Doctoral Thesis

Dynamic Robot Architecture for Robust Realtime Computer Vision

Author(s):
Meier, Lorenz

Publication Date:
2017

Permanent Link:
https://doi.org/10.3929/ethz-a-010874068

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Dynamic Robot Architecture for Robust Realtime Computer Vision

A thesis submitted to attain the degree of DOCTOR OF SCIENCES of ETH Zürich

(Dr. sc. ETH Zürich)

presented by

Lorenz Meier

MSc ETH in Computer Science
ETH Zürich

citizen of Germany

accepted on the recommendation of

Prof. Marc Pollefeys
Prof. Roland Siegwart
Prof. Davide Scaramuzza

2017
Abstract

This dissertation presents a novel and complete system architecture for dynamic systems using onboard computer vision. The architecture is derived from port-based automata in a sound theoretical framework and implemented in Linux and real-time operating systems. It improved over the existing state of the art by combining real-time robot control and high-performance computer vision in one architecture. The software framework supports deeply embedded devices like digital signal processors and central processing units. The communication architecture scales from basic sensor data to full camera frames. It networks low-level and high-level processors in a hybrid system and allows to run port-based objects on either core concurrently and communicating through a publish-subscribe framework. The dissertation leverages this base architecture to develop fundamental design patterns for robust realtime computer vision on resource constrained flying systems. A major unsolved challenge for obstacle detection for micro air vehicles are power-lines and other types of suspended wires. As the dynamics of a micro air vehicle are undamped, any loss of position tracking in the vision pipeline can lead to a catastrophic failure. We account for this with a tightly integrated software-hardware design for robot localization. Robust vision is equally important for obstacle avoidance. We contribute a failure prediction and fusion algorithm for stereo which enables the fusion of two stereo pairs. This contribution allows to reliably detect power-lines, which are invisible to traditional stereo setups. The dissertation contributes overall a holistic system architecture for real-time and robust computer-vision and control.
Zusammenfassung

Acknowledgement

I would like to thank my advisor Marc Pollefeys for his ongoing and exceptional support and trust through my masters and PhD. Marc has given me an exceptional level of academic freedom to explore and resources to execute successfully. He has encouraged me to take unbeaten paths and provided all his trust even if that exploration sometimes didn’t yield results in the short term. Friedrich Fraundorfer has equally supported me in creating the Pixhawk project and student team and has been instrumental in enabling me. I would also like to thank Roland Siegwart for welcoming me in his lab even before starting my masters and continuing to support my path throughout my studies and PhD. Raffaello d’Andrea has been an important mentor since I started my studies at ETH and I’m deeply grateful for his ongoing advice. I would also like to thank Davide Scaramuzza for our collaboration since the early days of my PhD during the sFly micro air vehicle swarm research project and the many fruitful discussions at ETH and later UZH and his ongoing support for the open source platform. I have learnt a tremendous amount about embedded system architecture and embedded programming from Michael Smith. His advice and contributions have been key to successfully apply the architectural contributions to the deeply embedded domain. James Goppert has been a collaborator since very early on and contributed key estimation, control and math libraries to the systems I created throughout my PhD. Thomas Gubler, Julian Oes, Laurens Mackay and Martin Rutschmann have helped to create the first generation of the PX4 implementation and have been critical in getting a lot of the ar-
chitectural concepts implemented. Roman Bapst has created the support for vertical takeoff planes (VTOL) in the system and proved with it the fundamental benefits of the architecture derived in this dissertation. Beat Küng has influenced the system architecture in very important dimensions and did cover so much of the ongoing system evolution that I was able to focus more on the architectural level. Very special thanks go to my research engineer Vilhjálmur Vilhjálmsson who has allowed me to focus during the last six months of my PhD purely on theoretical contributions and has taken away a tremendous load of software engineering and debugging. I would like to thank Chris Dobler for his excellent hardware design for Pixhawk which allowed many scientific experiments. He even exported his hardware to Stanford, MIT, CMU and UPenn and enabled many PhD students around the globe to publish using high-end onboard hardware that was easy to use. I have enjoyed working with Dominik Honegger since 2009 since the first Pixhawk team and our collaborations on almost all papers I published have been critical for the success of this thesis. Petri Tanskanen and Tobias Nägeli for the great collaboration over so many years on micro air vehicles and 3D reconstruction on drones. Gim-Hee Lee, Lionel Heng, Fabian Kaiser, Paul Riseborough and Otmar Hilliges have also been important collaborators which were critical to the success of this thesis. A very special thanks to Markus Achtelik for a fantastic collaboration across lab borders and for being instrumental in making ETH more interdisciplinary and different labs work more intensely together. Robin Ritz has enabled our venture into the VTOL domain with Raffaello d’Andrea and has been extremely supportive. I would like to thank Andrew Tridgell for all the great technical discussions, great collaboration on multiple protocols and codebases and all his contributions he has made. I would also like to thank him for being persistent in raising GIT workflow issues and advocating best practices in project maintenance. Chris Anderson has been a great supporter and advocate of open source in the drone space and has been instrumental in explaining the benefits of open source to the public. The wide adoption of open source on drones would have been impossible without him. David Sidrane has in the last years been absolutely critical for me to not get overwhelmed with maintaining the open source project and has put out every fire. Gregory Nutt has created with NuttX a convenient base firm real-time operating system
that made focusing on a high-level architecture easy through relying on the POSIX standard. Remo Masina, Christoph Dunkel, Philip Rowse, Jerome Perin, Kevin Alvarez, Phillip Kocmoud and Nick Arsov have been great collaborators on open source hardware and have helped to create a complete hardware ecosystem in support of the open source software created in this dissertation. I would like to thank Antoine Drouin and Pascal Brisset for effectively starting open source work on micro air vehicles over a decade ago. I would also like to thank all members of the open source and open hardware community I have not specifically mentioned for the great collaboration and learning experience they provided. I would also like to thank all students and lab-mates I have not explicitly mentioned for their collaboration and support.

This dissertation would be have been impossible without the unconditional support of my fiancée Sabrina Kaiser. She has endured all ups and downs a PhD thesis commonly yields. Without her support doing academic research and maintaining a major open source project would have been impossible.

I would like to thank our industry collaborators and funding agencies: ETH Zurich Foundation for the two fellowships it has provided, the EU Horizon 2020 program for its funding, Google ATAP / Project Tango for its funding on work on mobile phones which was not included in the dissertation, Amazon Prime Air for their undirected gift in support of research on obstacle avoidance, Qualcomm Technologies for their QInf fellowship and later contributions to PX4, the Dronecode Foundation and Linux Foundation for giving PX4 a home.

And most importantly I would like to thank my parents for their support, advice and backing over so many years and my dad for convincing me to not study politics but switch last minute to computer science. Concluding my PhD is the end of a long journey through the education system and getting to this point would have been impossible without their exceptional support and encouragement over so many years.
Contents

I. Overview 1

1. Introduction 3
   1.1. Preface 3
   1.2. Motivation 4
   1.3. Contributions 4
       1.3.1. Dynamic Robot Architecture 5
       1.3.2. Software Reusability 5
       1.3.3. Open Innovation in Aerial Robotics 6
       1.3.4. Real-time and Robust Vision for Localization and Avoidance 7

2. Foundations 9
   2.1. Previous Work 9
       2.1.1. Architecture 9
       2.1.2. Design Process 10
       2.1.3. Systems 10
   2.2. System Architecture Foundations 11
       2.2.1. Software Components 11
       2.2.2. Design Patterns 13
       2.2.3. Software Reusability 14
       2.2.4. Design Process 15
## Contents

2.2.5. Specification ........................................... 15  
2.2.6. Certification ......................................... 16

3. **Open and User Innovation** .................................. 21  
3.1. Lead User Innovation ...................................... 22  
3.2. Sustainability of Open Source Development .............. 23  
3.3. Scale and Impact .......................................... 24  
3.3.1. Academic Impact ....................................... 25  
3.3.2. Industrial Impact ...................................... 26

II. **Dynamic Robot Architecture** ................................ 27

4. **Architecture** .................................................. 29  
4.1. Port Based Objects ......................................... 29  
4.1.1. Dynamic Reconfiguration ............................... 30  
4.1.2. Task Set ................................................. 31  
4.1.3. Architecture Verification ............................... 31  
4.1.4. Configuration ........................................... 32  
4.2. Software Reuse .............................................. 34  
4.3. A Hybrid System Design .................................... 36  
4.4. Micro Air Vehicle and Middleware ....................... 42  
4.4.1. Flight and Processing Electronics .................... 43  
4.5. Aerial Middleware and Message Format .................. 45  
4.6. System Architecture ....................................... 47  
4.6.1. Vision-IMU Synchronization ............................ 49  
4.7. Visual Position Estimation ................................ 52  
4.7.1. ARTK+ Localization .................................... 52  
4.7.2. Vision-IMU fusion ...................................... 52  
4.7.3. Outlier Removal ........................................ 55  
4.8. Stereo Obstacle Detection ................................. 56  
4.8.1. Point Clouds from Stereo .............................. 56  
4.9. State Estimation and Control .............................. 58  
4.9.1. Discrete Kalman Estimation ............................ 58  
4.9.2. Position and Attitude Control ......................... 60
4.10. Computer Vision Timing ............................................. 60
  4.10.1. Stereo Head ..................................................... 61
  4.10.2. Mechanical Structure and Flight Time .................... 61
4.11. Experiments and Results ........................................... 62
  4.11.1. Visual localization ............................................. 63
  4.11.2. Stereo Obstacle Detection .................................... 68
4.12. Summary ............................................................. 72

5. Dynamic Systems ..................................................... 73
  5.1. Related Work ....................................................... 74
  5.2. Design Criteria for Deeply Embedded Systems ............... 75
    5.2.1. Controller Timing Requirements ............................ 76
    5.2.2. Reusability .................................................... 76
    5.2.3. Interoperability .............................................. 76
  5.3. Software Architecture ........................................... 78
    5.3.1. $\mu$ORB Middleware ........................................... 79
    5.3.2. Applications .................................................. 80
    5.3.3. Work Queue and HRT Callbacks ............................. 81
    5.3.4. Companion Computer ......................................... 81
  5.4. Implementation .................................................... 82
    5.4.1. Driver Layer .................................................. 83
    5.4.2. Hardware ....................................................... 83
  5.5. Results .............................................................. 87
    5.5.1. Performance ................................................... 87
    5.5.2. Reusability .................................................... 89
    5.5.3. Flexibility ..................................................... 90
  5.6. Summary ............................................................. 91

III. Dynamic Vision & Navigation ..................................... 93

6. Reliable Localization ................................................ 95
  6.1. Related Work ...................................................... 95
6.2. Fundamental Design Considerations for Robust Vision  
6.2.1. Sampling Strategies for Vibration and Lighting Variation  
6.2.2. Texturedness Prior  
6.2.3. Time Constraints  
6.3. Resulting Architectural Considerations for the Vision Pipeline  
6.4. Computer Vision in Discrete Hardware  
6.4.1. Background  
6.4.2. System Setup  
6.4.3. Flow Computation and Results  
6.4.4. Outdoor Test  
6.5. Summary  
7. Robust Vision  
7.1. Related Work  
7.2. Epipolar Geometry  
7.2.1. Fundamental limitations  
7.2.2. Realtime stereo in hardware  
7.3. Setup  
7.4. Failure Heuristic  
7.4.1. Fault prediction  
7.4.2. Final depthmap fusion and quality score thresholding  
7.5. Results  
7.5.1. Indoor Experiment  
7.5.2. Outdoor Urban Experiment  
7.5.3. Outdoor Powerlines Experiment  
7.5.4. Outdoor Brush Experiment  
7.6. Summary  
IV. Summary  
8. Conclusion  
9. Future Work
Contents

Bibliography 139

xv
Part I.

Overview
Chapter 1.

Introduction

1.1. Preface

At the beginning of the term of this PhD the word drone did not describe a small flying camera available in every retail electronics store. Instead it did exclusively refer to military remotely piloted aircraft (RPAS). What is known today as drone was referred to as micro air vehicle (MAV), a technical term coined 20 years ago and still used in technical literature. However, the general public and press refer to micro air vehicles exclusively as drones today.

The fact that the terminology has completely changed underlines how fundamentally this field has evolved and consolidated. Drones are today a significant industry and "drone software engineer" is a standard technical term like "embedded software engineer". In line with this, the first generation of hardware and software used in this thesis has been completely replaced. In fact this thesis covers two completely different implementations of the same software architecture. The current-generation implementation is used as middleware in thousands of drones. It underlines the validity of the architectural contributions as they successfully transitioned from early research systems into the gold standard of the drone industry.
1.2. Motivation

The starting point for this PhD thesis was the insight that small, lightweight, power-efficient sensors such as cameras are an optimal solution for flying vehicles. Active sensors like scanning laser range finders (LIDAR) had been successfully used on MAVs already. While the initial focus was purely on cameras, it quickly started to include the complete avionics package including the drone autopilot. This resulted from a general lack of availability of commercial-off-the-shelf solutions that were a match in terms of size, price point and quality. The combination of a real-time autopilot with a Linux computer for computer vision led to a hybrid system architecture which is what most products in the market employ.

1.3. Contributions

This PhD thesis has not only explored core concepts of a dynamic system architecture for robots, but also has profoundly influenced the whole drone industry by establishing architectural design patterns and a concrete reference implementation across multiple software components. PX4 is today the standard solution adopted by Dronecode, a non-profit organization administered by Linux Foundation to foster the use of open source software on flying vehicles. Apart from this very broad but short-term impact this thesis contributes fundamental architectural design patterns in the theoretical domain which are independent of today’s processor architectures or implementations. These design patterns are the result of a fundamental system analysis in terms of the specific communication and timing requirements of a port-based, reconfigurable software architecture for robotics. We contribute a novel port-based dynamic robot architecture which scales from microcontrollers to Linux systems. This architecture enables the re-use of software across different vehicles and robot types, fostering more collaboration between different research groups and industry. This enabled open innovation in the micro air vehicle research area. Finally we contribute tightly coupled and robust computer vision algorithms for localization and obstacle detection. These leverage the architecture we have established previously to
provide highly efficient algorithms for embedded vision applications.

1.3.1. Dynamic Robot Architecture

The first generation middleware, MAVCONN, has explored the hybrid system design pattern that is now the de-facto robot architecture on drones. It consists of a microcontroller performing the time-critical computations in a firm real-time setup and a high-level commodity Linux system with a standard kernel performing computer vision in a soft real-time setup. Despite various other architectures and attempts by the development community to move more of this onto a realtime Linux base it remains the prevailing design pattern in the industry. While MAVCONN was a loosely coupled system with only low-level tasks being performed on the microcontroller and without concurrent processing on the microcontroller, PX4 is a tightly coupled hybrid system and makes good on the promise of implementing a complete port-based automaton architecture on the complete system. This work builds on the theoretical frameworks derived by Steenstrup, Arbib and Manes [1] and Stewart [2] for the static system and adds communication bandwidth as an additional constraint in direct competition to response time. It then proposes a hybrid system architecture to resolve the combination of communication bandwidth and response time.

1.3.2. Software Reusability

This thesis shows that a robotics framework can be designed to allow for a very high level of software reusability in robotic vehicles. While earlier work mostly focused on generic robot architectures [2, 3] this thesis goes one level higher. It identifies common task sets which form minimum-viable subsystems such as the drivers plus attitude estimator to provide an attitude estimate of the vehicle. The validity of this concept and the strengths of the dynamic reconfigurability are shown with the vertical-takeoff-and-landing vehicle which reconfigures at runtime between a multi-rotor and plane.
1.3.3. Open Innovation in Aerial Robotics

The efforts in this thesis have contributed in turning ETH Zurich into a lighthouse for open innovation on drones. The main open source contributions of the thesis are today maintained by Dronecode, a non-profit organization fostering the further adoption of open source in drones. The open source community around the Pixhawk open hardware platform and the PX4 open source platform is the largest industry-backed development community in the drone space.

Since the technical implementation of this thesis has been open source with external contributions the question of authorship might arise, as in any collaborative academic work. Thankfully the GIT revision control system offers excellent and very detailed tracking of each author’s activity. While it is not perfect, it allows orders-of-magnitude assessment of contributions. The command

```
git shortlog -s -n
```

prints for revision

```
a701feefcf69c540d6328e2f7634e4e98e725373f
```

8013 commits for the author with the second ranking author having 1518 commits end of December 2016. Together with the system architecture published in multiple publications [4, 5, 6] it is trivial to show that performing academic research collaboratively with a global open source community does have great potential to improve the outcome. And it also provides very solid statistics which allow to attribute authorship much more quantitatively than usually possible in academia. All other authors combined do have of course a significant contribution which was essential for the outcome. However, this thesis covers the architectural foundations of the published source code and is not concerned with the actual implementation itself. The contribution therefore lies in the derived architecture, not its detailed implementation.
1.3. Contributions

1.3.4. Real-time and Robust Vision for Localization and Avoidance

This thesis contributes a number of fundamental design patterns in computer vision which provide more robust estimates. The improvement is achieved by a combination of a-priori measures such as implementing the processing on dedicated hardware or using fixed-time approaches, faster sample rates and inertial sensing. This generally makes the result less sensitive to the image content. We contribute in addition a novel approach to evaluate stereo matching for potential systematic error and propose a scoring approach that enables the fusion of two orthogonal stereo heads into one significantly more robust depthmap. We show that this improves the results on average scenes and leads to less outliers. For scenes which are problematic for stereo such as powerlines we show that this changes completely incorrect results into a highly accurate depthmap with very low noise. As the stereo processing is in hardware and the post-processing lightweight the approach is suitable for real-time implementation and runs on resource-constrained embedded hardware.
Chapter 2.

Foundations

2.1. Previous Work

This section only contains previous work which is general for system architecture and suitable as an introduction to the field. Each chapter contains where appropriate its own previous work section with much more detailed references.

2.1.1. Architecture

System architecture has always been a relevant area of research. The beginnings of contemporary architectural designs can be traced back to theoretical computer science in the early 1980’s. The concepts presented as theorems are today building the backbone of robot communication. Robotics architecture is as a research field as old as robotics itself. Initially it has been a topic for theoretical computer science when Steenstrup, Arbib and Manes introduced their concept of port automata in their 1983 paper [1]. This work is particularly interesting as it formally describes the full set of automata concepts required to describe an asynchronous publish-subscribe implementation. They focus entirely on analyzing the formal properties of
the automata but are concerned yet with communication latencies or locking. A more robot focused [7] Stewart analyzed robotics and sensor system architectures in his PhD thesis [2]. He did build on the port automata concept in much more concrete robotics concept. Kortenkamp and Simmons [8] provide an overview of the state of the art in 2008 which is architecturally close to contemporary robotic systems and introduce many of the concepts and building blocks still valid today.

### 2.1.2. Design Process

Coste-Maniere and Simmons [9] break down the process of defining a robotics system architecture into specification, execution and validation. Specification includes formal languages and tools where available. Execution can be with deterministic and real-time schedulers. Validation is typically done on unit tests and so-called black box tests. One of the most interesting observations from their paper is that the systems they surveyed in 2000 are not used any more today. This speaks to the fact that robotics as a field is still evolving rapidly and has not a settled design pattern yet, allowing further contributions.

### 2.1.3. Systems

The Object Management Group (OMG) Data Distribution Service (DDS) [10] is an industry standard for publish-subscribe communication in real-time applications. It is not a complete robotics middleware as it restricts itself to the communication layer, but it remains over a decade after its introduction the gold standard for messaging. LCM [11] is a lightweight alternative for high-rate messaging written for the DARPA Urban Challenge. CLARAty [12] and CARMEN [13] were first generation general-purpose robotics toolkits which found use in individual universities. GENOM [14] is a robotics toolkit used intensely for research on mars rovers. The main difference between them and ROS [3] is that ROS was developed in an open innovation model: The design is not just fulfilling the needs of one lab or company, but has been intended as a general toolkit and is co-developed
2.2. System Architecture Foundations

with a global development community. The Open Source Robotics Foundation has since continued this in ROS2, which interestingly mostly carries on with the development community model but completely replaces the technological base [15].

2.2. System Architecture Foundations

This section introduces the most relevant fundamental concepts used in this thesis. It covers fundamental theoretical concepts as well as more applied concepts required to understand the implementation and experiments.

2.2.1. Software Components

Software components are abstract software concepts. They are generally not implementation specific and can be applied to different implementations and architectures.

Port Based Objects

Port based objects in the notion introduced by Steenstrup et al. in 1983 and presented in one of the first complete system implementations by Stewart in 1994 [1, 2, 8, 3] are essentially blocks which have inputs and outputs and react to changes on these inputs or to an internal clock source. Their successful execution can be proven using the algebra of sets.

Operating Systems

Within the domain of robotics three classes of operating systems are relevant: Hard real-time, firm real-time and soft real-time. The three terms refer to different levels of timing guarantees. All of them perform a task which has an inherent maximum execution time or deadline. Typically the deadline is defined by the time the output is required to control the next update of an actuator position. For a hard real-time system missing the deadline leads to a total system failure. An example for this would be a motor controller which estimates the state of the motor continuously and cannot infer
its position from a single measurement. Missing the deadline will render its control commands out-of-phase and the electric motor will stop to spin. This is in fact a common failure case in drone crashes. In contrast to hard real-time systems a firm real-time system can tolerate to miss a deadline, but under degraded performance. A good example is a flight controller for a drone: A single miss of the attitude control loop will not result in the attitude becoming unstable, but it might lead to slightly worse tracking of the attitude reference. Many flight controller systems for drones are implemented using a firm real-time paradigm. The least requirements are present in a soft real-time system where the result of the computation is less useful after the deadline, but can still utilized and the system does not become unstable. Many consumer or general-purpose operations systems like Linux can be configured for soft real-time where they generally show less latency and variation in the scheduling but are not providing any formal guarantees. In robotics this mode of operation is generally satisfactory for high-level computer vision applications where the runtime of the algorithm is varying with the input. It might not be satisfactory for firm real-time computer vision where the system might become unstable if deadlines are regularly missed.

Middleware

In the context of this thesis middleware has a somewhat wide definition: The most lightweight implementation of a middleware might only provide multi-process communication facilities, while the most heavyweight implementations might offer not just communication, but also logging, sensor drivers, a software package manager, visualization and simulation tools. What is all common to the different levels is that a middleware is concerned with connecting high-level software such as computer vision and navigation (sometimes referred to as "business logic") with other processes and/or the driver and operating system layer. How hard or how easy it is to add components (port based objects) to an existing system and how well the extended system behaves in terms of timing and robustness is largely depending on how well the middleware is designed.
2.2. Design Patterns

Design patterns are essentially collections of recurring architectural design choices in software. Typical examples of design patterns include basic patterns such as a first-in-first-out queue (FIFO) or complex patterns such as a client-server model. For a more detailed discussion please refer to the seminal book of Gamma [16].

Client-Server

In the client-server design pattern one component acts as a central unit which is waiting for requests to come in and clients connect to the a-priori known address of the server. Commonly the clients also exchange information via server, instead of a peer-to-peer communication. This has several drawbacks: Going through the server adds communication latency (more than twofold since its two times the communication plus the server handling). It also means that the server becomes the single point of failure of the system and the bandwidth limitation: It receives and sends many more requests than the individual nodes.

Publish-Subscribe

In publish-subscribe the complete messaging in the system is not based on sender and receiver, but on content categories called topics: Messages are sent and received on a specific topic. This seems initially very simplistic, since only one sender and one receiver might be interested in a topic. However, it makes the configuration of the system much easier, as the sender and receiver do not need to know who they need to connect to a-priori - the complete message passing is focused on the content, rather than on the structure.

Locking data structures and Lockless data structures

For highly parallel systems concurrent access to data structures can be an issue, for example if a receiver of the data reads half of it and then the sender
updates the other half while reading the receiver can end up with an inconsistent sample, leading to a wrong system response. The naive solution would be to protect all critical sections with locks (semaphores or mutexes). However, in highly parallel systems which update some of these topics at a kilohertz rate, the probability that multiple processes want concurrent access and end up being put to sleep until the resource is available is rather high. This can be avoided with lockless data structures which operate by establishing a queue of depth $N$ at the receiver, commonly with $N = 1$. This allows the receiver to pull data from the queue independent of the receiver. In real-time systems it is also common to implement these queues as priority queues where on an overflow the oldest data is deleted, instead of dropping the newest sample.

**Synchronous and Asynchronous**

Synchronous system designs run at a fixed rate on a fixed timing, independent of any changes in sensor data or the environment. Synchronous execution has the downside that if the execution is not synchronized to its input it can add significantly to the overall system latency. The worst-case latency of a synchronous system can be the reciprocal of its update rate. In contrast to that an asynchronous system is even driven: Whenever an update is available its processed. Therefore an asynchronous system can have a very low update rate but also a very low latency.

In general asynchronous systems are more efficient and more effective at processing inputs. It is important that when updates happen at the same time the system is handling the high-priority work items first. It then reduces the backlog of updates according to their priority and interrupts that processing whenever new high-priority items come in.

**2.2.3. Software Reusability**

Research on software reusability is focused on identifying architectural constraints that allow software to be reused / re-purposed for different use cases.
2.2.4. Design Process

Coste-Maniere and Simmons [9] introduced a three-step process which stopped with the validation of the design. However, since 2000 the fundamental processes in software development have changed: Today continuous delivery has become the gold standard. Continuous delivery describes the method of continuously providing updates to users of a software. Some companies go as far as removing formal release processes entirely and new features are rolled in as part of the constant update process. Continuous delivery has emerged as the necessary counterpart for real-time performance and crash analytics built today into all software (and most aggressively used in apps). It has been a research topic for more than a decade (e.g. in the work of Orso et al. [17]). When a fix or an improvement of the user experience is finished it is immediately made available to users via continuous delivery. Typically the traditional Specification-Execution-Validation cycle is kept in place for the initial design effort and major features, but these are now driven decoupled from each other and deployed immediately when ready. As a result tracking how individual features perform, identifying failures of the deployed software and implementing fixes or complete overhauls is now a major part of the software lifecycle. Depending on how agile the software development process is implemented maintenance is becoming the main activity.

2.2.5. Specification

Robotics software falls in the category of timing-sensitive software which must fulfil particular constraints in addition to basic correctness. While every software (except for batch-processing software) is subject to some sort of time constraint (the user / use case typically has a time limit for the result), software in robotics is in a physical feedback loop. Depending on how quick or how slow the result is available, the problem to solve can be easier or harder. One typical example is optical flow: A pretty basic optical flow implementation can deliver excellent results because the difference frame-to-frame between each image is minimal. If the same software is run at a slower rate it will not provide good results because the vehicle has moved...
considerably.

Execution

The same is even more true when a controller is in the loop: If the controller runs very fast it can generally keep a system stable even if the design itself is simple, if the controller can observe the system state only at a low rate it has to predict the next system state and generally needs some type of model to make up for delays. The latency and update rate of the system architecture can therefore dictate the algorithmic approach of the higher-level software. Therefore the specification phase of robotics software does not only include the overall static structure, but also the timing requirements. Because events are not always in-order and predictable (inertial sensors are predictable, detected pedestrians and their number much less so), it needs to deal with varying workloads and update rates. Events can also be out-of-order, so the system must not assume in-order arrival of data or events.

Validation

Validating the result can involve some form of formal certification or an informal validation that the resulting robot architecture is fit for a particular purpose. The tools for this validation have been traditionally formal and procedural, including model checking and specific software development models. On the other hand modern, highly complex cloud computing software is validated using a black box approach: The software is run in a simulation or testing environment and the results of certain inputs (ranging from unit tests to tests with real data) are evaluated.

2.2.6. Certification

Software can be validated both as a final artifact but also as a design process. The dominant approach in regulation and safety inspection has been to inspect processes, code, documentation and applying certain rule sets and tests determining that a system is safe. However, this approach could not prevent the catastrophic failure of the first Ariane 5 rocket, which was later
attributed to a software failure: Both computers estimating the vehicle attitude failed in short sequence due to a sensor input not expected by the original designers. The resulting report recommends complete simulations, which is essentially treating the software on the vehicle as a black box and running it through a complete flight: "R2 Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained." [18]

The next recommendation is equally interesting: "R3 Do not allow any sensor, such as the inertial reference system, to stop sending best effort data." [18]. While it is common sense to never give up from a human perspective, the definition of "best effort data" is highly speculative. In-built into the definition is that the best-effort data is still better than no data, and hence that the best effort data is inherently of better quality than declaring a complete failure and stop sending any data. This is only true if the remaining accuracy can still be assessed - otherwise best effort data could have steered the rocket into the ground. Consequently the only way to define where best effort ends becoming worse than shutting down the flight computer is through extensive simulation, treating the system essentially as a statistical black box. Once this is however done for safety critical systems and as the last resort for safety, the valid question arises why this approach shouldn’t be appropriate for the overall system design. This very fundamental question is at the core of an emerging discussion about functional safety in the robotics industry. Certifying quality and reliability a predictive process: Because failure was costly (in terms of financial and moral cost) it was necessary to devise methods to predict the quality of a certain piece of software by auditing the process by which it was created. Of course this included testing of the software, however, it is often not practical to test every potential scenario and outcome, so exhausting testing is virtually impossible. While software for aerospace and automotive applications was reasonably simple, certifying the process and basic tests was the best option available. Algorithms were often even tractable for formal verification and model checking. However, with robotics making fast strides towards full autonomy without user input, the complexity of the software involved makes it
impossible to perform model checking. And even the design-process based techniques are reaching their limit: Even if individual components and the overall system design follow a design process, small differences in timing (potentially even within the specification) can lead to catastrophic events, such as e.g. debug output from one component loading a communication bus so that two other components can’t communicate at a fast enough rate or misinterpret this data as valid input. Recently deep learning has seen wide adoption in almost every artificial intelligence and control problem in existence. Some of the deep learning algorithms are a black box - their behavior can be studied with respect to concrete inputs and outputs, but it is unclear how much the output changes if the input is slightly changed. Changing the input only slightly can lead to a completely different output and so it is hard to make procedural predictions. Amodei, Olah et al. have discussed this in their paper [19] in full depth.

It is essentially impossible to ensure the safety or predict the performance of highly complex autonomous systems using existing certification techniques. It is however possible to test them in a very wide variety of simulated situations and to test them extensively in real world to approach this challenge quantitatively.

**Qualitatively / Process based**

Current aviation and automotive certification is strongly process based, resulting in e.g. requirements how code is written (even formatted). One of the examples is the MISRA [20] coding standard, another the DO-178A/B/C aviation standard.

**Quantitatively / Black Box**

The opposite testing method is quantitatively: Instead of certifying the process, the result is analyzed. The theory behind this type of testing is that all the important and typical functions (which are two different, but overlapping sets) can be executed in real-world system tests. The limitations of quantitative testing are getting enough coverage (the ability to put the system into all states that are important) and a sufficiently large sample size.
(enough test runs / test flights). Some of the tests involve destructive testing runs (such as emergency braking on a plane or testing the containment case of a turbine engine). All these factors limited quantitative testing in the past to a relatively small amount of hours compared to the actual usage hours (a few thousands compared to hundreds of thousands of hours usage).

In automotive or aviation it is morally unacceptable to subject the vehicle to real customer use before the absolutely latest and final levels of testing have been achieved. While there are also moral predicaments of testing beta software for drones in real use, they can be mitigated by limiting the damage to only the drone. By performing flight tests in a cordoned off area it is feasible to limit the damage to only the vehicle. This enables black-box flight testing of essentially completely unverified software (although in practice it is far more efficient to do a lot of this testing in simulation and only then move on to the real system). In order for this technique to become disruptive the second key element of having no pilot / no driver and a low-cost system is required: Being able to deploy hundreds of systems in parallel, clocking up thousands of flight hours in a very short amount of calendar days.
Chapter 3.
Open and User Innovation

Robotics was in the last decades a discipline which largely relied on custom systems created in one university or company and maintained over the course of a few years. However, as the complexity in the field has increased, more and more labs realized that the lifetime of a three-year research project is not enough to create custom hardware and software or even maintain it. Several platforms have emerged to address this, e.g. ROS for general-purpose robotics and PX4 for the specific case of deeply embedded systems like drones. For a detailed discussion of open and lead user innovation please refer to von Hippel, 2005 [21].

Overall the inherent complexity of the software infrastructure has led to duplicate efforts yielding very little international effect and traction. The report of [22] Bekey et al. from 2006 lists investments of more than $1M per year in robotics middleware in Japan. This funding level should have lead to a significant, if not leading, position in the field. Similar efforts have been made in many research institutions, including our own (now retired) MAVCONN middleware. However, the dominant middleware for general robotics is ROS [3], the Robot Operating System. The most notable difference between ROS and other alternatives is the open innovation model: While ROS has been dominantly developed at Willow Garage and later at its
successor, the Open Source Robotics Foundation. In addition, major parts have been contributed by a global open source community. This distributed open innovation is not particular to robotics but today a major contributor to overall innovation.

To summarize, the main difference between software open sourced by individual research groups and an open innovation model is the cooperation of multiple stakeholders during the innovation process. As von Hippel and von Krogh discuss in their seminal paper [23] open source is neither a private nor a public good but rather a compound model having aspects of private investment and collaboration on a public good. In the case of robotics middleware and infrastructure this difference is significant as the overall system architecture is designed to match the different use-cases of the different stakeholders. These stakeholders are often investors into the open platform in the sense of providing resources to advance the technical state in order to later leverage the overall system in their products [24]. This view is not limited to companies - volunteers working on open source in their spare time essentially also devote resources in order to arrive at a technical solution for their particular problem.

Amalgamated with open innovation is in our case lead user innovation: While a middleware provider soliciting community feedback operates with an open innovation model, these community members are often lead users. In the case of the ROS example the Open Source Robotics Foundation is driving an open innovation process and most of its external contributors are researchers who apply ROS to their research projects. These researchers are lead users of robotics middleware.

### 3.1. Lead User Innovation

The Pixhawk open hardware autopilot and the PX4 open source flight stack are both the result of a lead user innovation process [25, 21]: Our lab (the Computer Vision and Geometry Lab of the computer science department of ETH Zurich) is not an electrical engineering or controls lab. The primary driver behind creating an open hardware and open source autopilot was the need for a tool to build a drone capable of taking computer vision to the air.
3.2. Sustainability of Open Source Development

The avionics development was therefore originally outside of the scope of our lab and started as user innovation.

PX4 evolved from an user innovation in a research lab into a research project on system architecture and design patterns for real-time robots such as drones. Its profound impact on the field hinges in great parts on open innovation. This allowed individuals and companies to contribute significantly to its development.

3.2. Sustainability of Open Source Development

Open source development is commonly regarded as an activity carried out by enthusiasts in their spare time. In fact most open source projects with massive impact started out in a similar fashion. However, for the sustained development it is critical to allow the core developers to focus their full attention on the project. Taking Linux as an example, this results in the vast majority of development being done by professional developers, as noted in a survey by the Linux Foundation: "It is worth noting that, even if one assumes that all of the "unknown" contributors were working on their own time, well over 80% of all kernel development is demonstrably done by developers who are being paid for their work." [26]. As an open source project sustains development over many years this share seems to even further increase over time: "Interestingly, the volume of contributions from unpaid developers has been in slow decline for many years. It was 14.6% in the 2012 version of this paper, and 13.6% in 2013; now it is 11.8%." [26].

As open source projects become established the original main authors tend to focus more on the coordination, review and integration of the changes of others. For the Linux Kernel this has meant that the project maintainer (the founder and project leader) Linux Thorvalds does consistently not show up in the top 30 contributor list in the last years. However, the position of a maintainer is to coordinate and create agreement over the technical direction, not to contribute code.

For our projects this state is approaching fast, however, as all our code-
bases are just between four and five years old, the original authors are still the most frequent contributors. Today over 90\% of the contributors to the project are professional developers, a share similar to that in Linux. PX4 has been exponentially growing since the project has been created, effectively doubling the number of its contributors in the last year between 2015 and 2016. Figure 3.1 shows the number of contributors as tracked by the GIT version control system. The last month of December saw well over 90'000 lines of code (LOC) being added to the codebase.

![Contributors](chart.png)

**Figure 3.1.:** Number of contributors to PX4 per year, as logged by the GIT version control system. The growth is accelerating and the number of contributors has doubled between 2015 and 2016.

### 3.3. Scale and Impact

The work presented in this dissertation has one main component, PX4, which is a two-part software package containing a generic middleware layer and an estimation and control library consisting of port-based objects implementing specific estimators and controllers for individual airframes or
classes of airframes. However, in support of this main component additional software components were created which are today widely adopted: MAVLink, the micro air vehicle communication protocol and QGroundControl, an user interface system for drones. Neither MAVLink nor QGroundControl are within the scope of this dissertation but are key infrastructure required for PX4. The same applies for the open hardware electronics developed for PX4, most notably the flight-management-unit (FMU). This hardware unit is marketed as product under the commercial name "Pixhawk" by multiple companies.

All of these software components have been widely adopted in the drone industry - either individually or as a functional set, often referred to as stack. As computer vision is becoming commonplace on drones additional components will become part of this stack, which as a whole includes all the functionality to let a drone fly autonomously.

PX4 has been adopted since its creation in 2012, initially slowly, then very rapidly by major stakeholders in the industry. The field is rapidly consolidating and more and more drone manufacturers are abandoning in-house development in favor of basing on an open source solution. Figure 3.2 shows the publicly visible industry partners engaged with the open source project.

Even more important than company adoption has been the fact that components of the system could be reused for other software development efforts. The APM project is a great example which leveraged the hardware and middleware to run their own flight control solution as an application on top.

### 3.3.1. Academic Impact

This dissertation had a very wide academic impact by providing state-of-the-art infrastructure to a large number of research groups. The number of citations is too large to list them all, but they include very notable research groups which advanced the state of the art considerably [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42].

We surveyed 300 academic papers which contained the word Pixhawk. 144 of these did not just mention it in their prior work section but described using it. Every Pixhawk contains the FMUv2 open hardware design and the
PX4 middleware. Depending on the configuration used, it also contains additional estimation and control software components. We found in an online search more than 1000 papers mentioning Pixhawk, so we can conclude that hundreds of research projects (where one research group can have multiple projects) have been enabled by this dissertation. Most likely it is many more than this number, as not every research efforts results in a publication.

### 3.3.2. Industrial Impact

The impact of the Pixhawk open hardware platform and the PX4 software architecture has been profound: Many drone startups began adopting it early on, most notably 3D Robotics and companies like Skycatch. It is also safe to assume that virtually every other drone project like Google Wing, Kittyhawk, Amazon Prime Air started by ordering a Pixhawk.

![Figure 3.2.: Major industry partners of the PX4 development](image)
Part II.

Dynamic Robot Architecture
Chapter 4.

Architecture

This chapter extends our previous work on the theoretical level described in Meier, Lorenz, et al. "PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision." Autonomous Robots 33.1-2 (2012): 21-39. [4, 5]. It adds an in-depth analysis of port-based hybrid system designs as a theoretical robot architecture contribution. We developed in 2009-2010 a novel quadrotor MAV system, the PIXHWAK MAV, which was specifically designed to be a research platform for computer vision based flight control.

4.1. Port Based Objects

Port based objects are automata which act on an input on a port and provide an output on a different port. Input ports are denoted as \(X\), output ports \(Y\), as shown in Figure 4.1. These ports can be connected as direct links in a 1:1 relationship or be formulated as 1:N. It is trivial to see that the 1:N relationship is a superset of the 1:1 set.
Figure 4.1.: A port-based object is an automaton with inputs $X_i$ and outputs $Y_i$. It executes its internal state machine whenever the inputs change.

### 4.1.1. Dynamic Reconfiguration

The 1:N relationship can be designed by using named channels which contain a specific type of message, called topics. One publisher publishes a message on this topic (e.g. vehicle attitude) and one or multiple receivers can subscribe the topic and subsequently receive the messages.

Figure 4.2.: Reconfigurable port based architecture: The configuration of connected objects on the top row can be altered to the configuration of connected objects in the bottom row at runtime.

The overall execution of the global state machine formed by individual port based objects is non-deterministic: The order of execution depends on the timing of the individual inputs and their timing in turn can depend on
the computation time, varying with the message content.

4.1.2. Task Set

A task set as defined by Fielding [2] is a number of port based objects which have at least some of their ports connected and execute concurrently. The overall robot behavior is formed by task sets. Task sets can be reused between different robots (such as e.g. the task set consisting of a gyroscope driver, accelerometer driver, magnetometer driver and attitude estimator). Specific combinations of port-based objects are minimum-viable task sets that are typically reused as a set, rather than as individual components. This has interesting properties for the overall system architecture.

4.1.3. Architecture Verification

Figure 4.3.: Example task set for attitude control around the yaw axis ($\psi$)

The port-based object design can be verified following the notation of Steenstrup, Arbib and Manes [1] and the application to system implementations of Fielding [2] p. 27.

\[(Y_i \cap Y_j) = \emptyset, \text{ for all } i, j \text{ such that } 1 \leq i, j \leq k \land i \neq j \quad (4.1)\]

and

\[((\bigcup_{j=1}^k X_j) \subseteq (\bigcup_{j=1}^k Y_j)) \quad (4.2)\]

where $X_i$ is the set including the inputs of object $j$, $Y_j$ is the set including the outputs of object $j$, and $k$ is the number of modules in the task set as
defined above. We are validating the example task set of Figure (4.3). From left to right the module 1 is the gyroscope sensor driver, module 2 is the magnetometer sensor driver, module 3 is the attitude estimator and module 4 is the attitude controller. We define \( X_1 = \emptyset, Y_1 = \{ \dot{\psi}_r \}, X_2 = \emptyset, Y_2 = \{ \psi_r \}, X_3 = \{ \psi_r, \dot{\psi}_r \}, Y_3 = \{ \psi_s, \dot{\psi}_s \}, X_4 = \{ \psi_s, \dot{\psi}_s \} \) and \( Y_4 = \emptyset \).

To satisfy equation (4.2), the union of the input sets and the output sets must be compared.

\[
\bigcup X = X_1 \cup X_2 \cup X_3 \cup X_4 = \{ \psi_r, \dot{\psi}_r, \psi_s, \dot{\psi}_s \} \tag{4.3}
\]

and

\[
\bigcup Y = Y_1 \cup Y_2 \cup Y_3 \cup Y_4 = \{ \psi_r, \psi_r, \psi_s, \dot{\psi}_s \}. \tag{4.4}
\]

Since \( \bigcup X = \bigcup Y \), Equation (4.2) is also satisfied and thus the task set depicted in 4.3 is valid.

### 4.1.4. Configuration

Apart from the runtime components the configuration management is a key aspect of a robot architecture. A configuration consists of the set of port based objects and their inputs and outputs: \( \bigcup X, \bigcup Y \). Intuitively robots performing overlapping activities should be expected to have largely similar configurations. However, it is non-trivial to break down these configurations into common task-sets which would allow to maximize the reuse. Figure 4.4 shows two typical task sets and their relationship: The attitude sensing and estimation task set and the controller task set.

Typically the number of connected ports and the rate at which they update varies greatly throughout the network of port-based objects. Identifying re-usable task sets is therefore possible by dividing the full set of components into all possible task-sets and then choosing \( N \) task-sets. This optimization problem is exponential in complexity, but can be simplified intuitively with a simple heuristic: Starting with the first output \( Y_1 \) of object 1 which is connected to the input \( X_2 \) of object 2, walk the control flow of the robotic system and store the minimum of the number of connected inputs and outputs \( |X_i| + |Y_{i+1}| \). Intuitively in a micro air vehicle such local minima are
4.1. Port Based Objects

Figure 4.4.: Example task set. Connections between port-based objects are characterized by the width (number of states) and rate (in Hertz). Reusable task sets typically have a low number of states.

the attitude controller interface and the position controller interface. Figure 4.5 illustrates the interface width over several steps of the implementation pipeline. However, to properly judge the reusability of the configuration one would also need to identify the potential permutations of the inputs and outputs of each pipeline step. The possible permutations are provided not by the number of ports, but by the number of independent port sets. This is because the attitude quaternion or the position vector have their entries linked to each other. Consequently the number of potential options to express the attitude setpoint is 1: the quaternion plus throttle. In contrast, the options to output the motor signals for a multicopter is $4! = 24$. 
Figure 4.5.: Example configuration: The interface width varies with the different steps in the control pipeline from sensor input to actuator output.

4.2. Software Reuse

One of the key design goals of any port-based object architecture is software reuse. Figure 4.6 shows a very well suited example from micro air vehicles. It shows the block diagrams of a multicopter vehicle and a fixed wing vehicle. As both vehicles have similar sensors and navigation systems the main difference are the airframes-specific controllers for position and attitude. Recently a lot of interest was created around a hybrid of these two vehicle types: A vertical-takeoff-and-landing airplane. The port-based object architecture lends itself optimally to these use cases as the two vehicle nodes can be combined or parts replaced without reworking the overall system architecture. We showed in our VTOL design paper [43] that our architecture scales well for system topographies not originally foreseen when designing the architecture.

Software reuse is critical for the further evolution of robotics research as researchers need to be able to build on top of the current state of the art. However, the simple availability of software and hardware components implementing it is not sufficient: The average consumer drone does not lend
4.2. Software Reuse

**Figure 4.6.** Software reuse example: Vertical-takeoff-and-landing airplanes need rotary wing and fixed wing features. A port-based architecture is optimally suited to reconfigure existing components.
itself as a research platform. Instead, researchers need the ability to start with a baseline implementation but also to modify or replace components. Port-based object architectures are particularly strong as their components are only linked by ports.

4.3. A Hybrid System Design

The main contribution of this chapter is a complete micro air vehicle system architecture that is able to run both the vision based flight control and stereo vision based obstacle detection in parallel on an embedded computer onboard the MAV. The flight control system runs on a connected microcontroller. In addition, we show fully autonomous onboard computer vision based waypoint navigation using visual markers and demonstrate the integrated vision based obstacle avoidance system. The system integrates a computing board that is powerful enough to handle all image processing and flight control processes onboard onto a small scale quadrotor MAV. The proposed flying system carries up to $6 \times$ the computation power of comparable systems of the same size in 2012, e.g. [44]. With the possibility of performing all computational processes onboard without the requirement for a constant data link to a ground station, our design brings the vision of a fully autonomous quadrotor MAV significantly closer.

A key feature of our system is the hybrid system architecture which tightly integrates the microcontroller with the high-level Linux computer and allows hardware IMU-camera synchronization. This allows us to measure the USB image transmission delays in our system precisely. As a result, we are able to do visual pose estimation with the synchronized IMU measurements with improved efficiency and robustness. This algorithm is evaluated and compared to a vision only marker based pose estimation algorithm. A stereo vision based obstacle detection system is integrated onto our MAV system. The stereo computer vision system produces a depth map that gives detailed information about the obstacle as compared to other sensors such as infrared or sonar.

We demonstrate the capabilities of the system in the conducted experiments. We perform autonomous waypoint based flights using only vi-
4.3. A Hybrid System Design

sion and compare the accuracy of the vision pose estimation with Vicon groundtruth. In addition, we compare the vision-only pose estimation to a combined IMU-Vision pose estimation using the 2pt+gravity PnP algorithm [45]. Lastly, we show the functionalities of our stereo vision obstacle detection module. All our hardware and software designs are made open-source and are published on our web page \(^1\) with the goal to create an open research platform for the community.

Much of the previous research in autonomous unmanned aerial vehicles (UAVs) has been based on large UAVs in the weight range of 10-20kg. UAVs of this size are able to carry an extensive sensor suite, e.g. LIDAR, Radar, camera system and powerful onboard computers. Impressive results have been shown in terms of autonomous take-off, landing and navigation as well as obstacle avoidance [46, 47, 48, 49, 50, 51, 52, 53]. Specific adaptations to the algorithms and sensor hardware are needed to apply these results to small scale MAVs under 1.5kg. Recent works successfully demonstrated the use of small LIDAR sensors on such small scale MAVs for mapping and autonomous flight [54, 55, 56, 57, 58, 59]. However, pure vision based autonomous flight control and mapping for small scale MAVs has yet to reach the same level of maturity as with LIDAR sensors. One of the first works in visual MAV control was done by Kemp [60]. He used an a-priori generated 3D model of the flight area and 2D-3D edge matching to compute the MAV pose. He demonstrated on-spot hovering of an MAV. The MAV only carried a small analog camera with wireless image transmission. All processings were done off-board.

More recently, Blösch et al. [61] described visual autonomous flight using an Asctec Hummingbird and a downward looking camera. A visual SLAM algorithm was running off-board on a standard PC. The images of the on-board camera are streamed to the PC using a USB cable physically connected to the MAV thus limiting the autonomy. Control input was sent back to the MAV via a radio link. In their paper, they demonstrated on-spot hovering and waypoint following over a 10m trajectory. To ensure enough visual features for SLAM, their testbed was covered with textured posters. The off-board visual SLAM computed positions at varying frame rates be-

\(^1\)www.pixhawk.ethz.ch
Chapter 4. Architecture

tween 15-30Hz. An extension of [61] was described by Achtelik et al. [44]. They replaced the Asctec Hummingbird with a bigger model, the Asctec Pelican, and equipped it with an Intel Atom onboard computer. This allowed them to run a modified version of the visual SLAM of [61] on-board with a frame rate of 10Hz. However, this modification severely limits the size of the environment that can be mapped. They also described a position controller and successfully demonstrated closed-loop position control with only visual feedbacks. In addition, they demonstrated on-spot hovering in an indoor and outdoor setting. However, the vision based flight control used up all the processing power of the embedded computer, leaving none for the other processes, e.g. obstacle avoidance. Williams et al. [62] used line and point features for visual flight control of a MAV. They described three types of flight patterns: traversing, hovering and ingress. In their experiments, they computed the MAV trajectory offline using previously captured images on a desktop PC for the different flight patterns. An approach for higher level navigation implemented on a Parrot AR.Drone was described by Bills et al. [63]. The commercially available Parrot AR.Drone comes with a forward and downward looking camera and the capability of onboard on-spot hovering making the system easy to use. However, the system is closed and has no payload capability. It is only possible to stream the images of the camera using Wi-Fi and control it with Wi-Fi. In their work, Bills et al. controlled the direction of movement of the MAV from perspective cues obtained from images and from classification of the environment. This allowed the MAV to follow corridors and even make turns. However, there was no notion of a metric map and the image processing is completely done off-board on a desktop PC.

The image processing can be greatly simplified with the use of artificial markers. Artificial markers were used by Eberli et al. [64] for hovering, take-off, and landing. They described the use of one circular marker to compute the position and pose of the MAV. In their experiments, the MAV was connected via USB cable to a ground station and the image processing was done off-board on a desktop PC.

A different approach by Li et al. [65] showed hovering, take-off and landing of an Asctec Hummingbird equipped with an Intel Atom onboard computer. They described the use of an active LED marker pattern. In their
approach, the flight control was done on-board. They demonstrated hovering over a marker pattern which was mounted on top of a ground robot and as a result, the MAV was able to follow the trajectory of the ground robot.

A similar approach was described by Wenzel et al. [66]. They demonstrated hovering, take-off, and landing of an Asctec Hummingbird using a marker platform mounted on top of a ground robot. The marker pattern was made of IR LED’s and the MAV’s position was computed from a Wii-mote sensor fitted to the MAV. The Wii-mote sensor performs hardware image processing and directly outputs the point coordinates of the detected pattern. The final pose computation was then directly done on the low-level controller of the MAV. Substantial existing research relies on outside-in-tracking of the MAV, e.g. by means of a Vicon motion capture system, to measure the vehicle position [67, 68, 69, 70]. These works mainly focus on low-level control problems or higher-level tasks assuming prior knowledge of the MAV positions and use off-board control. The PIXHAWK MAV system design itself is an alternative to commercially available MAVs, such as Asctec MAVs \(^2\), MicroKopter \(^3\), MicroDrones \(^4\) or Parrot AR.Drone [71]. The hardware design is quite similar to commercially available MAVs. However, while most of the systems have a closed control architecture, our system is primarily designed as a research platform, and therefore, has an open control architecture that provides easy access to all the low level measurements and readily accepts control inputs from higher-level on-board computers. In contrast to the commercial products, our system is an open source and open hardware design, which allows researchers to adapt every single detail as needed. Many commercially open-source systems either only allow users to modify a part of the software or to replace the complete software stack or explicitly do not allow the modification and reuse. This requires users to completely rebuild the whole software stack, while our approach is to allow incremental improvements and specializations based on an initial state. Together with the software architecture, the ground control and operator software, and the easy to use marker based lo-

\(^2\)www.asctec.de
\(^3\)www.mikrokopter.de
\(^4\)www.microdrones.com
calization, the PIXHAWK system is a great testbed for MAV research.

**Comparison of PIXHAWK Quadrotor Platform (mid 2011)**

<table>
<thead>
<tr>
<th>System</th>
<th>Pixhawk</th>
<th>Asctec</th>
<th>AR.Drone</th>
<th>MK</th>
<th>ACM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU CORE 2</td>
<td>Intel Atom</td>
<td>ARM9</td>
<td>ARM9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CPUs</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Clock</td>
<td>1.86 GHz</td>
<td>1.2 GHz</td>
<td>468 MHz</td>
<td>90 MHz</td>
<td>-</td>
</tr>
<tr>
<td>RAM</td>
<td>2 GB</td>
<td>1 GB</td>
<td>128 MB</td>
<td>96 KB</td>
<td>-</td>
</tr>
<tr>
<td>Power</td>
<td>27 W</td>
<td>11 W</td>
<td>1 W</td>
<td>0.2 W</td>
<td>-</td>
</tr>
<tr>
<td>USB</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>PCIe</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>S-ATA</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>UARTs</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Autopilot</th>
<th>ARM7</th>
<th>ARM7</th>
<th>PIC24</th>
<th>ATMega</th>
<th>ATMega</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP MHz</td>
<td>60 MHz</td>
<td>60 MHz</td>
<td>24 MHz</td>
<td>16 MHz</td>
<td>16 MHz</td>
</tr>
<tr>
<td>AP RAM</td>
<td>32 KB</td>
<td>32 KB</td>
<td>8 KB</td>
<td>96 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>3D Gyro</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Accel.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Mag.</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Open HW</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Open SW</td>
<td>x</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>x</td>
</tr>
</tbody>
</table>

| Weight | 1.5 kg | 1.5 kg | 0.6 kg | 1.5 kg | 1.5 kg |

Table 4.3 shows the difference between different quadrotor platforms. While the autopilot capabilities of the PIXHAWK quadrotor are similar to
another competitive system, the onboard computational speed, RAM and solid-state disk interfaces are unmatched in the MAV domain. A "-" stands for not present, a "x" for present and "o" for partially present. While the power consumption of the Core 2 Duo onboard computer is substantially higher than the alternatives, its performance is up to 6x better than the Atom computer which consumes 11 W. The maximum consumption of 27 W does not substantially contribute to the overall consumption of the quadrotor, which is about 150-180 W only for the motors and about 200 W in total with all electronics and onboard computer.

### Comparison of MAVCONN middleware in MAV-specific features

<table>
<thead>
<tr>
<th>Software Name</th>
<th>MAVCONN</th>
<th>LCM</th>
<th>ROS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>MAV</td>
<td>RT Comm.</td>
<td>Robotics</td>
</tr>
<tr>
<td>Message format</td>
<td>x</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Ground Control avail.</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Radio modem support</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Serial comm support</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UDP support</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>UDP Transport Layer</td>
<td>LCM</td>
<td>LCM</td>
<td>ROS</td>
</tr>
<tr>
<td>UDP Latency</td>
<td>100-1100 us</td>
<td>100-1100 us</td>
<td>500-1100 us</td>
</tr>
<tr>
<td>Stereo triggering</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IMU sync</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.3 compares different middlewares. As MAVCONN uses internally LCM as transport layer, all features available in LCM are retained in MAVCONN. As LCM is a transport layer and not a full robotic middleware, it optimally combines with MAVCONN to a full solution for MAVs. Our middleware adds a layer on top of LCM, providing the MAVLink message format and interfaces to peripherals such as radio modems or USB machine vision cameras. The main difference to ROS is the distributed communica-
tion model without central server and the capability to fully communicate over radio links when needed. No message rewriting is necessary to communicate with MAVLink-enabled IMUs or a ground control station. MAVCONN thus resembles the upper layers of ROS, while LCM is a substitute for the ROS communication layers. MAVCONN is intended for a different application scenario as ROS and focuses more on small-scale system where different electronic modules are not interfaced by ethernet (in contrast to large ground robots), but often employ simple RS-232 serial communication. ROS requires in this scenario to rewrite ROS messages in a serial format, while MAVCONN messages can be directly passed between UDP and serial transport layers. This allows to use one generic bridge process to bridge between different transport media, in contrast to ROS which requires one customized bridge process per type of serial peripheral. Also other management functions, such as the parameter server, are decentralized in MAVCONN and support Linux processes and microcontrollers with the same API.

4.4. Micro Air Vehicle and Middleware

Off-board processing effectively makes the MAV dependent on the external processing unit, and severely limits the safety and operation range of the vehicle. Our system brings the multi-process architecture and onboard processing capabilities from the 20-100 kg class to vehicles with around 1.2 kg liftoff weight. In contrast to systems using local stabilization approaches on specialized microcontroller hardware (Parrot AR.Drone), the system is geared towards global localization and autonomous exploration of unknown environments using stereo vision. The presented initial results show that with a 30 Hz frame rate, our system consumes 10 % of the maximum CPU load (5 ms processing time per frame) for autonomous marker based flight and 40-60 % load (20 ms processing time per frame) for stereo-based obstacle avoidance, which leaves enough capacity for future work. Since the onboard computer offers two CPU cores, onboard parallel localization and mapping is within reach. GPS and, to a large extent, laser-based systems can offer a deterministic processing time to fuse the sensor data into the local-
4.4. Micro Air Vehicle and Middleware

ization. In contrast, computer vision has varying, and in comparison, often longer processing times, depending on the image content. Therefore, the estimation and control steps cannot assume a fixed interval length $\Delta t$ and a fixed processing delay $\Delta p$. Instead, they must use the actual timestamp of all measurements to calculate the correct latency. Thus, all data in our system is timestamped with a resolution in the order of microseconds. This data includes images from multiple cameras, the system attitude, acceleration data, and barometric data. For the stereo head, the middleware supports 2 Point Grey Firefly MV or MatrixVision Bluefox cameras that respectively capture $640 \times 480$ and $752 \times 480$ grayscale images at up to 60 Hz. The camera interface allows additional camera models to be supported. However, since there is no standard for trigger-support among different camera module manufacturers, it is necessary to implement a small interface class in the middleware for each camera type. The camera pair is rigidly mounted with a baseline of 5 cm on a carbon composite frame as shown in Fig. 4.15. The hardware trigger provided by the IMU enables synchronous capture of images from both cameras; this synchronous capture is crucial for accurate estimation of stereo disparity.

4.4.1. Flight and Processing Electronics

The PIXHAWK Cheetah quadrotor design was built from scratch for onboard computer vision. With the exception of the commercial off-the-shelf (COTS) motor controllers and cameras, all electronics and the mechanical frame are custom-designed. First, the payload consisting of the pxCOMEx processing module and up to four machine vision cameras (PointGrey Firefly MV USB 2.0 or MatrixVision Bluefox), was selected. The system design then followed the requirements of onboard computer vision. The onboard electronics consists of an inertial measurement unit and autopilot unit, px-IMU, and the onboard computer vision processing unit, pxCOMEx.

Autopilot Unit

The pxIMU inertial measurement unit/autopilot board (Fig. 4.8, left part) provides 3D linear acceleration (accelerometer, $\pm 6$ g), 3D angular velocity
Figure 4.7.: Onboard sensors and avionics with electronic buses.

(±500 deg/s), 3D magnetic field (± milligauss), barometric pressure (130-1030 Hectopascal (hPa)) and temperature. The onboard MCU for sensor readout and sensor fusion as well as position and attitude control is a 60 MHz ARM7 microcontroller. It can be flashed via an USB bootloader, and stores settings such as PID parameters in its onboard EEPROM. It provides the required I2C bus to the motor controllers, additional GPIOs, ADC input, and other peripherals. It is interfaced via UART to the computer vision processing unit, and it operates at a maximum update rate of 200-500 Hz.

Image Processing Unit

The processing unit is the core piece of the system and consists of a two-board stack. The pxCOMEx base board (Fig. 4.8, middle) provides the USB and UART peripherals to interface machine vision cameras, communication equipment and the pxIMU module. It can accept any micro COM express industry standard module. Currently, a Kontron etxExpress module with Intel® Core™2 Duo 1.86GHz and 2 GB DDR3 RAM is used (Fig. 4.8, right), but future upgrade options include Intel® Core™i7 CPUs. It has $4 \times$ UART, $7 \times$ USB 2.0 and $1 \times$ S-ATA 2.0 peripheral options. The typical onboard setup consists of $4 \times$ PointGrey Firefly MV monochrome cameras,
4.5. Aerial Middleware and Message Format

Figure 4.8.: From left to right: pxIMU Autopilot, pxCOMex image processing module, microETXExpress Core 2 DUO 1.86 GHz module

1 × USB 2.0 802.11n WiFi adapter, and 1 × S-ATA 128 GB SSD with more than 100 MB/s write speed. The pxIMU unit, the GPS module, and the XBee radio modem are connected via UART to the processing unit. With a weight of 230 g including cooling and only 27 W peak power consumption, the processing unit can be easily lifted by a wide range of aerial systems, and not limited to the quadrotor presented here.

4.5. Aerial Middleware and Message Format

Existing middleware solutions for ground robotics include ROS [3], CARMEN [13] and CLARAty [12]. CARMEN and CLARAty have paved the way for standardized robotics toolkits, but their use has declined with the wide adoption of ROS. Although ROS has been used on MAVs, all of these toolkits assume an Ethernet network to all connected nodes. However, MAV onboard-networks typically include no Ethernet device, but several devices connected via serial links and USB. As these toolkits do not scale down to this link type, every packet has to be transcoded by bridge processes. Therefore, we propose a new communication protocol and architecture that
can be transparently used on different hardware links and which minimizes the system complexity.

![Diagram of MAVCONN Network](image)

**Figure 4.9.** MAVCONN Network showing different physical links using the MAVLink protocol.

As shown in Fig. 4.9, the PIXHAWK Linux middleware consists of several layered software components. This architecture allows us to use the different base communication layers (ROS and LCM), and provides a convenient high-level programming interface (API) to the image distribution server. MAVLink messages from the IMU and the ground control station can also be directly received by any process. We rely on the Lightweight Communication Marshalling library (LCM) as the base middleware, as it was shown in [11] that LCM outperforms ROS in low-latency applications. Another benefit is the increased robustness of the overall software architecture when using LCM, as no central server exists and our communication over MAVLink is mostly stateless. This eliminates a single point of failure (the ROS central node), and also eliminates possible protocol lockups in stateful implementations (as in many ROS nodes). Our system can however still benefit from ROS software packages, such as the ROS Kinect interface, by using our ROS-MAVCONN bridge process that routes between the two software packages. The mission and control architecture of the presented robotics toolkit is based on a lightweight protocol called MAVLink, which
4.6. System Architecture

The PIXHAWK design includes a powerful onboard computer which makes it possible to run high-level tasks, in particular visual localization, onboard the MAV. The system design is depicted in Fig. 4.10. Visual localization, obstacle detection, and planning are implemented on the onboard high-level flight computer, an Intel® Core™ 2 Duo. The visual localization module computes the full 6-DOF pose of the MAV (see details in section 4.7). The scales from serial to UDP links. It serves also as a communication protocol between the flight computer (pxImu) and the onboard main computer (pxCOMex/pxOvero). As MAVLink is used on all communication links including the downlink to the operator control unit, it is particularly important that this protocol scales down to very low bandwidth links, and allows the use of several links in parallel. In turn, this parallel use allows several redundant links, which in our case, are long-range XBee radio modems and 802.11n Wifi (UDP). MAVLink has a small 8-byte overhead per packet, allows routing on an inter-system or intra-system level, and has in-built packet-drop detection. Due to the low overhead, it is both suitable for UDP and UART/radio modem transport layers. The efficient encoding also allows protocol execution on microcontrollers. These properties allow the building of a homogenous communication architecture across the PIXHAWK system. The MAVLink sentences are generated based on an XML protocol specification file in the MAVLink format. The code generator ensures well-formed messages, and generates C89-compatible C-code for the message packing and unpacking. This allows fast and safe extensions, and changes to the communication protocol, and ensures that no implementation errors will occur for new messages. Our current implementation supports the use of the Lightweight Communication Marshalling Library (LCM) and the Robot Operating System (ROS) as transport layers. While we use MAVLink to send system states and control commands, we do rely on a separate shared memory implementation of an image hub. This component allows sharing of images of all cameras with an arbitrary number of Linux processes and with the least overhead possible.
Chapter 4. Architecture

Stereo obstacle detection module computes real-time disparity maps from the front-looking stereo pair (see section 4.8). The output of the stereo module can be used for obstacle avoidance by the planning module. The planning module currently implements waypoint following. Both the attitude and position controllers are implemented on a low-level real-time controller (see section 4.9 for state estimation). The position controller takes as input both the poses from visual localization and the setpoints generated from the planner. For MAV control, the attitude measurements and the vision pose estimates need to be synchronized. In our system, the synchronization is solved by using the IMU to hardware-trigger the cameras and timestamp the measurements.

![Diagram of the PIXHAWK quadrotor system design.](image)

**Figure 4.10.:** The PIXHAWK quadrotor system design. The powerful onboard computer enables high-level tasks such as visual localization, obstacle detection, and planning to run onboard. Position and attitude estimation are implemented on a low-level real-time controller.
4.6. System Architecture

Figure 4.11.: System delays from camera shutter to final output

4.6.1. Vision-IMU Synchronization

Data from different sensors, in particular from multiple digital cameras, is synchronized by an electronic shutter signal, and assembled into a timestamped sensor message. IMU sensor data, for example, absolute attitude and angular rates, is available as part of the image metadata. Images are transmitted over the USB bus to the camera process, while the IMU measurements and the shutter timestamp are delivered via a serial interface. Image transmission from the cameras to main memory via USB takes approximately 16–19 ms (where the time differs slightly for each image), while the transmission of the shutter timestamp from the IMU to main memory via serial/MAVLink takes approximately 0.1–2 ms. As it is guaranteed that the IMU data arrives earlier than the image, the camera process can always deliver images labeled with IMU metadata via the middleware to subscribing processes. Fig. 4.11 shows the contributions of individual processing steps to the overall system delay. The transfer time of the image content from the camera module over USB 2.0 is substantial where fast localization techniques are concerned. Initially, the camera shutters are triggered by the inertial measurement unit. The first 19.5 ms after closing the shutter (outdoors after 0.3 ms, indoors after 2-5 ms) are consumed by the USB transfer of the image from the camera to the volatile memory of the onboard computer. To support multi-process computer vision, the image is transported to different processes via shared memory in RAM through the image hub software interface of MAVCONN. Processing the image in Linux requires between 5 ms for ARToolkit and several seconds for large-scale localization.
and mapping approaches. USB and UART transfer delays are only observable with hardware triggering; otherwise, they remain unknown.

Computer vision algorithms can exploit synchronized attitude and vision data to increase accuracy and robustness. Since machine vision cameras have a delay in the tens of milliseconds range due to USB / Firewire transfer time and operating system scheduling delays, the best solution is to synchronize the camera to the inertial measurement unit with a hardware shutter. Fig. 4.12 shows the USB transfer delays (red, bottom curve), the USB and shared memory interface delay (green, middle curve), and the total camera shutter to control output delay (blue, top). The measurements show that the overall delay is in the same range as the interval between two captured images (36 ms for the presented localization). This applies to a system with low and high CPU loads (both cores at maximum load). The spikes in the processing time are mostly attributed to the scheduling of the computer vision processes, as these results were obtained on a non-RT Linux system.
4.6. System Architecture

Figure 4.12.: Measured system delays. Top, blue: Total round-trip time from the camera trigger until the image is transferred to the onboard computer, vision is computed, state estimation is run and actuator outputs are active. Middle, green: Trigger to vision process delay. Bottom, red: USB transfer delay.
4.7. Visual Position Estimation

As the PIXHAWK middleware provides a precise time base, a standard textbook estimation and control pipeline was proven to perform well for autonomous flight. Fig. 4.10 shows the localization and control architecture. Images are read from the camera at 30 Hz and the position is estimated at the full camera rate, using additional inertial information. The current position is then used by the onboard mission planner to determine the desired position. The current and the desired positions are fed back to the position estimation and control software module running on the ARM7 autopilot.

4.7.1. ARTK+ Localization

As the main purpose of the vehicle and this contribution is the robot architecture, we use a localization test bed that uses markers with an adapted implementation of ARToolkit+ [72] for the localization. This has the benefit of being able to demonstrate vision-IMU fusion in a textbook-like setup. The marker positions are encoded in a global world map with the 6D position and orientation of each marker. By extracting the marker quadrangle, the global marker position can be estimated. The correct orientation on the quadrangle plane and the marker ID are encoded by a 2D binary code inside each marker. An example of a larger marker setup is shown in Fig. 4.13. However, the system itself is not dependent on this particular approach – any kind of localization algorithm can be used. The main benefit of using ARToolkit+ in the test bed setup is its relatively low delay (5-10 ms), its robustness with respect to suboptimal lighting conditions, and the high robustness to motion blur. It is therefore very suitable for system testing.

4.7.2. Vision-IMU fusion

The inertial measurement unit can be exploited to increase the robustness and accuracy of the vision-based localization. There are classical IMU-vision fusion methods but all these methods expect covariance estimates of the vision position data. Estimating the position covariance is not possible as vision-based localization typically works on 2D image features that
Figure 4.13.: Flight environment with the ARToolkit markerboard on the floor
are matched between two images. Since this matching is not perfect, outlier removal is necessary which again cannot be made error-free, and the remaining outliers in the feature correspondences will disturb the position estimate. The covariance resulting from a non-linear optimization of the vision algorithm cannot correctly capture these misestimates due to outliers in the feature correspondences. A more promising method is to directly include the IMU data in the vision estimation during the outlier filtering and position estimation. If the accuracy of the IMU-estimated vertical direction is better than the pure vision estimate, the IMU-estimated vertical direction can be used by the vision algorithm to improve the localization accuracy. Fig 4.19 shows that the localization accuracy increases when using two points and the known vertical from the inertial data instead of four points. In the case of the 2-point algorithm [73], the calculation steps for the pose estimation are significantly simplified when substituting parts of the four-point equation with the IMU roll and pitch (see Fig. 4.14 for the geometric relation where the image plane is aligned with the surface normal formed by the gravity vector). This speeds up RANSAC which is typically used to filter out outlier matches between image features. Hence, the benefits of directly combining IMU and vision are, depending on the methods used, improved accuracy and computation time. Since the roll and pitch angles of the camera are known through the inertial measurement unit, the lines connecting the camera center and the 3D points seen by the camera can be rotated to compensate for roll and pitch. The equation for projecting image points into the homogeneous camera space and rotating the rays is given in equation 1; the same operation is also depicted in Fig. 4.14. The pixel coordinate \( u \) is projected with the inverse camera matrix \( K^{-1} \) into the normalized homogeneous coordinate space. This ray is then rotated by the roll and pitch with the rotation matrix formed by multiplying the rotation matrices around the roll and pitch \( R_{\phi\theta} \). The resulting ray in homogenous coordinates is now fronto-parallel with respect to the ground plane.

\[
u' = R_{\phi\theta}K^{-1}u
\]  

(4.5)

The resulting fronto-parallel view has only \( x, y \) and \( \psi \) as free parameters. Kukelova et al. provided a closed-form solution to localize from two points.
4.7. Visual Position Estimation

Figure 4.14: Relation of gravity vector and camera angles. The right-fronto parallel view is obtained by rotating the image plane by the roll and pitch angles.

and the known vertical direction by solving for \( x, y, z \) and \( \psi \). By applying this algorithm in a least squares sense on the ARToolkit correspondences, we obtain the final position output, since \( \phi \) and \( \theta \) are already known. Fig. 4.19 shows that the solution of the vision-IMU 2-point algorithm outperforms in terms of accuracy the 4-point algorithm used in ARToolkit+. Both algorithms operate on the same set of visual correspondences based on the ARToolkit+ corners. In addition, for any non-global vision-based localization approach, the IMU information can provide the gravity vector and heading to be used as the global reference. This is especially important for loop closure in SLAM where global attitude information can facilitate loop detection and reduce convergence into local minimas. This also explains the advantage of storing the absolute attitude in image metadata rather than the attitude relative to the previous frame. Using the absolute attitude, it is always possible to extract the relative orientation between any pair of images.

4.7.3. Outlier Removal

The obtained position vector \( x, y, z \) and \( \psi \) is filtered with a 4×1D block Kalman filter in the next step; this implies that the filters are parameterized
with an error model of the computer vision approach. As IMU and vision both estimate the 3-DOF attitude of the helicopter, this redundant data can be used to detect and remove position outliers produced by the localization step. Any erroneous vision measurement will not only contain a wrong position estimate but also a wrong attitude estimate because of the projective dependency of the position and attitude. Position outliers can therefore be rejected based on the comparison of roll and pitch estimates from the IMU and from the visual localization. Notice the effect on position outliers when including the IMU data into the vision estimation in Fig 4.19. The variance of the position estimate is reduced by a significant amount such that there is no need for subsequent outlier removal.

4.8. Stereo Obstacle Detection

The front looking stereo camera allows us to get depth information in both indoor and outdoor environments, and with depth information, we can reliably detect obstacles in the MAV’s vicinity, and compute their locations. By running stereo processing algorithms onboard the MAV, we demonstrate the ability of the onboard computer to handle computationally intensive tasks which would otherwise be not possible on typical MAVs equipped with single-core processors. In our stereo processing pipeline, we compute disparity data from stereo image pairs, and subsequently, compute a point cloud which is used to update a 3D occupancy map. We determine a cell to be an obstacle if its occupancy probability exceeds a preset threshold, which in our case, is 0.5. If an obstacle is observed to be within the safety clearance of the MAV, an alert message is published. Any planning module that receives this alert can either perform an emergency stop or take evasive maneuvers.

4.8.1. Point Clouds from Stereo

With each stereo image pair, we rectify both images, and use a block-matching stereo correspondence algorithm to build a dense 640 x 480 disparity map. Subsequently, we compute the depth to the points in the scene relative to the camera coordinate system:
4.8. Stereo Obstacle Detection

\[ z = \frac{bf}{d} \]  \hspace{1cm} (4.6)

where \( d \) is the disparity. Differentiation of Equation 4.6 with respect to \( d \) yields:

\[ \Delta z = \frac{bf}{d^2} \Delta d \]  \hspace{1cm} (4.7)

\( \Delta z \) denotes the resolution of the range measurement corresponding to \( d \). To avoid spurious range measurements due to small disparities, we set the minimum disparity:

\[ d_{\text{min}} = \left\lceil \sqrt{\frac{bf\Delta d}{\Delta z}} \right\rceil \]  \hspace{1cm} (4.8)

In our case, we choose conservative values of \( \Delta z = 0.25 \) and \( \Delta d = 0.5 \). With these values, the maximum range of our stereo camera with a baseline of 5 cm and a focal length of 645 pixels is 4 m.

We compute the 3D coordinates of each pixel relative to the camera coordinate system:

\[
\begin{bmatrix}
x_c \\
y_c \\
z_c
\end{bmatrix} = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & f \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \]  \hspace{1cm} (4.9)

where \( z \) is the depth associated with the pixel, \((c_x, c_y)\) are the coordinates of the principal point of the camera, \( f \) is the focal length, and \((i, j)\) are the image coordinates of the pixel. The values of \( c_x, c_y, \) and \( f \), together with the stereo baseline \( b \) are obtained from an one-time calibration.

We then find the world coordinates of each point:

\[
\begin{bmatrix}
x_w \\
y_w \\
z_w
\end{bmatrix} = i^H_i^w H_c^i y_c
\begin{bmatrix}
x_c \\
y_c \\
z_c
\end{bmatrix} \]  \hspace{1cm} (4.10)
where $^iH$ is the homogeneous transform from the world frame to the IMU frame and is estimated by the visual odometry, and $^cH$ is the homogeneous transform from the IMU frame to the camera frame and is estimated using the InerVis calibration toolbox [74].

### 4.9. State Estimation and Control

The state estimation runs on the low-level controller to satisfy the real-time requirements of the system. In addition, this allows us to compensate for the loss of either visual localization data or the complete onboard computer connection. In the case of such a catastrophic event, the system performs an open-loop safety landing maneuver. To synchronize inertial data and vision estimates, the IMU maintains a buffer of attitude measurements corresponding to the last $n$ image frames. Once the IMU receives a vision position estimate, it reads out the buffered sensor values, and performs a state estimator update.

#### 4.9.1. Discrete Kalman Estimation

Similarly to vision algorithm we chose a textbook Kalman filter implementation over a more elaborate formulation in order to focus on the architectural properties of the system. It is trivial to see that any other form of recursive estimation and any state of the art technique can be a drop-in replacement if a baseline discrete Kalman filter is performing well. After the outlier rejection, the remaining positions are more conformant to the normal distribution, and thus, allow the use of a simple discrete Kalman filter. As the dynamics of a quadrotor are only loosely coupled in the $x$, $y$ and $z$ directions [75], the dynamics can be modeled as three independent dimensions. As the yaw angle taken from computer vision is of much better accuracy and resolution in indoor settings compared to the yaw angle from a magnetometer due to iron structures in the building, the yaw angle is taken as the fourth independent dimension for filtering. Given this quadrotor dynamic model, the Kalman filter is designed as a block of $4 \times 1$D Kalman filters with the position and speed as states. The Kalman filter assumes a constant speed
model, and takes the position estimate as input. The estimated velocity is critical in damping the system, as the only physical damping is the air resistance on the horizontal plane which is insignificant at the hovering and low-speed conditions the system is typically operating in. The states of the four Kalman filters are:

\[
\begin{align*}
\begin{bmatrix}
    x_k \\
    \dot{x}_k \\
    y_k \\
    \dot{y}_k \\
    z_k \\
    \dot{z}_k \\
    \psi_k \\
    \dot{\psi}_k
\end{bmatrix}
\end{align*}
\]

We estimate the current state of the vehicle \(x_k\) which is modeled by

\[
x_k = A \cdot x_{k-1} + w_{k-1}.
\]

Where the dynamics matrix \(A\) models the law of motion, \(x_{k-1}\) is the previous state and \(w_{k-1}\) the process noise.

\[
A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}
\]

This motion is measured at certain time steps where the measurements are expressed as the gain \(H\) times the current state plus the measurement noise \(v\).

\[
z_k = H \cdot x_k + v_k
\]

The speed in the model will therefore only be changed by measurements, and assumed constant during prediction. From this formulation, it is already obvious that varying time steps can be handled by the filter as long as they are precisely measured. As this filter does not include the control input matrix \(B\), the filter assumes a constant speed model which is a valid approximation if the filter update frequency is fast enough with respect to the change of speed of the physical object. Because the PIXHAWK system provides a precise time base, the filter uses the measured inter-frame interval as the time difference input \(\delta t\). If measurements are rejected as outliers, the filter only predicts for this iteration, and compensates in the next update step for the then longer time interval. This allows the system to estimate its egomotion for up to about 500 ms and recover from several dropped camera frames.
4.9.2. Position and Attitude Control

The current and the desired positions are fed back to the position estimation and control software module running on the ARM7 autopilot controller. The autopilot calculates the desired attitude, and controls the attitude using its onboard inertial sensor suite. The x- and y-positions are controlled with the angle of attack of the collective thrust vector by setting the desired pitch angle for x and the desired roll angle for y. The z-position can be controlled with the component of the collective thrust collinear to the gravity vector. The yaw angle can finally be controlled by the difference of rotor drag of the clockwise (CW) and counter-clockwise (CCW) rotor pairs. As the discrete Kalman filter contributes a smooth position and speed estimate with little phase delay, the controller can be designed as a standard PID controller implemented as four independent SISO PID controllers for x, y, z, and yaw, as in [76]. Attitude control is implemented following the standard PID based attitude control approach for quadrotors using one PID controller for each of the roll, pitch, and yaw. The craft is actuated by directly mixing the attitude control output onto the four motors.

4.10. Computer Vision Timing

Off-board processing effectively makes the MAV dependent on the external processing unit, and severely limits the safety and operation range of the vehicle. Our system brings the multi-process architecture and onboard processing capabilities from the 20-100 kg class to vehicles with around 1.2 kg liftoff weight. In contrast to systems using local stabilization approaches on specialized microcontroller hardware (Parrot AR.Drone), the system is geared towards global localization and autonomous exploration of unknown environments using stereo vision. The presented initial results show that with a 30 Hz frame rate, our system consumes 10% of the maximum CPU load (5 ms processing time per frame) for autonomous marker based flight and 40-60% load (20 ms processing time per frame) for stereo-based obstacle avoidance, which leaves enough capacity for future work. Since the onboard computer offers two CPU cores, onboard parallel localization and
mapping is within reach. GPS and, to a large extent, laser-based systems can offer a deterministic processing time to fuse the sensor data into the localization. In contrast, computer vision has varying, and in comparison, often longer processing times, depending on the image content. Therefore, the estimation and control steps cannot assume a fixed interval length $\Delta t$ and a fixed processing delay $\Delta p$. Instead, they must use the actual timestamp of all measurements to calculate the correct latency. Thus, all data in our system is timestamped with a resolution in the order of microseconds. This data includes images from multiple cameras, the system attitude, acceleration data, and barometric data.

### 4.10.1. Stereo Head

For the stereo head, the middleware supports 2 Point Grey Firefly MV or MatrixVision Bluefox cameras that respectively capture $640 \times 480$ and $752 \times 480$ grayscale images at up to 60 Hz. The camera interface allows additional camera models to be supported. However, since there is no standard for trigger-support among different camera module manufacturers, it is necessary to implement a small interface class in the middleware for each camera type. The camera pair is rigidly mounted with a baseline of 5 cm on a carbon composite frame as shown in Fig. 4.15. The hardware trigger provided by the IMU enables synchronous capture of images from both cameras; this synchronous capture is crucial for accurate estimation of stereo disparity.

### 4.10.2. Mechanical Structure and Flight Time

Our custom mechanical design (Fig. 4.15) effectively protects the onboard processing module in case of a system crash, and the fixed mounting of the four cameras allows inter-camera and camera-IMU calibration. Our system comes in two sizes, one optimized for very small indoor environments, and one standard size. As the processing board and up to four cameras represent a relatively large payload of 400-800g for the small diameter of 0.55 m (0.70 m for the larger version) of the quadrotor, the overall system structure has been optimized for low weight. It consists of lightweight sandwich material
Chapter 4. Architecture

with composite plates and an inner layer made of Kevlar. Each of the four motors with 8” or 10” propeller contributes a maximum of 450-600g thrust, enabling the system to lift a 400g payload at a total system weight of 1.0–1.2 kg, including the battery. This allows a continuous flight time of 7–9 minutes with 8” propellers and 14–16 minutes with 10” propellers. The propulsion consumes 150-180W for hovering, while the onboard computer consumes only 27 W peak. Therefore, flight time is governed by the weight of the system.

![Quadrotor Overview](image)

**Figure 4.15.:** Quadrotor Overview

4.11. Experiments and Results

We conduct experiments to evaluate the visual localization without and with the vision-IMU 2-point algorithm, and the stereo obstacle detection, and discuss the results.
4.11. Experiments and Results

4.11.1. Visual localization

We perform two experiments to determine the localization accuracy of our ARToolkit+ localization without and with the vision-IMU 2-point algorithm described in Section 4.7.2. In our experiments, we use ground truth data from a Vicon motion tracking system; the ground truth data is provided at a rate of 50 Hz and is very precise with < 1mm error. The objective of the experiments is two-fold: to quantitatively measure the ARToolkit+ localization error relative to the Vicon ground truth, and to examine whether the 2-point algorithm improves the localization accuracy by using vision-IMU fusion. To be able to localize the helicopter with the described vision system during the whole flight, ARToolkit+ markers were laid out on the floor in the flight area as shown in Fig. 4.13. In each experiment, we use our operator control software, QGroundControl, as shown in Fig. 4.16 to set relevant parameters for MAV software components, monitor the MAV’s status, send commands to the MAV, and preset waypoints for autonomous flight. Furthermore, in each experiment, the MAV executes an autonomous flight; at the beginning and end of the flight, open-loop takeoff and landing are performed respectively, using in the control loop only the estimated state of the MAV without any external position or attitude reference. During the flight, the MAV uses the localization output to follow the preset waypoints.

**Experiment 1 - Autonomous waypoint following**

In the first experiment, Fig. 4.17 shows the localization results using the ARToolkit+ localization without the 2-point algorithm. The plot shows a flight around a rectangular path and two crossings. The solid black line shows the planned flight path; the vertical line in the top left corner of the figure indicates takeoff while the vertical line in the bottom left corner indicates landing. The grey spheres indicate the waypoints; the radius of each sphere equals the acceptance radius within which the waypoint is marked as reached. The blue asterisks represent the position estimates computed by the unfiltered visual localization, and the red crosses represent the Vicon ground truth.

It is observed in Fig. 4.17 that the ARToolkit+ localization output without
Figure 4.16: QGroundControl view with live image streaming from the helicopter using MAVLink over UDP. The live view on the left shows the rectified and depth images from the stereo camera setup.
the 2-point algorithm closely follows the Vicon ground truth, but is subject to frequent large errors. This is because the localization output is computed purely based on vision, hence making it extremely sensitive to errors from the extracted image features. A small error in the position of the extracted image feature would translate into a large error in the localization output, thus explaining the frequent large errors.

![Figure 4.17: Trajectory of an autonomous flight using unfiltered AR-Toolkit+ localization (blue asterisks) including takeoff and landing plotted together with the Vicon groundtruth (red crosses) and planned path (solid line and spheres are the planned waypoints).](image)

**Experiment 2 - Comparison of IMU-aided localization**

The quadrotor autonomously flies a similar trajectory as in Fig. 4.17. The flight control is based on the ARToolkit localization without using the IMU,
and the state estimation is done with an attitude observer filter and four independent 1D Kalman filters for $x$, $y$, $z$ and yaw. In our vision-IMU 2-point algorithm, we first compute the 2D image features that correspond to the 4 corners of each ARTK marker in full view in the image; example 2D image features are shown as black circles in Fig. 4.18. We then establish 2D-3D correspondences through identification of the marker IDs and retrieval of the 3D coordinates of the identified markers from the ARToolKit+ configuration file. We use the same set of 2D-3D correspondences to compute the pose estimates for the ARToolKit+ localization without and with the 2-point algorithm.

Fig. 4.19 shows a comparison of the localization output from the ARToolkit+ localization without and with our 2-point algorithm as shown in red and blue respectively; the Vicon readings are shown as groundtruth in green. It is observed from Fig. 4.19 that the localization output with our 2-point algorithm is significantly smoother and more accurate than that without the 2-point algorithm; the 2-point localization output coincides more closely to the Vicon ground truth and is not subject to large jumps which occur for the localization without the 2-point algorithm. This is due to the additional roll and pitch information from the IMU which helps to reduce the sensitivity.
Figure 4.19.: Position estimates from ARToolKit+ localization without (in red) and with (in blue) the 2-point algorithm. The Vicon groundtruth is shown in green.
of the localization process to errors arising from the extracted image features. In both experiments, the helicopter hovered shortly above the landing position until it reached a steady hovering state, and then landed. It can be observed that there are significant cross-track errors between the actual and planned flight paths. As our focus is not on precise path following, our MAV system is equipped with basic PID position and attitude controllers which are not optimally tuned. Furthermore, the localization output does not reflect the actual position of the MAV, and therefore, deviations from the planned flight path are expected.

4.11.2. Stereo Obstacle Detection

We carry out an experiment in which the MAV flies autonomously along preset waypoints. We show a visualization of the stereo processing at one point of time; Fig. 4.20 shows an image from the left camera of the stereo rig, the resulting 3D point cloud computed from the corresponding stereo frame, and the same point cloud colored by distance from the MAV. In the latter two images, the MAV is shown in green. Fig. 4.21 shows the occupancy map that corresponds to the stereo frame depicted in Fig. 4.20.

![Figure 4.20](image)

**Figure 4.20.** Left: Left stereo image. Middle: Colorized point cloud. Right: Same point cloud colored by distance from the MAV.

The MAV publishes alert messages if it detects obstacles within a safety clearance of 0.75m. To test this functionality, we put obstacles (plant, person, cardboard) along one side of the flight path and closer than the pre-set safety clearance. Fig. 4.22 shows the outcome of the test flight; the flight trajectory is shown in blue, while the locations where the MAV published
Figure 4.21.: The occupancy map corresponding to the frame in Fig. 16. Obstacle cells in the map are marked in blue.
Table 4.1.: Breakdown of computational time for on-board stereo processing using 640 x 480 stereo images. (1.86 GHz Intel® Core™ 2 Duo)

<table>
<thead>
<tr>
<th>Process</th>
<th>Average Computational Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image rectification</td>
<td>5 ms</td>
</tr>
<tr>
<td>Disparity mapping</td>
<td>29 ms</td>
</tr>
<tr>
<td>Point cloud generation</td>
<td>1 ms</td>
</tr>
<tr>
<td>Occupancy mapping</td>
<td>50 ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>85 ms</strong></td>
</tr>
</tbody>
</table>

alert messages are marked with red circles. These alert messages could be used by a planning algorithm to change the flight plan.

The breakdown of computational time for the stereo processing on-board the MAV is described in Table 4.1.
Figure 4.22.: Autonomous test flight with obstacles. Red circles mark the locations where the MAV published alert messages (when obstacles are detected within the MAV’s safety clearance of 0.75m). The flight trajectory is in blue.
Chapter 4. Architecture

4.12. Summary

We have presented a complete hybrid system architecture starting with the fundamental concept of port-based objects. We have shown how high-level computer vision processes on Linux can be integrated with low-level processes on a microcontroller. We identified key architectural considerations that enable successful combination of inertial and visual sensing. Overall this system architecture which represented the state-of-the-art in 2012 is now widely adopted in academia and industry.
Chapter 5.
Dynamic Systems

After the introduction of the port-based object architecture and the high-level overview of the hybrid system we will introduce the detailed architecture for the low-level flight controller and focus on runtime properties. This chapter is based on the paper: Meier, Lorenz, Dominik Honegger, and Marc Pollefeys. "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms." 2015 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2015. [6].

Recently a number of large scale industry applications have been proposed which require higher levels of autonomy than deployed systems can offer today. As the field progresses and more sophisticated robotics problems are targeted, the software complexity of these systems increases rapidly. Similar to ground robotics recently, aerial robotics is reaching a level of complexity where individual research groups are unable to tackle the system design on their own.

We address this problem by providing a very flexible, standards oriented and low-cost research middleware supporting MAVs in the micro size scale but still offering a multi-threaded industry standard programming environment. This allows path planning and other high-level navigation and control research topics to benefit from a fully featured Unix system even on platforms
too small to carry a companion computer. With an object request broker (ORB) and a standardized API, it can be characterized as Robot Operating System (ROS) equivalent for small aerial or any other size and power constrained systems. The platform has also been successfully utilized in conjunction with Linux companion computers, when more processing power is necessary.

The contributions of this work are as follows: First we first discuss related works concerning existing deeply embedded systems. Second, we introduce the design criteria for such a system. Third, the general software architecture and implementation is presented. Last we show performance results and provide some possible applications for the framework.

## 5.1. Related Work

As robotics research is currently tackling a wide range of problems from estimation and control to high-level planning, communication and middleware systems have emerged to support the increased complexity. Huang et al. created a publish-subscribe design pattern object request broker for robotics systems [11] which is particularly lightweight and low latency. Quigley et al. designed a complete robotics meta operating system which not only includes a middleware layer providing different communication methods including publish-subscribe patterns, but also a library of robotics packages [3]. Our previous work was concerned with a different layer and application domain and was not focused on the deeply embedded scope. It leveraged LCM [11] as middleware and added a set of computer-vision specific interfaces and algorithms [5]. While the presented work has much stronger similarities to Linux / Unix based robotics systems in terms of architecture or features, the most prominent existing research system in the micro air vehicle domain is PPZ [77][78]. It features a complete fixed wing and rotary autopilot suite, but does not support native port-based objects in a publish-subscribe environment or a native ROS integration. The authors of APM [79] have created a complete autopilot system for fixed wing, multicopters and traditional helicopters. While the system shows excellent flight performance, it utilizes an internal function-scheduling routine without sup-
port for preemption. It does support PX4 hardware and middleware and leverages the multithreading capabilities of the platform by running background worker threads. However the design, whilst using worker threads, does not decouple individual tasks and does not implement a general inter process communication scheme or a port-based object architecture. Instead it utilizes different main loops for different vehicle types. The OpenPilot platform [80], which focuses on multicopters, implements a fully multithreaded port-based object solution, but uses an API that is custom to the operating system, therefore making the convergence and reuse between the deeply embedded and Unix platforms more complex. One design feature of the platform is the linkage of inter process message formats to telemetry, which is making interoperability challenging. For a complete discussion of open source systems, please refer to the excellent overview in [81]. Apart from pure onboard systems, hybrid onboard and off board research systems are widely used today. One of the first instances of this class, the system of Lupashin et al.[82] uses a hybrid system. It implements a custom off-board trajectory generation and position control, which is combined with on-board attitude and rate control. The architecture of Michael et al. [83] is very similar, but the system operates on top of the Robot Operating System (ROS) infrastructure.

5.2. Design Criteria for Deeply Embedded Systems

In this section we describe the design criteria for deeply embedded systems. We focus on the real-time aspects of the system. This is mostly defined by requirements for low-latency sensor acquisition and control responses and specialized low level interfaces, such as I2C, SPI, CAN and PWM outputs. The PX4 software architecture is modeled to address typical estimation and control tasks in deeply embedded platforms with a modular approach. We describe the common layered control architecture of robotic platforms, reusability considerations, and interoperability concerns.
5.2.1. Controller Timing Requirements

Robot controllers can generally be modeled as a set of nested control loops. Each control loop has a reference set point and the current vehicle state as input. It generates the reference for the next inner loop. Even more advanced control structures often can be described as a set of nested control loops. The outer loops generally have less strict timing requirements compared to the innermost control loops, thus giving the system designer more flexibility on which platform to implement the outer layers. Figure 5.1 illustrates the typical modeling and control of a generalised robot, following the notation of [83], but generalized to all vehicle types. For a complete example of this design scheme please refer to [84] for multi rotors and [85] for fixed wing aircraft.

5.2.2. Reusability

To date, when compared to open platforms, many research projects run custom hardware and software with very little testing and virtually no safety track record. An open and modular architecture is required to enable research groups to focus on their core research interests. We achieve a high level of reusability, while not sacrificing performance and safety by:

- Clear and clean layering of the software API from low-level to high-level controllers
- Multithreaded node-like architecture that decouples individual applications
- Availability of current sensors and actuator buses
- Expansion bus on the hardware for new sensors
- Safety hard-override in hardware at all times

5.2.3. Interoperability

The PX4 platform offers both compatibility and integration: The cross-platform API supports writing software packages which can be executed
5.2. Design Criteria for Deeply Embedded Systems

Figure 5.1.: Independent of the vehicle type, the dynamical model and nested control architecture can be generalized. Different levels have different update rate and latency requirements.
on the micro controller or on the ROS/Linux system. It provides an interface between the micro controller based solutions and Linux companion computers to run them as a distributed system, leveraging the deeply embedded platform where real-time performance is necessary. The PX4 API allows to use the same code on ROS and on NuttX. For example the code in the listing below executes on both platforms and allows to build and run the same controller on the micro controller, on Linux, or in the Linux Software-in-the-Loop environment.

```cpp
px4::Publisher<px4::rc_channels> *pub = n.advertise<rc_channels>();
```

## 5.3. Software Architecture

In the following section we describe the different layers of the developed software architecture. The software architecture is split into four main layers, as depicted in Fig 5.2: in the lower half, device drivers with device-specific code (e.g. for the particular microcontroller or bus type), and in the upper half, drivers which expose an interface for the system as device node. These are part of the operating system (NuttX [86]). The third layer is the micro object request broker which handles inter process communication efficiently and ensures data integrity between threads. The fourth layer is the application layer, which consists of individual applications (apps), such as flight control or state estimation modules.

Our contribution consists of the PX4 middleware which provides devices drivers and a micro object request broker (µORB). The presented experimental results were obtained using the PX4 flight stack, which is a selection of estimators and controllers developed in close collaboration with the open source community. All hardware plans and the complete source code are available under a permissive BSD (software) and CC-BY-SA license (hardware) on the project website [87]. Although not formally certified, the system design is oriented towards several industry standards: The device drivers and operating system are modeled after the POSIX [88] interface standards. The off-board communication is using the commonly
5.3. Software Architecture

![Software Architecture Diagram]

**Figure 5.2.:** The different layers of the software architecture make the system horizontally and vertically modular.

used MAVLink protocol [89]. The onboard networking is following the UAVCAN standard proposal [90].

### 5.3.1. μORB Middleware

The object request broker provides a data structure for data distribution. It follows the one-to-many publish-subscribe design pattern: A publisher wanting to share information advertises a topic. A topic is defined as a semantic message channel such as ‘attitude’ or ‘position’. A subscriber can subscribe to a topic, and after the subscription is established ask at his own pace for new data (polling), or be woken from the thread sleep state at the instant new data is available. As Figure 5.3 depicts a process can be both publisher and subscriber at the same time, and subscribe and publish to multiple topics.

Our implementation of this design pattern has particular strengths for realtime control applications:

- The topic handle is implemented as virtual file, allowing listeners to
do blocking waits on interfaces and drivers (such as serial ports) and topics in parallel. This is commonly not supported by middleware solutions but saves a complete worker thread.

- The read-write lock of the publication allows efficient concurrency and ensures atomic reads and writes of the complete topic content.

- Subscribers can ask for a notification limit, allowing a subscriber to receive the topic only every N milliseconds. This is important for the efficiency of high-rate topics such as the 1KHz accelerometer updates.

- The asynchronous / blocking wait approach combined with the task priority setup of the operating system allows for minimal latency and deterministic scheduling in the control pipeline. Low-priority tasks and high-priority real time control tasks can be mixed.

### 5.3.2. Applications

Each state estimator and controller in the PX4 stack is implemented as standalone application, which is started with a main() function and then
subscribes / publishes to different topics. Applications can be started and stopped at runtime.

### 5.3.3. Work Queue and HRT Callbacks

For applications that repeatedly only execute one function, such as device drivers, three different work queues to execute callbacks are available: The low priority and high priority work queues and the high resolution timer (HRT) callbacks. The two work queues execute in the normal application context, while the HRT callbacks operate in interrupt context for time-critical functions. Work queue entries are part of the normal scheduling and can access all operating system interfaces, HRT callbacks should be kept as short as possible and only support a subset of the OS API calls. However, HRT callbacks can publish to \(\mu\)ORB topics.

### 5.3.4. Companion Computer

As this system is designed as deeply embedded system, the average robotic application will also provide a companion computer, commonly running ROS on Linux [42][91][5]. We not only offer a ROS interface for feedback and control, but go one step further: Our framework supports the native operation of nodes originally designed for the autopilot on ROS. This is feasible as the node centered design of the deeply embedded solution has the same architecture as on a Linux platform. Therefore we also build our software-in-the-loop simulation based on the ROS native port.

Figure 5.4 shows the architecture of the joint deeply embedded + Linux setup. Some components are exclusive to one of the platforms, e.g. the actuator drivers on the embedded platform or e.g. a simultaneous localization and mapping pipeline on the Linux system. Nodes that suit both environments can be executed on either platform. This has the particular benefit of allowing a proven version of a controller to run on the safety-critical deeply embedded controller, while testing a new version or different implementation on the Linux companion computer. Instead of encapsulating and hiding ROS in our environment, we run native and standard ROS nodes on Linux, allowing researchers already familiar with ROS to easily adopt
the embedded codebase without having to learn a new API. Furthermore any time-critical nodes can be run on the deeply-embedded platform in real-time, whereas a Linux system does not offer that capability without using a real-time kernel on dedicated hardware. Our embedded solution is therefore suitable for low level control and allows to upgrade an existing ROS system to achieve real-time performance.

### 5.4. Implementation

In this section we describe an efficient implementation of the time critical driver layer and then introduce the major hardware components. The software platform is highly portable, but has been implemented for these results on the PX4 FMU hardware, with the main hardware features listed below and a schematic display of the inputs and outputs provided in Figure 5.7. The system offers a serial terminal interface to monitor the system
5.4. Implementation

status (for example system load via the ‘top’ command). System startup is managed through a set of shell scripts and parameters, which allows the full customization of the system startup if required. A MAVLink-enabled centralized parameter storage provides an easy to use (and GUI-supported) management of controller and general system parameters. Even non-experienced users can therefore setup a fully customized system. Parameters can be changed during flight and stored in permanent storage (FRAM).

5.4.1. Driver Layer

The system is clocked using a high resolution timer which supports callbacks on the interrupt level. These callback functions are ideally suited to read sensors efficiently and accurate at a high rate. The interrupt latency / jitter is below 4 microseconds. A wide variety of common peripherals is supported by the system, including MEMS sensors, external airspeed and pressure sensors, PWM, I2C and CAN motor controllers as well as PX4 specific peripherals, such as the flow sensor [92]. The current hardware supports up to 5 serial ports, which can be used to communicate point-to-point or point-to-multipoint with radio modems. The support for commercial-off-the shelf digital radio control systems (S.BUS1/2, PPM, Spektrum) allows the use of low-cost but well-proven manual override solutions.

5.4.2. Hardware

Our middleware supports different variants of microcontrollers, including ST Micro STM32F4, ST Micro STM32F7, Atmel SAM7 and most recently NXP Freescale K64.

FMUv1 Architecture

The autopilot consists of two independent sub-modules, the flight management unit (FMU) and the input-output unit (IO). Figure 5.6 shows the standalone flight management unit rev. 1.7 which is optimized for small scale research systems. We have made FMU + IO available as all-in-one board as well (Pixhawk). The main hardware features are:
Figure 5.5.: PX4 FMU v1.7 open hardware controller board, measuring 50 x 36mm with a weight of 8g. The input/output board (or a custom research module) can be connected.
5.4. Implementation

- 168 MHz Cortex M4F, 256 KB RAM, 2 MB flash
- Hardware single-precision FPU
- MPU6000 gyro/acc, L3GD20 gyro, LSM303D mag/acc
- 14 PWM (servo) outputs total (8 with hard override)
- Triple-redundant power supply inputs with failover
- 5 serial ports (2 with hardware flow control)
- 2 CAN ports, 1 I2C port, 1 SPI port, 3x ADC
- RC inputs: PPM, S.BUS1/2, DSM2/X, RSSI input

The following FMUv2-FMUv4 architectures were minor revision of the overall design, all with a 168 - 180 MHz STM32F4 with up to 384 KB RAM.

**FMUv5 Architecture**

The autopilot allows for triple-redundant designs using UAVCAN communication. This is also the reason no separate override processor was included in the initial design. The main hardware features are:

- 216 MHz Cortex M7, 512 KB RAM, 2 MB flash (effectively 2x the compute of the F4)
- Hardware double-precision FPU
- ICM-20602 accel / gyro, BMI-055 accel / gyro
- 8 PWM (servo) outputs total
- Triple-redundant power supply inputs with failover
- 7 serial ports (3 with hardware flow control)
- 3 CAN ports, 4 I2C port, 4 SPI port, ADC
Figure 5.6.: PX4 FMUv5 reference design implementation (designed in collaboration with TBS)
• RC inputs: PPM, S.BUS1/2, DSM2/X, RSSI input

The rationale for the separation of both units is to allow a hard override to manual control using the safety processor (see Figure 5.7). This considerably improves safety, particularly in research setups.

![Figure 5.7.](image)

**Figure 5.7.** Input and output data streams of the system.

### 5.5. Results

In this section, we show first the performance of the platform while running an attitude estimator. We then give a reusability comparison among existing platforms and finally show the flexibility of our platform.

#### 5.5.1. Performance

Despite running on a resource constrained system, our proposed architecture is capable of processing multiple sensors connected via different embedded
bus systems at 1000 Hz or more each, with an average interrupt latency in the sensor readout of less than 4 microseconds. Context switches between tasks require only 25 microseconds. As depicted in Figure 5.8 the inter process latency is very low. In contrast to other deeply embedded solutions, the sensor drivers directly publish to sensor topics, simplifying the development of higher-level nodes such as controllers, as no knowledge about the embedded interfaces is required. Compared to Linux, the interrupt latency and timings are lower and more consistent even under high load. The system design allows to log different sensors and controller outputs at variable rate when available (similar to a ROS bag), and Python and Matlab tools are provided to plot these logs for flight analysis or replay them in unit test harnesses for filter design.

Figure 5.8.: Latency from publication of topics in one process to reception in a second process, exhibiting very low latency.
5.5.2. Reusability

Recently a wide range of open aerial robotics platforms have become available. Due to their fast evolution, low cost and applicability to rovers and underwater vehicles, they represent general purpose robotic controllers. However, the reusability of these platforms depends on their modularity. Table 5.1 summarizes aspects relevant to the adoption and general reusability, including potential limits induced by the license. The BSD license does not limit the reuse in academic and industrial applications, while GPL licensed code is subject to certain restrictions. The column nodes describes whether one software module (e.g. a controller or estimator) is self-contained and can be easily exchanged against a different module without modifying the core system (equivalent to a ROS node). The column IPC describes if the system is multithreaded and offers a suitable generic inter process communication layer. The column ROS-IF (ROS interface) captures the ROS platform interface. The column ROS-N captures the native ROS support of flight control and guidance software. SITL refers to Software-in-the-loop, a pure software simulation mode. The license column describes the license model.

Table 5.1.: Platform reusability. The five selected platforms performed best out of all evaluated platforms.

<table>
<thead>
<tr>
<th>System</th>
<th>Nodes</th>
<th>IPC</th>
<th>ROS-IF</th>
<th>ROS-N</th>
<th>SITL</th>
<th>Lic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PX4</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>ROS</td>
<td>BSD</td>
</tr>
<tr>
<td>OpenPilot [80]</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>GPL</td>
</tr>
<tr>
<td>APM [79]</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>GPL</td>
</tr>
<tr>
<td>PPZ [77]</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>GPL</td>
</tr>
<tr>
<td>MultiWii [93]</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>GPL</td>
</tr>
</tbody>
</table>
5.5.3. Flexibility

For research applications the ability to adapt the system to new vehicles and setups without requiring fundamental software or hardware changes is critical. Our platform has been utilized in various non-standard vehicle setups, including a novel spherical blimp design [94], but also in various other platforms across very diverse fields [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39].

Here we present a vertical take-off and Landing (VTOL) vehicle setup as an example of a concrete system design, including a Linux companion computer. It is trivial to implement using the platform, despite no VTOL support designed into the system. As our platform design is airframe-agnostic, it was possible to combine the fixed wing and multicopter controllers with a transition controller to obtain first flight results without changing the system architecture. As the evolution of the VTOL control logic progresses, it will later be trivial replace this transition scheme with a custom VTOL controller, as depicted in Figure 5.9. In fact the controllers can be hot-swapped in flight. As the controllers can be shifted between systems or even run in parallel, test-flights are possible with experimental controllers in Linux, with proven controllers as fallback available on the deeply embedded system if the Linux system fails or the controllers implemented on it do not perform as desired.
5.6. Summary

We have presented the first node architecture oriented, fully multithreaded modular robotics framework for deeply embedded platforms using a publish / subscribe design pattern. By design, our architecture is well suited for experimental setups. The ability to run individual nodes as native ROS nodes allows for a very high flexibility in the design process. Our system is highly extensible both in terms of hardware and software, from the addition of individual sensors to using alternate controllers, filters and estimators. Researchers also can just leverage the base system and run completely custom control and estimation pipelines. The use of a POSIX programming model and the publish/subscribe API substantially lowers the barriers for researchers accustomed to ROS or similar non-embedded toolkits to implement deeply embedded solutions. The hardware of our platform is easily and internationally available as open hardware design from multiple vendors and has even sparked the creation of derivatives specialized for partic-
ular use cases.
Part III.

Dynamic Vision & Navigation
Chapter 6.

Reliable Localization

As computer vision is increasingly used in safety critical systems such as driver assistance and drones. This chapter covers design patterns for robust system engineering while the next chapters discusses software introspection and robustness analysis. Intuitively a robust computer vision pipeline to estimate egomotion from images would be one that is strong in rejecting outliers and applies robust photometric and geometric consistency checks in every step of the calculation.

6.1. Related Work

Combining computer vision with inertial data was already a well studied area of research and has since seen substantial further contributions in particular on visual-inertial odometry on mobile phones and robots. Kanade et al. showed vision and gyro fusion on aerial vision systems as early as 1993 in [95, 96]. In industrial applications optical mouse sensors use low-sensitivity silicon combined with very intense lighting to estimate displacement using optical flow. These computer mouse hardware sensors are successfully used for navigation and obstacle avoidance of micro aerial vehicles [97]. Due to the small tracked image area several mouse sensors are
used within one vehicle for multiple direction flow detection in [98]. More complex maneuvers as autonomous take-off and landing based on optical flow sensors have been presented in [99]. A mouse sensor based optical flow module for quadrotor control is shown in [100]. All these systems are not suitable for indoor applications since standard mouse sensors require stronger lighting than present in normal indoor conditions. Systems using a CMOS sensor for optical flow computation have been built using a wireless link for sending the camera images to a computer on the ground, which is processing the images and sending the computed flow value back to the MAV [101] [102]. Dedicated hardware designs implemented in field-programmable gate arrays (FPGA) are used to perform all computations onboard in real-time. The authors of [103] showed optical flow computation in real-time for micro air vehicles. In [104] a stereo camera pair computes optical flow and dense stereo for metric flow in 3D. FPGA systems are, compared to our approach, expensive, large and require further knowledge in hardware description languages.

### 6.2. Fundamental Design Considerations for Robust Vision

The traditional view on estimation is already fundamentally restricting the world to a set of images, very often driven by decades of lab experiments which ensured proper scene lighting and were handheld or ground based with low vibration. The reality for a robot is however very different: Its motion is continuous, subject to vibration, dust, improper sensor calibration and suboptimal lighting. Figure 6.1 illustrates the fundamental differences this induces for the system design: Rather than modelling the 3D geometry very accurately using the implied assumption of reasonably high-resolution images, a realtime vision pipeline leverages the time continuity and potentially additional sensors like IMUs to achieve robustness. The three main differences between computer vision from handheld data and on a drone are lighting / vibration, the texturedness prior and time constraints.
6.2. Fundamental Design Considerations for Robust Vision

Figure 6.1.: The classic frame-to-frame position estimation assumes a discrete, static world. A realtime vision approach considers instead a very high frame rate with very high overlap.

6.2.1. Sampling Strategies for Vibration and Lighting Variation

In computer vision applications on drones lighting and vibration are significant issues. Lighting because the system is moving fast enough that the frame-to-frame exposure can change significantly at a 20 or 30 Hz exposure rate, requiring intensity-invariant computer vision algorithms. A higher frame rate allows to neglect frame-to-frame exposure differences as they remain minimal and a simplistic constant-intensity algorithm can yield excellent results. Vibration is not a widely explored issue in robotic vision yet as most work in the field has been done in handheld and ground based setups where the camera assembly and the overall vehicle were sufficiently sturdy. However, as for any sampling problem the Nyquist frequency is critical to successfully reconstruct a signal and is in particular critical for integrative techniques such as visual-inertial odometry. By increasing the sampling frequency dramatically beyond typical camera frame rates even a very simplistic zero order hold yields very accurate estimates. This is in particular critical for drones as the propeller frequency is 100-180 Hz and the eigenfrequency of many mechanical airframes is around 30-50 Hz. Consequently sampling the camera image at 30-60 Hz is not sufficient to achieve a theoretically sound sampling. In contrast sampling at 400 Hz is sufficient to
reconstruct the full signal even in presence of vibration and one of the core design insights of the PX4FLOW sensor. Figure 6.2 shows data from a real flight log and illustrates that there are several peaks in the spectrum due to vibration.

Figure 6.2.: Frequency magnitude on a real flight. The propeller frequency of 160-180 Hz is very visible and is creating sampling accuracy issues for vision based approaches due to significant vibration on the system. Sampling at 400 Hz can overcome vibration-induced aliasing.

6.2.2. Texturedness Prior

Many structure-from-motion problems start with an object to reconstruct as the problem statement. In many cases these objects have an inherent complexity that makes them hard to measure with a tape or ruler. A typical example is an ancient building facade. This implicitly focused the attention on scenes that have at least average texture. The camera is mostly facing into the direction of the object to be reconstructed.

This prior explains why structure-from-motion pipelines filtered outliers already in some of the first steps, effectively throwing a lot of gradient in-
6.2. Fundamental Design Considerations for Robust Vision

Optimization over gradient in single image

Optimization over high rate image sequence and motion model

**Figure 6.3.** The classic frame-to-frame position estimation assumes a discrete, static world. A realtime vision approach considers instead a very high frame rate with very high overlap.

formation away. The issue for robot vision applications is that because the robot operates in arbitrary environments it typically experiences a lot less texture. It is also not sufficient for a robot to only retain best or correct results: Instead, it typically would want to retain a best-effort estimate with an associated covariance. The direct visual odometry research which has become popular in recent years is targeting exactly this problem [105, 106, 107] by using weak gradients and not just salient corner features.

### 6.2.3. Time Constraints

Many traditional computer vision algorithms have a runtime which depends on the input. Without runtime guards this can lead to self-induced instability: If the algorithm looses tracking of its best estimate the re-initialization can take a considerable amount more time than one tracking delta. If then the flying vehicle starts to pick up speed (colloquially called a "fly away") as the result of the loss of tracking it might leave the area of operation where the vision algorithm is able to achieve frame-to-frame tracking of the veloc-
6.3. Resulting Architectural Considerations for the Vision Pipeline

Combining all these qualitative insights into computer vision on micro air vehicles we are proposing a fundamentally different approach to onboard computer vision. Our approach guarantees worst-case timing, operates at 400 Hz and includes inertial sensors. This very high-speed operation ensures minimal frame-to-frame differences and allows a surprisingly naive implementation to achieve excellent results and vibration rejection.

6.4. Computer Vision in Discrete Hardware

This section describes the architectural considerations of our previous work published in Honegger, Dominik, et al. "An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications." Robotics and Automation (ICRA), 2013 IEEE International Conference on. IEEE, 2013 [92]. Figure 6.4 shows our visual-inertial optical flow sensor. The right hand side of the same figure shows a typical example of street texture at low altitude. Flow is estimated at 250-400 Hz (depending on lighting and resulting shutter speed) using an embedded, computationally optimized variant of the KLT algorithm [108]. Our open hardware and open source sensor estimates the linear motion using a coplanar mounted ultrasonic sensor. The combined hardware-software solution has been widely adopted and used widely across many different robotics research areas [109, 110, 111, 112, 113, 114].

6.4.1. Background

This section summarizes the relation between the pixel-based motion field and metric velocity.
6.4. Computer Vision in Discrete Hardware

(a) The PX4FLOW optical flow sensor

(b) Typical street texture as seen through a 16 mm M12 lens at d = 0.8 m.

Figure 6.4.: PX4FLOW sensor and example data
Basic Equations of the Motion Field

The motion field is created by projecting the 3D velocity field on the image plane. Let \( P = [X,Y,Z]^\top \) be a point in the three dimensional camera reference frame. Let the optical axis be the Z-axis of this frame, let \( f \) denote the focal length and let the center of projection be in the origin. The projected pixel coordinates of \( P \) on the image plane are given by

\[
p = f \frac{P}{Z}.
\] (6.1)

Since the focal length \( f \) is equal to the distance of the image plane to the origin, the third coordinate of \( p \) is constant \( p = [x,y,f]^\top \). The relative motion between the camera and \( P \) is given by

\[
V = -T - \omega \times P,
\] (6.2)

where \( \omega \) is the angular velocity and \( T \) the translational component of the motion. Taking the derivative with respect to time of both sides of (6.1) leads to the relation between the velocity of \( P \) in the camera reference frame and the velocity or the flow of \( p \) in the image plane

\[
\frac{\text{flow}}{\Delta \text{time}} \equiv v = f \frac{ZV - V_z P}{Z^2}.
\] (6.3)

Expressed in x and y components and substituting (6.2) the motion field can be written as

\[
v_x = \frac{T_z x - T_x f}{Z} - \omega_y f + \omega_z y + \frac{\omega x y^2 - \omega_y x^2}{f}
\] (6.4)

\[
v_y = \frac{T_z y - T_y f}{Z} + \omega_x f - \omega_z x + \frac{\omega x y^2 - \omega_y x y}{f}.
\] (6.5)

The motion field components are equal to pure translational parts plus pure rotational parts. The rotational parts are not dependent from \( Z \) and therefore the angular velocity does not carry scene depth information.

The translational components in (6.4) and (6.5) are scaled with the focal
length and the current distance $Z$ to the scene. If the translational velocity is needed, e.g., if the rotational velocity is zero or known (measured by a gyroscope) and compensated from the motion field, it is possible to compute the translational velocity in metric scale by

$$v_{m,\text{trans}} = \frac{Z}{f}.$$  

(6.6)

The combination of the translational part of the motion field and distance measurements of the scene leads to a translational velocity in metric scale if it can be assumed that the distance to the scene is approximately constant. This is especially the case if the camera is faced perpendicular to the ground.

### 6.4.2. System Setup

In the following, we describe an efficient system setup to perform computer vision tasks on a microcontroller. An overview of the setup is shown in Figure 6.5.

The sensor system performs optical flow calculation on images acquired from the CMOS machine vision sensor. It is directly connected to the ARM Cortex M4 microcontroller to a special imager bus peripheral. The microcontroller processes the images in real-time. A frame grabber module captures frames from the sensor and stores them in memory. Optical flow values between two succeeding frames are calculated in the flow module. A refinement to achieve subpixel accuracy and angular rate compensation using the measurements of a gyroscope are performed on the resulting flow values. Finally metric scaling of the flow values as shown in (6.6) is done using the distance measurements of an ultrasonic sensor.

**Frame Grabber**

Pixel data is streamed in the microcontroller using a parallel interface. The frame grabber module samples pixel values at the corresponding pixel clock of the CMOS sensor. Direct Memory Access (DMA) with double buffering
Figure 6.5.: PX4FLOW system setup, the CMOS imager is directly connected to the microcontroller. Image data from the frame grabber module as well as angular rate and distance measurements are stored in system memory using DMA. The flow values scaled with the corresponding distance are sent out.
is used to transfer image data to the memory. Only the current and the preceding frames are stored.

### 6.4.3. Flow Computation and Results

Optical flow is calculated between two successive frames using a reduced variant of the Kanade-Lucas-Tomasi feature tracker.

![Graph showing comparison between Optical Flow and Gyroscope readings](image.png)

**Figure 6.6.** A comparison is shown between angular velocity measurements from the optical flow sensor and gyroscope during a rotation around the vertical axis. Both measurements are shown to coincide closely.

We run two experiments to demonstrate the accuracy of the velocity measurements from the optical flow sensor. In the first experiment, we repeatedly rotate the optical flow sensor around the vertical axis, and plot in Figure 6.6 both the rotational component of the velocity estimates from the optical flow sensor and the angular velocity measurements from the gyroscope that
Chapter 6. Reliable Localization

is located on the optical flow sensor and aligned with the image sensor. The RMSE of the rotational velocity estimates from the optical flow sensor with respect to the gyroscope measurements is 0.192 rad/s. In the second experiment, the MAV moves twice along a square-shaped trajectory which starts and ends at the same point. To obtain the position of the MAV, we do a simple integration of the linear component of the velocity estimates from the optical flow sensor over time; the linear component is obtained by applying angular rate compensation to the velocity estimates from the optical flow sensor. We plot the estimated position of the MAV in Figure 6.8. Figure 6.7 shows the raw velocity output. The rotational noise is obvious in the plot, however, as the sensor samples fast enough to reconstruct the signal it correctly reconstructs the zero mean and outputs the correct integrated result. We observe a position error of 52 cm between the start and end points over a trajectory of 32 m; the resulting loop closure error is 1.63%. The dataset was captured with 250 Hz and block matching instead of KLT flow. This position error is mainly caused by small errors in the velocity estimates from the optical flow sensor that accumulate over time, and the limited accuracy of the magnetometer and accelerometer data used in the attitude estimation by the IMU and which angular rate compensation depends on. Both experiments show the accuracy of the velocity measurements from the optical flow sensor. One limitation of the optical flow sensor is that direct velocity measurements are only available as long as the MAV is at most 5 m above the ground. This is because the maximum range of the ultrasonic distance sensor is 5 m. If a different sensor is available or the ground distance is assumed to be continuous the sensor can be used to estimate ground speed further. Another limitation is the maximum velocity of the MAV. The maximum velocity \( v_{\text{max}} \) that can be measured with the optical flow sensor using the SAD algorithm with a search window of size \( w \) pixels and at \( x \) Hz can be calculated:

\[
v_{\text{max}} = \frac{whx}{f}
\]  

where \( f \) is the focal length, and \( h \) is the height of the optical flow sensor above the ground. In our experiments, the MAV typically flies 1 m above the ground, and we use the following values: \( w = 4 \), \( x = 400 \), and \( f = 666 \).
Hence, the maximum velocity of the MAV that the optical flow sensor works up to is \(2.4\, \text{m/s}\) at 1 meter altitude. At 5 meters altitude it already reaches already \(12\, \text{m/s}\).

\[\text{Optical Flow Velocity}\]

\[
\begin{align*}
\text{y velocity (m/s)} & \quad \text{x velocity (m/s)} \\
-2 & \quad -1.5 \\
-1 & \quad -0.5 \\
0 & \quad 0.5 \\
1 & \quad 1.5 \\
2 & \quad 2
\end{align*}
\]

\[\text{seconds}\]

\[\text{m/s}\]

\[\text{Optical Flow Velocity}\]

\[\text{y velocity (m/s)}\]

\[\text{x velocity (m/s)}\]

\[\text{seconds}\]

\[\text{m/s}\]

**Figure 6.7.** Square shaped indoor trajectory, x and y velocity values measured during movement along square shaped trajectory.

It is well-known that the ultrasonic distance sensor suffers from spikes in distance measurements due to specular reflectance and acoustic noise. We plot the distance measurements from the ultrasonic distance sensor located on the downward-looking optical flow sensor, and the \(z\)-component of the state estimates in Figure 6.9. There is a constant offset between the two
Figure 6.8.: Linear velocity measurements from the optical flow sensor are integrated over time as the MAV moves twice along a square-shaped trajectory which starts and ends at the same point. The green square marks the estimated position of the MAV at the beginning of the trajectory, and the blue triangle marks the estimated position of the MAV at the end of the trajectory.

plots as the optical flow sensor is located below the IMU whose reference frame coincides with the body reference frame. The figure shows that the state estimator gracefully handles the spikes in the distance measurements from the ultrasonic distance sensor. As a result, to maximize the robustness of the state estimator, we use the estimated $z$-component of the estimated state instead of the ultrasonic distance measurement to scale the optical flow measurements to metric velocity measurements.
Figure 6.9.: The ultrasonic distance measurements are plotted with the $z$-estimates from the state estimator over time. The state estimator successfully rejects outlier measurements from the ultrasonic distance sensor.
6.4.4. Outdoor Test

Figure 6.10 shows the integrated metric output of the PX4FLOW sensor with orthophoto overlay for a manual flight with the PX4FMU autopilot on a 7”-propeller small quad rotor along a promenade in a park. The plot shows that the overall consistency with the aerial photo is very high and that the trajectories when closing the loop largely overlap. The estimated trajectory is the pure integration of the measured velocity at each time step. In order to purely show the sensor accuracy, we did not include any further filtering or any motion model. The overall trajectory had a length of 192.16 meters.

Figure 6.10.: Integrated metric output of the PX4FLOW sensor with orthophoto overlay for a flight along a promenade.
6.5. **Summary**

We have introduced the main design considerations for dynamic vision. We proposed a deeply embedded baseline implementation and discussed alternative approaches in literature. Common to all these system designs is the trend towards visual-inertial fusion and the use of dense image data instead of single frames and patches.
Chapter 7.
Robust Vision


Stereo is today the sensing technology of choice for many robotics applications, ranging from autonomous cars to autonomous drones. Still, the reliable operation of a stereo system is non-trivial. Commonly stereo systems are complemented with a second sensor offering a different failure mode to address fundamental concerns about robustness. However, adding a scanning laser range finder or a radar module can be challenging for space- or weight constrained applications. Consumer drones like the DJI Phantom 4 or the Yuneec Typhoon H Realsense rely on a single stereo pair (the IR projector on the Realsense has a very limited outdoor range, making the vehicle mostly rely on passive stereo). This exposes both drones and any other autonomous car or system with a single stereo pair to a major and well-known limitation of the epipolar geometry used for stereo matching: They are susceptible to incorrectly sense the distance to power-lines even if clearly visible against the background if the power lines are close to parallel
Figure 7.1: a)-f): Onboard video showing how a Phantom 4 drone approaches a whole series of powerlines and crashes finally into one of them. Subfigure d) shows the moment immediately before impact and e) and f) show the uncontained failure of the attitude control of the vehicle.
to the axis connecting both camera centers. Power lines are just one out of a huge set of cases in man-made environments where linear structures with weak texture cause a normal stereo head to fail. Figure 7.1 shows six stills from a video captured by a Phantom 4 drone which is equipped with a stereo camera. The stereo camera is not able to detect horizontal lines.

The stereo correspondence search is completely random when a line is present in the image as illustrated in Figure 7.2 with a close-up of one of the failure regions from the results section. Even worse, because all of the potential matches along the search region look the same, their matching score is excellent, giving the wrong result a very high confidence score.

Instead of adding a different type of range sensor we propose an alternate solution which boosts the precision and recall over a single-baseline stereo system using additional cameras. We perform all computations on the same hardware as with the standard stereo system, adding only the cost and weight for two miniature, low-cost image sensors over a standard stereo head. We show that we can successfully reject wrong measurements and increase the recall in areas of the image where the stereo correlation confidence was low and otherwise no reliable measurement would have been available. Our contributions in this paper are:

1. We propose a novel heuristic to predict the failure of the stereo matching.

Figure 7.2: Line in the image aligned with the stereo search region along the epipolar line. The best match is dominated by image noise and random, but achieves a high score as the intensity difference is very low.
2. We propose a cost function which allows to merge two orthogonal stereo depth maps into a single depth map to resolve the failure

7.1. Related Work

We consider two main areas for related work: Realtime stereo systems and algorithms targeted at stereo obstacle detection and navigation applications. Denker and Umlauf [116] showed a four-camera stereo system generating a single depth texture. In contrast to the approach presented here they did not predict epipolar geometry failures but rather fused the depthmap based on the matching score. Since the matching score on a line is excellent this leads to a 3D artifact visible on a 2D line in their presented results.

Due to its parallel characteristics FPGAs are successfully used to accelerate computer vision algorithms. Especially stereo matching fits well within the parallel FPGA architecture. The system in [104] produces disparity maps and optical flow field at 127 fps and 376x240 pixels resolution based on block matching. Another system that leverages an FPGA to determine optical flow values and position information is shown in [103]. A real-time stereo matching implementation based on a census rank transformation by [117] is shown in [118]. Real-time implementations of global cost constraint algorithms as SGM [119] on FPGAs are shown in [120, 121]. A combination of FPGA, CPU and mobile CPU mounted on a quadrotor to estimate disparity values and odometry is shown in [122]. None of the related FPGA implementations output the underlying cost map of the disparity estimation.

Woo et al. [123] presented a method to reconstruct 3D lines from stereo depthmaps and a related technique to match lines in 3D from 2D images to obtain stereo lines [124]. Li et al. leverage neural filters to directly filter the image content for power lines [125]. Fraundorfer et al. [126] presented a system capable of fully autonomous flight, planning and avoidance using a forward-looking stereo head and optical flow sensor.
7.2. Epipolar Geometry

The depth estimation in stereo leverages the epipolar geometry of the stereo setup and searches for correspondences along the one-dimensional epipolar lines only. Stereo algorithms commonly reject wrong measurements using the correlation score for the best location: If the error is above a certain threshold the measurement is rejected. In addition commonly a texturedness indicator is used to avoid correlating areas without texture.

7.2.1. Fundamental limitations

However, there is one fundamental limitation which is unavoidable: If a well-textured element is repetitive along the epipolar line it satisfies the texturedness and the correlation score criteria. Even worse, its correlation score will be excellent. It is trivial to show that this fundamental property is un-

Figure 7.3.: Urban scene as photograph and corresponding depthmap.
avoidable. Unfortunately this corner case is very common in man-made environments where straight lines are extremely common. Figure 7.3 shows a typical urban scene which is reasonably well textured. However, it has enough features causing stereo correlation failures. This problem is well studied in the structure-from-motion domain and often solved implicitly in photo-consistency checks. For an excellent discussion please refer to [127]. However, because we are focused on a real-time application domain for robotics and in particular micro air vehicles we propose a novel and more efficient approach leveraging a property of the real-time stereo algorithm employed.

### 7.2.2. Realtime stereo in hardware

In order to understand the constraints of a hard-realtime stereo system we need to quickly introduce the operation of stereo on hardware: The field-programmable-gate-array (FPGA) does not store a full frame from the camera before processing. Instead it only buffers very few scanlines of the image sensor and calculates the depthmap continuously for new lines fed into the system. This allows for an extremely fast processing of up to 10 cameras with a single chip, but it imposes several limitations on how the images can be accessed and stored. Not every stereo algorithm is suitable for this type of operation. One suitable algorithm is semi-global-matching (SGM). It improves over plain block matching by optimizing the resulting depthmap in multiple directions. One such path is parallel to the epipolar direction.

The SGM algorithms operates on the image intensity values and does not propagate disparity values over large gradients in the image. However false disparity matches along the epipolar direction do not contain any perpendicular gradients and the SGM algorithm propagates the false match in direction of the epipolar line. We leverage this particular property of its smoothing step to detect failures.

In order to determine possible faults of the epipolar geometry we exploit that the matching score provided by SGM peaks when it encounters lines which are close to parallel to the epipolar line: We are comparing the line-to-line matching scores to identify regions where the estimation likely failed. The effect can be seen in Figure 7.3 where parts of the sidewalk of a bridge
are estimated at the wrong depth.

7.3. Setup

This section shows the general system structure of the orthogonal camera setup. An overview of the setup is shown in Figure 7.4. Four image sensors are connected to a single FPGA using low-voltage differential signaling (LVDS) interfaces. The cameras are arranged as two stereo pairs with orthogonal baseline. The data streams of all four cameras are processed within the FPGA in real-time. The FPGA synchronizes the four independent data streams and performs a warp operation on the images to align them for epipolar geometry and correct for lens distortion effects. A SGM stereo algorithm as described in [128] outputs disparity values as well as the corresponding matching score. The overall latency introduced by the FPGA system is less than 2ms. Processed and raw image data is sent to a host computer using a USB 3.0 interface.

All four cameras are rectified using the intrinsics and extrinsics obtained during an initial calibration run using 2D markers and the Kalibr framework as described in [129]. We compute stereo for the horizontal and vertical pairs and a horizontal and vertical epipole in hardware using semi-global-matching (SGM) stereo. The output of the computation is a depthmap and the per-pixel matching score, which will serve later as a confidence indicator. Both depthmaps and both confidence maps are then transferred via USB 3.0 to a connected Linux computer. This is an Odroid XU-4 for on-board applications or a common Linux laptop for bench testing. The images are published as ROS messages and then fused in a ROS node using the following algorithm.
Figure 7.4.: Sensor head with two orthogonal baselines. SGM stereo is calculated directly on the hardware for both pairs and output synchronized via USB 3.0
7.4. Failure Heuristic

The overall processing pipeline is depicted in Figure 7.5. As we match in two directions our overlap area between both orientations is a square region. The input depthmaps have already a left-right check applied to them.

7.4.1. Fault prediction

We evaluated several options for this classifier: Using the left camera image directly, using the resulting depthmap and using the matching scores of the SGM algorithm. The qualitative evaluation showed that the SGM scores were by a large margin the best predictor in all scenes, which is why we will consider them exclusively for the remainder of the paper.

As the input depth map is median filtered we adjust the scale space of matching score image by applying a $13 \times 13$ gaussian blur kernel with $3\sigma$. The next step is to predict the most likely fault locations. We leverage the previously discussed property of SGM to not optimize over a strong gradient, which results also in lines in the score image. By finding the gradient magnitude for these score differences we can reliably predict line structures parallel to the epipolar line. In order to achieve this, we convolve the score image with a one-dimensional Sobel kernel in perpendicular direction to the epipolar line with a 7 pixel region and $0.3\sigma$ Gaussian. The insight that a significant change in the matching score along the epipolar line is a reliable predictor for wrong measurements is at the core of this contribution. Because the perpendicular stereo pair will have an excellent response in this region, we error on the safe side and and grow the region where we detected a geometric fault by applying another Gaussian blur with a $21 \times 21$ region and $10\sigma$. This results in over-estimating the error region and taking the depthmap surrounding the fault prediction from the orthogonal stereo pair.

After the last step we obtain the final epipolar fault hypothesis image. The fault prediction map for the fault present in Figure 7.3 is shown in Figure 7.6. We calculate one such prediction image for each depth map. The last step is to create the final depth map and final score map.

The final score map at every pixel location $c_{i,j}$ is a weighted average of the matching score $m_{i,j}$ from image $a$ or $b$ from SGM, chosen from the image
Figure 7.5.: Block diagram of our pipeline: We use the stereo depthmap and confidence map as inputs and generate a single fused depthmap with the lower fault score $f_{i,j}$.

$$c_{i,j} = m_{i,j} + s \cdot f_{i,j}$$

We found empirically a value of $s = 2$ to deliver a stable tradeoff between the SGM matching score and fault prediction. All experiments where performed with the parameter fixed at this value. Bright areas in the image indicate likely failures of the stereo correlation whereas dark areas have a high confidence to not have a geometrical flaw and sufficient texture.

7.4.2. Final depthmap fusion and quality score thresholding

Earlier work concerned with real-time depth map fusion of multi-camera stereo setups fused the depth maps by using the weighted average of the matching score of each stereo pair. The issue with this approach is that the matches along a line are perfect and hence the matching score provides an improper weight to the depth map fusion. Our above matching function instead selects one source image for each pixel $i, j$ in normal operation, based on the fault score. However, if both depthmaps are within 10% range of each other at a pixel location $i, j$ we fuse these pixels by calculating
Figure 7.6.: Output of our algorithm. The confidence map indicates regions where a failure is very likely as bright regions.
their weighted average using their matching score $m_{i,j}$. This provides a robust cross-check for the depth estimate and allows to reliably accept pixels that do not have a very high correlation score assigned by SGM. We found the effect most noticeable in weakly textured surfaces like tarmac, where this step considerably increases recall.

7.5. Results

In this section we present a series of indoor and outdoor experiments in different settings to validate our algorithm in man-made and natural environments. We provide quantitative measurements against a handheld laser range finder for the indoor experiment and qualitative results showing the depth maps of the well-known SGM algorithm against our fused version. In general the results are presented with the horizontal stereo pair on the top left, the vertical stereo pair on the top right and the fused result in the center. The bottom row shows the horizontal epipolar failure prediction on the left, the vertical epipolar failure prediction on the right and the resulting confidence map fused from both in the center. The final confidence map also includes the SGM matching scores as a weighted average.

7.5.1. Indoor Experiment

Our indoor experiment shows a typical office setting where well textured surfaces are rare. We found that in many frames both the horizontal and vertical orientations both showed corner cases. We created a scene in our office with significant depth range. The office is naturally weakly textured, so a product package was added to also show the performance on well-textured indoor scenes. Fig 7.7 shows that the horizontal pair fails on part of the table / product packaging while the vertical pair fails on the table leg and door frame. Both failures are detected and compensated in the final output. The final disparity map contains structures aligned with the vertical and horizontal epipolar line, what is not feasible using a single epipolar line setup.
7.5. Results

Figure 7.7.: Indoor experimental results: Top left: Horizontal epipole disparity map, top right: Vertical epipole disparity map. Top center: Fused disparity map with less artifacts introduced by structures aligned with the epipolar line direction. Bottom left: Failure score of the horizontal epipole setup. Brighter areas are more likely to contain incorrect correlations. Bottom right: Failure score of the vertical epipole setup.
7.5.2. Outdoor Urban Experiment

The outdoor urban scene consists of residential buildings with a park. The regular horizontal and vertical structures induce major challenges for a single stereo system as they create significant artifacts whenever the epipolar line is aligned with the structure by chance. Figure 7.8 shows how the horizontal pair fails at properly identifying the depth of parts of a horizontal structure (orange). However, there is also a more subtle but equally significant failure on the ground distance: Although the ground looks like a nice gradient its distance is not actually measured correctly. It is an inherent property of the SGM optimization step to smooth weakly textured parts which can lead to visually pleasing results in terms of a nice depth gradient, which is however incorrect. Our fault detector correctly predicts this effect and Figure 7.8 shows that the correct estimate is used from the vertical stereo pair instead.

7.5.3. Outdoor Powerlines Experiment

The powerline footage was captured at about 8 meters altitude near train tracks to test a drone application scenario. The capture location is shown in Figure 7.10. Since powerlines are typically almost horizontal and weakly textured they present a major hazard to any horizontal-aligned stereo system like on current consumer drones (DJI Phantom 4 and Yuneec Typhoon H Realsense). Figure 7.10 shows that the horizontal stereo pair (as mounted on consumer drones) is not able to judge the distance of powerlines in the image correctly.

7.5.4. Outdoor Brush Experiment

To validate qualitatively that our fusion does not deteriorate the performance in well-textured scenes we performed this experiment in a natural environment, as depicted in Figure 7.11. While we did not expect or encounter corner cases in this setting we were able to show the fusion is not deteriorating the performance.
Figure 7.8: Outdoor experimental results: The horizontal stereo pair is failing on the ground level. The red box highlights the curbstone, which is incorrectly smoothed in the horizontal stereo pair. Top left: Horizontal epipole disparity map, top right: Vertical epipole disparity map. Top center: Fused disparity map with less artifacts introduced by structures aligned with the epipolar line direction. Bottom left: Failure score of the horizontal epipole setup. Brighter areas are more likely to contain incorrect correlations. Bottom right: Failure score of the vertical epipole setup.
Figure 7.9.: Powerline scene as photograph and corresponding depthmap.
Figure 7.10.: Powerline experiment results: The horizontal pair is completely failing at identifying the right distance. The powerlines are less defined after fusion as their weak texture against a textured background also led to a low correlation score in the vertical stereo pair in some locations. Top left: Horizontal epipole, top right: Vertical epipole. Top center: Fused depthmap. Bottom: Failure score. Brighter areas are more likely to contain incorrect correlations.
Figure 7.11.: Outdoor brush results: The horizontal stereo pair is failing on the ground level on the left side, but the overall recall is excellent as expected in a natural scene. Top left: Horizontal epipole, top right: Vertical epipole. Top center: Fused depthmap. Bottom: Failure score. Brighter areas are more likely to contain incorrect correlations.
We have demonstrated that well-known limitations for stereo in man-made environments can be predicted reliably using a highly efficient minimal processing step and can be addressed using an orthogonal stereo pair. As the fundamental error case is not preventable in sparsely textured environments without additional sensors this represents a viable solution for real-world applications. To keep the resulting processing burden in check FPGA-enabled multi-camera hardware can be a significant contribution to the future safety of autonomous systems.

We have contributed a model to predict the failure of stereo matching for the case of the real-time semi-global-matching implementation on field programmable gate arrays (FPGA) and we proposed a suitable cost function to merge orthogonal stereo pairs into a single depthmap.

We have presented a loosely coupled approach in this paper and will explore the option to directly embed the algorithm into the SGM matching step on the FPGA in our future work.
Part IV.

Summary
Chapter 8.

Conclusion

In this dissertation we presented a holistic software architecture for autonomous aerial robots navigating with computer vision. The presented architectural concepts are independent of the implementation and represent fundamental principles. Each architectural concept is fully implemented in software and has many flight hours in real vehicles. As a result, this dissertation has had already great impact. The middleware derived for this work has been deployed to hundreds of thousands of systems according to manufacturer data and is omnipresent in research labs around the globe. It has been adopted by a non-profit organization (Dronecode) designed to advance the use of open source software in the drone industry.

- We contribute fundamental design principles for a hybrid real-time control system using port-based objects. We have shown that the fundamental principles of the architecture satisfy formal requirements and can be verified using set theory. The implementation has been verified using extensive simulation, unit tests and intense flight tests in real vehicles. The proposed architecture is therefore sound from a theoretical as well as from an implementation perspective.

- Our architecture is fully multithreaded across hybrid systems and highly
scalable. We have developed a design pattern that allows to instantiate a port-based object on a CPU with 196 KB of RAM or 16 GB of RAM.

- The derived architecture has been shown to perform as expected in terms of dynamic reconfiguration on vertical-takeoff-and-landing airplanes. These airplanes use the rotary wing as well as the fixed wing controllers from PX4. We showed that the architecture can support either controller scheduling or replacing the different vehicle controllers with an unified VTOL controller.

- We have explored the fundamentals of robust computer vision and presented robust optical flow and robust stereo using specialized hardware. We have shown how this leads to robust results in resource-constrained environments and how currently unsolved problems like the reliable detection of power-lines can be addressed with tightly integrated vision systems.

The resulting software is widely used in the drone industry and forms the backbone of many academic and commercial vehicles. When the hybrid system architecture was conceived initially early in the timeframe of this PhD, it marked the state of the art in robotics system architecture and has become the standard and is ubiquitous today. While system architecture had already tremendous impact on the field, the robust realtime computer vision paradigms in this thesis are merely the foundations for what could be a long line of future research. Only the direct interfacing of the image sensor and running it in a special configuration allowed to increase its output rate to 400 Hz, which in turn completely changes the architecture for the motion estimation pipeline consuming this data. In the same spirit the hardware-synchronization of two stereo pairs and the access to the low-level matching scores obtained by the stereo hardware allowed completely new approaches to solve a fundamental reliability issue of stereo in real-world applications. These contributions should serve as examples that there is still a lot of potential for computer vision on deeply embedded hardware and we have shown that this could be a means to solve some long-standing reliability and robustness issues that have limited the potential applications for computer vision.
Chapter 9.

Future Work

While the presented work has had already much more impact in the field than ever anticipated, there are of course many directions that were left unexplored. While in the last decade the focus has been on the vehicle software and architecture, it is becoming increasingly important to focus on the overall global system of micro air vehicles. The industry is already starting to explore cloud-based solutions for drones, but the exact architectural patterns in this space have not been explored in great depth yet. As with any maturing software ecosystem there are three main considerations for the further evolution:

- With more and more micro air vehicles being airborne at the same time and manned autonomous aviation, also called flying cars, not unthinkable anymore, the airspace management at high-speed and micro-resolution has become a major concern. At the same time there is also not yet one obvious solution for this challenge, neither on the physical layer of the communication nor in the architecture.

- Software distribution and deployment has been solved for smartphones with app stores, but the exact pattern for head-less internet-of-things devices like micro air vehicles is still unclear and there is not one standard architecture or industry solution available. Research focused on
software reuse would be very beneficial to show industry options that allow for a free market, rather than oligopoly of 2-3 app stores. When it comes to safety and certification it also raises interesting questions as to how much the customer is able to change the configuration of a micro air vehicle.

- Related to distribution but much more concerned with packaging are software containers. These sandboxed containers are a common architectural pattern and are called apps on smartphones. Containers for micro air vehicles would need to allow execution on very low-level hardware, raising new research questions regarding their architecture.

Computer vision on autonomous robots has made tremendous progress in the last decade but its robustness is still a concern. This dissertation has shown that by working very close to the silicon of the image sensor modern computation hardware can be used to robustify the process greatly.

- Tightly coupled vision systems like visual-inertial-odometry or hardware-based stereo have not been explored to their full depth yet.

- Traditional computer vision pipelines reduce the amount of data as they go through their computation steps and do not carry the uncertainty estimate through every step of the 2D and 3D computations. In order to achieve higher levels of robustness, a new class of probabilistic algorithms is needed, and the field is already gravitating into this direction with photometric and dense approaches.

- Using online databases and online aiding sources will be an important part of future computer vision on drones. Rather than sensing the complete environment they could leverage existing representations and merely identify the difference, allowing them to achieve globally optimal avoidance and navigation solutions by leveraging a-priori knowledge.
Bibliography


Bibliography


Bibliography


[80] OpenPilot: OpenPilot project. Website (2014)


[87] PX4: PX4 project. Website (2014)


[90] UAVCAN: UAVCAN project. Website (2014)


150


Bibliography


