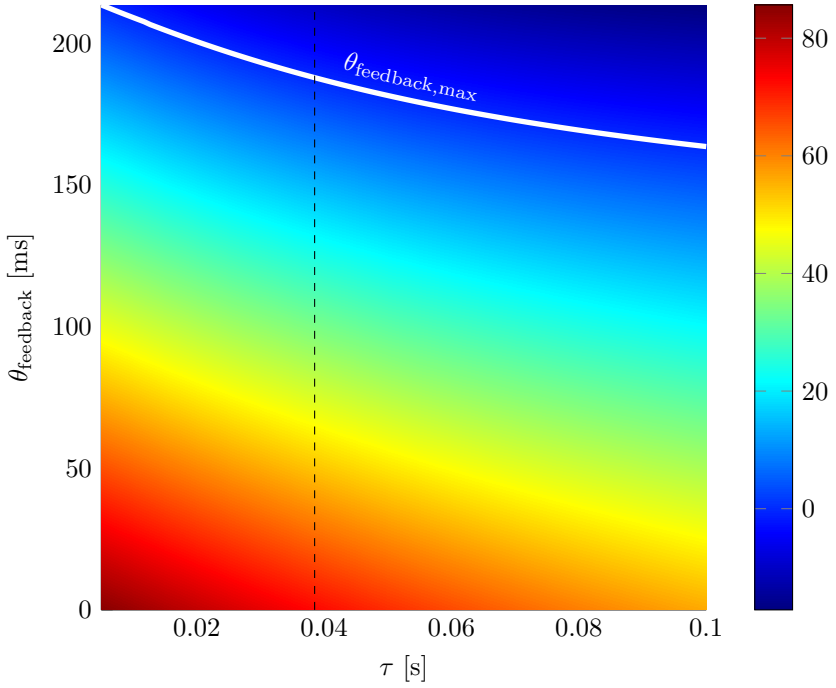
Velocity controller robustness is analyzed identically to the position controller. The control loop used for robustness analysis is shown in Figure 7.10b. As the only change, $K_{\mathrm{p}}(s)$ and $G_{\mathrm{p}}(s)$ in Algorithm 41 are replaced by $K_{\mathrm{v}}(s)$ (5.78) and $G_{\mathrm{v}}(s)$ (5.77) respectively. The goal of the robustness analysis remains the same: to determine the destabilizing feedback delay as a function of actuator time constant uncertainty.
Figure 7.14 visualizes the analysis results. As for position control, xy-velocity control destabilizes first as it relies on the lower-bandwidth attitude controller for thrust vectoring. However, because the plant $G_{\mathrm{v}}(s)$ is a simple integrator (as opposed to $G_{\mathrm{p}}(s)$, which is a double integrator) in has $90\,^\circ$ more positive phase than $G_{\mathrm{p}}(s)$. The velocity control loop therefore naturally has a higher phase margin and tolerates a larger $\theta_{\mathrm{feedback,max}} \approx 150\,\mathrm{ms}$. Again taking a phase margin of $30\,^\circ$ as the threshold for observing an oscillatory response, $\approx 40\,\mathrm{ms}$ of feedback delay are tolerated before the onset of tracking response oscillations. In real test flights, oscillatory velocity tracking was observed very rarely, yielding a hypothesis that **the real system has $< 40\,\mathrm{ms}$ of delay – enough to observe oscillations in position tracking, but not enough to observe oscillations in velocity tracking (most of the time)**. Chapter 8 suggests possible remedies.

Velocity Controller Tracking Performance

As mentioned in Section 5.8, the 2 DOF control architecture is used for velocity tracking. Like for position control, this follows from experience of this architecture



(a) Phase margin [°] color map and $\theta_{\text{feedback,max}}$ for xy-velocity control.



(b) Phase margin [°] color map and $\theta_{\text{feedback,max}}$ for z-velocity control.

Figure 7.14: Velocity controller phase margin and destabilizing feedback delay as a function of an uncertain τ_{actuator}. Black dashed line shows the nominal τ_{actuator} value.

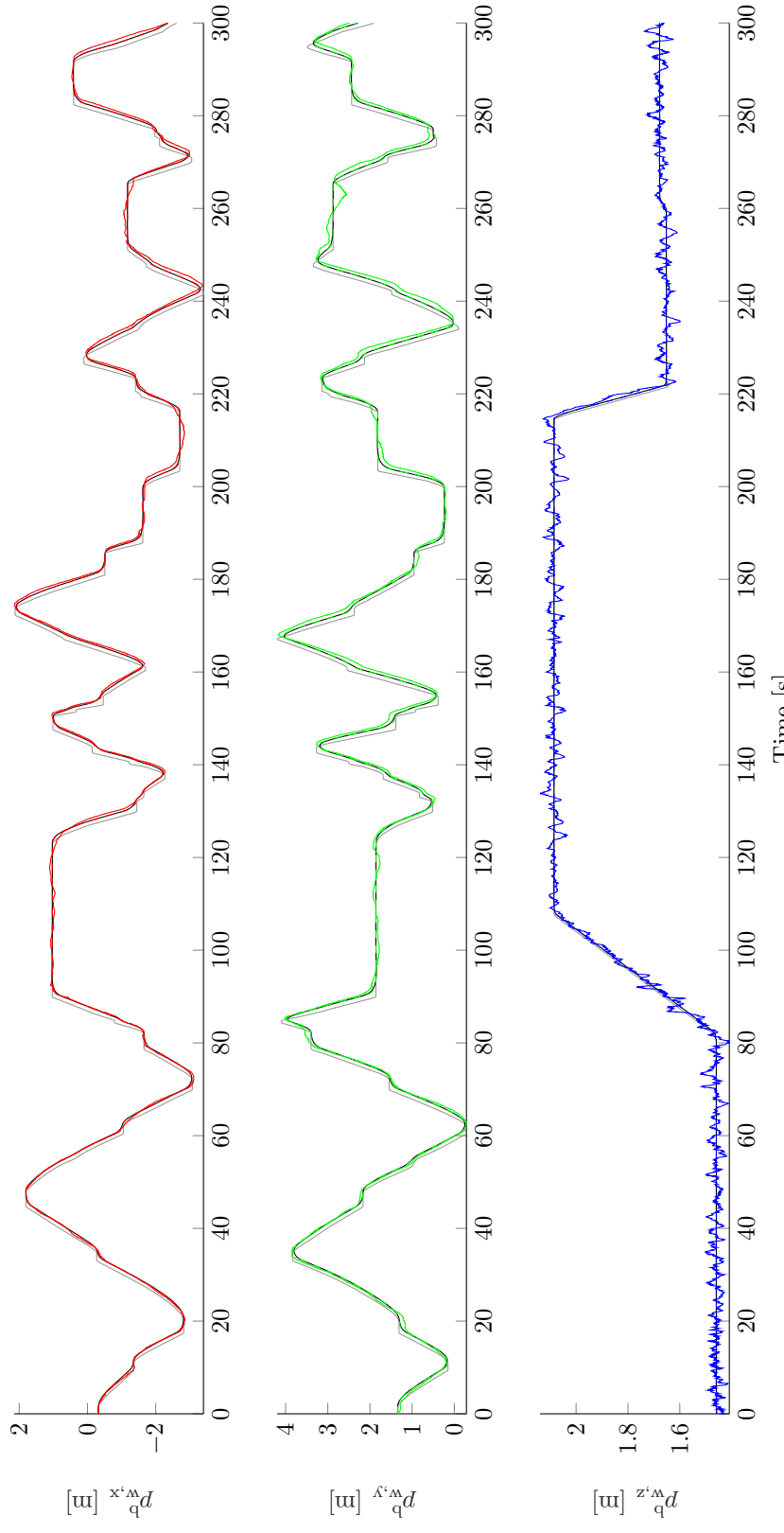


Figure 7.15: Position tracking in RotorS and in reality. Gray is reference, black is pre-filtered reference, solid color is outdoor estimated position and dashed color is RotorS ground-truth position.

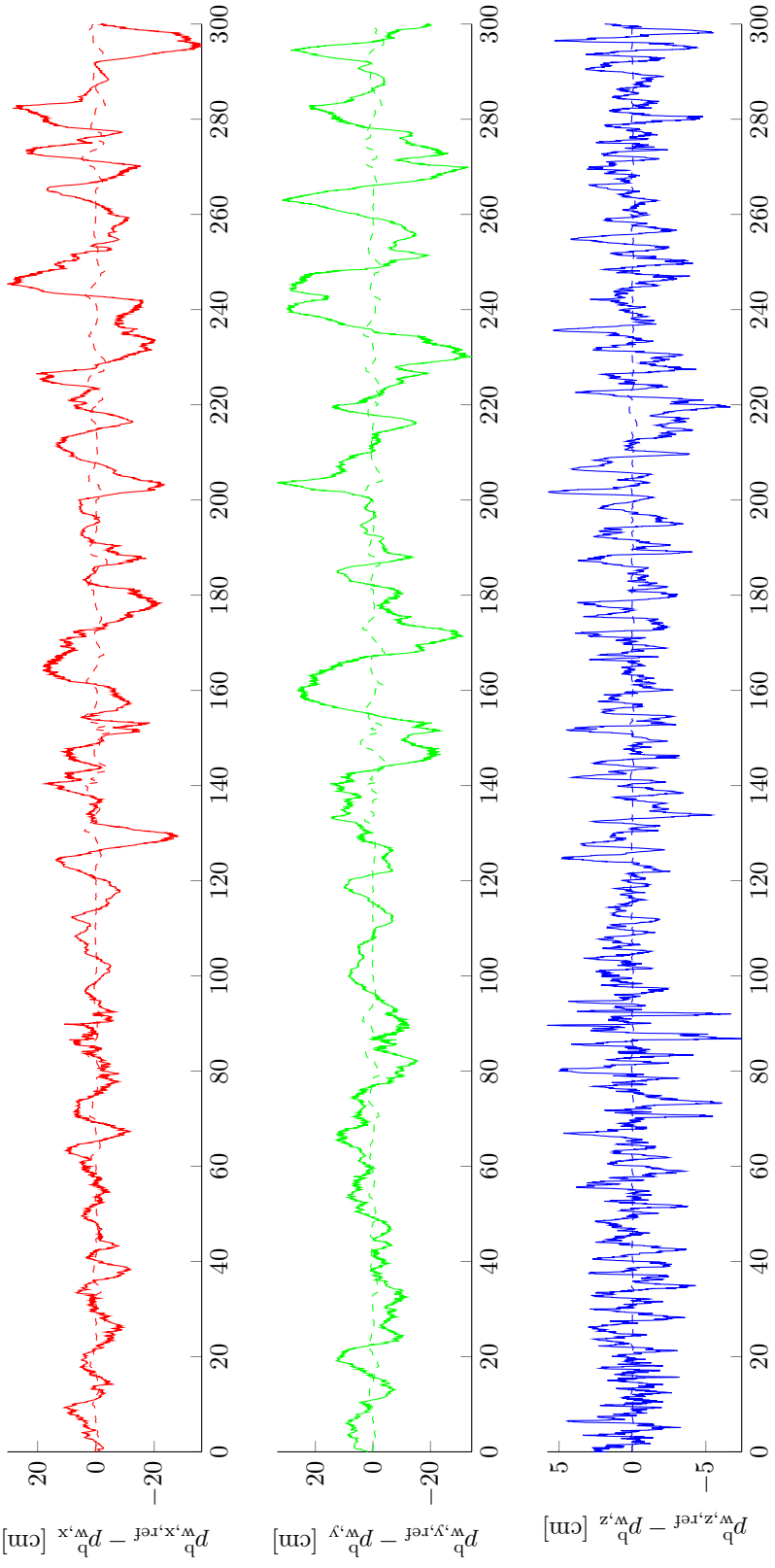


Figure 7.16: Position tracking error in RotorS and in reality, corresponding to Figure 7.15. Solid color is outdoor estimated tracking error and dashed color is RotorS ground-truth tracking error.

Tracked Variable	Outdoor RMSE [cm/s]	RotorS RMSE [cm/s]
Raw $v_{\mathrm{ref,x}}$	8.9	6.9
Pre-filtered $v_{\mathrm{ref,x}}$	5.9	2.1
Raw $v_{\mathrm{ref,y}}$	10.4	9.4
Pre-filtered $v_{\mathrm{ref,y}}$	4.8	2.8
Raw $v_{\mathrm{ref,z}}$	3	1.4
Pre-filtered $v_{\mathrm{ref,z}}$	2.8	1

Table 7.3: Velocity tracking RMSE for an outdoor and a RotorS test flight using the same reference (corresponding to Figure 7.17). In the first column, raw and pre-filtered values correspond respectively to v_{ref} and to the output of $F_{\mathrm{v}}(s)$ in Figure 5.20b.

causing a more predictable performance with almost no overshoot of the reference. Outdoor (with RTK GPS) velocity tracking performance is compared to RotorS simulation in Figure 7.17 by feeding RotorS the reference signal recorded during an outdoor flight. Figure 7.18 displays the corresponding tracking error and Table 7.3 computes the tracking error RMSE. As expected, the RotorS simulation has more accurate tracking for the same reason as for position control. Like for position, the price of using a pre-filter is again a tracking lag. However, this tracking lag is less pronounced for velocity than for position because $F_{\mathrm{v}}(s)$ has less negative phase than the $F_{\mathrm{p}}(s)$ (for xy-control, $\approx -30\,^\circ$ for $F_{\mathrm{v}}(s)$ versus $\approx -45\,^\circ$ for $F_{\mathrm{p}}(s)$ given Tables 5.4 and 5.5).

7.4.4 Emergency Landing Controller

Three test cases are presented in Figure 7.19 for the open-loop emergency landing controller developed in Section 5.9. The figures show three seconds prior to the emergency landing (i.e. the length of the velocity buffer), the open-loop emergency landing maneuver and part of the post-landing ground contact phase. In these tests, the emergency landing controller is triggered manually via a switch.

In Figure 7.19a, the quadrotor has no horizontal and some vertical motion. When the emergency landing controller is triggered, it successfully brings the quadrotor to $v^{\mathrm{b}}_{\mathrm{w,z}} \approx v_{\mathrm{em.z}} = -2\,\mathrm{m/s}$ at an acceleration of $\boldsymbol{a}^{\mathrm{b}}_{\mathrm{w}} \approx (0, 0, -2)\,\mathrm{m/s^2}$. When the open-loop acceleration phase is finished at $t \approx 4\,\mathrm{s}$, the quadrotor continues to travel down at a constant $v_{\mathrm{em.z}}$ vertical speed. This is due to perfect gravity cancellation due to exact mass knowledge of the quadrotor in the simulator. In reality, $v^{\mathrm{b}}_{\mathrm{w,z}}$ after the open-loop acceleration will not be exactly constant due to various disturbances, due to inexact thrust reproduction by the motors (due to thrust map limitations and aerodynamic effects) and perhaps due to imperfect mass knowledge.

In Figure 7.19b, the quadrotor has a small horizontal velocity component $v^{\mathrm{b}}_{\mathrm{w,x}} \approx 1\,\mathrm{m/s}$. When emergency landing is triggered, $v^{\mathrm{b}}_{\mathrm{w,z}}$ is reduced as before to $\approx v_{\mathrm{em.z}}$ while $v^{\mathrm{b}}_{\mathrm{w,x}}$ remains because the quadrotor is simply controlled to an upright orientation. Thus, the quadrotor travels forward along $\boldsymbol{e}^{\mathrm{w}}_{\mathrm{x}}$ while it descends.

In Figure 7.19c, the quadrotor is flown generically with a full 3 DOF velocity (all $\boldsymbol{v}^{\mathrm{b}}_{\mathrm{w}}$ components non-zero). Again $v^{\mathrm{b}}_{\mathrm{w,z}}$ is reduced to $\approx v_{\mathrm{em,z}}$ (now starting from an initial upward velocity) while the $v^{\mathrm{b}}_{\mathrm{w,x}}$ and $v^{\mathrm{b}}_{\mathrm{w,y}}$ components are quasi-maintained.

In summary, the emergency landing controller is capable of bringing the quadrotor to a level attitude and reducing the vertical velocity to a negative "landing velocity" value starting from a variety of initial conditions. Not being the focus of this thesis, the current emergency landing controller is a simple fallback that can help a human pilot take over without the quadrotor rapidly diverging due to a failed state estimate.

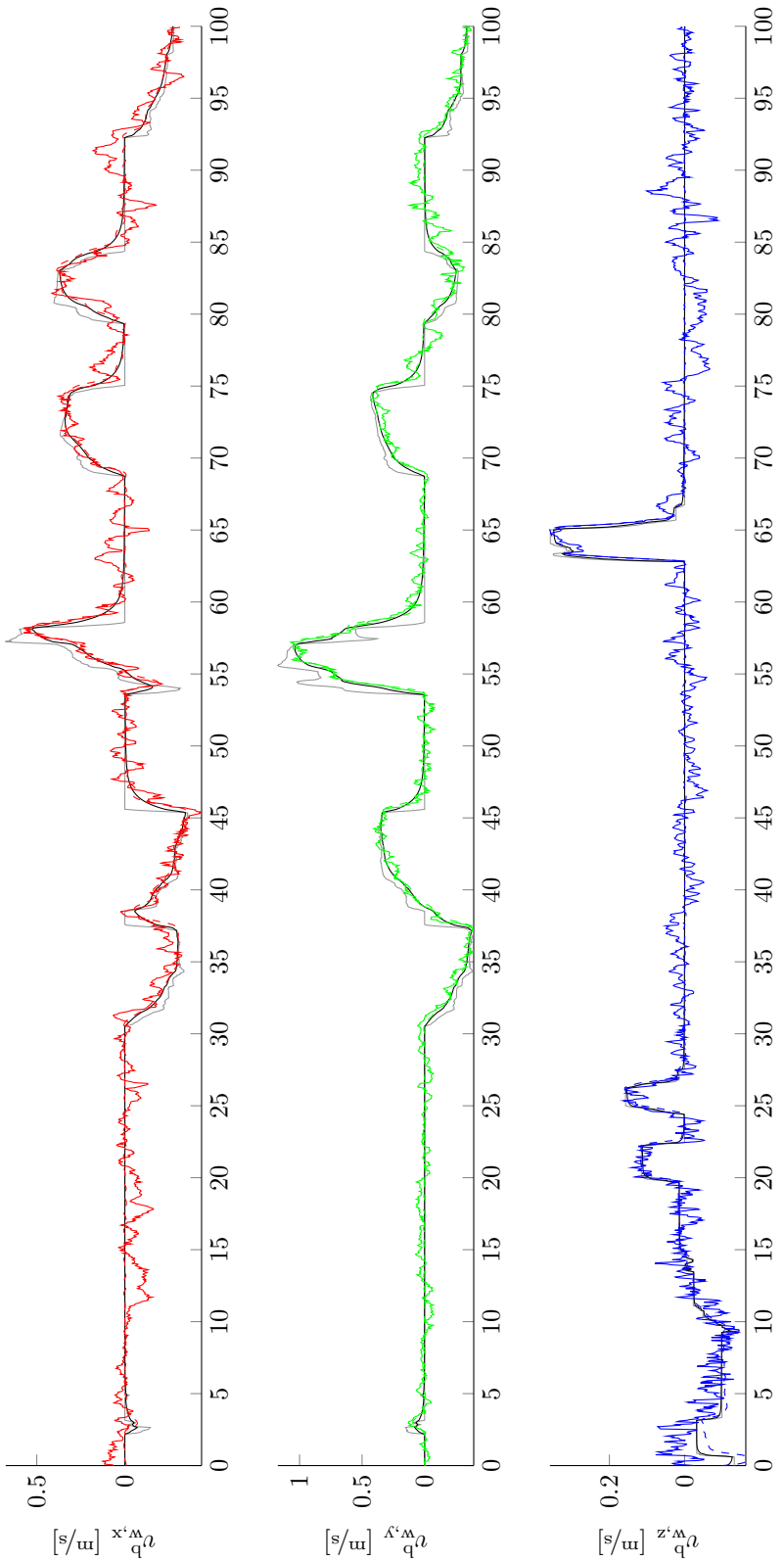


Figure 7.17: Velocity tracking in RotorS and in reality. Gray is reference, black is pre-filtered reference, solid color is outdoor estimated velocity and dashed color is RotorS ground-truth velocity.

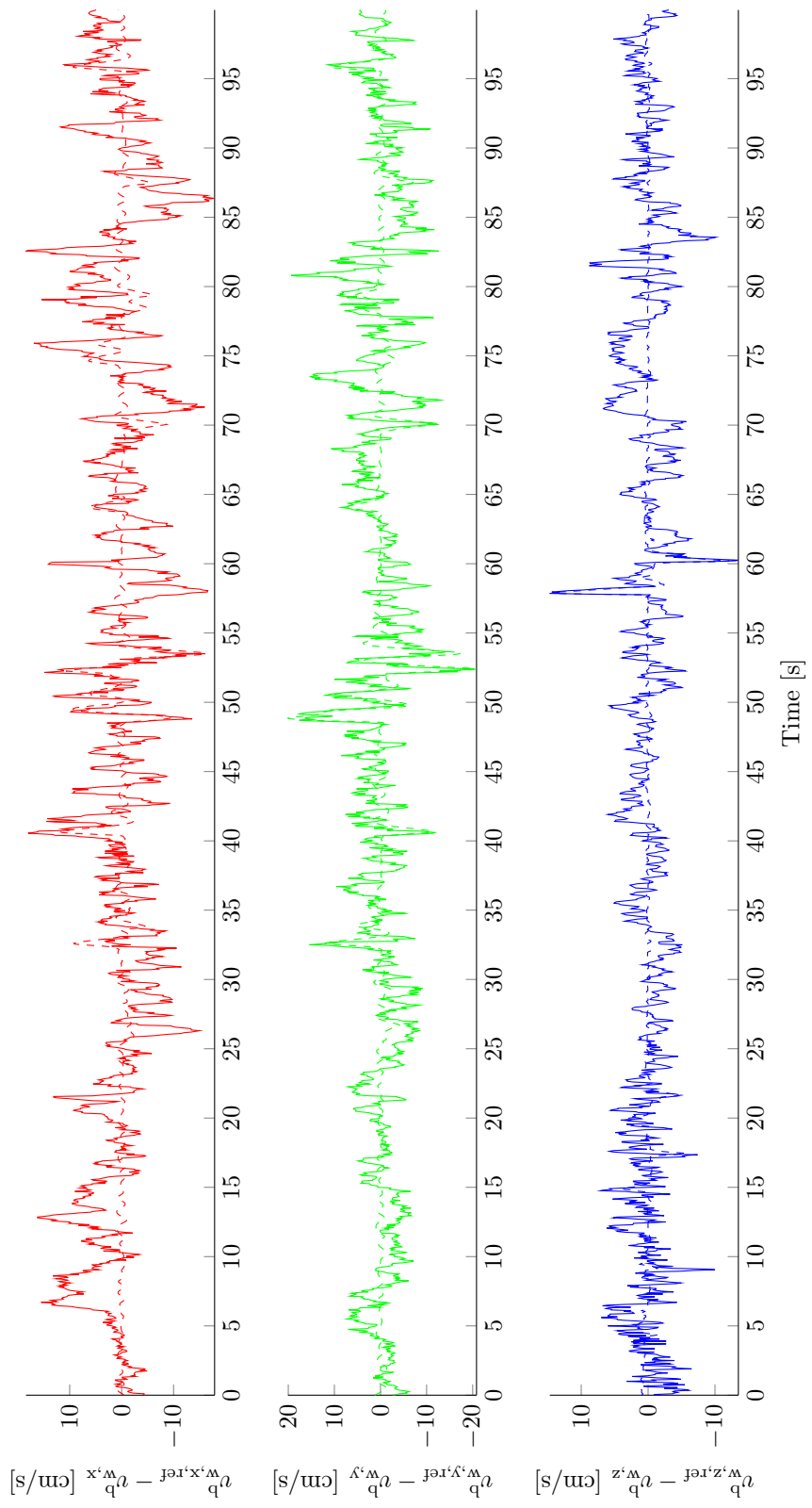
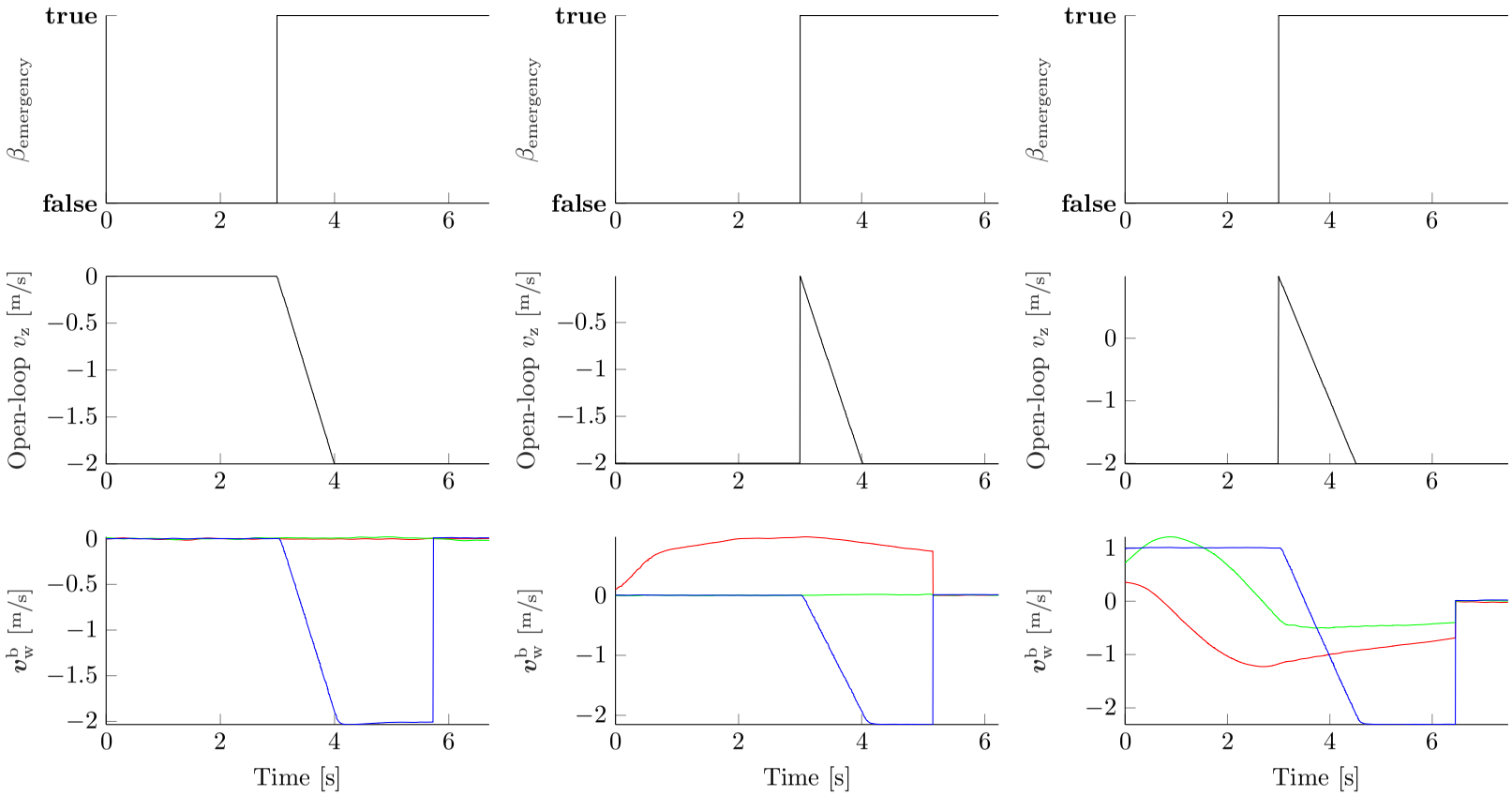


Figure 7.18: Velocity tracking error in RotorS and in reality, corresponding to Figure 7.17. Solid color is outdoor estimated tracking error and dashed color is RotorS ground-truth tracking error.



(a) Prior vertical motion.

(b) Prior unidirectional horizontal motion.

(c) Prior full 3 DOF motion.

Figure 7.19: Emergency landing controller test cases. Middle plots show v_z in Algorithm 33. Bottom plots' coloring follows the standard axis convention (Section 1.4.1 i.e. $v_{w,x}^b$ is red, $v_{w,y}^b$ is green, $v_{w,z}^b$ is blue). *RotorS data.*

7.5 Autonomy Engine

A set of flights is carried out to record the autonomy engine state machines' states. The flights are chosen so as to cover the major modes of operation of the autonomy engine which may occur over a multi-day full-cycle autonomous operation. Figure 7.24 shows the evolution of recorded states in time for the primary state machines introduced in Chapter 6. A narration of each flight is given below.

Flight 1. This flight shows a nominal Figure 6.1 cycle. Because the simulator does not implement a battery model, BATTERYCHARGED is emulated via FORCETAKEOFF manually by the operator at $t = 2\,\text{s}$. The quadrotor executes a data collection mission fully autonomously and lands itself back on the charging pad at $t = 104\,\text{s}$. Note that the landing autopilot spends negligible time in the ALIGN_POSITION state since the simulation does not model GPS drift, thus the return to home point exactly matches (in $\boldsymbol{e}_{\text{x}}^{\text{w}}$ and $\boldsymbol{e}_{\text{y}}^{\text{w}}$) the takeoff location.

Flight 2. This flight shows a case of aborting the mission phase due to a low battery. Again because battery dynamics are not modeled, BATTERYLOW is emulated using FORCELAND manually by the operator at $t = 150\,\text{s}$. Upon entering the RETURN_TO_HOME state, the master state machine commands the mission autopilot to return home (which thus transitions to the RETURNING_TO_HOME state). The landing autopilot subsequently brings the quadrotor to a safe touchdown on the charging pad at $t = 174\,\text{s}$. Note that because the quadrotor keeps its yaw aligned with the charging pad after Flight 1, negligible time is spent by the landing autopilot in the ALIGN_YAW state for the second landing. Figure 7.20 shows video stills of this scenario.

Flight 3. This flight shows a case of the landing bundle not being detected by the downfacing camera upon returning home. As shown in Figure 7.21, this edge case was achieved by displacing the charging pad by $\approx 9\,\text{m}$ from its original location. In reality, this edge case is most likely to occur due to GPS drift (i.e. the quadrotor returning to a position that is in reality far from the actual takeoff location). To make matters more complex, the landing pad was also yawed. The quadrotor returns home at $t = 267\,\text{s}$ and, after a 5 second landing bundle detection timeout, the landing autopilot transitions to the FIND_LANDING_PAD state at $t = 272\,\text{s}$. Flying the spiral grid search trajectory eventually finds the landing pad at $t = 335\,\text{s}$. The quadrotor subsequently aligns itself in position and yaw with respect to the new charging pad pose (states ALIGN_POSITION and ALIGN_YAW) and precisely lands at $t = 389\,\text{s}$. Figure 7.22 shows video stills of this scenario. Note that this flight also demonstrates the guidance spiral grid search trajectory generation (Section 4.1.8) in action.

Flight 4. This flight showcases a mid-mission emergency landing. Because battery dynamics are not modeled, BATTERYCRITICAL is emulated with EMERGENCYLAND manually by the operator at $t = 466\,\text{s}$. The mission autopilot is preempted and the emergency lander (Section 6.5) is awoken. The quadrotor descends vertically in velocity mode while maintaining the $p_{\text{w,x}}^{\text{b}}$ and $p_{\text{w,y}}^{\text{b}}$ that it had when the emergency lander was triggered. Touchdown is detected at $t = 482\,\text{s}$. With the master state machine being in the EMERGENCY_LANDING state, further interaction with the autonomy engine is forbidden and the operator must manually recover the system. Figure 7.23 shows video stills of this scenario.

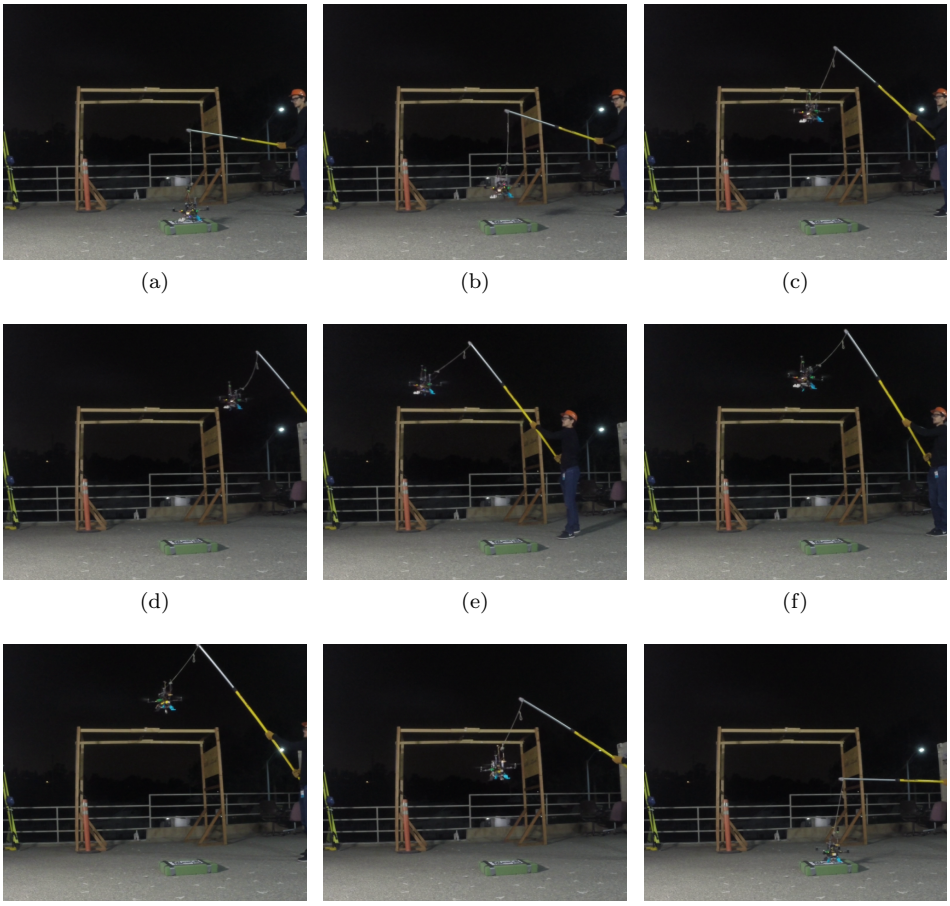


Figure 7.20: Time instances of an outdoor flight (tether required by JPL safety rules), with a smaller landing bundle for convenience. The quadrotor checks motor performance in (a) and accelerates to takeoff velocity in (b). Takeoff is finished in (c) with h_{takeoff} limited by human height. A square mission trajectory is flown in (d) an (e). After the operator triggers ForceLand, the quadrotor returns to home in (f) and aligns in yaw in (g). The quadrotor descends vertically in (h) and touches down in (i).

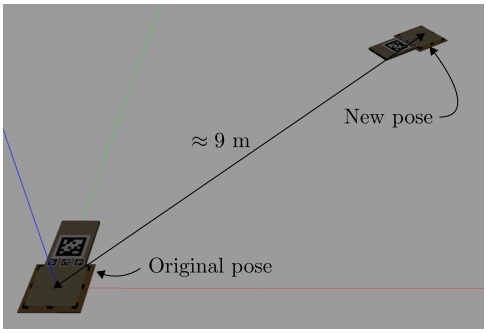
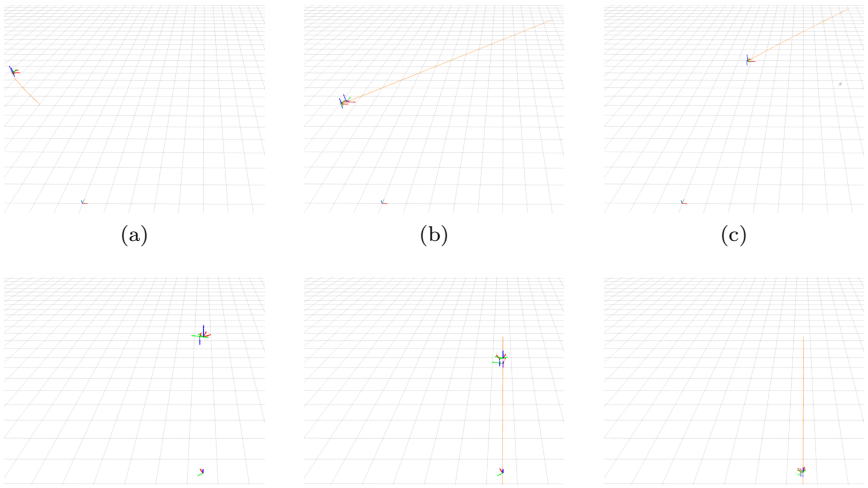


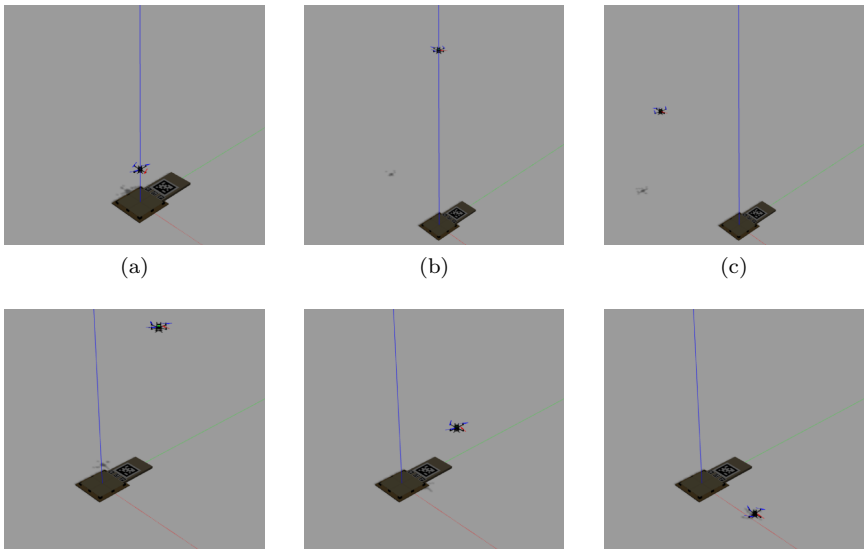
Figure 7.21: Manual horizontal position and yaw change of the charging pad in order to trigger a spiral grid search upon return to home.



(a) (b) (c)

(d) (e) (f)

Figure 7.22: Time instances of a flight with a spiral grid search after returning home, corresponding to the scenario in Figure 7.21. After flying the mission, the quadrotor returns to its original takeoff location, the origin, in (a). With the pad not visible, the first spiral segment is generated in (b) (aligned with the camera x-axis). By luck, the displaced pad is in the same direction and is spotted off in the distance in (c). A position alignment trajectory is generated. Once the trajectory is flown, the quadrotor aligns with the new pad yaw in (d) and descends as usual in (e). Precise landing is achieved in (f). *RotorS data.*



(a) (b) (c)

(d) (e) (f)

Figure 7.23: Time instances of a flight with a mid-mission emergency landing. Stills (a) and (b) show the start and end of the velocity mode takeoff. In (c) the quadrotor flies the mission trajectory. The quadrotor stops and hovers in (d) when the emergency lander is triggered. In (e) a vertical velocity mode descent is performed and in (f) the quadrotor touches down and turns off its motors. *RotorS data.*

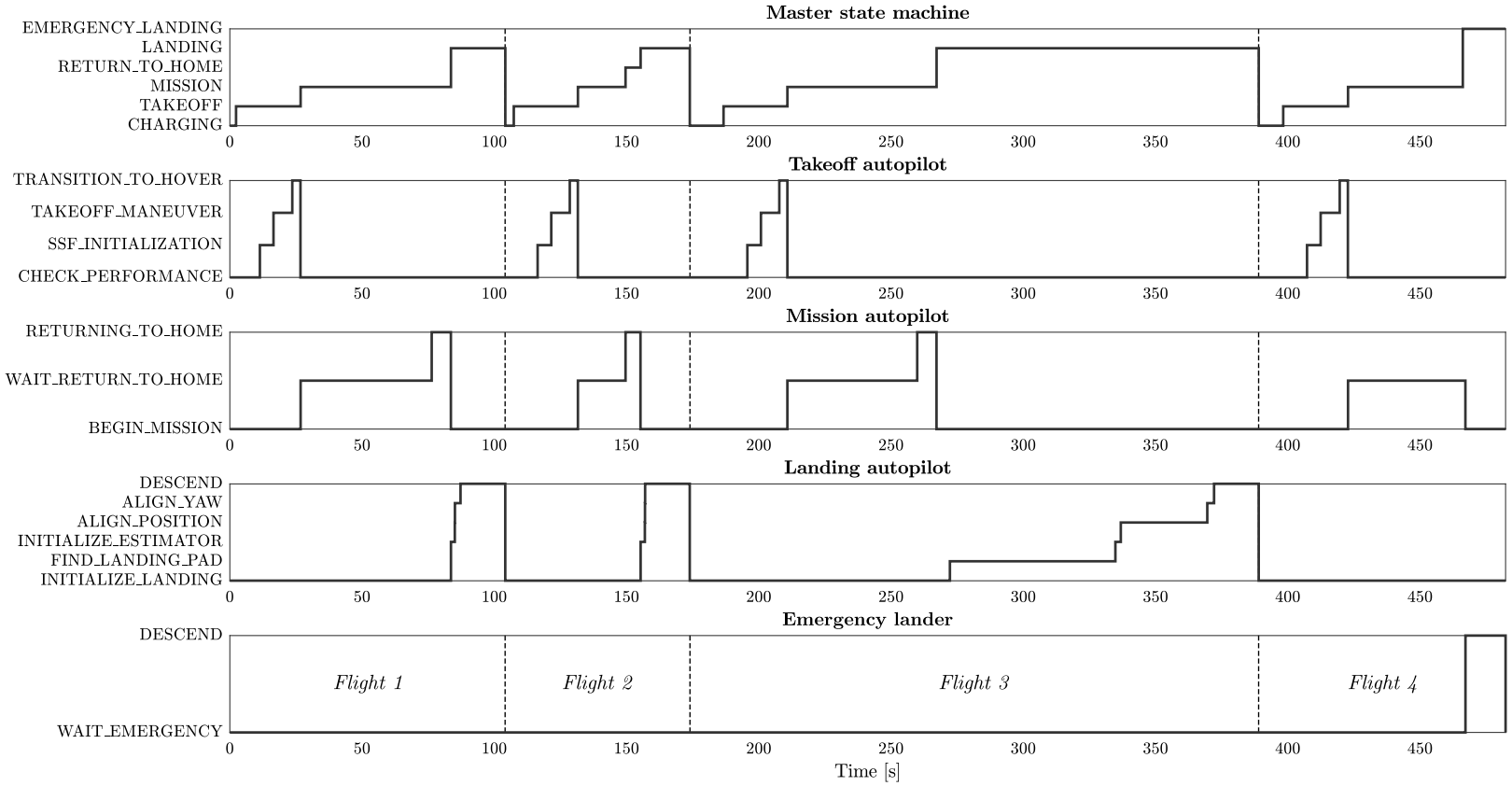


Figure 7.24: Autonomy engine states over four flights that are representative of the autonomy engine's dominant modes of operation. Dashed lines delimit individual flights (including on-pad time) and the flights are numbered in the bottom subplot. Flight (1) shows a nominal cycle, (2) shows an aborted mission due to a low battery, (3) shows a case where the landing pad is invisible upon returning home and (4) shows an emergency landing due to a critically low battery. *RotorS data.*

7.6 Precision Landing Tests

Precision landing is one of the most complex and mission-critical phases of full-cycle autonomy. Quadrotor structural integrity relies on a soft touchdown while the ability to sustain full-cycle autonomous operation depends on a repeatable, robust precision landing on the charging surface.
Precision landing, like no other flight phase, requires the correct operation of the entire GNC system and autonomy engine. All possible guidance trajectories are potentially involved during a landing. Vision-based landing navigation runs throughout the landing to provide a charging pad pose estimate. State estimation is required to provide a reliable state all the way until touchdown. Control is responsible for tracking the generated trajectories. Finally, the autonomy enables the synchronized operation of these four subsystems. As a result, precision landing, like nothing else, is a benchmark test for the successful operation of the GNC system and autonomy engine developed in this thesis.

7.6.1 Indoor Precision Landing

A set of precision landing tests is carried out indoors with the objective of calculating a *landing error ellipse* defined as an ellipse covering a 3 standard deviation landing probability. A single test begins with the quadrotor starting at some given position and yaw, then the landing autopilot (Section 6.4) is triggered and the quadrotor lands. To get meaningful statistics, a large number of landings must be performed. To automate the data collection process, an *auto landing tester*[5] was created similarly to the master state machine. The auto landing tester calls takeoff, commands the quadrotor to hover at some position and then calls the landing autopilot. The initial positions are setup in a grid as shown in Figure 7.25 while the initial yaws are randomized. Starting positions can be set to be repeated multiple times per position. The auto landing tester repeats this cycle until all starting positions and yaws have been tested. An indoor precision landing experiment has the following trade-offs:

+ The auto landing tester enables a completely automatic and hassle-free way for the human operator to collect data for a large number of landings;

+ A flight experiment exposes the system to the many uncertainties associated with reality (dynamics uncertainties, disturbances, delays, camera motion blur, etc.);

– State estimation with VICON is much more precise than with the outdoor GPS-based sensor suite, therefore landings are expected to be more accurate than in the quadrotor's target operating environment (i.e. outdoors);

– A single battery charge permits at most ≈ 20 landings, therefore collecting a larger sample requires intermittent human input to swap the battery (unless the quadrotor is charged from the charging pad between tests, which was not done here).

Figure 7.26 shows the trajectories for 14 such landings with the initial position grid parameters defined in Table 7.4a. The grid is incomplete (not all points were visited twice) due to running out of battery early. Figure 7.27 compares the 14 actual touchdown positions to the ideal position. Let $\boldsymbol{p}_{\text{landing},i} = (p^{\text{b}}_{\text{w,x}}, p^{\text{b}}_{\text{w,y}})$ be the i-th landing location (plotted as red crosses in Figure 7.27). The error ellipse is computed via a maximum likelihood fit given a total of N_{landing} landings:

[5] `/alure_landing_test/include/alure_landing_test/landing_test_state_machine.h`

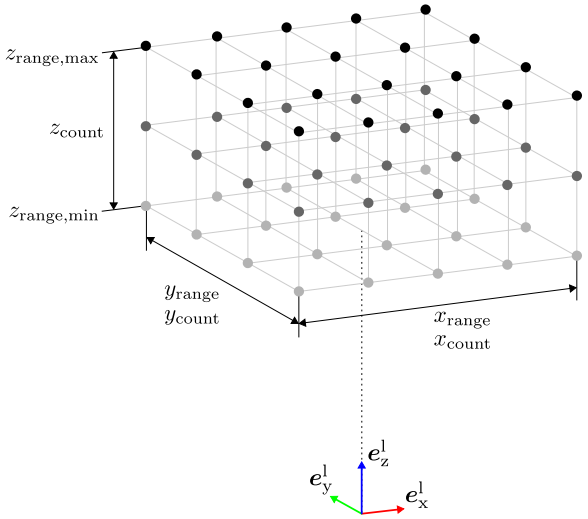


Figure 7.25: Grid of initial positions generated by the auto landing tester. The count variables define the number of positions along each axis. Here, $x_{\text{count}} = 5$, $y_{\text{count}} = 4$ and $z_{\text{count}} = 3$. The point grid is centered around the landing pad origin. Node shading is used purely for visual distinction between the z-layers.

Variable	Value
x_{range}	1 m
y_{range}	0.5 m
$z_{\text{range,min}}$	2 m
$z_{\text{range,max}}$	2 m
x_{count}	3
y_{count}	3
z_{count}	1
repeat	2

(a) VICON test.

Variable	Value
x_{range}	5 m
y_{range}	5 m
$z_{\text{range,min}}$	7 m
$z_{\text{range,max}}$	10 m
x_{count}	5
y_{count}	5
z_{count}	4
repeat	2

(b) RotorS with wind test.

Table 7.4: Initial position grid definitions for the auto landing tester. The "repeat" field specifies how many times each initial position is repeated.

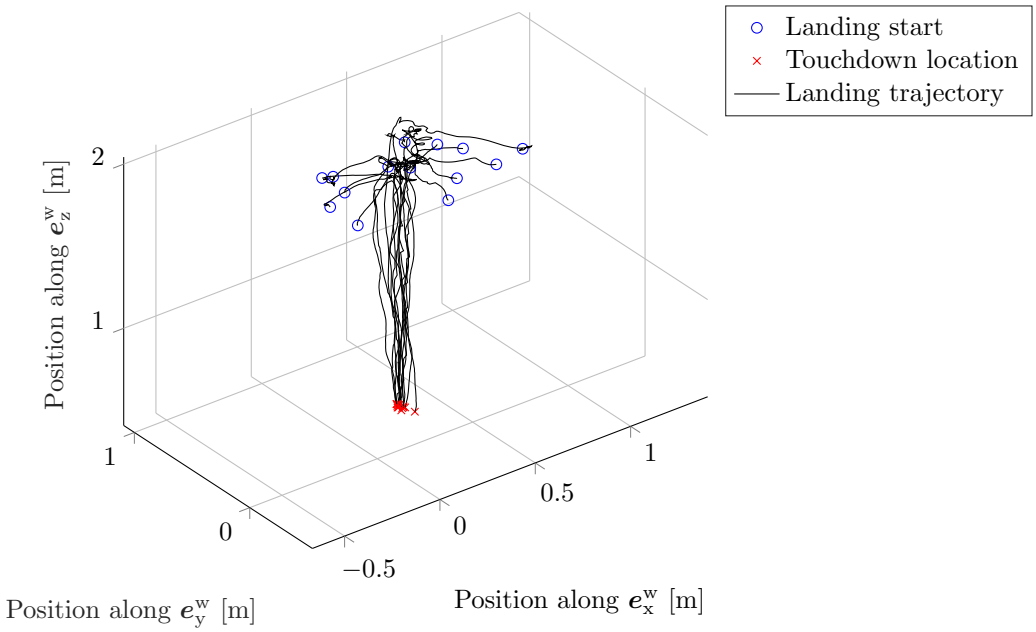


Figure 7.26: 14 indoor landing trajectories. *Indoor data.*

$$\begin{aligned}\boldsymbol{\mu}_{\text{landing}} &= \frac{1}{N_{\text{landing}}} \sum_{i=1}^{N_{\text{landing}}} \boldsymbol{p}_{\text{landing},i}, \\ \Sigma_{\text{landing}} &= \frac{1}{N_{\text{landing}} - 1} \sum_{i=1}^{N_{\text{landing}}} (\boldsymbol{p}_{\text{landing},i} - \boldsymbol{\mu}_{\text{landing}})(\boldsymbol{p}_{\text{landing},i} - \boldsymbol{\mu}_{\text{landing}})^{\mathsf{T}},\end{aligned} \tag{7.4}$$

where $\boldsymbol{\mu}_{\text{landing}}$ is the mean landing position and Σ_{landing} is the landing covariance in the world frame (constrained to the $(\boldsymbol{e}_x^w, \boldsymbol{e}_y^w)$ plane). We expect that $\boldsymbol{\mu}_{\text{landing}}$ should be near the ideal landing position because there should be no landing error bias. Figure 7.27 verifies this. The primary quantifiers of landing precision are the landing ellipse major and minor axes, computed via the singular value decomposition:

$$\Sigma_{\text{landing}} = U_{\text{landing}} S_{\text{landing}} V_{\text{landing}}^*, \tag{7.5}$$

where V_{landing}^* is the Hermitian transpose of V_{landing}. The major and minor axes are given by the diagonal terms of $\sqrt{S_{\text{landing}}}$ and correspond to the standard deviation along the longest and shortest ellipse directions, respectively. Taking 3 standard deviations, **precision landing with VICON achieves major and minor axes of** 8.4 cm **and** 2.6 cm **respectively**. Note that VICON is used only for state estimation – the knowledge of landing pad position and yaw is obtained from our visual landing pipeline (Section 3.1). We conclude that landing in VICON is largely precise enough for the 90×90 cm surface of the charging pad.

An unbiased landing precision should have equal major and minor axes, i.e. the ellipse should be a circle centered at the ideal landing position. That this is not the case for Figure 7.27 is attributed to a small sample size. The "outlier" red cross with the smallest y-value in particular exaggerates the major axis.

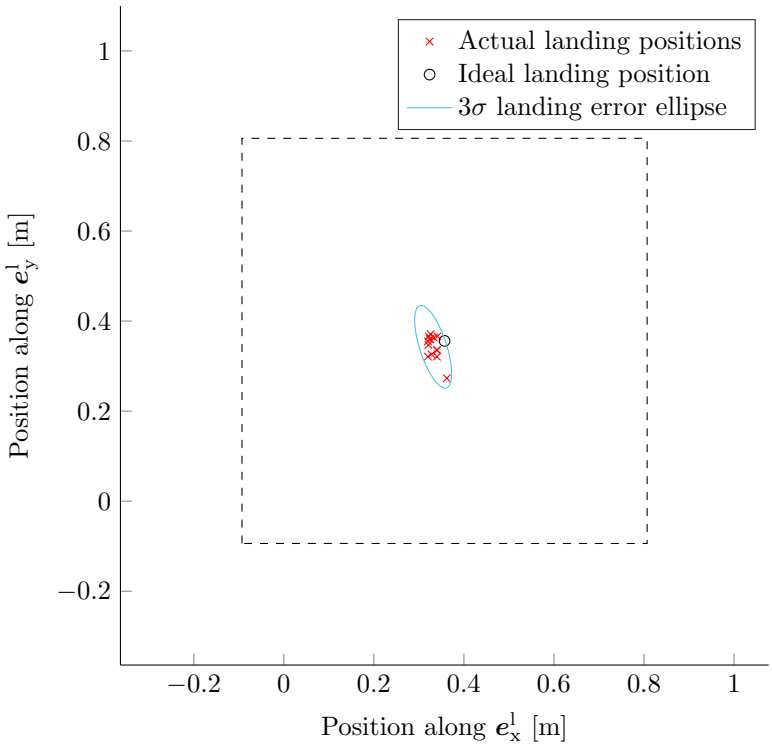


Figure 7.27: Landing error ellipse corresponding to Figure 7.26. The dashed square represents the charging surface boundary. *Indoor data.*

7.6.2 Precision Landing with Wind

The auto landing tester of Section 7.6.1 is used in the RotorS simulation to perform 200 landings in severe wind. The goal is to evaluate the landing precision when large disturbances act on the quadrotor. A RotorS precision landing experiment has the following trade-offs:

+ An arbitrary number of landings can be tested since the simulated quadrotor does not run out of battery;

+ The disturbance force can be precisely controlled, unlike in reality where large fans could simulate wind only to coarse precision;

– The data suffers from all the limitations of RotorS as described in Section 7.1.1. Obvious limitations are e.g. perfectly known quadrotor dynamics and usage of ground truth state for control. More subtle limitations are e.g. reliability of SSF and its sensor suite during landing, which would be revealed only by actually running them through an exhaustive number of landings.

Wind Modeling

Wind is modeled as a stochastic disturbance force[6], $\boldsymbol{f}_\mathrm{d}$. As illustrated in Figure 7.28, $\boldsymbol{f}_\mathrm{d}$ is defined in the {w} frame as:

[6] /simulation_packages/alure_simulation/include/alure_simulation/disturbance_force.h

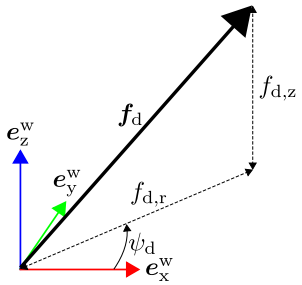


Figure 7.28: Geometry of the stochastic disturbance force.

$$\boldsymbol{f}_{\mathrm{d}} := \begin{bmatrix} f_{\mathrm{d,r}} \cos(\psi_{\mathrm{d}}) \\ f_{\mathrm{d,r}} \sin(\psi_{\mathrm{d}}) \\ f_{\mathrm{d,z}} \end{bmatrix}. \tag{7.6}$$

We shall now assume that $\boldsymbol{f}_{\mathrm{d}}$ is sampled with a sampling period Δt_{d} (currently $50^{-1}\,\mathrm{s}$). The stochastic variables $f_{\mathrm{d,r}}$, $f_{\mathrm{d,z}}$ and ψ_{d} are defined for iteration k to be:

$$f_{\mathrm{d,r}}[k] \sim \mathcal{N}(\mu_{\mathrm{r}}, \sigma_{\mathrm{r}}^2), \tag{7.7}$$

$$f_{\mathrm{d,z}}[k] \sim \mathcal{N}(0, \sigma_{\mathrm{z}}^2), \tag{7.8}$$

$$\psi_{\mathrm{d}}[k] = \psi_{\mathrm{d}}[k-1] + \Delta t_{\mathrm{d}} n_{\psi}, \quad n_{\psi} \sim \mathcal{U}\left(0, \left(\tfrac{\mathrm{d}\psi_{\mathrm{d}}}{\mathrm{d}t}\right)_{\max}\right). \tag{7.9}$$

In other words, $f_{\mathrm{d,r}}$ is Gaussian with non-zero mean (the "average" radial disturbance), $f_{\mathrm{d,z}}$ is Gaussian zero-mean and ψ_{d} is a random walk which allows the disturbance force to act on the quadrotor from different directions.

Results For 200 Landings

We set the disturbance force parameters:

$$\mu_{\mathrm{r}} = 3\,\mathrm{N},\ \sigma_{\mathrm{r}} = 0.3\,\mathrm{N},\ \sigma_{\mathrm{z}} = 0.2\,\mathrm{N},\ \left(\tfrac{\mathrm{d}\psi_{\mathrm{d}}}{\mathrm{d}t}\right)_{\max} = 150\,^{\circ}/\mathrm{s} \text{ and } \psi_{\mathrm{d}}[0] = 0\,^{\circ}.$$

Let us assume that f_{d} corresponds to the aerodynamic drag force which acts on the quadrotor as a result of wind (5.4):

$$\|\boldsymbol{f}_{\mathrm{d}}\| = \frac{1}{2}\rho c_{\mathrm{D}} \|\boldsymbol{v}_{\mathrm{air}}\|^2, \tag{7.10}$$

where $\rho = 1.225\,\mathrm{kg/m^3}$ is the air density and $c_{\mathrm{D}} = 0.08\,\mathrm{m}^2$ is taken from Nguyen Khoi Tran [134] for the AscTec Pelican. Given the radial disturbance force $f_{\mathrm{d,r}} \sim \mathcal{N}(3, 0.3^2)$, this represents a severe wind of over $30\,\mathrm{km/h}$ which borders the no-fly wind speed recommended by AscTec [135]. Indeed, a disturbance force of $4\,\mathrm{N}$ represents $\approx 20\,\%$ of our AscTec Pelican's weight. The precision landing test therefore measures landing accuracy at the limit of recommended wind speeds for flight[7].
Figure 7.29 shows 200 landing trajectories[8] performed using the auto landing tester and with $\boldsymbol{f}_{\mathrm{d}}$ always acting on the quadrotor. The grid parameters are given in Table 7.4b. Figure 7.30 shows the landing error ellipse, computed in the same way as in Section 7.6.1. As expected for an unbiased landing, the landing error ellipse is

[7] A future implementation may link to a local weather stations or a network of anemometers in order to prevent flights at higher wind speeds.

[8] Note that many trajectories contain an initial spiral grid search phase when the quadrotor does not see the landing pad from the landing start location.

quasi-circular and quasi-centered at the ideal landing location. **Precision landing in RotorS with severe wind achieves the major and minor axes of** 13 cm **and** 11.7 cm **respectively**. We conclude that landing in simulation with severe wind is largely precise enough for the 90×90 cm surface of the charging pad.

7.7 Autonomous Recharging During an 11-Hour Flight Test

An early version of the system presented herein was operated full-cycle autonomously indoors for 11-hours. The primary objective was to test autonomous recharging, a fundamental component of full-cycle autonomy. The experiment went as follows:

1. The quadrotor sits on the charging pad until the BatteryCharged event, then it takes off;
2. The mission consists of a simple hover point beside the charging pad;
3. Once the battery is depleted (BatteryLow), the quadrotor lands[9] and repeats the cycle from 1.

Figure 7.31 shows the recorded battery voltage where periods of flight are visible as distinct downward spikes while periods of recharge are the gradual upward voltage trends. When this test was performed, the charging pad had two shortcomings:

1. Charging was done with a limited 4 A current. As shown in the time breakdown in Figure 7.32, this leads to an over-exaggerated down-time when the quadrotor is charging[10];
2. Charging occasionally stopped before full battery charge, as seen e.g. during $t \in [6.6, 7.1]$ hours in Figure 7.31.

The first shortcoming has been improved by increasing the charging current to 6 A while the second shortcoming is now fixed.

[9] Landing in this early version of the GNC and autonomy system did not include visual landing navigation from Section 3.1 (precision landing relied on the VICON system's accuracy).

[10] One of the major advantages of battery hot-swapping, an alternative autonomous recharging technology used by e.g. Matternet [136], is that it essentially removes the charging period. This drastically increases the system up-time.

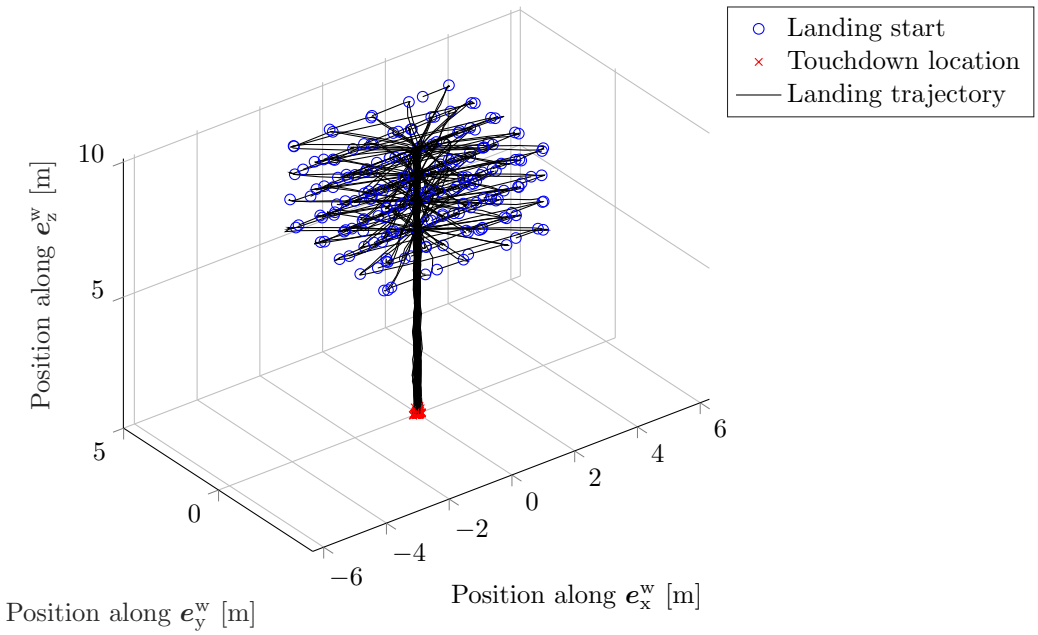


Figure 7.29: 200 landing trajectories with severe wind. *RotorS data.*

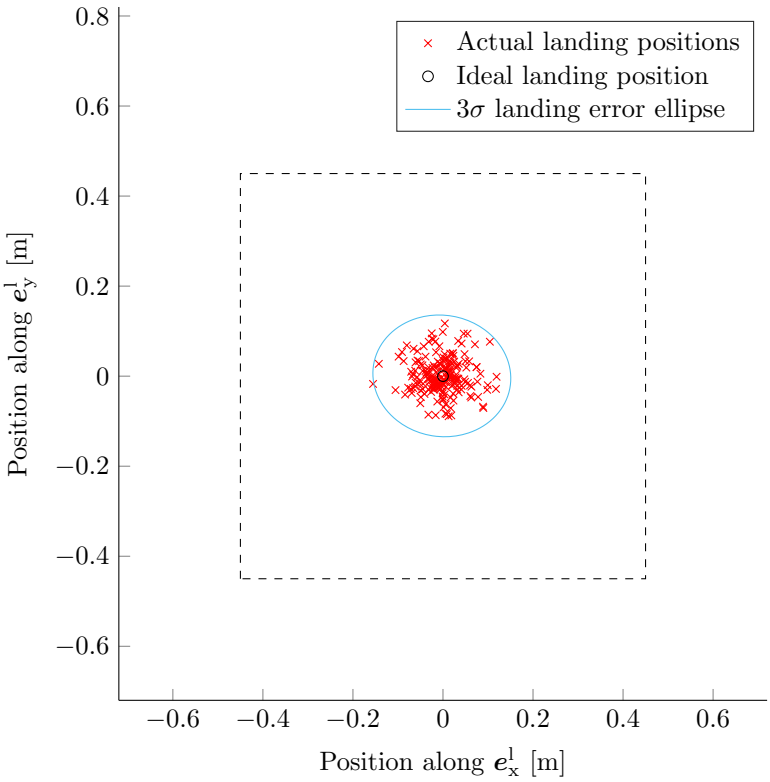


Figure 7.30: Landing error ellipse corresponding to Figure 7.29. The dashed square represents the charging surface boundary. *RotorS data.*

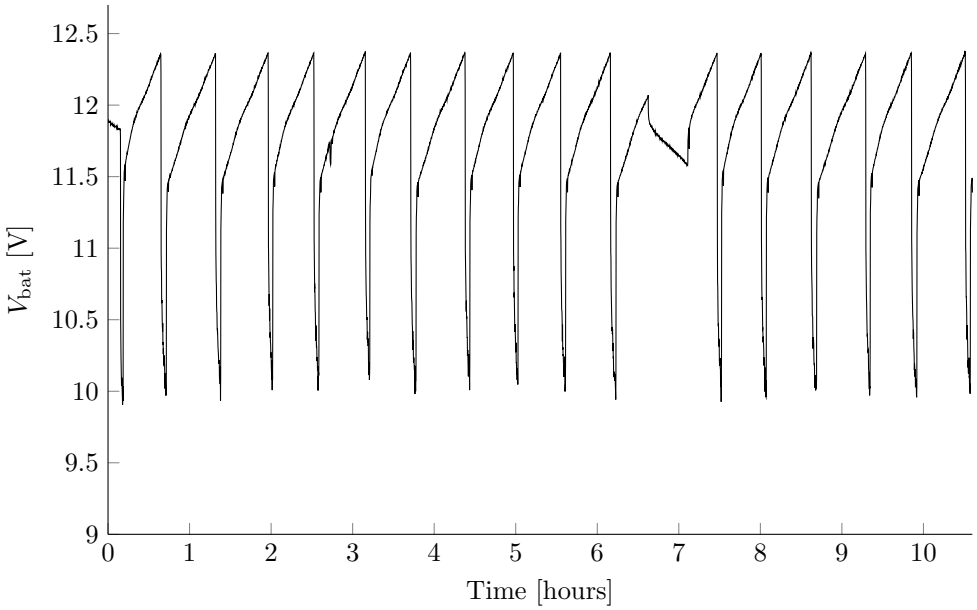


Figure 7.31: Battery voltage over an 11-hour indoor autonomous flight.
Indoor data.

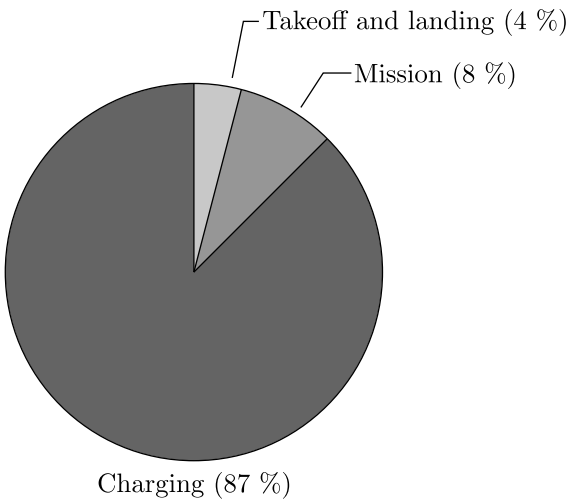


Figure 7.32: Time breakdown of the 11-hour flight test.

This page is intentionally left blank.

Chapter 8

Discussion

8.1 Results

This report presented an end-to-end development of a quadrotor GNC system and an autonomy engine capable of executing a full-cycle autonomous data acquisition mission. The primary contribution of this thesis is the complete development and assembly of many subsystems into a coherent, working system that has been demonstrated in flight.

The results presented herein are a major step towards delivering a much needed full-cycle autonomous MAV to the precision farming, environmental surveying and other markets. The advantage of this work is that it is almost entirely software-based. Together with our use of a charging rather than a battery hot-swapping approach, **the framework presented herein can be deployed on almost any multirotor**. As a result, many existing platforms can potentially be rendered full-cycle autonomous. Nevertheless, it is entirely possible to switch to a battery hot-swapping solution – this will require a specific battery mechanical interface which not all multirotors might satisfy, but the framework presented herein still applies.

Navigation. The navigation system developed in Chapter 3 consists of a state estimator (SSF) and a visual landing navigation pipeline based on AprilTag detection. The specific contribution of this thesis has been the augmentation of the AprilTag 2 algorithm with a bundle pose estimation algorithm, wherein a PnP solution, a bundle calibration method and an RLS filter were introduced. Furthermore, an ideal bundle pose geometry was carried out in Appendix C.

Guidance. The guidance subsystem developed in Chapter 4 consists of a trajectory sequencer, a trajectory tracker and several polynomial trajectory generators. The specific contribution of this thesis has been the trajectory sequencer which implements a rigorous set of internal mechanics for chaining together trajectories in real time while providing the autonomy code with a high-level interface. Our use of VOLATILE trajectories allows to simplify guidance by, when sufficient, providing target positions and letting the tracker automatically generate a trajectory towards them. Large disturbances such as wind also trigger VOLATILE realignment trajectories that provide smoother guidance than if we had let the controller take care of all (even large) tracking errors. Finally, an implemented spiral grid search trajectory solves the crucial practical problem of an invisible landing pad upon return to home, caused by GPS drift during the data acquisition phase.

Control. The control subsystem developed in Chapter 5 consists of a three-stage cascaded control loop that was implemented from scratch based on the latest and most robust multirotor control methods. An emergency landing controller has also been implemented as a fallback measure in case of state estimator failure. The primary contribution of this thesis has been the complete assembly, deployment and analysis of the control loop which consists of otherwise individually presented algorithms in the literature. Importantly, a more computationally efficient yaw controller was presented in Section 5.7. The control system is deployable on any multirotor and is amenable to SIL simulation unlike the previous control loop which relied on the inaccessible AscTec LLP attitude controller. SIL simulation allows for vastly faster design iterations and frees the physical MAV for use by other engineers. Good performance of the control system has been shown in simulated, indoor and outdoor flights. Robustness has also been evaluated and shall be further discussed in Section 8.2.

Autonomy Engine. A state machine based hierarchical autonomy engine has been developed from scratch in Chapter 6. This engine implements logic specific to the full-cycle data acquisition mission. Beyond nominal operation, multiple robustness features have been included to account for a low or critical battery charge and an invisible landing pad after return to home.

8.2 Future Work

Experience with developing the system from the ground up has highlighted multiple avenues for future development, which are discussed below.

Navigation Improvements

Latency Analysis. Section 7.4.3 puts forwards a hypothesis that a variable feedback delay of up to 40 ms may be present in the system. This would cause position and but not velocity mode oscillations on an intermittent basis, as was observed during flight tests. It is advised, as a top priority, to carry out a rigorous full-feedback-loop latency analysis to determine exactly how much latency (i.e. feedback delay) a controller must be robustly peformant to. Lupashin et al. [103] cite 22 ms to 46 ms in their system, of which up to 24 ms "represents variable delays due to the independent asynchronous loops in the system". Importantly, this is the same order of magnitude as our hypothesis and would cause position control, but not velocity control, to exhibit oscillatory tracking. The results from the latency analysis may help, as a first step, to modify the system so as to reduce the latency.

AprilTag Bundle Pose Measurement Frequency. As shown in Figure 3.5, the current bundle pose measurement caps at ≈ 7 Hz due to computational constraints of the Odroid XU4. `cv::solvePnP` is the potential bottleneck (that should be verified via profiling) as it is a heavy iterative method which uses Levenberg-Marquardt optimization OpenCV [66]. Increasing this frequency would be beneficial for a faster detection rate which can help to get a better bundle pose estimate (since more measurements would be collected) and increase reaction time since measurements would arrive faster.

Downfacing Camera Resolution. The current 752×480 px downfacing camera limits the maximum bundle detection range due to sheer pixel count (an AprilTag is physically not detectable when one bit becomes smaller than one pixel). Larger detection ranges require disproportionately large tags (> 1 m in size). A better

approach would be to increase camera resolution such that tags may be detected from a larger distance, e.g. 32 m using 1080p video as achieved in Fnoop and Hadaway [49].

RotorS Implementation. SSF and its array of sensors can be implemented as part of the current RotorS SIL simulation. Eventually, even a HIL simulation can be done. Implementing SSF in RotorS will increase the simulation's realism and potentially serve as a very useful tool for further developments of SSF itself.

Guidance Improvements

More Advanced Trajectory Generation. Current trajectories are simple fully constrained polynomials. To extend flight time, energy-minimizing optimal trajectories may be generated, for which an optimization problem will have to be solved. A potential avenue here is to adapt existing research on using convex optimization to plan reliable minimum-fuel trajectories for planetary landers Acikmese and Ploen [96], Blackmore et al. [97], Blackmore and Acikmese [98], Acikmese et al. [99], Szmuk et al. [100]. There is also more direct research for quadrotor minimum-energy and minimum-time trajectory generation Vicencio et al. [94], Morbidi et al. [95], Hehn and D'Andrea [137], Ritz et al. [138], Mueller and D'Andrea [91], Mueller et al. [92]. Of course, standard time-optimal minimum-snap polynomial trajectories may also be implemented Mellinger and Kumar [81], Richter et al. [85], Burri et al. [86], however our experience with them has been negative as the generation takes both a long time (which grows with the number of waypoints) and the resulting geometry can be quite different from a straight-line path, requiring the addition of more waypoints. Open-source frameworks for time-optimal minimum-snap trajectory generation already exist Oleynikova et al. [139]. Note that because our quadrotor's aim is to be robust, not necessarily highly dynamic, strictly dynamically feasible trajectory generation takes a back seat to robust/energy optimal trajectory generation.

Yaw Trajectory Generation. The current framework only generates position trajectories (i.e. in $\mathbb{R}^3$) while yaw may be passed as target values to the trajectory tracker. If required, the implementation may be extended to trajectories in $\mathbb{R}^4$ to incorporate yaw.

Feed-Forward Terms. The trajectory tracker currently passes only $\boldsymbol{p}_{\text{ref}}$ for position control and $\boldsymbol{v}_{\text{ref}}$ for velocity control. If tracking lag is removed (e.g. by using the 1 DOF control architectures in Section 5.8), it may be possible to feed forward $\boldsymbol{a}_{\text{ff}}$ and $\boldsymbol{\omega}_{\text{ff}}$ from the dynamically feasible polynomial trajectory generation.

Control Improvements

Translation Control Tuning. Section 7.4.3 puts forward a hypothesis that a variable feedback delay of up to 40 ms may be present in the system. This would cause position and but not velocity mode oscillations on an intermittent basis, as was observed during flight tests. It was already mentioned as a navigation improvement to determine exactly how much latency is present in our indoor (VICON-based) and outdoor (GPS-based) system configuration. The result of latency is an oscillatory position tracking which leads to non-repeatable performance between flights. One avenue for fixing this was already mentioned – reduce the latency. Another is to tune down the position controller, or employ an altogether different control technique that is robust to higher latencies (e.g. $\mathcal{H}_{\text{inf}}$ Skogestad and Postlethwaite

[102] or MPC approaches). The aim would be for the position controller to be robustly performant to latencies (i.e. feedback delays) of up to the identified system maximum latency.

Pre-Filter Numerical Stability. It was mentioned in Section 5.8.4 that the discrete transfer function form of our pre-filters diverges while the observer canonical state space realization is stable. However, the observer canonical form is still not the most numerically stable. It is suggested to implement a *balanced state space realization* of the pre-filters for maximum numerical stability Silverman [140], De Schutter [141]. Furthermore, the current implementation is not robust to a zero derivative term ($k_\mathrm{d} = 0$) in the position control or a zero proportional term ($k_\mathrm{p} = 0$) in the velocity control due to a division by zero singularity in Table 5.6 coefficients. For safety, it is suggested to handle these edge cases such that the operator may not accidentally compromise the control system numerics due to a poor tuning choice.

Better Emergency Landing Control. The emergency landing controller in Section 5.9 is a simple fallback algorithm in case of state estimation failure. A more advanced and robust emergency landing controller may be implemented via e.g. Mueller and D'Andrea [116], Faessler et al. [105].

Closed Loop Motor Control. As discussed in Appendix D, it can be beneficial for control accuracy to implement closed loop motor control. Not only will this enable usage of the more physically accurate map from propeller angular speed (instead of ESC command) to motor thrust and torque, but it will also render the motor thrust and torque characteristic independent from the battery charge.

Two-Stage Cascaded Control. The weak time scale separation discussed in Chapters 5 and 7 may potentially be improved by a two-stage cascaded controller consisting of just the translation and body rate controllers Achtelik et al. [109]. The translation controller in this case exploits quadrotor differential flatness to output $\boldsymbol{\omega}_\mathrm{ref}$ directly.

Autonomy Engine Improvements

State Machine Implementation. The autonomy engine currently uses self developed state machine mechanics that are loosely based on the UML formalism (see Appendix B). More predictable and expandable behavior can likely be achieved by porting the existing state machines to a framework such as the State Machine Compiler (SMC) Rapp et al. [142]. Besides implementing a much more robust and well-defined set of state machine mechanics (that are also close to UML), SMC follows the Open-Closed Principle (OCP) design pattern which stipulates that "software entities must be open for extension but closed for modification". In practice for the autonomy engine state machines it means that new states and behaviors may be added without modifying source code for existing states and behaviors. This is not the case for the current implementation where the entire state machine lives in a single stateMachineDefinition() method. Note that porting the autonomy engine to SMC (or another similar framework) is particularly easy due to the fact that the state machines in Chapter 6 are already designed based on UML state machine diagram syntax which SMC supports.

Autonomy Engine Verification & Validation. Some autonomous car companies have begun using formal methods for Verification & Validation (V&V) of their cars' complex autonomy logic Ackerman [143], Almeida et al. [144]. Although the

autonomy engine presented herein is much simpler than that of an autonomous vehicle, a push to commercial operation of a system such as the one presented herein will likely see both a significant increase in autonomy engine complexity (to handle more edge cases and offer more features) and a serious V&V requirement to ensure that the system is safe to operate at will. Formal methods are a potential V&V tool.

Software Crash Automatic Recovery. Failure of any ROS node in the current implementation will result in a breach of full-cycle autonomy, if not a crash. For example, if the AscTec HLP interface fails, the state estimate can no longer be sent to the HLP and will trigger the emergency landing controller. If the AprilTag bundle pose measurement node crashes, landing navigation becomes not possible (the quadrotor will think that the landing pad is invisible and will execute a spiral grid search in vain). A potential solution is to implement a highly reliable supervisory software that is able to restart failed ROS nodes, or the entire ROS network if necessary. Some progress has been made on this front to already store the return to home location in permanent memory (in a local SQLite database) such that data can be recovered once the network is running again. Certain failure modes have been observed that required a hard restart of the HLP in order to re-establish serial communication – therefore, a hardware solution based on e.g. a relay may be implemented to give this supervisory software the ability to restart the HLP. Extreme levels of reliability that are most likely not achievable in a research project (i.e. requiring a dedicated company) would likely require hardware and OS customization in order to support a fully self-healing system.

Landing Autopilot Improvements. The current landing autopilot cannot exit from the DESCEND state once it has entered it. However, if a disturbance occurs very close to the ground or the landing pad moves for some reason without the RLS filter having time to re-converge to its new location, then precision landing is unlikely (the quadrotor will touch down somewhere next to the charging pad instead). A virtual landing cone could be defined within which the quadrotor must remain throughout the descent phase. If it exits this cone, logic can be implemented to halt the descent and re-enter the cone before descending further. If landing nevertheless misses the charging pad, logic could be implemented to takeoff and re-attempt a landing (provided sufficient battery charge is available).

Miscellaneous Improvements

GUI Interface. A Graphical User Interface (GUI) may be implemented for a more user-friendly interaction with the system than through a combination of the terminal and ROS dynamic reconfigure.

Mission Data Synchronization. The current mission data synchronizer is a rudimentary ROS node based on McClung [129]. A more advanced implementation would be required in a commercial (or simply more user-focused) platform that would allow the user to seamlessly interact with collected data without worrying about the MAV itself (which remains full-cycle autonomous).

Weather and Lighting Awareness. Rain, extreme wind and low visibility (due to fog or night time) all potentially represent no-fly conditions for the quadrotor. An any-day full-cycle autonomous system will have to autonomously verify the local weather and visibility so as to prevent flight in out-of-envelope conditions.

Better RotorS Wind Simulation. The generic stochastic disturbance force presented in Section 7.6.2 can be exchanged for a more realistic wind force model such as the Dryden wind turbulence model Sytsma and Ukeiley [145], Department of Defense [146].

8.3 Conclusion

This master thesis has developed from the ground up a GNC system and an autonomy engine for a full-cycle autonomous quadrotor that can fly data acquisition missions for applications like precision agriculture and environmental surveying. In this sense, the thesis goal (Section 1.3) has been fulfilled. Furthermore, the modified RotorS simulator provides future JPL engineers with an end-to-end baseline SIL simulation environment for further developing the present system. This work represents a solid step towards full-cycle autonomous data collection drones, of which the author believes that there will be many in the near future.

Bibliography

[1] Jet Propulsion Laboratory, "Jet Propulsion Laboratory NASA Facts," https://www.jpl.nasa.gov/news/fact_sheets/jpl.pdf, accessed: 2017-12-10.

[2] JPL Infographics, "JPL Mission History," https://www.jpl.nasa.gov/infographics/infographic.view.php?id=10743, accessed: 2017-12-10.

[3] Goldman Sachs, "Drones: Reporting for Work," http://www.goldmansachs.com/our-thinking/technology-driving-innovation/drones/, accessed: 2017-12-10.

[4] The Economist, "Taking flight," http://www.economist.com/technology-quarterly/2017-06-08/civilian-drones, accessed: 2017-12-10.

[5] Andrew Meola, "Drone market shows positive outlook with strong industry growth and trends," http://www.businessinsider.com/drone-industry-analysis-market-trends-growth-forecasts-2017-7, accessed: 2017-12-10.

[6] Jonathan Vanian, "The Multi-Billion Dollar Robotics Market Is About to Boom," http://fortune.com/2016/02/24/robotics-market-multi-billion-boom/, accessed: 2017-12-10.

[7] FAO, "2050: A third more mouths to feed," http://www.fao.org/news/story/en/item/35571/icode/, accessed: 2017-12-10.

[8] Ascending Technologies, "UAS Precision Farming in Spain," http://www.asctec.de/en/uas-precision-farming-in-spain/, accessed: 2017-12-10.

[9] SenseFly, "Drones for Environmental Protection and Conservation," https://www.sensefly.com/applications/environmental-protection.html, accessed: 2017-12-10.

[10] Ascending Technologies, "UAV Precision Agriculture," http://www.asctec.de/en/uav-uas-drone-applications/uav-precision-agriculture/, accessed: 2017-12-10.

[11] ——, "UAV Surveying and Mapping," http://www.asctec.de/en/uav-uas-drone-applications/uav-surveying-mapping-orthophoto-photogrammetry/, accessed: 2017-12-10.

[12] senseFly SA, "Accessories," https://www.sensefly.com/drones/accessories.html, accessed: 2017-12-10.

[13] DJI and PrecisionHawk, "Smarter Farming Package," https://store.dji.com/product/smarter-farmer-kit, accessed: 2017-12-10.

[14] PrecisionHawk, "PrecisionMapper," http://www.precisionhawk.com/precisionmapper, accessed: 2017-12-10.

[15] ——, "PrecisionViewer," http://www.precisionhawk.com/precisionviewer, accessed: 2017-12-10.

[16] ——, "PrecisionFlight," http://www.precisionhawk.com/precisionflight, accessed: 2017-12-10.

[17] Thuy Ong, "The first autonomous drone delivery network will fly above Switzerland starting next month," https://www.theverge.com/2017/9/20/16325084/matternet-autonomous-drone-network-switzerland, accessed: 2017-12-10.

[18] AeroVinci, "Insight at Scale," http://www.aerovinci.com/, accessed: 2017-12-10.

[19] Matternet, "Our Mission," https://mttr.net/company, accessed: 2017-12-10.

[20] D. Kane, "Set Theory Notation," Stanford University, Tech. Rep.

[21] Matrix Vision GmbH, "USB 2.0 board-level camera - mvBlueFOX-MLC," https://www.matrix-vision.com/USB2.0-single-board-camera-mvbluefox-mlc.html?camera=mvBlueFOX-MLC200wG, accessed: 2017-12-10.

[22] FLIR, "FLIR Ax5-Series," http://www.flir.com/automation/display/?id=56341, accessed: 2017-12-10.

[23] Hardkernel, "WiFi Module 4," http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141630348024, accessed: 2017-12-10.

[24] Trimble, "BD930-UHF," http://www.trimble.com/gnss-inertial/bd930-uhf.aspx?dtID=overview, accessed: 2017-12-10.

[25] Honeywell, "3-Axis Digital Compass IC HMC5843," https://aerocontent.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Defense_Brochures-documents/HMC5843.pdf, accessed: 2017-12-10.

[26] Skysense, "Charging Pad Technical Specifications," http://www.skysense.co/charging-pad/specs/, accessed: 2017-12-10.

[27] Casalrc, "C6S Instruction Manual," http://modellflybutikken.no/docs/default-source/CE-dokumenter/casalrc_c6s_c4s_manual.pdf?sfvrsn=0, accessed: 2017-12-10.

[28] Hardkernel, "Odroid-XU4," http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825, accessed: 2017-12-10.

[29] HobbyKing, "ZIPPY Compact 6200mAh 3s 40c Lipo Pack," https://hobbyking.com/en_us/zippy-compact-6200mah-3s-40c-lipo-pack.html?___store=en_us, accessed: 2017-12-10.

[30] Ascending Technologies, "Pinout and Connections," http://wiki.asctec.de/display/AR/Pinout+and+Connections, accessed: 2017-12-10.

[31] NXP, "LPC2146 ARM7 Microcontroller Datasheet," https://www.nxp.com/docs/en/data-sheet/LPC2141_42_44_46_48.pdf?, accessed: 2017-12-10.

[32] Memsic, "MXR9500G/M Tri Axis Accelerometer Datasheet," http://www.memsic.com/userfiles/files/Datasheets/Accelerometer-Datasheets/MXR9500GM__RevD.pdf, accessed: 2017-12-10.

[33] Analog Devices, "ADXRS620 Gyro Datasheet," http://www.analog.com/media/en/technical-documentation/data-sheets/ADXRS620.pdf, accessed: 2017-12-10.

[34] NXP, "MPXH6115A Integrated Pressure Sensor Datasheet," http://cache.freescale.com/files/sensors/doc/data_sheet/MPXA6115A.pdf, accessed: 2017-12-10.

[35] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[36] M. Achtelik, M. Achtelik, S. Weiss, and L. Kneip, "asctec_mav_framework," http://wiki.ros.org/asctec_mav_framework, accessed: 2017-12-10.

[37] C. Sharp, O. Shakernia, and S. Sastry, "A vision system for landing an unmanned aerial vehicle," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 2. IEEE, pp. 1720–1727.

[38] S. Saripalli, J. Montgomery, and G. Sukhatme, "Visually guided landing of an unmanned aerial vehicle," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 3, pp. 371–380, 2003.

[39] J. Shaogang, Z. Jiyang, S. Lincheng, and L. Tengxiang, "On-board vision autonomous landing techniques for quadrotor: A survey," *Chinese Control Conference, CCC*, vol. 2016-Augus, pp. 10 284–10 289, 2016.

[40] S. Yang, S. A. Scherer, and A. Zell, "An Onboard Monocular Vision System for Autonomous Takeoff, Hovering and Landing of a Micro Aerial Vehicle," *Journal of Intelligent & Robotic Systems*, vol. 69, no. 1-4, pp. 499–515, jan 2013.

[41] G. Klein and D. Murray, "Parallel tracking and mapping for small AR workspaces," *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR*, 2007.

[42] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*. IEEE, nov 2011, pp. 2564–2571.

[43] C. Forster, M. Faessler, F. Fontana, M. Werlberger, and D. Scaramuzza, "Continuous on-board monocular-vision-based elevation mapping applied to autonomous landing of micro aerial vehicles," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2015, pp. 111–118.

[44] C. Forster, M. Pizzoli, and D. Scaramuzza, "SVO: Fast semi-direct monocular visual odometry," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2014, pp. 15–22.

[45] M. Huber, "Autonomous Rotorcraft Landing with Structured Light Stereo Vision," Master thesis, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109, USA, May 2017.

[46] R. Brockers, P. Bouffard, J. Ma, L. Matthies, and C. Tomlin, "Autonomous landing and ingress of micro-air-vehicles in urban environments based on monocular vision," *Proc.SPIE*, vol. 8031, pp. 8031 – 8031 – 12, 2011.

[47] R. Brockers, S. Susca, D. Zhu, and L. Matthies, "Fully self-contained vision-aided navigation and landing of a micro air vehicle independent from external sensor inputs," *Proc.SPIE*, vol. 8387, pp. 8387 – 8387 – 10, 2012.

[48] V. R. Desaraju, N. Michael, M. Humenberger, R. Brockers, S. Weiss, J. Nash, and L. Matthies, "Vision-based landing site evaluation and informed optimal trajectory generation toward autonomous rooftop landing," *Autonomous Robots*, vol. 39, no. 3, pp. 445–463, Oct 2015.

[49] Fnoop and K. Hadaway, "vision_landing," https://github.com/fnoop/vision_landing, accessed: 2017-12-10.

[50] M. Fiala, "ARTag, a Fiducial Marker System Using Digital Techniques," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2. IEEE, pp. 590–596.

[51] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[52] R. Bencina, M. Kaltenbrunner, and S. Jorda, "Improved Topological Fiducial Tracking in the reacTIVision System," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, vol. 3. IEEE, pp. 99–99.

[53] A. Xu and G. Dudek, "Fourier tag: A smoothly degradable fiducial marker system with configurable payload capacity," *Proceedings - 2011 Canadian Conference on Computer and Robot Vision, CRV 2011*, pp. 40–47, 2011.

[54] F. Bergamasco, A. Albarelli, L. Cosmo, E. Rodola, and A. Torsello, "An Accurate and Robust Artificial Marker Based on Cyclic Codes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 12, pp. 2359–2373, 2016.

[55] J. Wang and E. Olson, "AprilTag 2: Efficient and robust fiducial detection," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[56] Ling, Kevin, "Precision Landing of a Quadrotor UAV on a Moving Target Using Low-Cost Sensors," Master's thesis, University of Waterloo, 2014.

[57] S. Kyristsis, A. Antonopoulos, T. Chanialakis, E. Stefanakis, C. Linardos, A. Tripolitsiotis, and P. Partsinevelos, "Towards Autonomous Modular UAV Missions: The Detection, Geo-Location and Landing Paradigm." *Sensors (Basel, Switzerland)*, vol. 16, no. 11, nov 2016.

[58] A. Borowczyk, D. Nguyen, A. P. Nguyen, D. Q. Nguyen, D. Saussié, and J. L. Ny, "Autonomous Landing of a Multirotor Micro Air Vehicle on a High Velocity Ground Vehicle," *CoRR*, vol. abs/1611.07329, 2016.

[59] S. M. Chaves, R. W. Wolcott, and R. M. Eustice, "NEEC research: Toward GPS-denied landing of unmanned aerial vehicles on ships at sea," *Naval Engineers Journal*, pp. 1–10, 2015.

[60] Amazon, "Amazon Prime Air's First Customer Delivery," https://www.youtube.com/watch?v=vNySOrI2Ny8, accessed: 2017-12-11.

[61] J. Vincent, "Google's Project Wing has successfully tested its air traffic control system for drones," https://www.theverge.com/2017/6/8/15761220/google-project-wing-drone-air-traffic-control-tests, accessed: 2017-12-10.

[62] Wingtra, "How it Works," https://wingtra.com/workflow/, accessed: 2017-12-10.

[63] F. Devernay and O. Faugeras, "Straight Lines Have to Be Straight: Automatic Calibration and Removal of Distortion from Scenes of Structured Enviroments," *Mach. Vision Appl.*, vol. 13, no. 1, pp. 14–24, Aug. 2001.

[64] APRIL Laboratory, "AprilTags Visual Fiducial System," https://april.eecs.umich.edu/software/apriltag/, version 2015-03-18.

[65] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, ISBN: 0521540518, 2004.

[66] OpenCV, "cv::solvePnP()," https://docs.opencv.org/3.3.1/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d, accessed: 2017-12-10.

[67] Blender, "Blender – a 3D modelling and rendering package," http://www.blender.org, Blender Foundation, Blender Institute, Amsterdam, 2017, accessed: 2017-12-10.

[68] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman, "Averaging Quaternions," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 4, pp. 1193–1197, jul 2007.

[69] N. Trawny and S. I. Roumeliotis, "Indirect Kalman filter for 3D Attitude Estimation," *Dept. of Comp. Sci. & Eng.*, no. 2005-002, pp. 1–25, 2005.

[70] J. Solà, "Quaternion kinematics for the error-state kalman filter," *CoRR*, vol. abs/1711.02508, 2017.

[71] Martin John Baker, "Maths – Conversion Matrix to Quaternion," http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/, accessed: 2017-12-10.

[72] R. Derek, "Introduction to Recursive-Least-Squares (RLS) Adaptive Filters," MIT, Tech. Rep., 2008.

[73] ISO, "Programming Languages – C++," ISO, Tech. Rep., 2013.

[74] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[75] D. E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison-Wesley Pub. Co, 1973.

[76] T. Finch, "Incremental calculation of weighted mean and variance," University of Cambridge Computing Service, Tech. Rep., 2009.

[77] S. Weiss, M. Achtelik, M. Chli, and R. Siegwart, "Versatile distributed pose estimation and sensor self-calibration for an autonomous MAV," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 31–38, 2012.

[78] S. Weiss and R. Siegwart, "Real-time metric state estimation for modular vision-inertial systems," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 231855, pp. 4531–4537, 2011.

[79] Terabee, "TeraRanger One," http://www.teraranger.com/products/teraranger-one/, accessed: 2017-12-10.

[80] M. S. Grewal, L. R. Weill, and A. P. Andrews, *Global Positioning Systems, Inertial Navigation, and Integration*. Hoboken, NJ, USA: John Wiley & Sons, Inc., jan 2007.

[81] D. Mellinger and V. Kumar, "Minimum Snap Trajectory Generation and Control for Quadrotors," *2011 IEEE International Conference on Robotics and Automation*, pp. 2520–2525, 2011.

[82] R. M. Murray, M. Rathinam, and W. Sluis, "Diffierential Flatness of Mechanical Control Systems: A Catalog of Prototype Systems," *ASME International Mechanical Engineering Congress and Expo*, 1995.

[83] P. Martin, "Endogenous feedbacks and equivalence," in *Proceedings of the Internationnal Symposium MTNS*, Regensburg, Germany, 1993.

[84] V. Mistler, A. Benallegue, and N. M'Sirdi, "Exact linearization and noninteracting control of a 4 rotors helicopter via dynamic feedback," in *Proceedings 10th IEEE International Workshop on Robot and Human Interactive Communication. ROMAN 2001 (Cat. No.01TH8591)*. IEEE, pp. 586–593.

[85] C. Richter, A. Bry, and N. Roy, "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments," *International Conference on Robotics and Automation*, no. Isrr, pp. 1–16, 2013.

[86] M. Burri, H. Oleynikova, M. W. Achtelik, and R. Siegwart, "Real-time visual-inertial mapping, re-localization and planning onboard MAVs in unknown environments," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2015-Decem, pp. 1872–1878, 2015.

[87] M. M. de Almeida and M. Akella, "New numerically stable solutions for minimum-snap quadcopter aggressive maneuvers," in *2017 American Control Conference (ACC)*. IEEE, may 2017, pp. 1322–1327.

[88] H. Oleynikova, Z. Taylor, R. Siegwart, and J. I. Nieto, "Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles," *CoRR*, vol. abs/1710.00604, 2017.

[89] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar, "Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1688–1695, jul 2017.

[90] L. Campos-Macias, D. Gomez-Gutierrez, R. Aldana-Lopez, R. de la Guardia, and J. I. Parra-Vilchis, "A Hybrid Method for Online Trajectory Planning of Mobile Robots in Cluttered Environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 935–942, apr 2017.

[91] M. Mueller and R. D'Andrea, "A model predictive controller for quadrocopter state interception," *Proceedings of the European Control Conference (ECC), 2013*, pp. 1383–1389, 2013.

[92] M. Mueller, M. Hehn, and R. D'Andrea, "A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[93] M. Hehn and R. D'Andrea, "Real-Time Trajectory Generation for Quadrocopters," *IEEE Transactions on Robotics*, vol. 31, no. 4, pp. 877–892, 2015.

[94] K. Vicencio, T. Korras, K. A. Bordignon, and I. Gentilini, "Energy-optimal path planning for six-rotors on multi-target missions," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, sep 2015, pp. 2481–2487.

[95] F. Morbidi, R. Cano, D. Lara, F. Morbidi, R. Cano, D. Lara, M.-e. P. Generation, F. Morbidi, R. Cano, and D. Lara, "Minimum-Energy Path Generation for a Quadrotor UAV," *International Conference on Robotics and Automation*, no. May, pp. 2–8, 2016.

[96] B. Acikmese and S. R. Ploen, "Convex Programming Approach to Powered Descent Guidance for Mars Landing," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 5, pp. 1353–1366, 2007.

[97] L. Blackmore, B. Acikmese, and D. P. Scharf, "Minimum-Landing-Error Powered-Descent Guidance for Mars Landing Using Convex Optimization," *Journal of Guidance, Control, and Dynamics*, vol. 33, no. 4, pp. 1161–1171, 2010.

[98] L. Blackmore and B. Acikmese, "Lossless Convexification of a Class of Optimal Control Problems with Non-Convex Control Constraints," *Systems & Control Letters*, no. September 2009, pp. 0–6, 2012.

[99] B. Acikmese, J. Carson, and L. Blackmore, "Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 6, pp. 2104–2113, 2013.

[100] M. Szmuk, B. Acikmese, and A. W. Berning, "Successive Convexification for Fuel-Optimal Powered Landing with Aerodynamic Drag and Non-Convex Constraints," *AIAA Guidance, Navigation, and Control Conference*, no. January, pp. 1–16, 2016.

[101] Wikipedia, "Underwater Searches," https://en.wikipedia.org/wiki/Underwater_searches#Spiral_box_search, accessed: 2017-12-10.

[102] S. Skogestad and I. Postlethwaite, *Multivariable feedback control : analysis and design*. John Wiley, 2005.

[103] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea, "A platform for aerial robotics research and demonstration: The flying machine arena," *Mechatronics*, vol. 24, no. 1, pp. 41 – 54, 2014.

[104] M. Faessler, F. Fontana, C. Forster, E. Mueggler, M. Pizzoli, and D. Scaramuzza, "Autonomous, Vision-based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle," *Journal of Field Robotics*, vol. 7, no. PART 1, pp. 81–86, 2015.

[105] M. Faessler, F. Fontana, C. Forster, and D. Scaramuzza, "Automatic reinitialization and failure recovery for aggressive flight with a monocular vision-based quadrotor," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, no. June, pp. 1722–1729, 2015.

[106] D. Brescianini, M. Hehn, and R. D'Andrea, "Nonlinear Quadrocopter Attitude Control. Technical Report," ETH Zurich, Tech. Rep., 2013.

[107] M. Faessler, D. Falanga, and D. Scaramuzza, "Thrust Mixing, Saturation, and Body-Rate Control for Accurate Aggressive Quadrotor Flight," *IEEE Robotics and Automation Letters*, vol. PP, no. 99, pp. 1–7, 2016.

[108] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart, "Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, may 2011, pp. 3056–3063.

[109] M. Achtelik, S. Lynen, M. Chli, and R. Siegwart, "Inversion based direct position control and trajectory following for micro aerial vehicles," *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2933–2939, 2013.

[110] T. Lee, M. Leok, and N. Harris McClamroch, "Control of Complex Maneuvers for a Quadrotor UAV using Geometric Methods on SE(3)," *ArXiv e-prints*, Mar. 2010.

[111] T. Lee, M. Leok, and N. H. Mcclamroch, "Nonlinear robust tracking control of a quadrotor UAV on SE(3)," *Asian Journal of Control*, vol. 15, no. 2, pp. 391–408, 2013.

[112] D. Gurdan, J. Stumpf, M. Achtelik, K.-M. Doth, G. Hirzinger, and D. Rus, "Energy-efficient Autonomous Four-rotor Flying Robot Controlled at 1 kHz," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, apr 2007, pp. 361–366.

[113] D. Lucena de Athayde Guimarães, "AscTec Pelican CAD Model," http://wiki.asctec.de/display/AR/CAD+Models, accessed: 2017-12-10.

[114] W. F. Phillips, *Mechanics of Flight*. John Wiley, 2010.

[115] P. Martin and E. Salaun, "The true role of accelerometer feedback in quadrotor control," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, may 2010, pp. 1623–1629.

[116] M. Mueller and R. D'Andrea, "Critical subsystem failure mitigation in an indoor UAV testbed," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, oct 2012, pp. 780–785.

[117] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, "Precision flight control for a multi-vehicle quadrotor helicopter testbed," *Control Engineering Practice*, vol. 19, no. 9, pp. 1023 – 1036, 2011, special Section: DCDS'09 – The 2nd IFAC Workshop on Dependable Control of Discrete Systems.

[118] H. Melkote, F. Khorrami, S. Jain, and M. Mattice, "Robust adaptive control of variable reluctance stepper motors," *IEEE Transactions on Control Systems Technology*, vol. 7, no. 2, pp. 212–221, mar 1999.

[119] A. Kapun, M. Čurkovič, A. Hace, and K. Jezernik, "Identifying dynamic model parameters of a BLDC motor," *Simulation Modelling Practice and Theory*, vol. 16, no. 9, pp. 1254–1265, oct 2008.

[120] F. F. Khorrami, P. Krishnamurthy, and H. H. Melkote, *Modeling and adaptive nonlinear control of electric motors*. Springer, 2003.

[121] M. Bangura, "Aerodynamics and Control of Quadrotors," Ph.D. dissertation, College of Engineering and Computer Science, The Australian National University, 2017.

[122] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "Robot operating system (ros)," *Studies Comp.Intelligence Volume Number:625*, vol. The Complete Reference (Volume 1), no. 978-3-319-26052-5, p. Chapter 23, 2016, iSBN:978-3-319-26052-5.

[123] AscTec, "AscTec Pelican," http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-pelican/, accessed: 2017-12-10.

[124] R. Zanasi and F. Grossi, "Open and closed logarithmic Nyquist plots," *2014 European Control Conference, ECC 2014*, pp. 850–855, 2014.

[125] K. J. Åström and T. Hägglund, *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society, 2006.

[126] C.-T. Chen, *Linear system theory and design*. Oxford University Press, 1999.

[127] A. Megretski, "The Tustin Transform," MIT Department of Electrical Engineering and Computer Science, Tech. Rep., 2004.

[128] R. Hipp, D. Kennedy, and J. Mistachkin, "SQLite," https://www.sqlite.org/index.html, accessed: 2017-12-10.

[129] A. McClung, "rsync_ros," https://github.com/clungzta/rsync_ros, accessed: 2017-12-10.

[130] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3. IEEE, pp. 2149–2154.

[131] Russell Smith, "Open Dynamics Engine v0.5 User Guide," http://ode.org/ode-latest-userguide.html, accessed: 2017-12-10.

[132] OGRE, "OGRE - Open Source 3D Graphics Engine," http://www.ogre3d.org/, accessed: 2017-12-10.

[133] VICON, "VICON Bonita," https://www.vicon.com/products/archived-products/bonita, accessed: 2017-12-10.

[134] Nguyen Khoi Tran, "Modeling and Control of a Quadrotor in a Wind Field," Master thesis, McGill University, Dec. 2015.

[135] AscTec, "Safety Guidelines," http://wiki.asctec.de/display/AR/Safety+Guidelines, accessed: 2017-12-10.

[136] Matternet, "Product," https://mttr.net/product, accessed: 2017-12-10.

[137] M. Hehn and R. D'Andrea, "Quadrocopter Trajectory Generation and Control," *IFAC World Congress*, vol. 18, no. 1, pp. 1485–1491, 2011.

[138] R. Ritz, M. Hehn, S. Lupashin, and R. D'Andrea, "Quadrocopter performance benchmarking using optimal control," *IEEE International Conference on Intelligent Robots and Systems*, pp. 5179–5186, 2011.

[139] H. Oleynikova, R. Baehnemann, M. Popovic, A. Millane, and Z. Taylor, "mav_trajectory_generation," https://github.com/ethz-asl/mav_trajectory_generation, accessed: 2017-12-10.

[140] L. Silverman, "Realization of linear dynamical systems," *IEEE Transactions on Automatic Control*, vol. 16, no. 6, pp. 554–567, 1971.

[141] B. De Schutter, "Minimal state-space realization in linear system theory: an overview," *Journal of Computational and Applied Mathematics*, 2000.

[142] C. Rapp, E. Suez, F. Perrad, C. Liscio, T. Arnold, and G. Krajcovic, "SMC: The State Machine Compiler," accessed: 2017-12-10.

[143] E. Ackerman, "After Mastering Singapore's Streets, NuTonomy's Robo-taxis Are Poised to Take on New Cities," https://spectrum.ieee.org/transportation/self-driving/after-mastering-singapores-streets-nutonomys-robotaxis-are-poised-to-take-on-new-cities, accessed: 2017-12-10.

[144] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa, *Rigorous Software Development*, ser. Undergraduate Topics in Computer Science. London: Springer London, 2011.

[145] M. Sytsma and L. Ukeiley, "Low Order Turbulence Modeling Methods for MAVs Flight Environment," in *AIAA Atmospheric Flight Mechanics Conference*. Reston, Virigina: American Institute of Aeronautics and Astronautics, aug 2010.

[146] Department of Defense, *Department of Defense Handbook Flying Qualities of Piloted Aircraft MIL-HDBK-1797*. Department of Defense, 1997.

[147] J. Holt and S. Perry, *SysML for Systems Engineering : A Model-Based Approach*, 2nd ed. The Institution of Engineering and Technology.

[148] L. Delligatti, *SysML distilled : a brief guide to the systems modeling language*. Addison-Wesley, 2014.

[149] M. Fowler, *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley, 2004.

[150] The Object Management Group, "Unified Modeling Language: Superstructure Version 2.0," *OMG document formal/05-07-04*, no. August, pp. 1–710, 2004.

[151] E. Marder-Eppstein and V. Pradeep, "actionlib," http://wiki.ros.org/actionlib, accessed: 2017-12-10.

[152] M. R. Jardin and E. R. Mueller, "Optimized Measurements of Unmanned-Air-Vehicle Mass Moment of Inertia with a Bifilar Pendulum," *Journal of Aircraft*, vol. 46, no. 3, pp. 763–775, 2009.

[153] A. R. Kim, P. Vivekanandan, P. McNamee, I. Sheppard, A. Blevins, and A. Sizemore, "Dynamic Modeling and Simulation of A Quadcopter with Motor Dynamics," *AIAA Modeling and Simulation Technologies Conference*, no. January, pp. 1–16, 2017.

[154] S. Widnall and J. Peraire, "3D Rigid Body Dynamics: The Inertia Tensor," Massachusetts Institute of Technology, Tech. Rep., 2008.

[155] RCbenchmark, "Series 1580 Thrust Stand and Dynamometer," https://www.rcbenchmark.com/dynamometer-series-1580/, accessed: 2017-12-10.

[156] P. Groves, *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, Second Edition*, 1st ed. Artech House, 2008.

[157] H. E. Rauch, C. T. Striebel, and F. Tung, "Maximum likelihood estimates of linear dynamic systems," *AIAA Journal*, vol. 3, no. 8, pp. 1445–1450, aug 1965.

[158] B. J. Lurie and P. J. Enright, *Classical feedback control with MATLAB and Simulink*. CRC Press, 2012.

[159] M. J. Van Nieuwstadt and R. M. Murray, "Real-time trajectory generation for differentially flat systems," *International Journal of Robust and Nonlinear Control*, vol. 8, no. 11, pp. 995–1020, 1998.

[160] D. Mellinger, "Trajectory generation and control for quadrotors," Ph.D. dissertation, University of Pennsylvania, 2012.

[161] R. S. Smith, "Control Systems 2," ETH Zurich Lecture Slides, 2016.

[162] Google, "glog," https://github.com/google/glog, accessed: 2017-12-10.

[163] Autonomous Systems Lab, "rotors_simulator," accessed: 2017-12-10.

[164] The MathWorks, "Embedded Coder," https://www.mathworks.com/products/embedded-coder.html, accessed: 2017-12-10.

[165] AscTec, "SDK Manual," http://wiki.asctec.de/display/AR/SDK%2BManual, accessed: 2017-12-10.

[166] ——, "Pinout and Connections," http://wiki.asctec.de/display/AR/Pinout+and+Connections, accessed: 2017-12-10.

[167] M. Achtelik and K. D. Hansen, "vicon_bridge," https://github.com/ethz-asl/vicon_bridge, accessed: 2017-12-10.

This page is intentionally left blank.

Appendix A

Generic Helper Functions

This appendix provides several algorithms that are reused throughout the thesis.

Algorithm 42 Modulo operation ($x \bmod y$).
`/general/alure_common/include/alure_common/generic_helper_functions.h:modulo`

function MODULO(x, y)
  assert $y \neq 0$
  $z \leftarrow x - y\lfloor \frac{x}{y} \rfloor$
  return z
end function

Algorithm 43 Convert $\theta \in \mathbb{R}$ to $\theta_{\text{wrapped}} \in [0, 2\pi)$.
`/general/alure_common/include/alure_common/generic_helper_functions.h:wrapToZeroTo2Pi`

function WRAPTOZEROTO2PI(θ)
  $\theta_{\text{wrapped}} \leftarrow$ MODULO($\theta, 2\pi$)
  return θ_{wrapped}
end function

Algorithm 44 Extract yaw in $[0, 2\pi)$ from a quaternion.
`/general/alure_common/include/alure_common/generic_helper_functions.h:convertQuaternionToYaw`

function CONVERTQUATERNIONTOYAW($\boldsymbol{q}$)
  $\boldsymbol{q} \leftarrow \boldsymbol{q}/\|\boldsymbol{q}\|$ ▷ Ensure the quaternion is normalized
  $\psi \leftarrow$ `std::atan2`$(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2))$
  return WRAPTOZEROTO2PI(ψ)
end function

Algorithm 45 Convert yaw to an only-yaw quaternion.
`/general/alure_common/include/alure_common/generic_helper_functions.h:convertYawToQuaternion`

function CONVERTYAWTOQUATERNION(ψ)
  return $(\cos(\frac{\psi}{2}), 0, 0, \sin(\frac{\psi}{2}))$
end function

Algorithm 46 Computation of an error quaternion which rotates vector $\boldsymbol{a}$ into vector $\boldsymbol{b}$. Handles the two singularities for parallel and anti-parallel vectors (i.e. 0 and π error angle respectively).

`/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/controller_math.c:computeErrorQuaternion`

function COMPUTEERRORQUATERNION($\boldsymbol{a}$, $\boldsymbol{b}$)
  $\boldsymbol{q}_\text{e} \leftarrow (1, 0, 0, 0)$
  $c \leftarrow \boldsymbol{a} \cdot \boldsymbol{b}$ ▷ cos(error angle)
  if $c < 1$ **and** $c > -1$ **then** ▷ $\boldsymbol{a}$ and $\boldsymbol{b}$ are not (anti-)parallel
    $\boldsymbol{n} \leftarrow \frac{\boldsymbol{a} \times \boldsymbol{b}}{\|\boldsymbol{a} \times \boldsymbol{b}\|}$
  Compute error quaternion using the half-angle formulae and knowledge that the error angle $\in (0, \pi)$**:**
    $\boldsymbol{q}_\text{e} \leftarrow (\frac{1}{\sqrt{2}}\sqrt{1 + c}, \frac{\boldsymbol{n}}{\sqrt{2}}\sqrt{1 - c})$
  end if
  Compute error quaternion for the case of π **error angle:**
  if $c = -1$ **then** ▷ $\boldsymbol{a}$ and $\boldsymbol{b}$ are anti-parallel
    $\boldsymbol{n} \leftarrow (0, 0, 0)$ ▷ Populate $\boldsymbol{n}$ s.t. $\boldsymbol{n} \perp \boldsymbol{a}$ by construction
    if $\boldsymbol{a}[1] \neq 0$ **then**
      $\boldsymbol{n}[2] \leftarrow \boldsymbol{a}[1]$
      $\boldsymbol{n}[1] \leftarrow -\boldsymbol{a}[2]$
    else if $\boldsymbol{a}[2] \neq 0$ **then**
      $\boldsymbol{n}[3] \leftarrow \boldsymbol{a}[2]$
      $\boldsymbol{n}[2] \leftarrow -\boldsymbol{a}[3]$
    else ▷ $\boldsymbol{a}[3] \neq 0$
      $\boldsymbol{n}[1] \leftarrow \boldsymbol{a}[3]$
      $\boldsymbol{n}[3] \leftarrow -\boldsymbol{a}[1]$
    end if
    $\boldsymbol{n} \leftarrow \frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}$
    $\boldsymbol{q}_\text{e} \leftarrow (0, \boldsymbol{n})$
  end if
  return $\boldsymbol{q}_\text{e}$
end function

Appendix B

State Machine Basics

A *state machine* is a program which implements state-based behavior. State machines are a powerful tool for developing systems whose behavior is related to the system being in a specific state.
In its simplest form, a Finite State Machine (FSM) is a program which can be in exactly one of a finite number of states at any given time. It is defined by a set of states, state transition conditions and an initial state. *States* describe what is happening within a system while *transitions* define the possible paths between the states Holt and Perry [147].
The Unified Modeling Language (UML) extends the FSM with advanced features to create a *UML state machine*. The language defines a *state machine diagram* which is a precise and unambiguous specification of the UML state machine. While this complexity is not needed for the autonomy engine in Chapter 6, the UML state machine diagram offers a notational super-set over FSM diagrams which was found to be useful for development. It also means that the autonomy engine can be ported to rigorous UML-based state machine frameworks as discussed in Section 8.2. This appendix:

1. Introduces the UML state machine diagram notation utilized in Chapter 6;

2. Explains how state machines are implemented in this thesis.

The text below shall use the standard Linux command usage pattern (<obligatory>, [optional] and * for repeating) for specifying diagram label structure.

B.1 UML State Machine Diagram

A *state* defines the current system status. In a "flat" (single level) state machine like the one implemented in this thesis, exactly one state may be active at any given time. While UML defines several state types, the implementation in this thesis concerns itself only with a *simple state*. As shown in Figure B.1, a simple state has a name and up to three *internal behaviors*:

1. The *entry* behavior is always the first behavior to be executed upon entering the state;

2. The *exit* behavior is always the last behavior to be executed upon exiting the state;

3. The *do* behavior begins executing immediately after the state entry behavior.

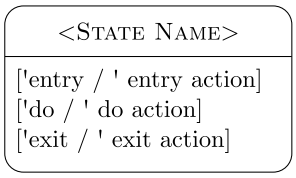


Figure B.1: A simple state and its three internal behaviors.

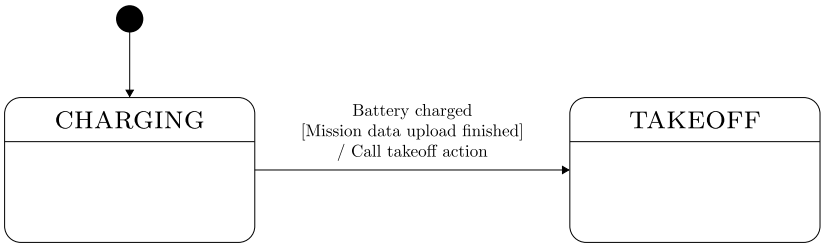


Figure B.2: Transition example with a trigger event (Battery charged), a guard (Mission data upload finished) and an effect (Call takeoff action). CHARGING is indicated to be the initial state.

This thesis does not implement the entry and exit behaviors. Furthermore, the do behavior is guaranteed to finish executing for a given iteration before a transition is made to the next state (unlike in UML, where the do behavior is interruptible).
A *transition* represents a change of state. It is drawn as an arrow from the source to the target state and has three optional properties: a set of triggers, one guard and one effect. These are written in a string next to the transition arrow:

transition	:=	[triggers] [guard] ['/' effect]
triggers	:=	trigger [',' trigger]*
guard	:=	'[' constraint ']'

Triggers are a set of one or more events chained via **or** logic such that, when the set evaluates to **true** (i.e. at least one of the events occurs), a transition is attempted. A *guard* is a Boolean expression which, when **false**, is the only element capable of preventing a transition when triggers evaluate to **true**. Finally, an *effect* specifies a possible action (which can encapsulate several actions) that gets performed when the transition occurs. A transition with no trigger and no guard occurs immediately after executing the entry behavior of its source state. Our implementation allows to specify all three transition properties.
Figure B.2 shows a typical transition example. It also introduces a new element illustrated by the black circle, the *initial pseudostate*. The transition from it is used to indicate the first state machine state. A transition with an initial pseudostate target is not allowed.
UML supports a feature called *event deferring*. Deferred events are listed in the state body as follows:

deferred events	:=	[events '/ defer']
events	:=	event [',' event]*

When an event occurs in a state which has the event listed in its deferred events list, it will be preserved for later. When a state is reached where the event is not deferred, it will be invoked. This thesis implements slightly different mechanics. Events can have a deferred "type"[1] such that they are only cleared when a transition checks

[1] `/general/alure_common/include/alure_common/core_state_machine.h`

them. Optionally, a deferred-type event can be checked and then deferred – meaning that the next state to check this event will see that it has been invoked. This is used e.g. to implement the pause functionality in the master state machine (Section 6.6). In summary, a state machine is composed of an initial pseudostate, one or more simple states and transitions between these. Each state may furthermore have associated with it a do action which executes for as long as the simple state is active. This is as much knowledge as needed to understand the autonomy engine state machines and Figures B.1 and B.2 provide the required notation. There is much more depth to UML state machines which the interested reader may consult in Delligatti [148], Fowler [149], The Object Management Group [150].

B.2 State Machine Implementation

This section provides a global overview of the state machine implementation for the autonomy engine described in Chapter 6. The goal is to help the reader to more easily understand the source code structure rather than to give a complete but unnecessary implementation formalization. For a complete picture, the interested reader is advised to explore the provided source code links within this section.
The core low-level state machine functionality is implemented as the base class CoreStateMachine[2]. This class spawns a parallel thread which repeatedly executes at frequency f_{sm} a user-defined stateMachineDefinition() **which defines the state machine**. The following functionality is provided by this class as illustrated in Figure B.3:

- The state machine can be run via play(), can be externally paused via pause() and can pause itself via selfPause(). A paused state machine consumes no computational resources (via passive wait). A running state machine handles events in busy-wait fashion (i.e. checks their occurrence at f_{sm} Hz);
- Event deferring as described in Section B.1 via deferEvent();
- The stateMachineDefinitionWrapper() automatically publishes the target state to the ROS network during a state transition.

While the CoreStateMachine provides all the necessary functionality for creating a basic state machine, it lacks the functionality to be controlled by external logic running in a ROS environment. Therefore two more base classes, ServiceBasedStateMachine[3] and ActionBasedStateMachine[4], extend the CoreStateMachine functionality to allow control via a ROS service and a ROS action respectively. The inheritance relationship is shown in Figure B.4.
The ActionBasedStateMachine allows to control the state machine from other ROS nodes via a ROS action Marder-Eppstein and Pradeep [151]. This is useful when it is desired to know that the functionality provided by the state machine has run to completion. For example, in Chapter 6 the master state machine needs to know when the takeoff phase is finished. Furthermore, an ActionBasedStateMachine can be preempted which is used by e.g. the autonomy engine to abort the mission phase in case of emergency landing (see Figure 6.17).
The ServiceBasedStateMachine allows to interact with the state machine from other ROS nodes via a ROS service. This is useful when it is desired simply to start or to stop the state machine without caring for when it finishes. This is used by

[2] /general/alure_common/include/alure_common/core_state_machine.h
[3] /general/alure_common/include/alure_common/service_based_state_machine.h
[4] /general/alure_common/include/alure_common/action_based_state_machine.h

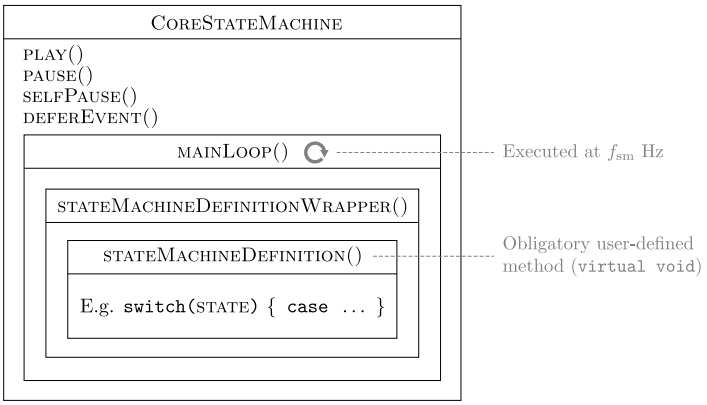


Figure B.3: Core state machine illustration. This diagram highlights the repeatedly executed mainLoop() and the location of the user-defined stateMachineDefinition() within it.

e.g. the master state machine which the user runs once and, after an arbitrary time period, may pause or stop.
The autonomy engine and guidance code implement a total of 11 state machines whose inheritance structure is shown in Figure B.4. The choice of which base class to inherit from is systematized as follows:

- State machines which execute an action that requires feedback on completion inherit from ActionBasedStateMachine;
- State machines which execute an action with no completion feedback requirement inherit from ServiceBasedStateMachine;
- "Slave" state machines (i.e. those that execute as members of higher-level classes) inherit directly from CoreStateMachine.

Listing 56 outlines the typical structure of a state machine child class.

```
// my_state_machine.h

#include "alure_common/core_state_machine.h"

// Inheriting from ActionBasedStateMachine or ServiceBasedStateMachine
// is identical, just include the corresponding header file
class MyStateMachine : public CoreStateMachine
{
 private:
  // ** State machine specifics ************************************
  enum MyState
  {
    STATE_1 = 0,
    STATE_2,
    ...
  };
  void stateMachineDefinition();
  // ***************************************************************
 private:
  ...
 public:
  MyStateMachine();
  ...
};
```

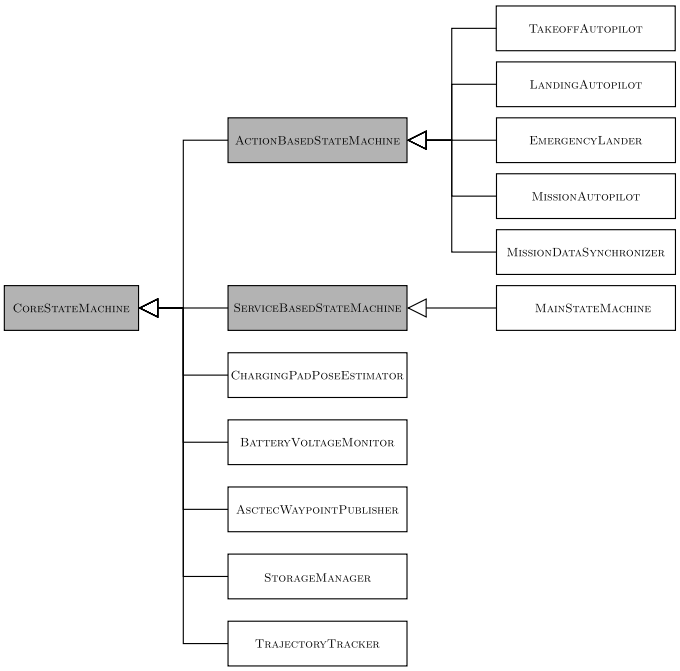


Figure B.4: State machine inheritance. Standard UML class diagram notation is used. Gray highlights the state machine base classes.

```
// my_state_machine.cpp

#include "my_state_machine.h"

MyStateMachine::MyStateMachine ()
{
  UPDATE_STATE(STATE_1); // Sets the initial state
  ...
}

void MyStateMachine::stateMachineDefinition ()
{
  switch (state_)
  {
    case STATE_1:
      {
        // Do action
        if (/* trigger list */ && /* guard */)
        {
          UPDATE_STATE(STATE_2);
          // Transition effect
        }
        // More transitions in other if statements
        break;
      }
    case STATE_2:
      ...
    QUIT_ON_BAD_STATE(); // Exit program on unknown state
  }
}
```

Listing B.1: Main elements of a fictitious MyStateMachine state machine. This sketches the essential structure a typical state machine's source code.

This page is intentionally left blank.

Appendix C

AprilTag Measurement Noise

This appendix analyzes the measurement noise of our AprilTag based bundle pose sensor presented in Section 3.1.3. The goal is to determine what tag layout constitutes an ideal tag bundle with the following properties:

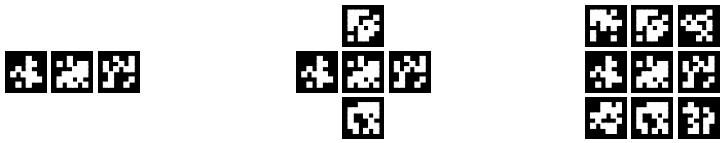
- It is positioned outside of the landing pad such that it does not obscure the charging area;
- It has a large detection range while being reasonably small with respect to the landing pad size;
- It provides reasonable measurement accuracy at large distances which improves to sub-centimeter accuracy for small distances. This allows a naturally coarse landing approach at large heights which refines to pinpoint-accuracy before touchdown.

C.1 Data Collection Method

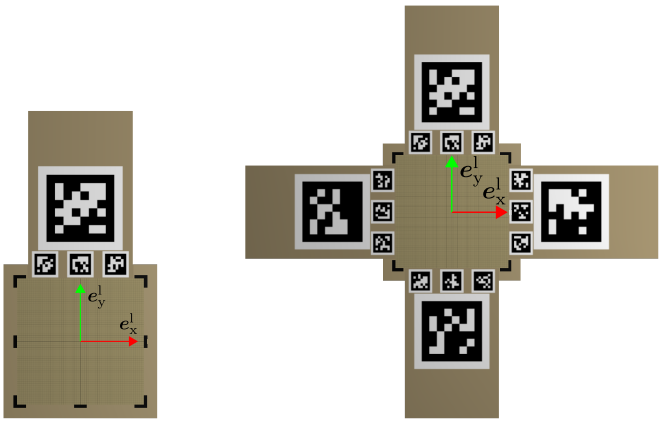
We carry out our analysis in the Gazebo simulation environment. The reason for doing so is that measurement noise analysis is done primarily for good bundle design. A real test suffers from a multitude of uncontrolled variables such as the external lighting, motion of the quadrotor and thus motion blur and the unavailability of a large enough space with available ground truth for testing the entire detection range of the larger tags (the available VICON area was a $\approx 3 \times 3 \times 3\,\mathrm{m}$ cube). Measurement noise will vary with lighting conditions, camera calibration quality, vibration amount due to spinning motors, etc. As a result, accurate characterization of real bundle pose measurement noise is a difficult if not impossible task. The simulation environment, meanwhile, is a clean way to extract fundamental properties about good bundle design because it allows to have a single independent variable and to control all other variables.
A simulated OpenGL pinhole camera is used with a $752 \times 480\,\mathrm{px}$ resolution, a field of view of $2\,\mathrm{rad}$ (similar to the real mvBlueFOX [21] downfacing camera) and an additive zero-mean Gaussian noise with a standard deviation of 0.003[1] added independently to each pixel. The latter is used to introduce some level of variability that is present in the real image due to lighting, electronic noise, etc.
Bundle pose measurement noise data is collected for bundles shown in Figure C.1. The data collection procedure is to place the bundle such that the {l} frame overlays the {w} frame. Then, the camera is placed at $(0, 0, 0.4)\,\mathrm{m}$ in the {w} frame and is moved up along $\boldsymbol{e}_z^w$ at $5\,\mathrm{mm/s}$. This z-movement of the camera is the experiment's

[1] This value is with respect to the pixel's grayscale value, which is in $\in [0, 1]$.

(—+—) 100 mm. (—●—) 150 mm. (—■—) 180 mm. (—×—) 250 mm. (—♦—) 400 mm.



(—+—) 150mm bundle of 3. (—●—) 150mm bundle of 5. (—■—) 150mm bundle of 9.



(—♦—) Used landing bundle. (—×—) Improved landing bundle.

Figure C.1: Bundles for simulated measurement noise data collection. The last two landing bundles consist of 480mm and 150mm tags.

only independent variable, thus we obtain data about bundle measurement noise as a function of $p^c_{l,z}$. Two reasons motivate choosing z-movement for the noise analysis:

1. The variable that changes most during a landing is the quadrotor height – and consequently $p^c_{l,z}$;

2. Introducing more independent variables would lead to a combinatorial explosion that drastically increases the testing time required to cover the independent variables' multidimensional domain. The decision-making value of the results will likely also suffer from a strong case of diminishing returns.

C.2 Measurement Noise Definition

Consider a generic variable with an actual value x and a measured value $\hat{x}$. Define the *measurement error* for measurement i as:

$$e_x[i] := \hat{x}[i] - x[i]. \tag{C.1}$$

Consider now a sample of measurements $\mathcal{M}$. We define the usual measurement noise mean and standard deviation:

$$\mu_x := \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} e_x[i], \tag{C.2}$$

$$\sigma_x := \sqrt{\frac{1}{|\mathcal{M}| - 1} \sum_{i \in \mathcal{M}} (e_x[i] - \mu_x)^2}. \tag{C.3}$$

In our analysis, $\mathcal{M}$ is obtained as the set of bundle pose measurements corresponding to a range of ground-truth camera heights above the bundle, $p^{\mathrm{c}}_{\mathrm{l,z}}$, binned into 0.4 m increments.

C.3 Noise Analysis Results

Let us retain the {l} frame notation for the bundle frame. We collect raw bundle pose measurement data (a set of $\hat{\boldsymbol{p}}^{\mathrm{l}}_{\mathrm{c}}$ and $\hat{\boldsymbol{q}}^{\mathrm{l}}_{\mathrm{c}}$) using the method of Section C.1. We then post-process the raw data by applying (C.2) and (C.3) to $p^{\mathrm{l}}_{\mathrm{c,x}}$, $p^{\mathrm{l}}_{\mathrm{c,y}}$, $p^{\mathrm{l}}_{\mathrm{c,z}}$ and $\psi_{\mathrm{l}} :=$ convertQuaternionToYaw($\boldsymbol{q}^{\mathrm{l}}_{\mathrm{c}}$). These are the only variables of interest as they are used for visual landing navigation (see Section 3.1).
The results are plotted in Figures C.3 and C.4 for the bundles of Figure C.1. The following sections discuss and draw conclusions from these results.

C.3.1 Noise In $p^{\mathrm{l}}_{\mathrm{c,x}}$ And $p^{\mathrm{l}}_{\mathrm{c,y}}$

Consider the $p^{\mathrm{l}}_{\mathrm{c,x}}$ and $p^{\mathrm{l}}_{\mathrm{c,y}}$ measurement noise. For all bundles except the used landing bundle (red markers), adding more tags does not significantly improve measurement accuracy. As expected, the mean noise is ≈ 0 since the measurement is unbiased (e.g. $\hat{p}^{\mathrm{l}}_{\mathrm{c,x}}$ is equally likely to under- and to over-estimate $p^{\mathrm{l}}_{\mathrm{c,x}}$). The standard deviation appears to slightly decrease for bundles composed of more tags, but this is not significant since the standard deviation is already sub-millimeter.
Curiously, the used landing bundle (red curve) is the worst in accuracy and in precision. This is due to a *lever arm effect* as, unlike all other bundles, the used landing bundle has all tags offset in the $+\boldsymbol{e}^{\mathrm{l}}_{\mathrm{y}}$ direction from the bundle origin (see Figure C.1). To gain intuition about the lever arm effect, consider a single tag displaced from the origin by a distance l. A small angular error θ in this tag's attitude measurement causes the origin to displace by $\approx l\theta$, i.e. l introduces an *error gain*. Similarly for a bundle, it means that **bundles that are not homogeneously distributed around their origin have, on average, a higher translation measurement error**. Define a *balanced bundle* as one that has a radially symmetric tag distribution about its origin. All bundles in Figure C.1 are balanced except for the used landing bundle. For precision landing, it is desirable to have a balanced bundle. Our good landing accuracy (Section 7.6) despite an unbalanced landing bundle is testimony to our landing method's robustness.

C.3.2 Noise In $p^{\mathrm{l}}_{\mathrm{c,z}}$

Consider the $p^{\mathrm{l}}_{\mathrm{c,z}}$ measurement noise. In this case, a larger tag decreases measurement noise (as seen for the top row single-tag bundles of Figure C.1). Importantly, the detection distance of larger tags is larger (limited in simulation only by the distance at which the 6×6 bit tag payload becomes smaller than 6×6 px in the image). Measurement noise for bundles in row 2 of Figure C.1 is smaller than for the single-tag bundles, but curiously the 5- and 9-tag bundles do not reduce noise with respect to the 3-tag bundle. The used, unbalanced, landing bundle again has a poor measurement accuracy due to the aforementioned lever arm effect. The alternative,

balanced, landing bundle is the most accurate with $< 2\,\text{cm}$ standard deviation at up to 7 m distance (where its larger 450 mm tags are the only ones visible and are a mere 15 px in side length in the image).

C.3.3 Noise in ψ_l

Consider the ψ_l measurement noise. In this case, a balanced tag layout makes no difference because the yaw measurement does not suffer from the same lever arm effect. It appears that yaw measurement becomes more accurate with more tags in the bundle, as μ_{ψ_l} and σ_{ψ_l} are smallest for the alternative landing bundle. With more tag corner coordinates entering the PnP problem, it is intuitive that yaw would be more accurately measured. Note however that this accuracy is not required and even a single tag suffices, since yaw measurement accuracy is sub-degree (i.e. more accurate than yaw control).

C.3.4 Summary

The main results of the simulated noise analysis are:

- $p^\text{l}_{\text{c,x}}$ and $p^\text{l}_{\text{c,y}}$ measurement accuracy is not improved by using more tags, but is greatly worsened for unbalanced tag layouts;
- $p^\text{l}_{\text{c,z}}$ measurement accuracy is improved by using more tags and is greatly worsened for unbalanced tag layouts;
- ψ_l measurement accuracy is not affected by an unbalanced tag layout and is improved by using more tags.

An ideal bundle consists of multiple tags arranged in a balanced fashion about the bundle's origin. The alternative landing pad bundle (last in Figure C.1) is such an ideal bundle. The advantages of the alternative landing bundle are the following:

- Its detection range is up to 14 m (see Figure C.2);
- Large 450 mm tags in combination with small 150 mm tags make for a smooth transition during the landing descent from 450 mm tags that are only visible from far away to 150 mm tags that are only visible up close. During the final portions of descent, bundle pose measurement accuracy becomes sub-millimeter and sub-degree;
- The bundle does not obscure the charging area.

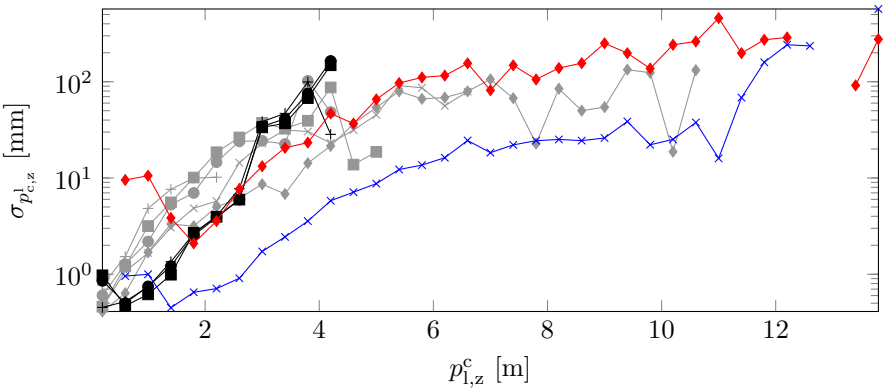


Figure C.2: $\sigma_{p^\text{l}_{\text{c,z}}}$ for the full detection range of the simulated bundles.

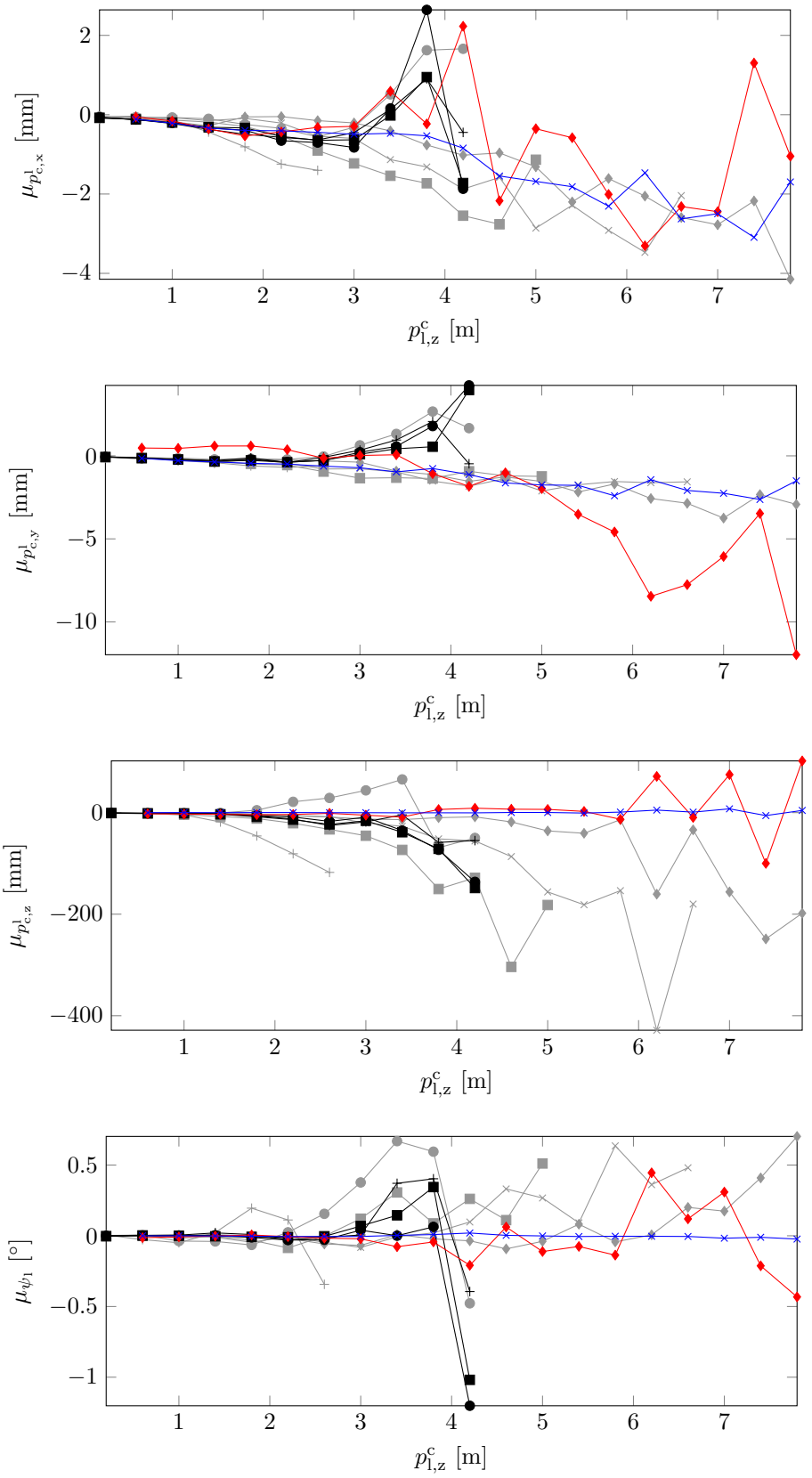


Figure C.3: Bundle pose measurement noise mean.

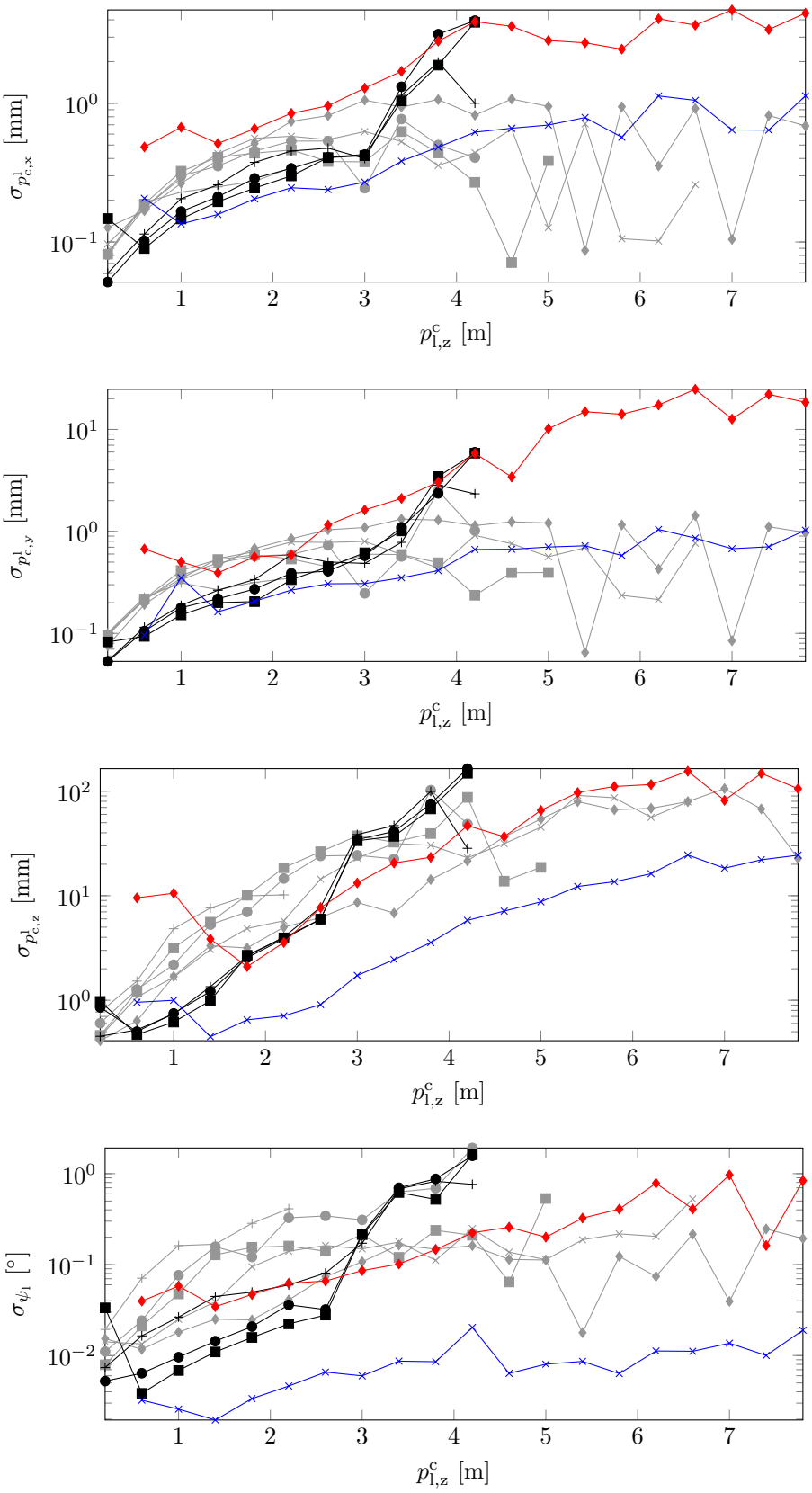


Figure C.4: Bundle pose measurement noise standard deviation.

Appendix D

System Identification

This appendix identifies the simplified flight dynamics model (5.30) parameters for the AscTec Pelican MAV. These parameters are m, l_{arm}, J, $\tau_{\text{m,up}}$, $\tau_{\text{m,dn}}$, $\{k_{\text{f},2}, k_{\text{f},1}, k_{\text{f},0}\}$ and $\{k_{\eta,2}, k_{\eta,1}, k_{\eta,0}\}$. Their final values are listed in Table 5.1 while this appendix describes the methods behind their identification.

D.1 Simple Parameters

Using a digital scale, $\hat{m} = 1.956\,\text{kg}$. Using a ruler, the rotor-to-rotor distance is measured and halved, giving $l_{\text{arm}} = 0.21\,\text{m}$.

D.2 Inertia Tensor

The inertia tensor J is estimated using a bifilar pendulum. The generic setup for a bifilar pendulum test is shown in Figure D.1. The aim of the test is to measure the 1 dimensional moment of inertia J of the test object about the $\boldsymbol{e}_{\text{z}}^{\text{w}}$ axis. By suspending the test object in different orientations, the moment of inertia about different body axes may be measured. The measurement procedure is as follows:

1. (**Calibration Step**) Determine the pendulum length L via a simple pendulum test, as shown in Figure D.2a. Provide a small angular displacement θ and release to let the setup oscillate. Take 5 measurements of the time taken for 5 periods, divide each measurement by 5 and take their median to obtain the simple pendulum's period estimate, $\tilde{T}_{\text{simple}}$. The length estimate $\tilde{L}$ is then given by the simple pendulum equation:

$$\tilde{L} = \|\boldsymbol{g}\| \left(\frac{\tilde{T}_{\text{simple}}}{2\pi} \right)^2 . \tag{D.1}$$

2. (**Measurement Step**) Determine J via a bifilar pendulum test, as shown in Figure D.2b. Provide a small angular displacement θ about $\boldsymbol{e}_{\text{z}}^{\text{w}}$ and release to let the setup oscillate. Take 20 measurements of the time taken for 10 periods, divide each measurement by 10 and take their median to obtain the bifilar pendulum's period estimate, $\tilde{T}_{\text{bifilar}}$. The moment of inertia estimate $\tilde{J}$ about $\boldsymbol{e}_{\text{z}}^{\text{w}}$ is then given by:

$$\tilde{J} = \frac{m\|\boldsymbol{g}\|\tilde{T}_{\text{bifilar}}^2 b^2}{4\pi^2 \tilde{L}}. \tag{D.2}$$

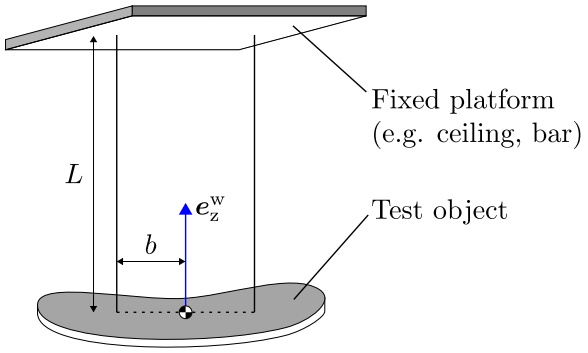
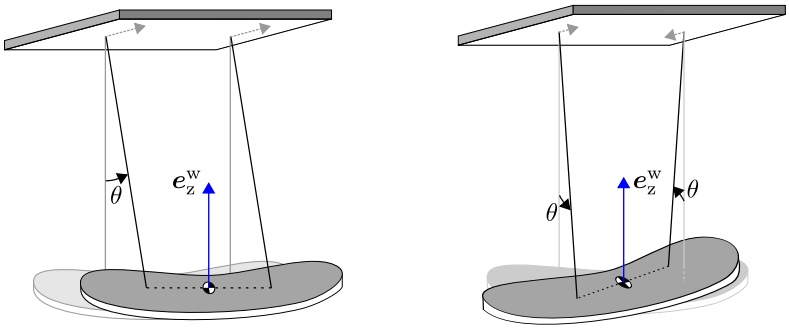


Figure D.1: Bifilar pendulum model as used for moment of inertia measurement.



(a) Initial displacement for the calibration step ("simple pendulum test).

(b) Initial displacement for the measurement step ("bifilar pendulum test").

Figure D.2: Initial displacements for the simple and bifilar pendulum tests. Dashed arrows on the ceiling are projections of the two pendulum wires that help to visualize their displacement.

Formula (D.2) makes the following assumptions[1] that simplify the J estimation problem from a nonlinear regression problem to a simple equation (D.2) Jardin and Mueller [152]:

1. The oscillations are undamped (in reality, there is damping from aerodynamic drag, viscous friction and friction in the wire attachment points);

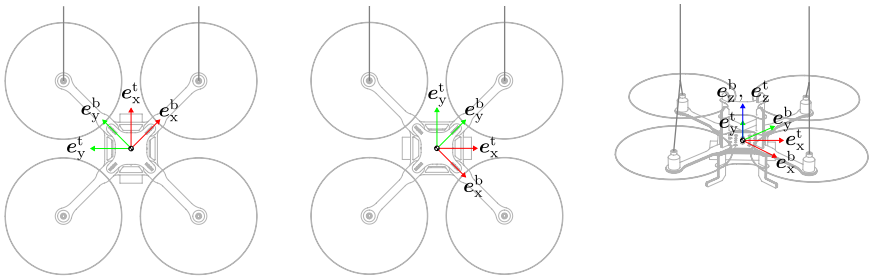
2. The pendulum wires are rigid (strain dynamics are neglected), which is reasonable due to L being much greater than the expected strain displacement and the strain dynamics being much faster than those of the pendulum;

3. The wires are massless and offer no bending or torsional resistance;

4. There is no influence by the surrounding air (beyond aerodynamic drag, this means that entrained air momentum effects are ignored, which leads to overestimating J by not removing the "air mass" which moves with the test object Jardin and Mueller [152])[2];

[1] More complex formulae exist which remove assumptions 1, 3 and 4, but make the inertia estimation process more involved than was deemed necessary for our case. The interested reader is directed to Jardin and Mueller [152].

[2] This effect is negligible for objects with small surface area, such as the quadrotor.



(a) Theoretical J'_{xx} setup. (b) Theoretical J'_{yy} setup. (c) Theoretical J'_{zz} setup.

(d) Actual J'_{xx} setup. (e) Actual J'_{yy} setup. (f) Actual J'_{zz} setup.

Figure D.3: Practical (top) and actual (bottom) setups of a bifilar pendulum for moment of inertia measurement. A spirit level tool is used to ensure proper alignment in the real setup.

5. Angular motion is small;
6. The wires are parallel at rest;
7. The test object's center of mass lies on the midpoint of the edge joining the two wires' attachment points, with a possible offset along $\boldsymbol{e}_z^w$. The edge is assumed to itself be contained in the $(\boldsymbol{e}_x^w, \boldsymbol{e}_y^w)$ plane;
8. Axis $\boldsymbol{e}_z^w$ is aligned with a principal axis of the test object (otherwise, precession occurs). This effect is small – a misalignment of 10° theoretically yields only a 1.5 % measurement error Jardin and Mueller [152];

As shown in Figure D.3, we adopt the method of Kim et al. [153] for estimating the quadrotor's moment of inertia J. Notably, the AscTec Pelican's geometry makes it easier to estimate the inertia tensor J' of the $\{t\}$ *test frame* which is yawed with respect to the $\{b\}$ frame by 45° about $\boldsymbol{e}_z^b$:

$$C_{(\boldsymbol{q}_b^t)} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{D.3}$$

Figure D.6 shows the result of applying the measurement procedure to moment of inertia estimation about the $\boldsymbol{e}_x^t$, $\boldsymbol{e}_y^t$ and $\boldsymbol{e}_z^t$ axes. By assuming a lognormal distribution[3] of T_{simple} and T_{bifilar} in (D.1) and (D.2), we are able to compute the mean and standard deviation of each moment of inertia. The mean plus/minus one standard deviation are also plotted in Figure D.6 and tabulated in Table D.1.

[3]The lognormal distribution is well suited for modeling positive uncertain quantities, such as time in this case.

Variable	Mean [m]	Standard Deviation [m]
$\tilde{L}'_{\mathrm{x}}$	2.74	0.06
$\tilde{L}'_{\mathrm{y}}$	2.73	0.03
$\tilde{L}'_{\mathrm{z}}$	2.82	0.05

(a) Simple pendulum test results for wire length.

Variable	Mean [N·m]	Standard Deviation [N·m]
$\tilde{J}'_{\mathrm{xx}}$	$1.41 \cdot 10^{-2}$	$3.18 \cdot 10^{-4}$
$\tilde{J}'_{\mathrm{yy}}$	$1.33 \cdot 10^{-2}$	$1.62 \cdot 10^{-4}$
$\tilde{J}'_{\mathrm{zz}}$	$1.64 \cdot 10^{-2}$	$3.25 \cdot 10^{-4}$

(b) Bifilar pendulum test results for moment of inertia.

Table D.1: Moment of inertia estimation results for the {t} frame.

We then use the inertia tensor frame transformation formula to obtain the inertia tensor in the {b} frame Widnall and Peraire [154]:

$$\tilde{J} = C_{(\boldsymbol{q}^{\mathrm{t}}_{\mathrm{b}})} \tilde{J}' C^{\mathsf{T}}_{(\boldsymbol{q}^{\mathrm{t}}_{\mathrm{b}})}. \tag{D.4}$$

By applying (D.3) and (D.4) to the data in Table D.1, our AscTec Pelican quadrotor is estimated to have the following inertia tensor:

$$\tilde{J} = \begin{bmatrix} 1.37 \cdot 10^{-2} & 4.35 \cdot 10^{-4} & 0 \\ 4.35 \cdot 10^{-4} & 1.37 \cdot 10^{-2} & 0 \\ 0 & 0 & 1.64 \cdot 10^{-2} \end{bmatrix}. \tag{D.5}$$

D.3 Motor Thrust and Torque Maps

The motor thrust (5.25) and torque (5.26) map coefficients are identified on an RCbenchmark [155] load cell test stand. The setup is pictured in Figure D.4 and the corresponding signal flow diagram is shown in Figure D.5. A testing script was provided by the JPL test stand maintainers[4] while we wrote an Arduino UNO[5] and an Odroid XU4[6] script in order to interface our system to the test stand.
The testing procedure is to run motor i from $u_{\mathrm{esc},i} = 20$ to $u_{\mathrm{esc},i} = 200$ in increments of 10. Each value is held for 2 seconds before moving on to the next one, such that the map is built only from the steady-state thrust and torque components of each interval.
Raw data shown in Figure D.7 exhibits thrust and torque saturation past $u_{\mathrm{esc},i} = 180$, meaning that the motor does not spin faster past this ESC value. Presumably this is due to an $\approx 18\,\mathrm{A}$ current limit in the ESC or in the AscTec LLP or PowerBoard since the utilized Zippy 6200 mAh 40C LiPo battery [29] can deliver up to 248 A continuous current. As a result, an ESC range of 20-180 is taken for the map fit and shall consequently also be the operating range for the body-rate controller output in Section 5.6.
The full quadratic maps (5.25) and (5.26) are fit using MATLAB's `polyfit` routine which performs a least-squares regression[7,8]. Similarly, a pure quadratic is

[4] `/load_cell_test_tools/scripts/ContinuousStep.js`
[5] `/load_cell_test_tools/arduino/pwm_read/pwm_read.ino`
[6] `/load_cell_test_tools/scripts/time_constant_step_signal.py`
[7] `/scripts/motor_thrust_torque_id/thrust_model_id.m`
[8] `/scripts/motor_thrust_torque_id/torque_model_id.m`

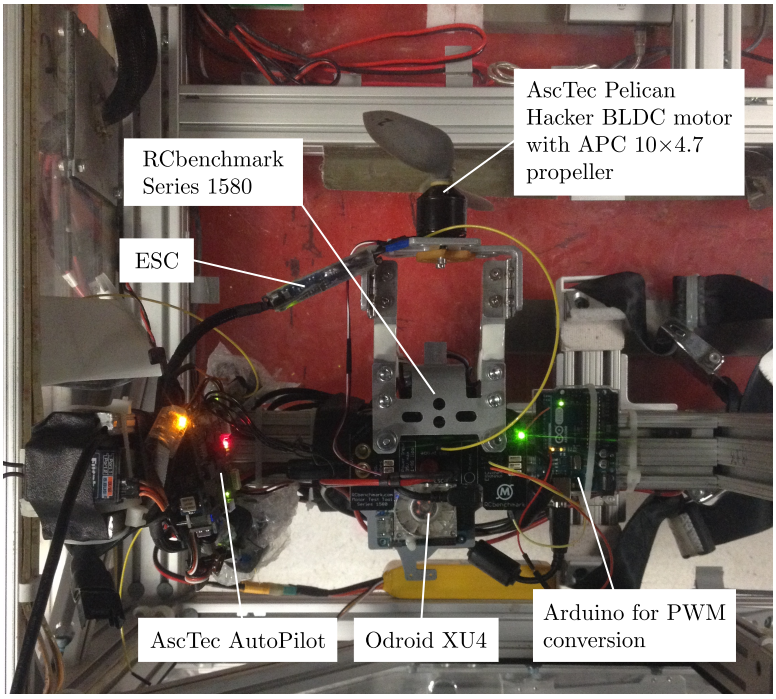


Figure D.4: Setup for motor thrust and torque map identification.

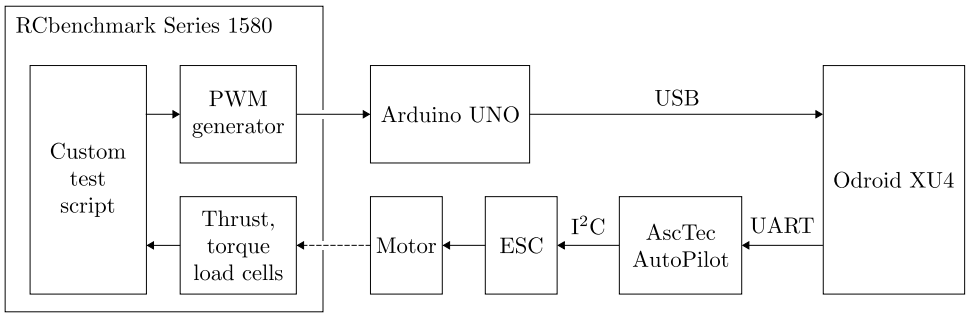


Figure D.5: Information flow diagram for the setup in Figure D.4. The dashed signal line indicates an implicit link – the load cells measure the thrust and torque generated by the motor as a result of the ESC command.

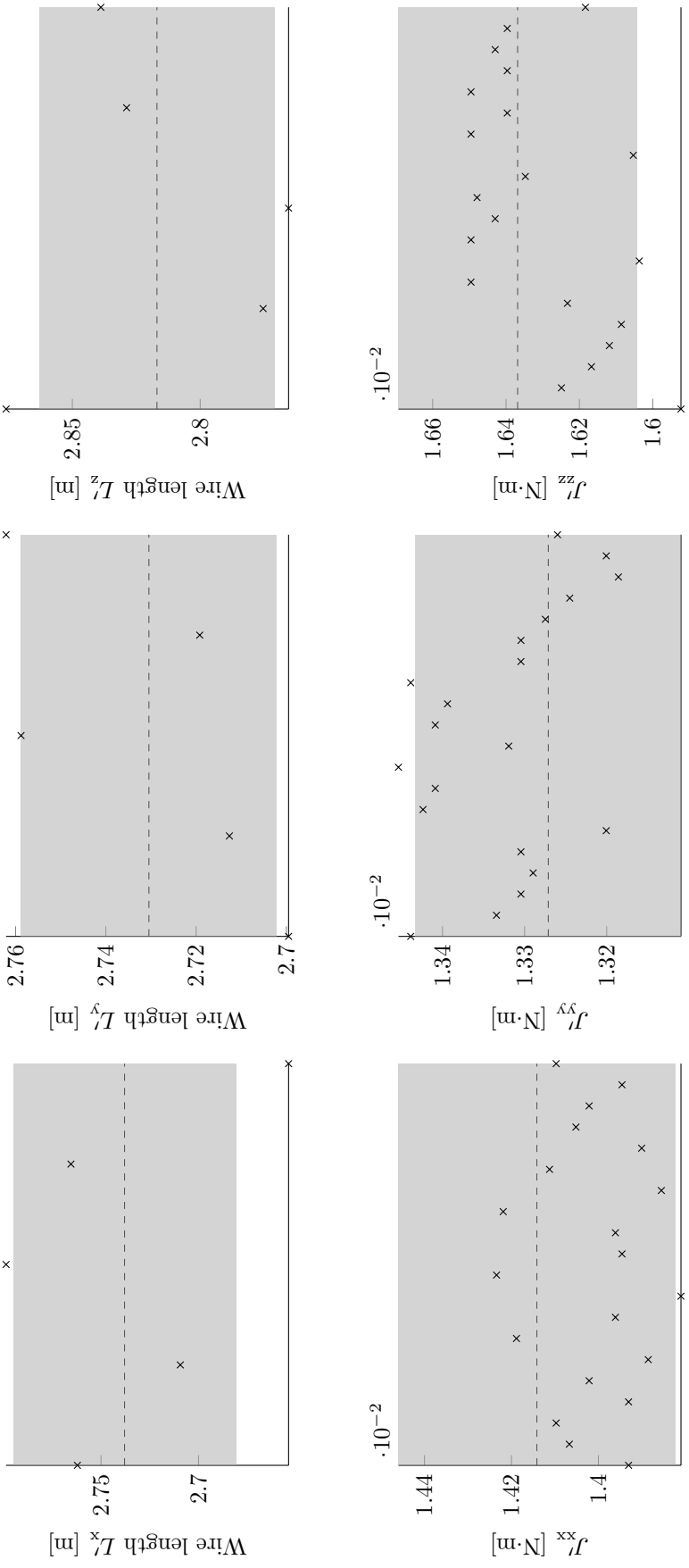


Figure D.6: Moment of inertia estimation along the {t} frame axes. Crosses show raw output of (D.1) and (D.2) while the dotted line and shaded area represent the mean plus minus one standard deviation as obtained from a lognormal distribution assumption about the period samples $\hat{T}_{\text{simple}}$ and $\hat{T}_{\text{bifilar}}$.

fit for comparison by solving the following least-squares problem using MATLAB's `fminsearch` routine[9]:

$$\underset{k_{\text{pure}}}{\text{minimize}} \quad \sum_{k=1}^{M} \left(k_{\text{pure}} u_{\text{esc},i}[k]^2 - y[k]\right)^2 \quad , \tag{D.6}$$

where k_{pure} is the pure quadratic's coefficient, M is the total number of measurements and y is $\hat{\bar{f}}_i$ for thrust regression and $\hat{\bar{\eta}}_i$ for torque regression[10]. Least squares is a subclass of convex optimization and thus (D.6) has a global optimum. Therefore, the initial value of k_{pure} does not matter and is set to zero for `fminsearch`[11].
Figure D.7 shows the resulting fits and Table D.2 provides the numerical values. We note that:

- The full quadratic fits the data much better than the pure quadratic, thus motivating the use of this more complicated model (this result agrees with Faessler et al. [107]);

- The individual motor fits are, as expected, slightly different from the "all-motor" fit which uses all the data at once. This is due to electrical and mechanical differences in the motor/propeller combination as well as measurement noise and potential load cell alignment errors (although a calibration was performed prior to each test). These differences motivate the thrust calibration procedure described in Section 5.6.4;

- As noted in Bangura [121] and as confirmed during load cell tests, a decreasing battery voltage leads to a decreasing $\omega_{\text{p},i}$ for a given $u_{\text{esc},i}$ in the upper ESC command ranges. This leads to a further dependence on the integrator in the position/velocity controllers of Section 5.8 and is undesirable. An improvement to use closed loop $\omega_{\text{p},i}$ control is suggested in Section 8.2.

- The thrust and torque maps are valid only for the AscTec Pelican's default Hacker motor and APC 10×4.7 propeller combination. A manual test applying load torque with one's fingers to the motors when they are spinning showed that they can be stopped easily – suggesting that the motors are driven open-loop by the ESCs (i.e. they are torque-controlled rather than RPM-controlled, where in the latter case the expected behavior would be a torque resistance to being stopped). Without feedback, the ESCs would have to assume a certain load torque on the motors in order to drive them to a precise RPM. Using a different motor or a different propeller would invalidate the ESCs' load torque assumption.

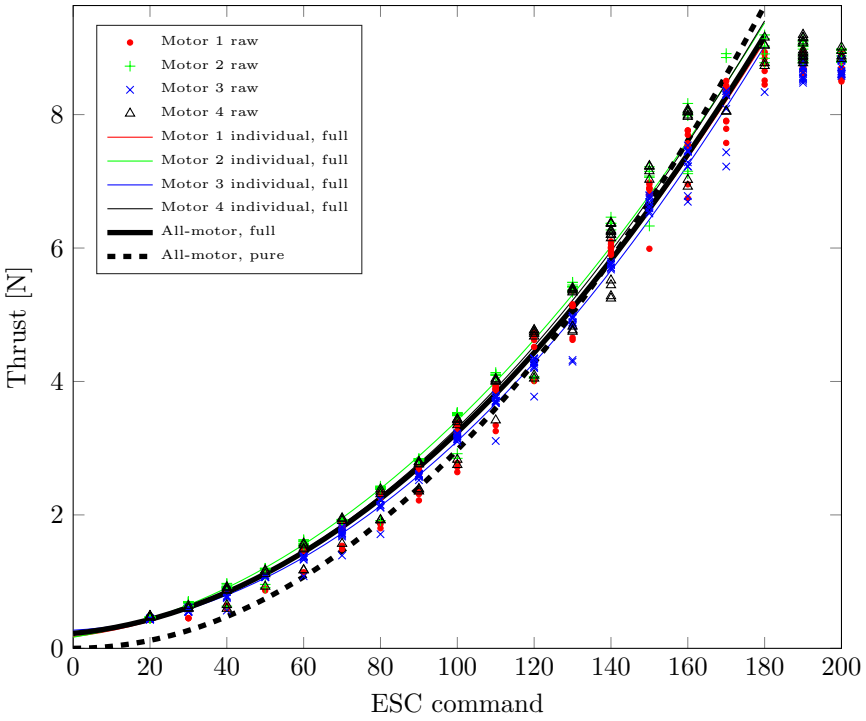
D.4 Motor Thrust Dynamics

The motor spin-up and spin-down time constants, $\tau_{\text{m,up}}$ and $\tau_{\text{m,dn}}$ respectively, are identified by solving the following regression problem:

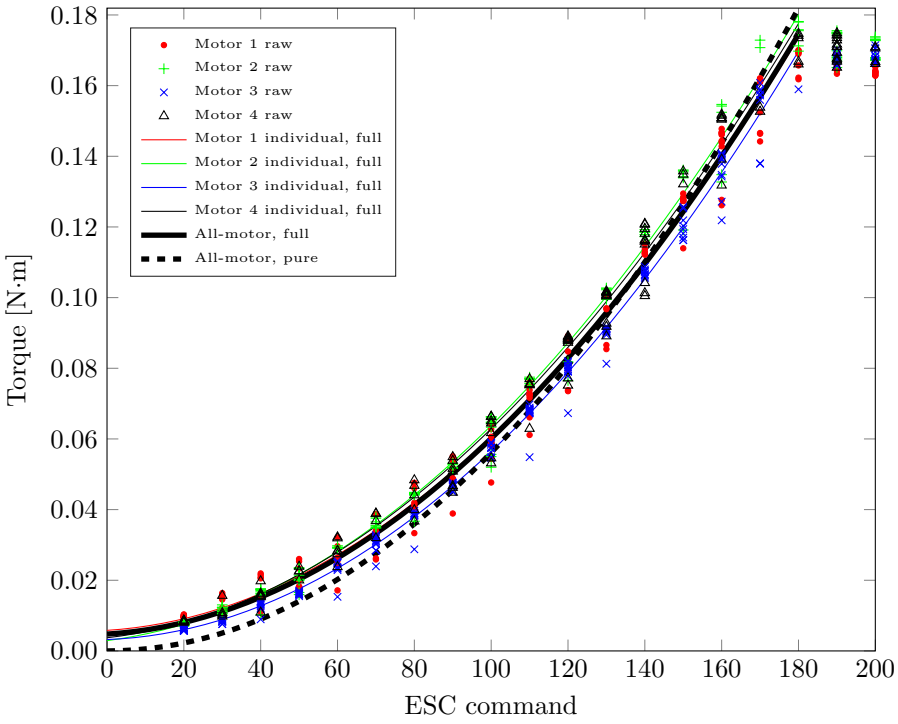
[9] The detour through `fminsearch` is necessary because MATLAB is not known to have a built-in routine for pure quadratic fitting.

[10] True to Section 5.6.2, the bar signifies load-cell thrust and torque.

[11] As a sanity check, it was confirmed that setting different initial values for k_{pure} did not alter the optimal solution returned by `fminsearch`.



(a) Motor thrust map (5.25).



(b) Motor torque map (5.26).

Figure D.7: Identified motor thrust and torque maps.

Coefficient	Value
$k_{\mathrm{f},2}$	$2.455 \cdot 10^{-4}$
$k_{\mathrm{f},1}$	$5.613 \cdot 10^{-3}$
$k_{\mathrm{f},0}$	$2.234 \cdot 10^{-1}$
$k_{\eta,2}$	$4.868 \cdot 10^{-6}$
$k_{\eta,1}$	$6.693 \cdot 10^{-5}$
$k_{\eta,0}$	$4.764 \cdot 10^{-3}$

Table D.2: Numerical values of the identified coefficients for (5.25) and (5.26).

$$
\begin{aligned}
\underset{\tau_{\mathrm{m,up}},\tau_{\mathrm{m,dn}}}{\text{minimize}} \quad & \sum_{i=1}^{4}\sum_{k=2}^{M}(f_i[k]-\hat{f}_i[k])^2 \\
\text{subject to} \quad & \Delta t[k] = (t[k]-t[k-1]) \\
& f_i[k] = \begin{cases} f_i[k-1] + \frac{\Delta t[k]}{\tau_{\mathrm{m,up}}}(f_{i,\mathrm{ref}}[k]-f_i[k-1]) & \text{if } f_{i,\mathrm{ref}}[k] \geq f_i[k-1] \\ f_i[k-1] + \frac{\Delta t[k]}{\tau_{\mathrm{m,dn}}}(f_{i,\mathrm{ref}}[k]-f_i[k-1]) & \text{otherwise} \end{cases} \\
& f_i[1] = \hat{f}_i[1]
\end{aligned}
$$

which selects $\tau_{\mathrm{m,dn}}$ and $\tau_{\mathrm{m,up}}$ such that the Euler-discretized thrust dynamics model (5.24) optimally fits the measured thrusts $\hat{f}_i$ in the least-squares sense. The reference thrust $f_{i,\mathrm{ref}}$ is computed from the logged $u_{\mathrm{esc},i}$ by applying (5.25). To compute $\hat{f}_i$ we pass through the motor angular speed measurement $\hat{\omega}_{\mathrm{p},i}$ (in RPM) using the AscTec-provided map:

$$\hat{\omega}_{\mathrm{p},i} = 43\left(25 + \frac{175 u_{\mathrm{esc},i}}{200}\right) := \mathcal{M}_{\mathrm{esc}}(u_{\mathrm{esc},i}). \tag{D.7}$$

The actual AscTec motor RPM measurement $\hat{\tilde{\omega}}_{\mathrm{p},i}$ is scaled such that:

$$\hat{\omega}_{\mathrm{p},i} = 64\hat{\tilde{\omega}}_{\mathrm{pm},i}. \tag{D.8}$$

Using (5.25), (D.7) and (D.8), we can compute $\hat{f}_i[k]$:

$$\hat{f}_i[k] = \mathcal{M}_{\mathrm{f}}(\mathcal{M}_{\mathrm{esc}}^{-1}(64\hat{\tilde{\omega}}_{\mathrm{pm},i})). \tag{D.9}$$

Note that this method of computing $\hat{f}_i[k]$ contains a certain error due to inaccuracies in the thrust map $\mathcal{M}_{\mathrm{f}}$ and neglects any transient thrust dynamics. It is unfortunately necessary to use this limited approach because the RCbenchmark [155] test stand does not sample fast enough (up to 50 Hz) to capture the fast motor dynamics with a high enough resolution.

The regression problem is further simplified to render it less nonlinear and thus better behaved. We define $\bar{\tau}_{\mathrm{m,dn}} := \tau_{\mathrm{m,dn}}^{-1}$, $\bar{\tau}_{\mathrm{m,up}} := \tau_{\mathrm{m,up}}^{-1}$ and make the corresponding substitution in the thrust dynamics, thus removing the reciprocal function nonlinearity. Furthermore, we define $\Delta\bar{\tau}_{\mathrm{m}} := \bar{\tau}_{\mathrm{m,up}} - \bar{\tau}_{\mathrm{m,dn}}$. This allows us to write the following, equivalent, regression problem:

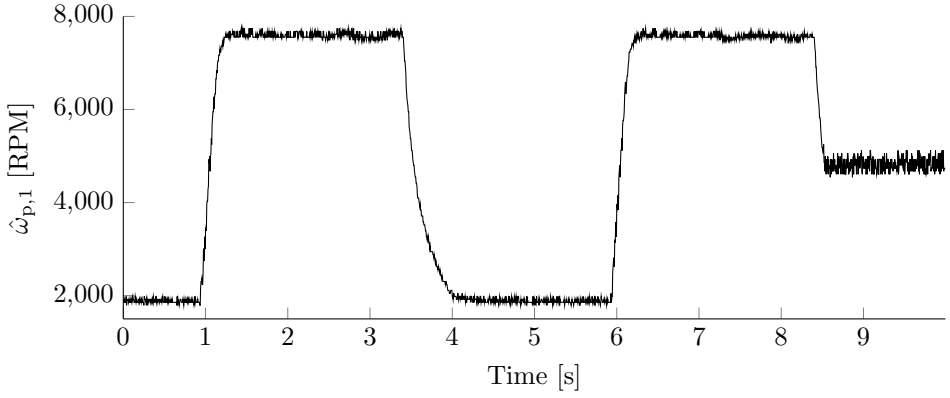


Figure D.8: $\tau_{\mathrm{m,up}} < \tau_{\mathrm{m,dn}}$ occurs when propeller inertia is dominant over parasitic torques (e.g. heavy propeller or low RPM).

$$
\begin{aligned}
\underset{\bar{\tau}_{\mathrm{m,up}},\Delta\bar{\tau}_{\mathrm{m}}}{\text{minimize}} \quad & \sum_{i=1}^{4}\sum_{k=2}^{M}(f_i[k]-\hat{f}_i[k])^2 \\
\text{subject to} \quad & \zeta_i[k] = f_{i,\mathrm{ref}}[k] - f_i[k-1] \\
& \Delta t[k] = (t[k]-t[k-1]) \\
& f_i[k] = f_i[k-1] + \Delta t[k]\Big(\zeta_i[k]\bar{\tau}_{\mathrm{m,up}} + \frac{1}{2}(|\zeta_i[k]| - \zeta_i[k])\Delta\bar{\tau}_{\mathrm{m}}\Big) \\
& f_i[1] = \hat{f}_i[1].
\end{aligned}
\tag{D.10}
$$

The regression problem (D.10) is solved using MATLAB's `fminsearch` nonlinear programming solver[12] using data from an indoor flight test[13]. **The optimal solution is** $\tau_{\mathrm{m,up}} = 53\,\mathrm{ms}$ **and** $\tau_{\mathrm{m,dn}} = 24\,\mathrm{ms}$. Figure D.9 shows a very good model fit to a validation set (a different time interval from the same flight).
Because (D.10) is not convex due to the nonlinear thrust dynamics constraint, locally optimal solutions may exist. In order to mitigate the chance of a locally optimal solution, and to check that the solution is robust to the initial condition, we solve (D.10) on an initial value grid of 10 equally spaced $\tau_{\mathrm{m,up}} \in [5, 500]$ms and 10 equally spaced $\tau_{\mathrm{m,dn}} \in [5, 500]$ms. The resulting optimal solution is constant up to $1 \cdot 10^{-7}$ tolerance and the resulting cost function optimal value is constant up to $1 \cdot 10^{-8}$. For reference, the solution tolerance and the cost tolerance were both set to $1 \cdot 10^{-6}$. We may therefore be reasonably confident that the obtained time constants represent the optimal solution to the regression problem.
The present result of $\tau_{\mathrm{m,up}} > \tau_{\mathrm{m,dn}}$ (i.e. spin-up dynamics are slower than the spin-down dynamics) is contrary to Lupashin et al. [103] and Faessler et al. [107], both of which state that the spin-down dynamics are slower. The present result, however, appears to be physically reasonable. When spinning up, the motor torque has to work against the propeller drag torque, viscous and Coulomb friction in the motor, etc. in order to increase $\omega_{\mathrm{p},i}$ and thus the thrust. When spinning down, these parasitic torques help to decrease $\omega_{\mathrm{p},i}$. Therefore, a higher net torque acts in spin-down than in spin-up. It is therefore to be expected that, all else being equal, a torque-controlled BLDC motor should have $\tau_{\mathrm{m,up}} > \tau_{\mathrm{m,dn}}$. Since neither Lupashin

[12] `/motor_thrust_torque_id/time_constants_id.m`
[13] Thrust dynamics identification based on flight rather than load cell data is that the motor thrusts undergo the same changes as they will in a real flight, thus making the model fit more appropriate for flying.

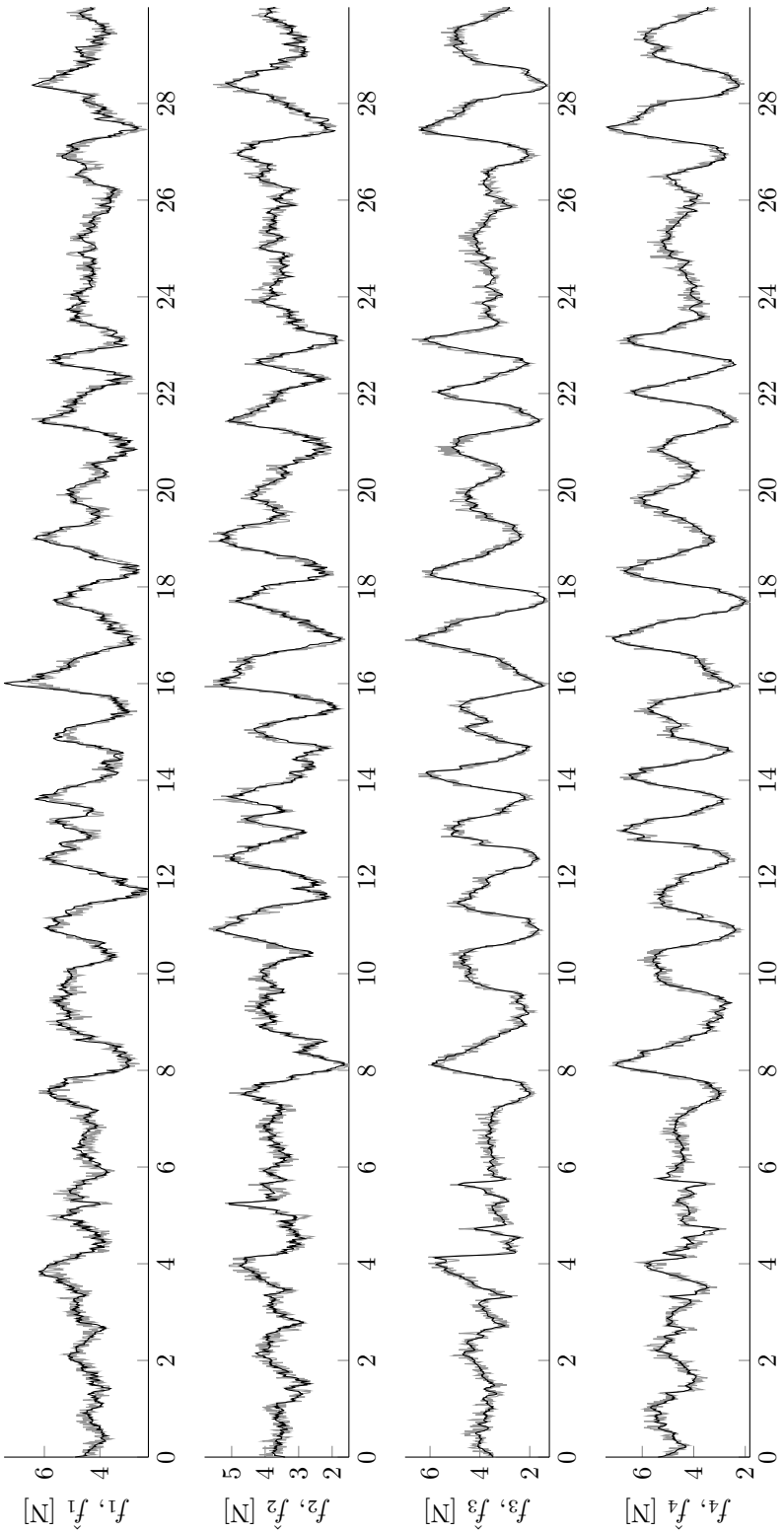


Figure D.9: Comparison of validation data to the thrust dynamics model (5.24). Gray shows $\hat{f}_i$ (measured) and black shows f_i (model predicted).

et al. [103] nor Faessler et al. [107] give details about their identification method, it is not clear why their result is different. As shown in Figure D.8, a possible reason is doing time constant identification in the low RPM region or with a very heavy propeller such that propeller inertia is the dominant effect over parasitic torques. In this case, the propeller's inertial tendency to continue rotating leads to slower spin-down dynamics (first step down in Figure D.8). However, in the case of the AscTec Pelican, parasitic torques are dominant in the flight regime and the spin down dynamics are faster (second step down in Figure D.8).

D.5 Model Validation

To check that the model is representative of reality, we feed it recorded ESC commands and compare the bias-compensated accelerometer and gyro measurements, $\hat{\boldsymbol{a}}_{\text{acc}}$ and $\hat{\boldsymbol{\omega}}$ respectively, to model output $\boldsymbol{a}_{\text{w}}^{\text{b}}$ and $\boldsymbol{\omega}$ respectively. Doing so, however, is not trivial for two reasons:

1. $\hat{\boldsymbol{a}}_{\text{acc}}$ is the *specific acceleration* in the {b} frame Groves [156] whereas the model outputs the *kinematic acceleration* $\boldsymbol{a}_{\text{w}}^{\text{b}}$ in the {w} frame;

2. The dynamics of $\boldsymbol{\omega}$ (fourth equation of (5.30)) are unstable (they are a simple integrator), hence the angular velocities predicted by the model will drift when the model is open-loop fed the recorded ESC commands.

The first complication is readily solved by transforming the specific acceleration into the world frame and accounting for the gravity vector Groves [156]:

$$\hat{\boldsymbol{a}}_{\text{w,m}}^{\text{b}} = C_{\hat{\boldsymbol{q}}_{\text{w}}^{\text{b}}} \hat{\boldsymbol{a}}_{\text{acc}} + \boldsymbol{g}.$$

Two approaches exist to solve the second complication:

1. The angular acceleration can be estimated from the gyro angular velocity measurements, $\hat{\boldsymbol{\omega}}$, and compared to the model prediction (5.22). Since the measurements are available offline, a non-causal zero-phase filtering approach may be used, such as MATLAB's `filtfilt` or a Kalman Smoother Rauch et al. [157];

2. The angular velocity dynamic model (fourth equation of 5.30) may be integrated with resetting to the most recent $\hat{\boldsymbol{\omega}}$ at fixed time intervals. This periodically zeroes-out the drift and lets one visually check if there is a reasonable match between the evolution of the measured $\hat{\boldsymbol{\omega}}$ and the modeled $\boldsymbol{\omega}$. As explained below, we use this approach because it yields a better behaved bundle optimization problem for thrust factors.

We additionally compute the thrust factors γ_i for the motors. Thrust factors are introduced in Section 5.6.4 but we compute them here via a bundle regression problem:

$$\begin{aligned}
\underset{\gamma_i}{\text{minimize}} \quad & \sum_{k=2}^{M} \|\boldsymbol{a}_{\text{w}}^{\text{b}}[k] - \hat{\boldsymbol{a}}_{\text{w}}^{\text{b}}[k]\|^2 + \|\boldsymbol{\omega}[k] - \hat{\boldsymbol{\omega}}[k]\|^2 \\
\text{subject to} \quad & \text{(5.22), (5.23) but using } \gamma_i f_i \text{ instead of } f_i \\
& \boldsymbol{a}_{\text{w}}^{\text{b}}[1] = \hat{\boldsymbol{a}}_{\text{w}}^{\text{b}}[1] \\
& \boldsymbol{\omega}[1] = \tilde{\boldsymbol{\omega}}[1] \\
& \boldsymbol{\omega}[k] = \tilde{\boldsymbol{\omega}}[k] \quad \text{if } t[k] - t[k_{\text{reset,last}}] > \Delta t_{\text{reset}},
\end{aligned} \tag{D.11}$$

where the last constraint resets the integrated angular velocity every Δt_{reset} seconds to $\tilde{\boldsymbol{\omega}}$, the estimate body rate via a Kalman Smoother. We set $\Delta t_{\text{reset}} = 0.015\,\text{s}$ which corresponds to the time constant (i.e. one over the crossover frequency) of the 1 dimensional angular velocity dynamics $\dot{\omega} = \xi/J$ with $J \approx 0.015\,\text{kg} \cdot \text{m}^2$ as according to the diagonal terms of $\tilde{J}$ in Table 5.1. Regression problem (D.11) was noted to converge more consistently in the γ_i decision variables than the approach using directly the estimated angular acceleration. Using flight data, we find $\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\} = \{1.26, 1.46, 1.08, 1.00\}$.
The model is validated on a separate time interval of the same flight test. We run the same simulation as given by the constraints of (D.11). Figures D.10 and D.11 show the results together with the mean and 3 standard deviations of the $\tilde{\boldsymbol{\omega}}$ estimate. We remark that:

- The acceleration match (Figure D.10) is very good. Some discrepancies (e.g. $a^{\text{b}}_{\text{w,z}}$ for $t \in [44, 50]$ seconds) may be due to neglected aerodynamic drag and propeller aerodynamic effects such as increased efficiency at higher angles of attack Hoffmann et al. [117], Mueller and D'Andrea [116];

- The model angular velocity stays mostly within the 3 standard deviation uncertainty bounds of $\tilde{\boldsymbol{\omega}}$ in Figure D.11. This may be considered a good fit, especially due to the fast nature of the body rate dynamics which, coupled with gyro noise, make it a difficult quantity to validate. Periods of bad model fit (e.g. when $\boldsymbol{\omega} \notin$ (3 standard deviation bound)) may be attributed to the numerous approximations in the simple body rate model (5.23): neglected propeller H-force Martin and Salaun [115], other propeller high-order aerodynamic effects Hoffmann et al. [117], neglected propeller gyroscopic effect, exogenous aerodynamic torque, etc. In practice, however, it was found that body rate control benefits more from large feedback Lurie and Enright [158] than from a highly accurate body rate model. A large gain proportional controller and motors with fast dynamics (for time scale separation) suffice for good body rate control, as is commonly done in literature Achtelik et al. [109], Lupashin et al. [103], Faessler et al. [105, 104].

Overall, the model exhibits a good match in acceleration and a satisfactory match in the body rates. A practical testimony to the accuracy of the identified parameters herein is that the RotorS simulation using these parameters flies well with the same controller tuning as in reality except for $k_{\omega,\text{x}}$ and $k_{\omega,\text{y}}$ (Table 5.2), which is reduced from 45 in reality to 30 in the simulation.

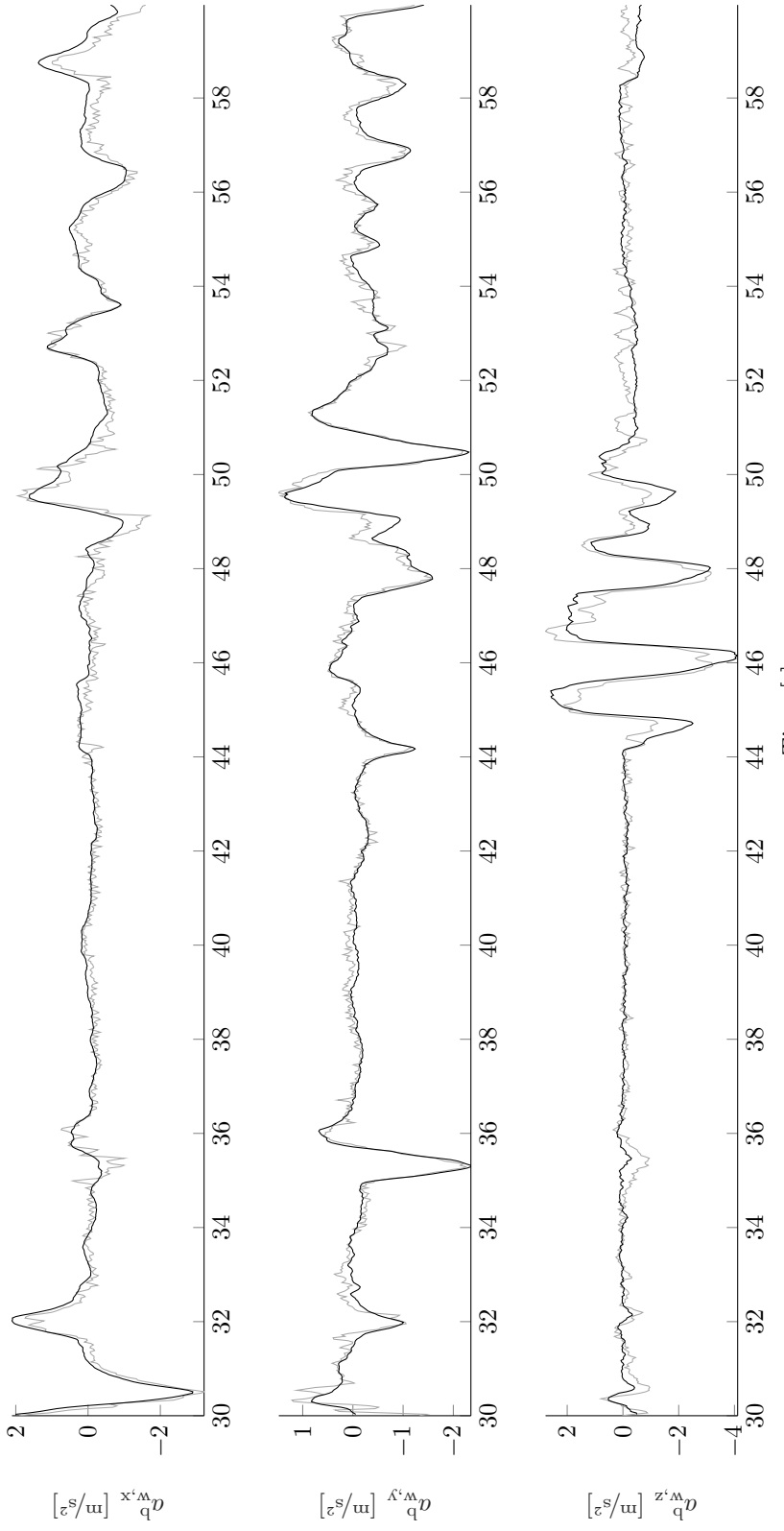


Figure D.10: Comparison of measured $\hat{\boldsymbol{a}}^{b}_{w}$ (gray) to modeled $\boldsymbol{a}^{b}_{w}$ (black).

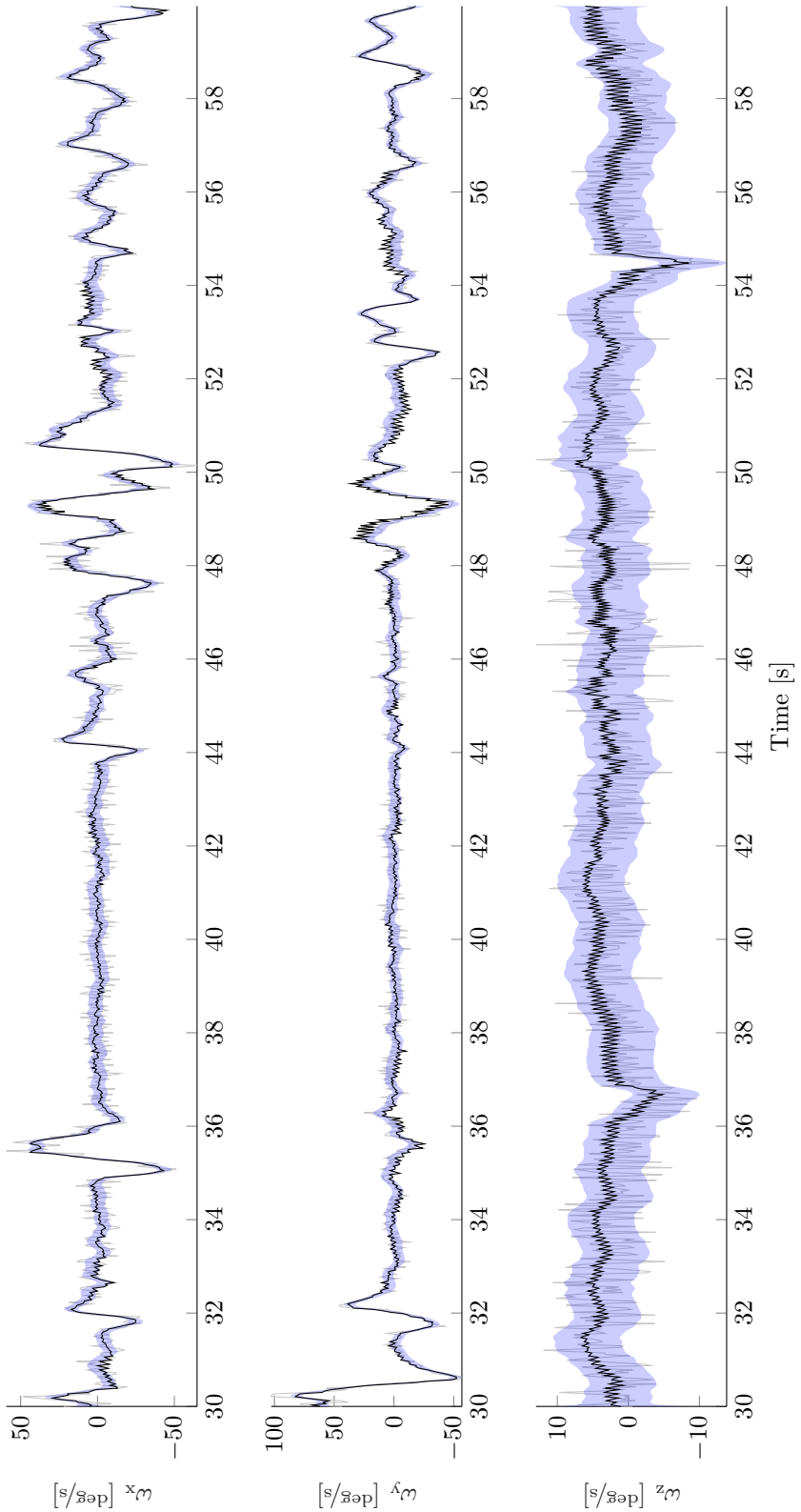


Figure D.11: Comparison of measured $\hat{\boldsymbol{\omega}}$ (gray) to modeled $\boldsymbol{\omega}$ (black). The blue area is the 3 standard deviation uncertainty in $\tilde{\boldsymbol{\omega}}$ (Kalman Smoother output).

This page is intentionally left blank.

Appendix E

Multirotor Differential Flatness

A multirotor is a *differentially flat* system Van Nieuwstadt and Murray [159], Mellinger and Kumar [81]. A differentially flat system's states and inputs may be represented by a set of *flat outputs* (equal in number to the number of inputs) and their derivatives. This appendix:

- Develops in Section E.1 a map from a multirotor's flat outputs $\boldsymbol{\sigma} \in \mathbb{R}^4$ (a multirotor has four inputs – collective thrust and three body torques) and their derivatives to its states and inputs. The derivation follows that of Mellinger and Kumar [81], Mellinger [160] but details all the steps in order to provide a complete derivation;
- Remarks on dynamically feasibly trajectory generation in Section E.2;

E.1 Flat Output To State And Input Map

Consider the quadrotor simplified flight dynamics model as developed in Section 5.4.2:

$$m\ddot{\boldsymbol{p}}_{\mathrm{w}}^{\mathrm{b}} = -m\|\boldsymbol{g}\|\boldsymbol{e}_3 + f\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}, \tag{E.1}$$

$$\dot{\boldsymbol{\omega}} = J^{-1}\left(\boldsymbol{\xi} - \boldsymbol{\omega} \times (J\boldsymbol{\omega})\right), \tag{E.2}$$

where motor dynamics are omitted. Consider the following flat outputs:

$$\boldsymbol{\sigma} := (\boldsymbol{p}_{\mathrm{w}}^{\mathrm{b}}, \psi), \tag{E.3}$$

where ψ is the yaw angle about the $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}}$ axis. We furthermore consider the Euler angle Z-X-Y convention[1], as illustrated in Figure E.1, where:

1. The multirotor first yaws by ψ about $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}}$, rotating the $\{\mathrm{w}\}$ frame into the $\{\mathrm{c}\}$ frame;
2. The multirotor then rolls by φ about $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}$, rotating the $\{\mathrm{c}\}$ frame into the $\{\mathrm{c}_2\}$ frame;
3. The multirotor finally pitches by θ about $\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}$, rotating the $\{\mathrm{c}_2\}$ frame into the $\{\mathrm{b}\}$ frame.

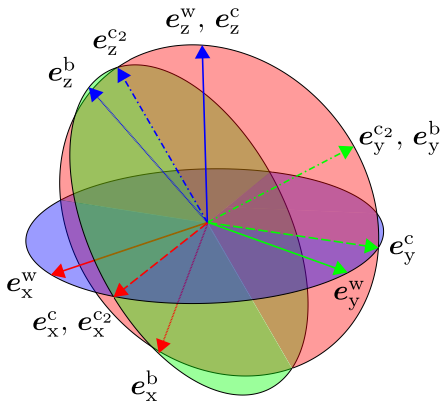


Figure E.1: Frames involved in determining the map from $\boldsymbol{\sigma}$ to $C_{(\boldsymbol{q}_w^b)}$.

The translation states are trivially written as a function of $\boldsymbol{\sigma}$:

$$\boldsymbol{p}_w^b = (\sigma_1, \sigma_2, \sigma_3), \tag{E.4}$$
$$\boldsymbol{v}_w^b = (\dot{\sigma}_1, \dot{\sigma}_2, \dot{\sigma}_3), \tag{E.5}$$
$$\boldsymbol{a}_w^b = (\ddot{\sigma}_1, \ddot{\sigma}_2, \ddot{\sigma}_3), \tag{E.6}$$

where we let $\sigma_i = \sigma[i]$ i.e. the i-th element of $\boldsymbol{\sigma}$. Higher derivatives of the translation states (i.e. jerk, snap, etc.) are equally naturally expressed as the derivatives of $\boldsymbol{\sigma}$. From (E.1) it follows that:

$$\boldsymbol{e}_z^b = \frac{m}{f}(\boldsymbol{a}_w^b + \|\boldsymbol{g}\|\boldsymbol{e}_3), \tag{E.7}$$

therefore, by using (E.6) and the fact that $\|\boldsymbol{e}_z^b\| = 1$, we have:

$$\boldsymbol{e}_z^b = \frac{\boldsymbol{t}}{\|\boldsymbol{t}\|} \tag{E.8}$$
$$\boldsymbol{t} := (\ddot{\sigma}_1, \ddot{\sigma}_2, \ddot{\sigma}_3 + \|\boldsymbol{g}\|) \tag{E.9}$$

Given $\psi = \sigma_4$, the $\{c\}$ frame x-axis is given directly by:

$$\boldsymbol{e}_x^c = (\cos(\sigma_4), \sin(\sigma_4), 0). \tag{E.10}$$

Then, given the Euler angle Z-X-Y convention, the remaining axes of the $\{b\}$ can be determined:

$$\boldsymbol{e}_y^b = \frac{\boldsymbol{e}_z^b \times \boldsymbol{e}_x^c}{\|\boldsymbol{e}_z^b \times \boldsymbol{e}_x^c\|}, \tag{E.11}$$
$$\boldsymbol{e}_x^b = \boldsymbol{e}_y^b \times \boldsymbol{e}_z^b, \tag{E.12}$$

provided that $\boldsymbol{e}_z^b \times \boldsymbol{e}_x^c \neq 0$. Then:

$$C_{(\boldsymbol{q}_w^b)} = \begin{bmatrix} \boldsymbol{e}_x^b & \boldsymbol{e}_y^b & \boldsymbol{e}_z^b \end{bmatrix}. \tag{E.13}$$

[1]Note that this convention is different from the Tait-Bryan Z-Y-X Euler angle convention (i.e. Y and X are permuted). This is the convention used in Mellinger and Kumar [81] and is important for the derivation.

We now consider the map from $\boldsymbol{\sigma}$ to the angular velocity $\boldsymbol{\omega}$. As a first step, differentiate (E.2):

$$m\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} = \dot{f}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + \boldsymbol{\omega} \times f\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}. \tag{E.14}$$

Projecting (E.14) along $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}$:

$$\begin{aligned} m\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} &= \dot{f}\underbrace{\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}_{=1} + \underbrace{(\boldsymbol{\omega} \times f\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}_{=0}, \\ \Rightarrow \dot{f} &= m\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}. \end{aligned} \tag{E.15}$$

Substituting (E.15) into (E.14) and rearranging, we obtain:

$$\boldsymbol{h}_{\boldsymbol{\omega}} := \boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} = \frac{m}{f}(\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} - (\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}). \tag{E.16}$$

Given the fact that:

$$\boldsymbol{\omega} = \omega_{\mathrm{x}}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} + \omega_{\mathrm{y}}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} + \omega_{\mathrm{z}}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}, \tag{E.17}$$

we have directly:

$$\omega_{\mathrm{x}} = -\boldsymbol{h}_{\boldsymbol{\omega}} \cdot \boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}, \tag{E.18}$$

$$\omega_{\mathrm{y}} = \boldsymbol{h}_{\boldsymbol{\omega}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}}. \tag{E.19}$$

To obtain ω_{z}, we recognize that $\boldsymbol{\omega}$ can also be written in terms of Euler angles:

$$\begin{aligned} \boldsymbol{\omega} &= \dot{\psi}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} + \dot{\varphi}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \dot{\theta}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}, \\ \Rightarrow \boldsymbol{\omega} &= \dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} + \dot{\varphi}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \dot{\theta}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}. \end{aligned} \tag{E.20}$$

By substituting (E.17) into (E.20), projecting along $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}$ and simplifying $\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} = 0$, we obtain:

$$\omega_{\mathrm{z}} = \dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + \dot{\varphi}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}. \tag{E.21}$$

It remains to find $\dot{\varphi}$, which we do by again substituting (E.17) into (E.20) and this time projecting along $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}$:

$$\begin{aligned} \boldsymbol{\omega} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= (\dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} + \dot{\varphi}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \dot{\theta}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}, \\ \boldsymbol{\omega} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= \dot{\sigma}_4\underbrace{\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}}_{=0 \text{ via (E.10)}} + \dot{\varphi}\underbrace{\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}}_{=1} + \dot{\theta}\underbrace{\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}}_{=0 \text{ via (E.11)}}, \\ \boldsymbol{\omega} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= \dot{\varphi}, \\ (\omega_{\mathrm{x}}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} + \omega_{\mathrm{y}}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} + \omega_{\mathrm{z}}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= \dot{\varphi}, \\ \omega_{\mathrm{x}}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \omega_{\mathrm{y}}\underbrace{\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}}_{=0 \text{ via (E.11)}} + \omega_{\mathrm{z}}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= \dot{\varphi}, \\ \Rightarrow \omega_{\mathrm{x}}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \omega_{\mathrm{z}}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} &= \dot{\varphi}. \end{aligned} \tag{E.22}$$

Substituting (E.22) into (E.21) and rearranging, we obtain ω_{z}:

$$\omega_{\mathrm{z}} = \frac{\dot{\sigma}_4(\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \omega_{\mathrm{x}}(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}})(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})}{1 - (\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})^2}. \tag{E.23}$$

Therefore, the map from $\boldsymbol{\sigma}$ to $\boldsymbol{\omega}$ is given by (E.18), (E.19) and (E.23) which are assembled via (E.17).
We now consider the map from $\boldsymbol{\sigma}$ to the angular acceleration $\dot{\boldsymbol{\omega}}$. As a first step, differentiate (E.14):

$$m\ddot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} = \ddot{f}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + 2\dot{f}\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + f\dot{\boldsymbol{\omega}} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + f\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}). \tag{E.24}$$

Projecting (E.24) along $\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}$:

$$\begin{aligned} m\ddot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} &= \ddot{f}\underbrace{\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}_{=1} + 2\dot{f}\underbrace{(\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}_{=0} + f\underbrace{(\dot{\boldsymbol{\omega}} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}}_{=0} + f(\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}, \\ \Rightarrow \ddot{f} &= m\ddot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} - f(\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}. \end{aligned} \tag{E.25}$$

Substituting (E.15) and (E.25) into (E.24) and rearranging, we obtain:

$$\begin{aligned} \boldsymbol{h}_{\dot{\boldsymbol{\omega}}} := \dot{\boldsymbol{\omega}} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} = &\frac{m}{f}(\ddot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} - (\ddot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) - \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \\ &(\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} - \frac{2m}{f}(\dot{\boldsymbol{a}}_{\mathrm{w}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})(\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}). \end{aligned} \tag{E.26}$$

Given the fact that:

$$\dot{\boldsymbol{\omega}} = \dot{\omega}_{\mathrm{x}}\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} + \dot{\omega}_{\mathrm{y}}\boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}} + \dot{\omega}_{\mathrm{z}}\boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}, \tag{E.27}$$

we have directly:

$$\dot{\omega}_{\mathrm{x}} = -\boldsymbol{h}_{\dot{\boldsymbol{\omega}}} \cdot \boldsymbol{e}_{\mathrm{y}}^{\mathrm{b}}, \tag{E.28}$$

$$\dot{\omega}_{\mathrm{y}} = \boldsymbol{h}_{\dot{\boldsymbol{\omega}}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}}. \tag{E.29}$$

We now compute $\dot{\omega}_{\mathrm{z}}$ by direct differentiation of ω_{z} (E.23). To make the formulation more concise, define two quantities derived from (E.23):

$$\begin{aligned} \omega_{\mathrm{z,num}} &= \dot{\sigma}_4(\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \omega_{\mathrm{x}}(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}})(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}), \\ \omega_{\mathrm{z,denom}} &= \frac{1}{1 - (\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})^2}. \end{aligned}$$

We have:

$$\begin{aligned} \dot{\omega}_{\mathrm{z,num}} = &\ddot{\sigma}_4(\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \dot{\sigma}_4(\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \cdot (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})) + \dot{\omega}_{\mathrm{x}}(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}})(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \\ &\omega_{\mathrm{x}}((\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}}) \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} + \boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot (\dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \times \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}))(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}) + \\ &\omega_{\mathrm{x}}(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{b}} \cdot \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}})((\dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \times \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})), \end{aligned} \tag{E.30}$$

$$\dot{\omega}_{\mathrm{z,denom}} = \frac{2(\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})((\dot{\sigma}_4\boldsymbol{e}_{\mathrm{z}}^{\mathrm{w}} \times \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}}) \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} + \boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot (\boldsymbol{\omega} \times \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}}))}{(1 - (\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}})^2)^2}. \tag{E.31}$$

Then using (E.30) and (E.31), we obtain $\dot{\omega}_{\mathrm{z}}$ via the product rule given that $\omega_{\mathrm{z}} = \omega_{\mathrm{z,num}}\omega_{\mathrm{z,denom}}$:

$$\dot{\omega}_{\mathrm{z}} = \dot{\omega}_{\mathrm{z,num}}\omega_{\mathrm{z,denom}} + \omega_{\mathrm{z,num}}\dot{\omega}_{\mathrm{z,denom}}. \tag{E.32}$$

With the map from $\boldsymbol{\sigma}$ to the states obtained, we now compute the map from $\boldsymbol{\sigma}$ to the inputs f and $\boldsymbol{\xi}$. For f, we identify (E.7) with (E.8) and get directly:

$$f = m\|\boldsymbol{t}\|. \tag{E.33}$$

For $\boldsymbol{\xi}$, we use directly (E.2) given that the maps for $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$ are both available from the above calculations:

$$\boldsymbol{\xi} = J\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (J\boldsymbol{\omega}). \tag{E.34}$$

Note that (E.11), (E.23) and (E.31) all suffer from a division by zero singularity when $\boldsymbol{e}_{\mathrm{x}}^{\mathrm{c}} \cdot \boldsymbol{e}_{\mathrm{z}}^{\mathrm{b}} = 0$. Mellinger and Kumar [81] describe that in practice this causes large changes in $C_{(\boldsymbol{q}_{\mathrm{w}}^{\mathrm{b}})}$ close to the singularity. They suggest a remedy to consider either the direct, or the yawed by 180 °, attitude based on whichever is closest to the quadrotor actual attitude $\tilde{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{b}}$. Because the data collection quadrotor is not expected to execute a 90 ° roll or pitch maneuver – the thrust pointing constraint of Section 5.7 prevents this from happening – and because the current implementation does not use the attitude map anyway (see below), this singularity is left present in the current implementation. Future work may remove it. To summarize, the map from flat outputs $\boldsymbol{\sigma}$ to the multirotor states and inputs is:

- Map to the states:
  - To $\boldsymbol{p}_{\mathrm{w}}^{\mathrm{b}}$: (E.4);
  - To $\boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}$: (E.5);
  - To $\boldsymbol{a}_{\mathrm{w}}^{\mathrm{b}}$: (E.6);
  - To $\boldsymbol{\omega}$: (E.17), (E.18), (E.19) and (E.23);
  - To $\dot{\boldsymbol{\omega}}$: (E.27), (E.28), (E.29) and (E.32).
- Map to the inputs:
  - To f: (E.33);
  - To $\boldsymbol{\xi}$: (E.34).

This map is present in the current implementation[2], but only the trivial $\boldsymbol{p}_{\mathrm{w}}^{\mathrm{b}}$ and $\boldsymbol{v}_{\mathrm{w}}^{\mathrm{b}}$ are used due to control limitations explained in Chapter 5. The remainder of the map can potentially be used by future implementations for control feed-forward inputs, leading to higher performance trajectory tracking.

E.2 Dynamic Feasibility

Chapter 4 calls trajectories that are C^4 continuous in $\boldsymbol{\sigma}$ (i.e. are continuous up to *snap*, the fourth derivative of position) as *dynamically feasible*. In other words, the trajectories satisfy the multirotor dynamics such that they can actually be executed. If $\boldsymbol{\sigma}$ is C^4 continuous, then $\ddot{\ddot{\boldsymbol{\sigma}}}$ is C^0 continuous. This means that $\boldsymbol{h}_{\dot{\boldsymbol{\omega}}}$ in (E.26) is continuous (due to $\ddot{\boldsymbol{a}}_{\mathrm{z}}^{\mathrm{b}}$) which means that $\dot{\boldsymbol{\omega}}$ and $\boldsymbol{\xi}$ are also continuous. All other states and inputs depend on lower order derivatives of $\boldsymbol{\sigma}$, thus they are continuously differentiable (i.e. they are class C^k, $k > 0$).
However, **it is not strictly true that a C^4 continuous trajectory in $\boldsymbol{\sigma}$ is dynamically feasible**. Namely, C^4 guarantees *continuity* but not *differentiability* of $\boldsymbol{\xi}$ (due to its $\dot{\boldsymbol{\omega}}$ dependency). However, as modeled in Section 5.6, there is an algebraic relationship between $\boldsymbol{\xi}$ and the motor thrusts. Without loss of generality, let us consider in 1 dimension a generic ξ which is either ξ_{x} or ξ_{y}, which are achieved via differential thrust. Consider the motor i thrust, f_i. As given by (5.25), an

[2] `/general/alure_common/include/alure_common/state_representations.h`

algebraic relationship exists between f_i and $\omega_{\mathrm{p},i}$, the angular velocity of propeller i. Then, an algebraic relationship exists between ξ and $\omega_{\mathrm{p},i}$, therefore an algebraic relationship exists between $\dot{\xi}$ and $\dot{\omega}_{\mathrm{p},i}$. If $\dot{\xi}$ is discontinuous (i.e. it "jumps"), can $\dot{\omega}_{\mathrm{p},i}$ jump? As given by (5.21), $\omega_{\mathrm{p,i}}$ has second-order dynamics with respect to the motor input voltage, **therefore the propeller angular velocity is differentiable (C^1) and $\dot{\omega}_{\mathrm{p},i}$ cannot jump**. Hence a discontinuous $\dot{\boldsymbol{\xi}}$ is dynamically infeasible due to motor current dynamics.
However, motor current dynamics are fast. If we assume that the motor current can be instantly set, then indeed $\dot{\omega}_{\mathrm{p,i}}$ can be discontinuous. This is the simplification that we make when saying that a snap-continuous trajectory is dynamically feasible. For all but the most extreme maneuvers, the dynamic infeasibility with respect to the motor current dynamics is likely imperceptible. For the case of this thesis, where the quadrotor flies slow data acquisition trajectories, this dynamic infeasibility is a non-issue.

Appendix F

Controller Tuning Procedure

This appendix describes the *tuning* (i.e. parameter selection) procedure for the control architectures developed in Section 5.8.

F.1 PID Tuning

This section describes the tuning procedure for the 1 DOF and 2 DOF PID control architectures of Section 5.8.2. The 1 DOF architecture requires selecting k_p, k_i and k_d. In addition to these, the 2 DOF architecture also requires selecting ω_n and ζ. Our approach is to set the same k_p, k_i and k_d for both architectures and to set ω_n and ζ a posteriori for the 2 DOF architecture.

(Step 1) PD Controller Tuning

The 1 DOF PID controller is first tuned with $k_\mathrm{i} = 0$ (no integral action). It is noted that with no integral term the complementary sensitivity of Figure 5.16a becomes:

$$T_\mathrm{PD}(s) = \frac{k_\mathrm{p}(1 + \frac{k_\mathrm{d}}{k_\mathrm{p}}s)}{s^2 + k_\mathrm{d}s + k_\mathrm{p}},$$

which is similar to (5.71) except for the Left Half Plane (LHP) zero at $k_\mathrm{p}/k_\mathrm{d}$. This zero corresponds to an overshoot in the time response Skogestad and Postlethwaite [102] and increases the bandwidth. Nevermind this difference, for tuning we neglect this zero and identify k_p and k_d with an ideal response of a second order system (5.71):

$$k_\mathrm{p} = \omega_\mathrm{n}^2 \qquad k_\mathrm{d} = 2\zeta\sqrt{k_\mathrm{p}}. \tag{F.1}$$

By setting a fixed value for ζ (e.g. 1 for critical damping or 1.2 for slight over-damping) and slowly increasing ω_n starting from a small value such as $1\,{}^{\mathrm{rad}}/_{\mathrm{s}}$, k_p and k_d tuning reduces to a 1 dimensional search for a good ω_n with eventual slight modifications to ζ. In this tuning procedure:

- Increasing ω_n corresponds to increasing the closed-loop bandwidth;
- Increasing ζ corresponds to reducing overshoots (the D term adds phase to the system, thus increasing robustness).

Each $(\omega_\mathrm{n}, \zeta)$ pair is evaluated via a flight test and the decision to increase/decrease ω_n and ζ is made by visual evaluation of tracking quality. Eventually, a too high value of ω_n will violate time scale separation from the attitude inner loop and a too high value of ζ will introduce jitter due to increased usage of the noisier velocity estimate.

(Step 2) Introducing Integral Action

Uncertainties in the plant (such as propulsion unit differences, center of mass offset, inaccurate mass, etc.) create a steady-state offset in the position response using just a PD controller. The k_i term is increased from zero in small increments and the removal of steady-state offset is noted via flight tests for each k_i setting. A large k_i may make the response oscillatory since an integrator adds negative phase. When this happens, k_d may be increased to provide yet more positive phase or one may simply stop increasing k_i. At this point, the PID gains are tuned.

(Step 3) Choosing the Pre-Filter Parameters

This step is carried out only for the 2 DOF architecture in Figure 5.17b. While the PID controller determines the response dynamics to load disturbance, to output disturbances and to measurement noise, we can choose the pre-filter ω_n and ζ parameters in (5.73) to achieve an ideal second order system's reference-to-output response[1]. We choose the initial ω_n and ζ to be those determined by (F.1). These initial values are subsequently changed e.g. to increase bandwidth or damping with the same intuition and limitations as in Step 1. Notably, the pre-filter will remove reference-to-output response overshoots if $\zeta \geq 1$ (overdamping may be required to hedge model uncertainty).

(Step 4) MIMO Tuning

Steps 1-3 describe the tuning procedure for a single axis. However, there are 3 axes ($\boldsymbol{e}_\mathrm{x}^\mathrm{w}$, $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$) to consider. Because axes $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ both require thrust vectoring (which the quadrotor can achieve equally well in any xy-direction) while $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ simply requires changing motor thrusts, we choose the same tuning for the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ axes and a more aggressive one for the $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ axis. While it is possible to tune all axes at once (i.e. have all three axes' controllers active), our implementation provides the possibility to selectively turn off axes leaving only manual acceleration control for them[2]. This enables one to tune each axis individually.

F.2 PI Tuning

This section describes the tuning procedure for the 1 DOF and 2 DOF PI control architectures of Section 5.8.3. The 1 DOF architecture requires selecting k_p and k_i. In addition to these, the 2 DOF architecture also requires selecting τ. Our approach is to set the same k_p and k_i for both architectures and to set τ a posteriori for the 2 DOF architecture.

(Step 1) P Controller Tuning

The 1 DOF PI controller is first tuned with $k_\mathrm{i} = 0$ (no integral action). It is noted that with no integral term the complementary sensitivity of Figure 5.20a becomes:

$$T_\mathrm{P}(s) = \frac{1}{\frac{1}{k_\mathrm{p}}s + 1}, \tag{F.2}$$

which means that $k_\mathrm{p} = \tau^{-1}$. Selecting a desired first-order reference-to-velocity response time constant, one directly obtains the k_p value. Checking each value

[1] The usual caveats apply, like model uncertainty and time scale separation requirement with the attitude inner loop.

[2] `/control/asctec_mav_framework/asctec_hl_interface/cfg/NewPositionController.cfg`

via a flight test, k_p is increased until the reference-to-velocity response becomes oscillatory (due to time scale separation violation with the attitude inner loop).

(Step 2) Introducing Integral Action

As in Section F.1, uncertainties in the plant as well as exogenous effects like wind will introduce steady-state offset in reference velocity tracking. Integral action is introduced in small steps starting from a low k_i value until steady-state offset is removed while making sure that the response does not become oscillatory due to a large k_i (i.e. too much negative phase). At this point, the PI gains are tuned.

(Step 3) Choosing Pre-Filter Parameters

This step is carried out only for the 2 DOF architecture in Figure 5.20b. While the P controller could match an ideal first-order response, integral action needed to be introduced for robustness. The pre-filter is an opportunity to again match the ideal first-order response. Thus, we begin by setting $\tau = k_\mathrm{p}^{-1}$ as explained for (F.2). This initial value may be subsequently changed until a satisfactory response is achieved (verified via flight tests).

(Step 4) MIMO Tuning

Steps 1-3 describe the tuning procedure for a single axis. With reasons identical to Section F.1, we choose the same tuning for the $\boldsymbol{e}_\mathrm{x}^\mathrm{w}$ and $\boldsymbol{e}_\mathrm{y}^\mathrm{w}$ axes and a more aggressive one for the $\boldsymbol{e}_\mathrm{z}^\mathrm{w}$ axis. Again, our implementation provides the ability to tune axes individually by turning other axes off (and leaving the operator to manually control these in acceleration mode).

This page is intentionally left blank.

Appendix G

Zero Order Hold Delay Modeling

The objective of this appendix is to develop a continuous-time transfer function of the ZOH element for control design in the continuous time domain. The ZOH element is located between a discrete time controller implementation and the plant, as shown in Figure G.1. It serves the purpose of converting a discrete signal into a continuous one by "holding" the output at the latest discrete input value until a new one arrives. Modeling the ZOH block is important when the sampling time T is large compared to the control loop bandwidth because, as shall be seen, the ZOH element introduces delay. The derivation below appeared in Smith [161].

As shown in Figure G.2, consider a unit impulse input to a ZOH block. We have:

$$\delta[k] = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \end{cases} \tag{G.1}$$

$$h(t) = u_{\text{step}}(t) - u_{\text{step}}(t - T). \tag{G.2}$$

We take advantage of the fact that the transfer function is the Laplace transform of the impulse response, $h(t)$. Given that $\mathcal{L}\{u_{\text{step}}(t - T)\} = e^{-Ts}/s$, we have:

$$\mathcal{L}\{h(t)\} := H(s) = \frac{1 - e^{-Ts}}{s}. \tag{G.3}$$

We now consider the frequency response of $H(s)$ by letting $s = j\omega$:

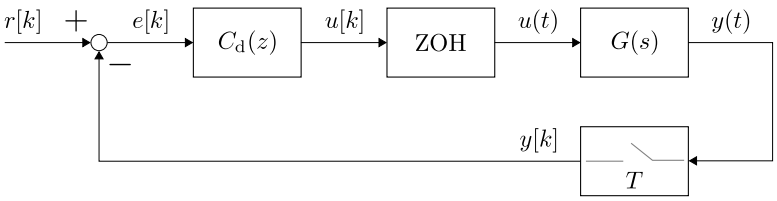


Figure G.1: Typical SISO control loop with a discrete controller $C_{\text{d}}(z)$ and a continuous plant $G(s)$. The ZOH block (i.e. Digital to Analogue Converter (DAC)) and the *sampler* block (i.e. Analogue to Digital Converter (ADC)) are shown.

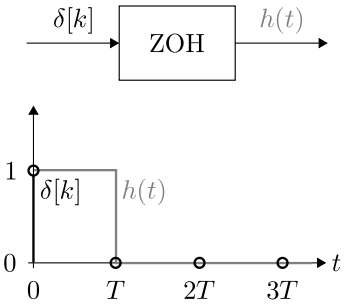


Figure G.2: Unit impulse response of the ZOH block.

$$\begin{aligned}
\mathrm{ZOH}(j\omega) &= \frac{1 - e^{-j\omega T}}{j\omega} \\
&= e^{-j\omega T/2} \left(\frac{e^{j\omega T/2} - e^{-j\omega T/2}}{j\omega T} \right) T \\
\Rightarrow ZOH(j\omega) &= T e^{-j\omega T/2} \operatorname{sinc}\left(\frac{\omega T}{2} \right), \qquad \text{(G.4)}
\end{aligned}$$

where $\operatorname{sinc}(x) := \sin(x)/x$. The Bode plot of (G.4) is visualized in Figure G.3, where the ideal ZOH response also appears. The Nyquist frequency, $\omega_{\mathrm{N}} = \pi$, is also noted (given $T = 1$, the sampling frequency is $\omega_{\mathrm{sampling}} = 2\pi/T = 2\pi$ and $\omega_{\mathrm{N}} := \omega_{\mathrm{sampling}}/2$). It becomes clear why it is common practice to sample at least 10 times faster than the control system bandwidth: then the ZOH gain is ≈ 1 and the ZOH phase is $< 10\,^\circ$ at the bandwidth.
The biggest concern for the control engineer is the delay term $e^{-j\omega T/2}$ in (G.4). The ZOH block therefore introduces, fundamentally, a $T/2$ delay into the control loop. This is visualized in Figure G.4. The continuous-time transfer function used for the ZOH block for control design in the continuous time domain is then:

$$\mathrm{ZOH}(s) := e^{-\frac{Ts}{2}}. \qquad \text{(G.5)}$$

This result is used in e.g. Section 5.6 for body rate controller design.

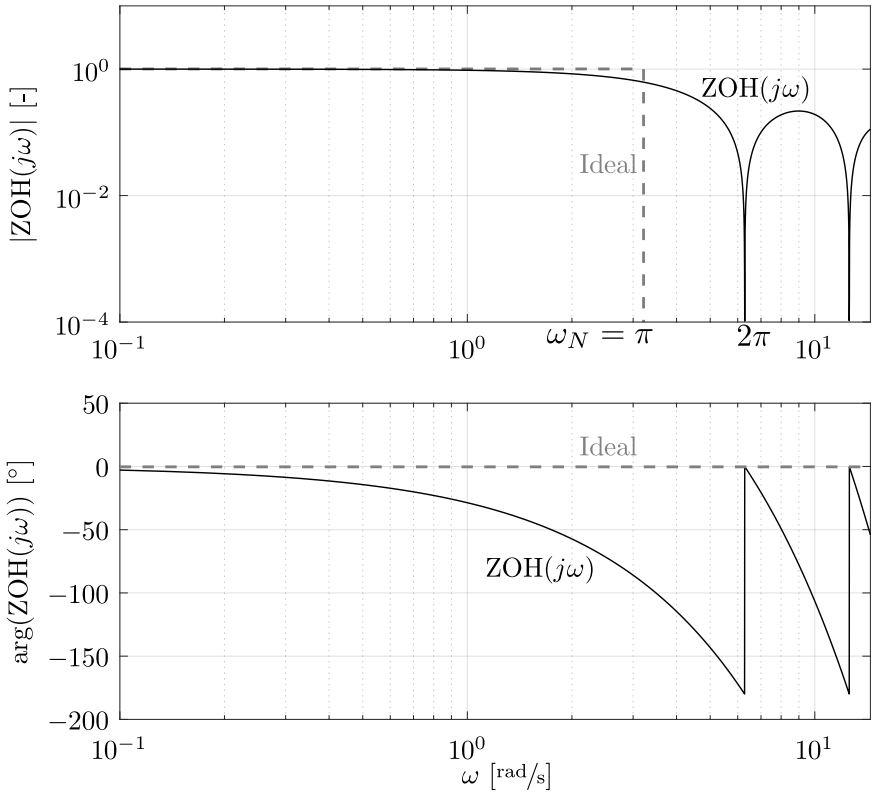


Figure G.3: Bode plot of ZOH$(j\omega)$ in (G.4) for $T = 1\,\mathrm{s}$.

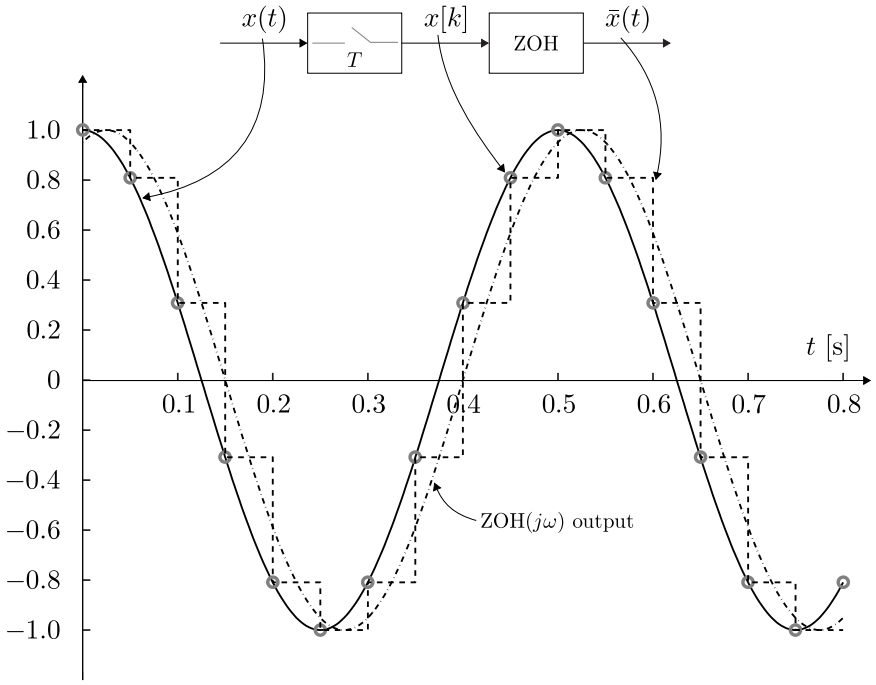


Figure G.4: Illustration of the time delay $\approx T/2$ introduced by a ZOH block when the sampling time is T, modeled by ZOH$(j\omega)$ in (G.4).

This page is intentionally left blank.

Appendix H

Source Code Organization

This appendix outlines the source code structure. As explained in Section 1.4.1, all of the source code is contained in the `~/material` directory where `~` represents the base directory of this thesis.

H.1 High Level Overview

The source code is split into three logical sections:

1. **Scripts**: located in `/` (see Section 1.4.1), this contains MATLAB code used for analysis, offline routines such as thrust calibration (Section 5.6.4) and scripts for generating the plots used in this report;
2. **Flight Software**: located in `/` (see Section 1.4.1), this contains C++ and C code compiled and run on-board the quadrotor to enable it to carry out its mission. This includes the GNC system, the autonomy engine and all supporting elements (camera drivers, AscTec HLP interface, etc.);
3. **Testing Software**: located in `/` (see Section 1.4.1), this contains C++ and Python code used for testing different features of the flight software, routines for controller tuning, routines for AprilTag noise analysis, the RotorS simulation, etc. This is software for testing the system, but is not part of the deployed flight software.

The sections below further develop the contents of the `/`, `/` and `/` directories.

H.2 Scripts

Upon first entry into the `/` directory, the user must execute `/init_dir.m` which configures MATLAB's path and loads external packages in `/external_packages/`. From here on, the user can execute any script in this directory, which is subdivided into logical groups of scripts:

- `/apriltag2_frequency`: bundle pose measurement runtime frequency analysis that is used in Section 3.1.3;
- `/apriltag2_noise_analysis`: AprilTag bundle pose measurement noise analysis. The base folder contains legacy code for real life noise analysis while `/apriltag2_noise_analysis/apriltag_gazebo` performs the simulated noise analysis of Appendix C;

- /bundle_calibration: implements the bundle pose calibration routine described in Section 3.1.4;
- /common: common functions reused by scripts in /;
- /control_debug: helper file for loading the controller info ROS topics useful for debugging control issues that may occur during a flight test;
- /control_design: controller design and analysis scripts used in Chapter 5;
- /inertia_tensor_id: inertia tensor identification used in Appendix D;
- /model_validation: model validation routine used in Section D.5;
- /motor_thrust_torque_id: motor thrust map, torque map and thrust dynamics identification used in Appendix D;
- /results_plots: results plots used in Chapter 7;
- /rls: scripts relating to the bundle pose RLS estimator of Section 3.1;
- /thrust_calibration: implements the thrust calibration algorithm of Section 5.6.4;
- /trajectory_generation: MATLAB version of the waypoint trajectory generator of Section 4.1.6.

H.3 Flight Software

H.3.1 Overview

The flight software in /src is subdivided into groups of guidance, navigation, control, autonomy engine, sensors and supporting software:

- /src/autonomy_engine: autonomy engine implementation of Chapter 6;
- /src/control: control implementation of Chapter 5 as well as AscTec HLP firmware and AscTec HLP interface. The controller itself is in /src/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control;
- /src/navigation: SSF;
- /src/guidance: guidance implementation of Chapter 4;
- /src/sensing: AprilTag core and bundle pose measurement code, camera image undistorter and sensor interfaces/drivers.
- /src/general: code that is reused by the above software groups;

H.3.2 Autonomy Engine

The /src/autonomy_engine directory consists of:

- /src/autonomy_engine/alure_takeoff: the takeoff autopilot of Section 6.2;
- /src/autonomy_engine/alure_mission: the mission autopilot of Section 6.3;
- /src/autonomy_engine/alure_landing: the landing autopilot of Section 6.4 and the emergency lander of Section 6.5;
- /src/autonomy_engine/alure_main_sm: the master state machine of Section 6.6.

H.3.3 Control

The /control directory consists just of /control/asctec_mav_framework. This is a heavily modified version of the original asctec_mav_framework Achtelik et al. [36]. Namely, the control software implements Chapter 5 and is located in /control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control. This is a self-contained Git repository that can potentially be implemented on any multirotor platform. The control software completely replaces that used in Achtelik et al. [36] and is called from /control/asctec_mav_framework/asctec_hl_firmware/sdk.c.

H.3.4 Navigation

The /navigation directory consists just of /navigation/jpl_sensor_fusion which contains our modified SSF algorithm based on the original SSF Weiss and Siegwart [78], Weiss et al. [77].

H.3.5 Guidance

The /guidance directory consists just of /guidance/alure_guidance, in which (besides several supporting classes):

- /guidance/alure_guidance/hover_point.h: implements the hover point trajectory of Section 4.1.5;
- /guidance/alure_guidance/waypoint_trajectory.h: implements the hover point trajectory of Section 4.1.6;
- /guidance/alure_guidance/transfer_trajectory.h: implements the transfer trajectory of Section 4.1.7;
- /guidance/alure_guidance/spiral_grid.h: implements the spiral grid search trajectory of Section 4.1.8;
- /guidance/alure_guidance/trajectory_list.h: implements the trajectory sequencer of Section 4.2;
- /guidance/alure_guidance/trajectory_tracker.h: implements the trajectory tracker of Section 4.3;

H.3.6 Sensing

The /sensing directory consists of:

- /sensing/alure_battery_monitor: battery rate of change of voltage estimator. Outputs the measured battery voltage $\hat{V}_{\text{bat}}$ (raw AscTec value), estimated $\frac{\mathrm{d}\hat{V}_{\text{bat}}}{\mathrm{d}t}$ and the energy flow direction (into battery, i.e. charging, or out of battery, i.e. discharging). Used by the master state machine (Section 6.6);
- /sensing/apriltags2_ros: AprilTag 2 ROS wrapper and implementation of bundle pose measurement (Section 3.1.3);
- /sensing/jpl_atan_camera_rectifier: raw downfacing camera image undistorter as used in Figure 3.1;
- /matrixvision_camera: downfacing camera driver and ROS interface as used in Figure 3.1;
- /rtk_publish: GPS ROS interface (used by SSF in Section 3.2).

H.3.7 General

The `/general` directory consists of:

- `/general/alure_comm`: ROS message, service and action definitions for the autonomy engine;
- `/general/alure_common`: general classes and utility functions for the autonomy engine and guidance. Includes the state machine implementation's base classes (Appendix B), classes that conveniently wrap communication with the AscTec HLP interface, classes that define certain useful entities (e.g. controller reference), classes that implement low-level functionality (e.g. a univariate sample buffer and a generic RLS estimator (Section 3.1.5)), etc.;
- `/general/alure_data_sync`: wraps `rsync_ros` McClung [129] for mission data synchronization with a ground computer;
- `/general/rsync_ros`: a dependency of `/general/alure_data_sync`;
- `/general/alure_logging`: implements local SQLite database read/write functionality as used in Chapter 6;
- `/general/alure_visualization`: implements visualization helper functions to draw the control reference and trajectory path in ROS rviz;
- `/general/glog_catkin`: a catkin wrapper for `glog` Google [162], which the autonomy and guidance software uses for message logging and necessary condition checking (e.g. obligatory user-defined ROS parameters);
- `/general/catkin_simple`: a dependency of `/general/glog_catkin`.

H.4 Testing Software

H.4.1 Overview

The testing software in `/` is subdivided into two groups:

- Generic testing software ROS packages which are placed directly in `/`;
- RotorS simulation-related packages in `/simulation_packages`. These implement the SIL simulation functionality described in Section 7.1.1.

H.4.2 Generic Testing Software

The generic testing software consists of:

- `/alure_control_standard_inputs`: allows sending step, sinusoid and sine sweep inputs to the controller in body rate (i.e. bypass translation, attitude control), acceleration, velocity or position control modes. Steps are useful for controller tuning (Appendix F) while sine sweeps can be used for Estimate Empirical Transfer Function (ETFE) identification. A sinusoid input may be useful for testing a particular frequency of the sine sweep;
- `/alure_landing_test`: implements the auto landing tester (Section 7.6);
- `/alure_trajectory_tracker_test`: a trajectory sequencer test node used during guidance subsystem refactoring;

- /apriltag_noise_data_collector: small Gazebo environment for simulated bundle pose measurement noise data collection (Appendix C);
- /asctec_cpu_load_calc: computes the average AscTec HLP Central Processing Unit (CPU) load. Was used when developing /control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control to ensure that the HLP has the computational capacity to run the new control implementation[1];
- /controller_tuning: useful scrips for controller tuning. Currently just a conversion script from the estimated quaternion attitude $\tilde{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{b}}$ and the desired quaternion attitude $\boldsymbol{q}_{\mathrm{e}}\tilde{\boldsymbol{q}}_{\mathrm{w}}^{\mathrm{b}}$ (Figure 5.12) to Tait-Bryan Euler angles;
- /load_cell_test_tools: scripts for working with the RCbenchmark [155] test stand as explained in Section D.3.

H.4.3 RotorS Simulation

The /simulation_packages directory consists of:

- /simulation_packages/rotors_simulator: a modified version of the original RotorS simulator for MAVs Furrer et al. [122], Autonomous Systems Lab [163]. Contains our autonomy engine and control interface[2];
- /simulation_packages/mav_comm: a (modified) dependency of /simulation_packages/rotors_simulator;
- /simulation_packages/alure_simulation: the central SIL simulation package. Contains launch files for launching the full SIL simulation (RotorS, autonomy engine, etc.), utility scripts for working with the SIL simulation and a (wind) disturbance force emulator[3] (Section 7.6.2);

[1] The current implementation uses about $\approx$ 10% CPU load while the original controller in ascctec_mav_framework Achtelik et al. [36] used $\approx$ 40%.

[2] /simulation_packages/rotors_simulator/rotors_control/include/rotors_control/jpl_multirotor_control_interface_node.h

[3] /simulation_packages/alure_simulation/include/alure_simulation/disturbance_force.h

This page is intentionally left blank.

Appendix I

Operation Tutorial

This appendix explains how to compile and operate the complete software suite developed in this thesis.

I.1 Compilation

This section details the steps to compile all of the software written in this thesis. The reader is recommended to first familiarize themselves with the source code organization by reading Appendix H. At a high level, the procedure is:

1. Install software dependencies;
2. Compile the flight software;
3. Compile the testing software;
4. Compile and flash the AscTec HLP firmware.

These instructions were tested on a fresh install of `x86_64 Ubuntu 14.04.5 LTS`. It is recommended that new readers execute **all** of the steps described below in order to minimize the chance of any problems. Returning readers that are already experienced with the software may use this appendix as reference for compiling only parts of the software.

I.1.1 Dependency Installation

ROS Indigo

The flight and testing software require ROS Indigo to be installed. For this, execute the following commands:

```
$ # Setup sources.list
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /
    etc/apt/sources.list.d/ros-latest.list'
$ # Set up keys
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421
    C365BD9FF1F717815A3895523BAEEB01FA116
$ # Update package index
$ sudo apt-get update
$ # Desktop-Full install of ROS Indigo
$ sudo apt-get install -y ros-indigo-desktop-full
$ # Initialize rosdep
$ rosdep init
```

```
$ rosdep update
$ # Add ROS environment variables automatically to any bash session
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Follow up ROS Indigo installation by installing Catkin Command Line Tools:

```
$ sudo apt-get install python-catkin-tools
```

ROS Dependencies

The flight software depends on some ROS packages:

```
$ # ssf_updates dependency for GPS
$ sudo apt-get install -y ros-indigo-gps-common
$ # Google glog logging library for ALURE software
$ sudo apt-get install -y libgoogle-glog-dev
```

The testing software depends on some additional ROS packages:

```
$ # RotorS simulator dependencies
$ sudo apt-get install -y protobuf-compiler ros-indigo-joy ros-indigo-octomap-ros ros
    -indigo-mav-msgs
```

Simulation Dependencies

The RotorS simulator's keyboard joystick functionality (to be able to control the quadrotor with a keyboard) depends on `python-uinput`:

```
$ # Install python-uinput (Python interface to Linux uinput kernel module)
$ cd /tmp
$ git clone https://github.com/devbharat/python-uinput
$ cd python-uinput
$ python setup.py build
$ sudo python setup.py install
$ modprobe -i uinput
$ # Configure device permissions
$ cd udev-rules
$ sudo cp 40-uinput.rules /etc/udev/rules.d
$ sudo addgroup uinput
$ sudo adduser $USER uinput
$ # Restart the machine for changes to take effect
$ sudo reboot
```

The keyboard joystick GUI additionally depends on `pygame`:

```
$ sudo easy_install pip
$ sudo -H pip install pygame
```

The `jstest` program helps to debug a joystick and to determine its port (see Section I.2.2):

```
$ sudo apt-get install joystick
```

Firmware Dependencies

Firmware compilation requires the `arm-elf-gcc` compiler:

```
$ cd /tmp
$ # Download arm-elf-gcc from AscTec
$ wget http://www.asctec.de/downloads/software/sdk/arm7-gcc-2012-03-28-bin.tar.bz2
$ # Extract arm-elf-gcc
$ mkdir arm-elf-gcc
$ tar -jxvf arm7-gcc-2012-03-28-bin.tar.bz2 -C arm-elf-gcc
$ # Install arm-elf-gcc
$ cd arm-elf-gcc/
$ sudo chmod +x install-arm7-toolchain.sh
$ sudo ./install-arm7-toolchain.sh
```

For a 64-bit (i.e. `x86_64`) version of Ubuntu, the following additional dependencies need to be installed:

```
$ sudo apt-get install -y libncurses5-dev:i386 libx11-dev:i386 zlib1g:i386
```

OpenOCD must be installed for flashing and debugging the AscTec HLP:

```
$ sudo apt-get install -y openocd
```

I.1.2 Flight Software Compilation

Camera Driver Installation

> The camera driver needs to be installed only when compiling on the Odroid XU4. If compiling the flight software on a desktop PC for use with the RotorS simulation (see Section I.2.2), this step should be skipped.

The downfacing camera is an mvBlueFOX-MLC200wG type Matrix Vision GmbH [21]. Open a new terminal and move to `/sensing/matrixvision_camera`. Install the driver as follows[1]:

```
$ cd mv_camera/mv_driver/
$ sudo ./install_mvBlueFOX.sh
[...]
Do you want to continue (default is 'yes')?
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
yes
Do you want to remove previous installation (default is 'yes')?
This will remove mvIMPACT Acquire for ALL installed Products!!!
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
yes
[...]
Do you want to install >wxWidgets< (default is 'yes')?
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
This is highly recommended, as without wxWidgets, you cannot build wxPropView.
n
[...]
Not installing wxWidgets

Do you want the tools and samples to be built (default is 'yes')?
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
n
[...]
```

[1] The `[...]` abbreviate superfluous output.

```
Do you want to copy 51-mvbf.rules to /etc/udev/rules.d for non-root user support (default
    is 'yes')?
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
yes

-------------------------------------------------------------------------------
                                Installation successful!
-------------------------------------------------------------------------------

Do you want to reboot now /default is 'yes')?
Hit 'n' + <Enter> for 'no', or just <Enter> for 'yes'.
yes
```

After the restart, again open a terminal in `/sensing/matrixvision_camera` and execute:

```
$ touch mv_camera/mv_driver/driver_installed
```

The downfacing camera driver is now installed.

ROS Package Compilation

> If senseless errors come up during `catkin build` that the reader is certain have been fixed, it may be due to CMake's cache which makes the builder unaware of the recent changes that have fixed the issue causing the build to break. A trick to fix this is to `rm -rf build/ devel/ logs/` in the software folder and to rerun `catkin build`.

Open a new terminal and move to `/../`, then execute:

- Handle the camera driver:

```
$ # If you did not compile the camera driver
$ catkin config --blacklist mv_camera
$ # If you did compile the camera driver
$ catkin config --no-blacklist
```

- Compile the flight software:

```
$ catkin build -j1
```

For interacting with the system through a groundstation computer the flight software must also be built on the groundstation computer.

I.1.3 Testing Software Compilation

Assume that the current directory is `/../`. The testing software is built with:

```
$ # Source the flight software, which contains dependency packages of the testing software
$ source ../alure_flight_software/devel/setup.bash
$ # Build the testing software
$ catkin build -j1
```

The testing software needs to only be built on an engineer's personal computer. It is not necessary to have the testing software on the quadrotor.

I.1.4 AscTec HLP Firmware Compilation And Flashing

Compilation

Assume that the current directory is ✈/. The HLP firmware is compiled as follows:

```
$ # Switch to the firmware directory
$ cd control/asctec_mav_framework/asctec_hl_firmware
$ # Compile the firmware
$ make clean
$ make
```

Simulink Controller Code Generation

⚠ Automatic code generation requires the Embedded Coder toolbox The MathWorks [164].

The control software implementing the low-level linear feedback controllers of Chapter 5 is located in ✈/control/asctec_mav_framework/asctec_hl_firmware/jpl_multirotor_control/matlab. Let CTRL_DIR denote this directory. Auto-generate C code for the translation and body rate core controllers as follows:

1. Open MATLAB and move to CTRL_DIR;
2. Run make_controllers;
3. Auto-generate C code for the core translation controller:
   (a) Open CTRL_DIR/simulink_models/translation_controller_core.slx;
   (b) In the Simulink menu panel, Code ▶ C/C++ Code ▶ Build Model (Ctrl+B). This generates the translation_controller_core_ert_rtw folder in the current working directory (we assume this is still CTRL_DIR/matlab);
   (c) Copy the controller source code to jpl_multirotor_control:

```
$ cd CTRL_DIR
$ cp matlab/translation_controller_core_ert_rtw/translation_controller_core.* .
```

   (d) In a text editor change the two occurrences of #include "rtwtypes.h" in the CTRL_DIR/translation_controller_core.h to #include "controller_types.h".
4. Auto-generate C code for the core body rate controller:
   (a) Open CTRL_DIR/simulink_models/body_rate_controller_core.slx;
   (b) In the Simulink menu panel, Code ▶ C/C++ Code ▶ Build Model (Ctrl+B). This generates the body_rate_controller_core_ert_rtw folder in the current working directory (we assume this is still CTRL_DIR/matlab);
   (c) Copy the controller source code to jpl_multirotor_control:

```
$ cd CTRL_DIR
$ cp matlab/body_rate_controller_core_ert_rtw/body_rate_controller_core.* .
```

   (d) In a text editor change the two occurrences of #include "rtwtypes.h" in the CTRL_DIR/body_rate_controller_core.h to #include "controller_types.h".

If only the translation or only the body rate controller is changed then naturally step 3 or step 4, respectively, may be skipped. Once these changes are made, re-compile the firmware as in the above "Compilation" section.

Flashing

Flashing instructions are provided by AscTec [165] and are replicated here for completeness. *Flashing* is the process of uploading the compiled firmware binary (`main.hex`) onto the AscTec HLP. The flashing steps are:

1. Turn on the AscTec AutoPilot board (attach a power source like a battery or a wall plug and flip its power switch);
2. Connect the female JTAG end of the AscTec JTAG cable into the male JTAG connection on the AutoPilot (see Figure I.1 and I.2) and the USB end into your personal computer;
3. Check that the HLP is plugged in and is recognized (the second line is output, which should be similar):

```
$ ls /dev/ttyUSB*
/dev/ttyUSB0
```

4. Create a new terminal window and move to `/control/asctec_mav_framework/asctec_hl_firmware`. Then run:

```
$ sudo openocd -f lpc2xxx_asctecusbjtag05.cfg
```

   If the output contains warnings and errors, it was empirically found that unplugging the USB-end of the JTAG cable and plugging it back in, then executing the above command again, always fixes the issues. When successful, the command output should be similar to:

```
Open On-Chip Debugger 0.5.0 (2011-08-26-10:27)
Licensed under GNU GPL v2
For bug reports, read
 http://openocd.berlios.de/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
5 kHz
RCLK - adaptive
adapter_nsrst_delay: 200
jtag_ntrst_delay: 200
trst_and_srst srst_pulls_trst srst_gates_jtag trst_push_pull srst_open_drain
fast memory access is enable
ddcc downloads are enabled
Info : RCLK (adaptive clock speed) not supported - fallback to 500 kHz
Info : JTAG tap: lpc2148.cpu tap/device found: 0x4f1f0f0f (mfg: 0x787, part: 0xf1f0,
    ver: 0x4)
Info : Embedded ICE version 4
Info : lpc2148.cpu: hardware has 2 breakpoint/watchpoint units
```

5. Create a new terminal window and move to `/control/asctec_mav_framework/asctec_hl_firmware`. Then open a telnet connection to OpenOCD (lines 2 and onwards are output):

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

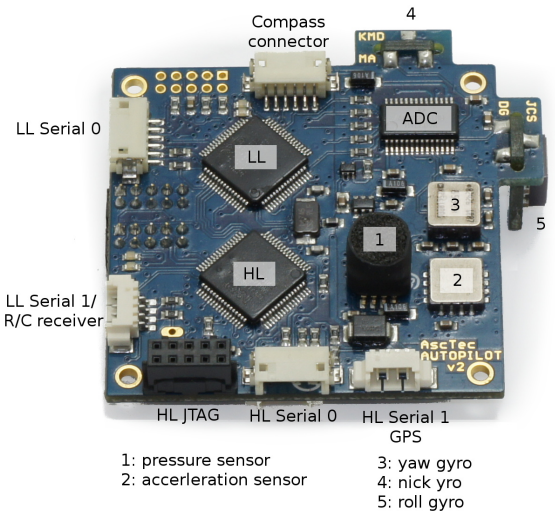


Figure I.1: AscTec AutoPilot Board pinout. *Photo source AscTec [166].*

6. In the telnet terminal, execute the flashing commands (only lines with > are commands, the rest is output):

```
> reset halt
JTAG tap: lpc2148.cpu tap/device found: 0x4f1f0f0f (mfg: 0x787, part: 0xf1f0, ver: 0
    x4)
srst pulls trst - can not reset into halted mode. Issuing halt after reset.
target state: halted
target halted in ARM state due to debug-request, current mode: User
cpsr: 0x00000010 pc: 0x0000d66c
> flash write_image erase main.hex
auto erase enabled
Verification will fail since checksum in image (0xe1a00000) to be written to
flash is different from calculated vector checksum (0xb8a06f58).
To remove this warning modify build tools on developer PC to inject correct LPC
vector checksum.
wrote 196608 bytes from file main.hex in 8.103316s (23.694 KiB/s)
> reset run
JTAG tap: lpc2148.cpu tap/device found: 0x4f1f0f0f (mfg: 0x787, part: 0xf1f0,
ver: 0x4)
> exit
Connection closed by foreign host.
```

That's it, the firmware is now flashed onto the HLP.

I.2 Operation

Now that the reader knows how to compile the software (Section I.1), this section describes how to operate it.

I.2.1 Real Flight Test

This section explains how to run the software suite for a flight test.

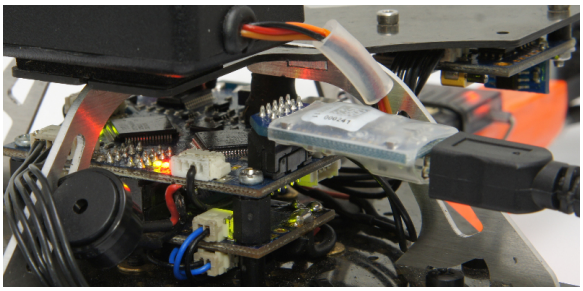


Figure I.2: AscTec AutoPilot JTAG connection. *Photo source AscTec [165].*

Configuration Files

The following configuration files are available to the user for configuring the system at start-up:

1. `fcu_parameters.yaml`[2]: default topic rates from the HLP;
2. `new_pos_ctrl_parameters.yaml`[3]: translation, attitude and emergency landing controllers' parameters (Chapter 5);
3. `new_att_ctrl_parameters.yaml`[4]: body rate controller parameters (Section 5.6);
4. `quadcopter_parameters.yaml`[5]: flight dynamic model parameters from Section 5.5 (including thrust calibration from Section 5.6.4);
5. `general_parameters.yaml`[6]: glog logging directory and parameters for trajectory visualization in ROS rviz;
6. `pelican_parameters.yaml`[7]: camera horizontal and vertical FOV;
7. `sensor_parameters.yaml`[8]: parameters for the battery voltage rate of change RLS estimator (Section H.3.6);
8. `data_sync_settings.yaml`[9]: source and target directories for mission data synchronization;
9. `storage_parameters.yaml`[10]: SQLite database file path;
10. `apriltags2_parameters.xml`[11]: core AprilTags 2 algorithm parameters Wang and Olson [55];
11. `tags.yaml`[12]: AprilTag landing bundle calibration definition (Section 3.1.4);

[2] `/control/asctec_mav_framework/asctec_hl_interface/launch/fcu_parameters.yaml`
[3] `/control/asctec_mav_framework/asctec_hl_interface/launch/new_pos_ctrl_parameters.yaml`
[4] `/control/asctec_mav_framework/asctec_hl_interface/launch/new_att_ctrl_parameters.yaml`
[5] `/control/asctec_mav_framework/asctec_hl_interface/launch/quadcopter_parameters.yaml`
[6] `/general/alure_common/config/general_parameters.yaml`
[7] `/general/alure_common/config/pelican_parameters.yaml`
[8] `/general/alure_common/config/sensor_parameters.yaml`
[9] `/general/alure_data_sync/config/data_sync_settings.yaml`
[10] `/general/alure_logging/config/storage_parameters.yaml`
[11] `/sensing/apriltags2_ros/apriltags2_ros/config/apriltags2_parameters.xml`
[12] `/sensing/apriltags2_ros/apriltags2_ros/config/tags.yaml`

12. `landing_parameters.yaml`[13]: landing autopilot and emergency lander one-time configuration parameters (Sections 6.4 and 6.5);
13. `takeoff_parameters.yaml`[14]: takeoff autopilot one-time configuration parameters (Section 6.2);
14. `mission_parameters.yaml`[15]: mission autopilot one-time configuration parameters (Section 6.3);
15. `mission_master_parameters.yaml`[16]: master state machine one-time configuration parameters (Section 6.6).

Operation

With the configuration files correctly filled out, the system may be started for a test flight. It is assumed that in every new terminal the flight software directory is sourced (`source devel/setup.bash` in `/../`). The system is then started as follows:

1. In a new terminal:

```
$ roscore
```

2. In a new terminal, start the downfacing camera driver:

```
$ roslaunch mv_camera camera.launch
```

3. In a new terminal, start the camera undistorter:

```
$ roslaunch jpl_atan_camera_rectifier atan.launch
```

4. In a new terminal, start the AprilTag bundle pose sensor:

```
$ roslaunch apriltags2_ros continuous_detection.launch
```

5. In a new terminal, verify that the quadrotor time is synced with the ground station (necessary only when using RTK GPS or VICON):

```
$ # Set the groundstation IP address in /etc/chrony/chrony.conf
$ # Then run:
$ chronyc tracking
$ # If 'System time' reading is small, stop here — time is synced.
$ # If 'System time' is large or IP address was incorrect, run:
$ sudo service chrony restart
$ chronyc
chrony version 2.1.1
Copyright (C) 1997−2003, 2007, 2009−2015 Richard P. Curnow and others
chrony comes with ABSOLUTELY NO WARRANTY.  This is free software, and
you are welcome to redistribute it under certain conditions.  See the
GNU General Public License version 2 for details.

chronyc> password 1234
chronyc> makestep
```

[13] `/autonomy_engine/alure_landing/config/landing_parameters.yaml`
[14] `/autonomy_engine/alure_takeoff/config/takeoff_parameters.yaml`
[15] `/autonomy_engine/alure_mission/config/mission_parameters.yaml`
[16] `/autonomy_engine/alure_main_sm/config/mission_master_parameters.yaml`

```
16 chronyc> exit
17 $ chronyc tracking # Make sure 'System time' if very small, if not repeat from line 6
```

6. In a new terminal, start the AscTec HLP interface:

```
1 $ roslaunch asctec_hl_interface fcu.launch
```

7. Start the global sensor interface

   - If running an outdoor test, in a new terminal start the GPS interface:

   ```
   1 $ rosrun rtk_publish parser device:=/dev/ttyUSB0
   ```

   - If running an indoor test, in a new terminal start the VICON bridge on the groundstation computer Achtelik and Hansen [167] (which must be available on the groundstation PC connected to VICON, it is not part of the flight or testing software):

   ```
   1 $ roslaunch vicon_bridge vicon.launch
   ```

8. In a new terminal, start SSF

   - If running an outdoor test:

   ```
   1 $ roslaunch ssf_updates magnetometer_position_sensor.launch
   ```

   - If running an indoor test:

   ```
   1 $ roslaunch  ssf_updates vicon_pose_sensor.launch
   ```

9. In a new terminal, start ROS rqt:

```
1 $ rqt
```

   Open Plugins ▶ Configuration ▶ Dynamic Reconfigure. In the SSF tab (`pose_sensor` indoors, `position_sensor` outdoors) click `set_scale`. It is important that the state estimator is initialized before the next step (starting the autonomy engine) as, on start-up, the latter requires the downfacing camera-to-IMU calibration which SSF publishes once it is initialized;

10. In a new terminal, start the autonomy engine:

    - If running an outdoor test:

    ```
    1 $ roslaunch alure_common control_system.launch
    ```

    - If running an indoor test:

    ```
    1 $ roslaunch alure_common control_system.launch vicon:=true
    ```

At this point, all necessary components of the system are running. With the quadrotor positioned on the charging pad, the autonomy engine is operated as follows:

1. Interaction with the autonomy engine happens through ROS service calls to the master state machine (Section 6.6). By default, the master state machine is paused i.e. it will not parse ROS service calls. Start it via:

```
1 $ rosservice call /alure/start_experiment "sm_play_action: 0"
```

2. In principle, the system is from here on full-cycle autonomous as described in Chapter 6 so no further actions are required from the operator. Nevertheless, functionality is provided to force state transitions in the master state machine (which is very useful for flight testing):

   - Invoke the PAUSE event (Section 6.6) via:

   ```
   $ rosservice call /alure/sm_manager "command: 0"
   ```

   - Invoke the FORCETAKEOFF event (Section 6.6) via:

   ```
   $ rosservice call /alure/sm_manager "command: 1"
   ```

   - Invoke the FORCELANDING event (Section 6.6) via:

   ```
   $ rosservice call /alure/sm_manager "command: 2"
   ```

   - Invoke the EMERGENCYLAND event (Section 6.6) via:

   ```
   $ rosservice call /alure/sm_manager "command: 3"
   ```

The above instructions are complete for starting the full system and interacting with the autonomy engine through the nominal interface of the master state machine. Some other functionalities are listed below:

- The dynamic reconfigure GUI in rqt allows the user to also set the downfacing camera parameters. Because the AprilTag bundle pose sensor runs at $\approx 7\,\text{Hz}$ (Figure 3.5), the camera frame rate is typically set to $10\,\text{Hz}$. For automatic exposure setting in variable lighting conditions, `auto_shutter` is enabled;

- The dynamic reconfigure GUI in rqt allows to modify the controller parameters whose defaults are set in configuration files 1-4;

- The phase-specific state machines of the autonomy engine may be activated manually (bypassing the master state machine) via a terminal call to the topic that their actionlib Marder-Eppstein and Pradeep [151] interface provides. For example, the landing autopilot can be triggered directly via:

```
$ rostopic pub -1 /alure/landing_action/goal alure_comm/
    ActionBasedStateMachineActionGoal "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
goal_id:
  stamp:
    secs: 0
    nsecs: 0
  id: ''
goal: {}"
```

  Similar calls can be issued to the takeoff, mission and emergency lander state machines.

I.2.2 RotorS Simulation

As explained in Section 7.1.1 and Appendix H, the testing software provides a SIL simulation of the GNC system and autonomy engine (except for SSF). This section explains how to operate the SIL simulation.
Assume that the current directory is `/../`. The SIL simulation is run like so:

```
$ # Source the testing software directory
$ source devel/setup.bash
$ # Run a simulation with the outdoor charging pad setup using:
$ roslaunch alure_simulation simulation_outside.launch
$ # Run a simulation with the indoor foam pad landing bundle using:
$ roslaunch alure_simulation simulation_vicon.launch
```

From this point on, the autonomy engine can be interacted with in exactly the same way as in a real flight test (Section I.2.1, except for the 10 "starting system" steps which are replaced by the above `simulation_*.launch` files). Note that all configuration files except for `fcu_parameters.yaml` continue to apply. The SIL environment has two specialties described below.

Wind Plugin

There is a wind plugin that can be activated in `simulation_*.launch` via the `wind` argument (used for Section 7.6.2).

Keyboard Joystick

To be able to use the keyboard for controlling the quadrotor, one must first determine the correct joystick number. With the simulation running, open a new terminal and execute:

```
$ sudo jstest /dev/input/js <TAB> <TAB>
js0 js1
```

Where the last keyboard TAB-TAB prints out something similar to `js0 js1`. Check all the printed `js*`. The correct one should produce a similar output to the following:

```
$ sudo jstest /dev/input/js1
Driver version is 2.1.0.
Joystick (python-uinput) has 4 axes (X, Y, Throttle, Rudder)
and 4 buttons (Btn0, Btn1, Btn2, Btn3).
Testing ... (interrupt to exit)
Axes:  0:     0  1:     0  2:-32767  3:-32767 Buttons:  0:on   1:off  2:off  3:off
```

With the joystick GUI window active, the `Axes:...` values should change as you press the `W A S D` and arrow keys. Supposing that `jsX` is the correct joystick, modify the `dev` parameter of `joy_node` to this value in `alure_outside.launch`[17] and `alure_vicon.launch`[18].
With the joystick correctly configured, it can be used as follows:

- `W`/`S` keys are for throttle up/down;
- `A`/`D` keys are for positive/negative yaw;
- Up/down keys are for positive/negative pitch;

[17] `/simulation_packages/rotors_simulator/rotors_gazebo/launch/alure_outside.launch`
[18] `/simulation_packages/rotors_simulator/rotors_gazebo/launch/alure_vicon.launch`

- Right/left keys are for positive/negative roll;
- `1/2/3` keys are for acceleration, velocity and position modes of the translation controller (Section 5.8);
- `M` key toggles the motors on/off (default motor state is off when the simulation is launched).;
- `E` key toggles the emergency landing controller of Section 5.9.

This page is intentionally left blank.