Portland State University

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical & Computer Engineering

DRAFT II: A Multi-Agent System for Adaptive Control of a Flapping-Wing Micro Air Vehicle

Author:
Michal Podhradsky

Advisor:
Dr. Garrison Greenwood

Dissertation Committee:
Dr. Richard Tymerski
Dr. Marek Perkowski
Dr. Wayne Wakeland

September 9, 2016

Abstract

Biomimetic flapping-wing vehicles have attracted recent interest because of their numerous potential military and civilian applications. In this dissertation is described the design of a multi-agent adaptive controller for such a vehicle. This controller is responsible for estimating the vehicle pose (position and orientation) and then generating four parameters needed for split-cycle control of wing movements to correct pose errors. These parameters are produced via a subsumption architecture rule base. The control strategy is fault tolerant. Using an online learning process, an agent continuously monitors the vehicle's behavior and initiates diagnostics if the behavior has degraded. This agent can then autonomously adapt the rule base if necessary. Each rule base is constructed using a combination of extrinsic and intrinsic evolution. Details of the vehicle, the multi-agent system architecture, agent task scheduling, rule base design, and vehicle control are provided.

Acknowledgments

Here comes the Acknowledgments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Biomimetic *flapping-wing micro-aerial vehicles* (FWMAV) have been the focus of much recent research due to their potential for both civilian and military applications. Because of heir insect-like size and relative simplicity in comparison with more traditional unmanned aerial systems (such as quadrotors or fixedwing airplanes), they can be made relatively cheaply and be a part of personal equipment of soldiers, first responders, law enforcement members etc. Probably the most well-known is their application in military reconnaissance – a soldier launches the drone in the air to get a better view of the battlefield, and to spot enemies and obstacles. Small pocket-size drones are already being tested for military use [1]. Similarly, insect-like robots can be used for law enforcement and surveillance – helping SWAT teams locate suspects, or monitor crowds. Ground robots capable of stealthy surveillance are already available [2].

Firefighters use drones to monitor fire from above [3], but an insect-like drone can fly inside a burning building and help access the fire damage without exposing fire crew to danger. First responders would benefit from miniature drones that map dangerous environments before humans step in, and that measure radiation and toxic levels in contaminated areas (if equipped with proper sensors) after a fire or an

environmental disaster. One application unique for miniature flapping-wing robots is artificial pollination – in the catastrophic event that there is not enough bees available, robotic pollinators can take over and help farmers where needed [4]. Yet another, perhaps more exotic use is planetary exploration as suggested by [5].

In either possible application area, adaptive, fault-tolerant, control and autonomous operation is paramount. One way how to achieve the desired level of autonomy is with a *Multi Agent System* (MAS) – where different agents inside of the robot's controller are responsible for specific tasks. For example, one agent is responsible for adapting the flight trajectory, while a second agent is responsible for monitoring the "health" of the robot, and a third agent can help during a fault recovery. This way even a very complex system can be split into simpler tasks. *Subsumption architecture*, another concept well known in robotics described in Section 2.3, is used for prioritization over different goals, which gives the robot the desired autonomy. The concept of MAS combined with the *subsumption architecture* is general enough so it can be used on various robots for different applications. In our research we use evolution algorithms, which can quickly evolve the robotic control system and tune it for a particular application.

1.1 Problem Statement

In the aforementioned examples, the existing drones are remotely controlled by a human pilot. They have some sense of autonomy (i.e. a "hold" function that keeps the drone hovering on a spot while the operator for example pans the on-board camera), but are not able to perform fully autonomous missions. Nonetheless, greater autonomy is not always well-received by the public (for example see [6]), so it is important to keep the related ethical questions in mind [7] [8].

FWMAVs, unlike their counterparts with rotating blades and fixed wings, have

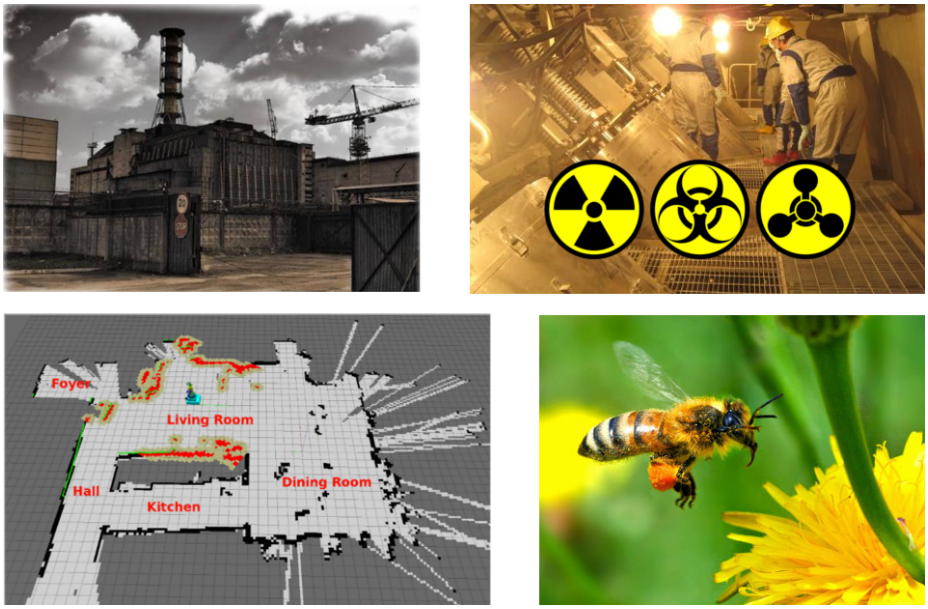

Figure 1.1: An example of applications of flapping-wing robots. Top left: Detecting toxic materials; Top right: Measuring radiation levels in contaminated areas; Bottom left: Mapping of dangerous environment; Bottom right: Artificial pollination

the potential to be truly insect-size. The reason is that the conventional technologies used for macroscale aircrafts do not scale well, because the decreased size brings an increased dominance of surface forces, causing revolute joints or sliding sliding surfaces (for example propellers shafts and motor bearings) inefficient or even infeasible [9]. In addition, the lift-to-drag ratio for fixed aerofoils decreases at small scales because of the greater effect of viscous forces relative to lift-generating inertial forces at low Reynolds numbers [10].

A FWMAV qualifies as a *Cyber Physical System* (CPS). A CPS is a system where a controlled physical process and information processing is tightly coupled. Working with CPS introduces unique challenges, which is described in Section 2.2. This work describes a MAS for adaptive control of a FWMAV. The vehicle employed in this research is similar to a minimally-actuated FWMAV introduced by Wood [11], [12] with core control laws introduced and subsequently refined by Doman et al. [13], [14]. Our vehicle [15], [16], [17] operates similarly to the minimally actuated vehicles considered by Wood and Doman et al. in that all propulsion and control are provided by two

wings, each of which possesses a single active and a single passive degree of freedom. As previously mentioned, the two wing configuration allows full 6 DoF control and is mimicking a bee or a fly.

Previous work employed variants of the controllers discussed in [13], [14] augmented with adaptive wing beat oscillators [18], [19], [20] that provided adaptation at the inner-most layer of vehicle control (wing flapping patterns). The goal of that work was to provide adaptation not by changing control laws that related desired forces and torques to stereotyped wing motions, but rather, to change the wing stereotyped motions to adapt generated forces to the needs of control laws designed for undamaged wings. In other words, the salient adaptation was that damaged wings learned to move in ways that mimicked the force and torque generation of undamaged wings. As a result the higher level controllers would not be able to distinguish between normal operation and damaged wings.

In contrast to that work, this dissertation describes a design for a MAS where control laws are directly adapted at a higher level of abstraction in the control law hierarchy. In this system, agents are responsible for collecting and estimating vehicle pose, recording waypoint locations for trajectory following, generating inputs needed by the split-cycle oscillator, monitoring vehicle behaviour and, when necessary, conducting diagnostics and adapting the control rule base. In this case the higher level control system monitors (directly or indirectly) the wing performance and adapts if necessary, while the inner-most layer of control is fixed. The initial set of control laws is designed using a combination of extrinsic and intrinsic evolution [21]. The ultimate vision is that both forms of adaptation co-exist in the vehicle so that the benefits of each approach are equally available.

1.2 Research Objectives

The former approach [15], [16], [17] focused on lower level control, and didn't provide a complete solution. The later approach, described in this dissertation, utilizing a MAS, provides a complete high-level control as well as autonomy to the vehicle. In combination with evolution inspired techniques, the control system is able to adapt to different vehicles and to different conditions (for example damaged wings). The main contribution of our research is that it proves the viability of a MAS for a FWMAV by demonstrating the autonomous waypoint following, and the concept of likelihood based fault detection procedure. The major outcomes of this research are:

1. understanding the viability of a MAS for control of flapping wing vehicles
2. developing a multi-agent control system allowing the vehicle to follow trajectory in experimental settings
3. developing fault detection and fault recovery mechanisms based on a combination of extrinsic and intrinsic evolution
4. developing high degree of autonomy of the vehicle, including trajectory following and fault recovery procedures

This research is supported in part by the National Science Foundation under Grant Numbers CNS-1239196, CNS-1239171, and CNS-1239229.

Chapter 2

Background

This Section provides background information related to our research. Basic concepts of FWMAV as well as the state-of-the-art research in the area is discussed in Section 2.1. Readers familiar with these concepts can skip the later sections, specifically Sections 2.2, 2.3, 2.4 and 2.5.

2.1 Related Research

Flapping-wing micro-aerial vehicles are relatively new research topic, with first concept papers appearing in early 2000's [22] [23] [24] [25]. To this date, the principles of flapping-wing flight are relatively well understood, including understanding of insect flight dynamics [26] [27] [28] and related aerodynamics phenomena [29] [30]. Researchers were able to successfully apply system identification methods (that were used on small-scale UAVs before, for example [31]) on insects [32], as well as on insect-size FWMAV [33]. To this date, a free-flight of a fly-size FWMAV was achieved [34] (although the power supply and data processing was external to the vehicle), as well as an untethered flight of a locust-size FWMAV with battery, sensors, radio and control unit on-board [35]. A comprehensive overview of modelling techniques is presented in [36].

First we will address the modelling and design of FWMAV, which is indeed very important for development of new vehicles and improvement of the existing prototypes. Then we will review several different approaches to the power system of a FWMAV, and discuss the commonalities and differences between them. Finally we will present most common on-board sensors that are a precursor of an autonomous FWMAV flight.

The most important part of the flapping-wing vehicle are the actuators (or "flight muscles"), since they provide propulsion to the robot. There are two main power technologies used – either a brushed or brushless Direct-Current (DC) motor, or a piezoelectric oscillator. The DC motors are commonly used in larger robots and MAVs, but unfavourable scaling of magnetic forces limits the achievable power densities in small electromagnetic motors, making them unusable for the smallest fly-size FWMAVs [9] [10]. Piezoelectric oscillators offer a better weight-to-light ratio, and be manufactured small enough for sub-centimetre robots. Yet they have a limited range of frequencies they can operate at, and require relatively high voltage of at least 150 Volts [37] which makes them more difficult to work with. DC motors are more suitable for larger FWMAVs with take-off weight of a few grams. Piezoelectric oscillators are suitable for vehicles lighter than 1 gram. In both cases the actuators require a transmission, because neither DC motors, nor oscillators are suited to connect directly to the wings.

2.1.1 Oscillators

Piezoelectric bimorph oscillators consist of two layers of piezoelectric material, with another layer of flexible material in between as shown in Figure 2.1. When sufficiently high voltage is applied across the piezoelectric layers, one layer will expand while the other layer will shrink, causing a bending effect. Application of sinusoidal signal will result into oscillatory movement. Note that amplitude modulation of the driving signal changes minimal and maximal position of the oscillator, while applying an

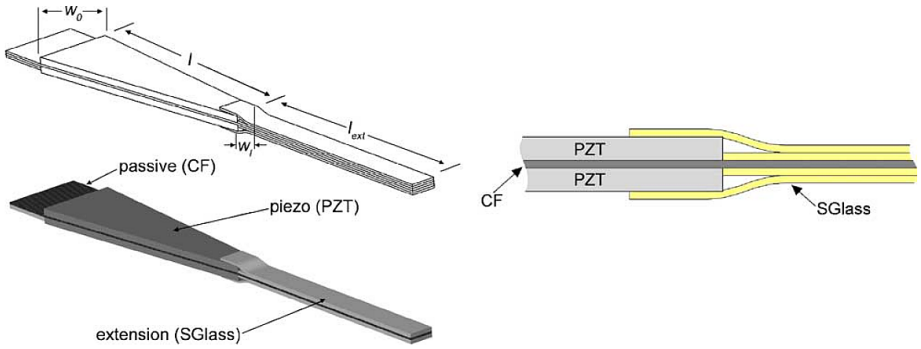

Figure 2.1: Piezoelectric bimorph oscillator [37]. CF -Carbon Fiber, SGlass - a high stiffness fiberglass, PZT - Lead zirconate titanate

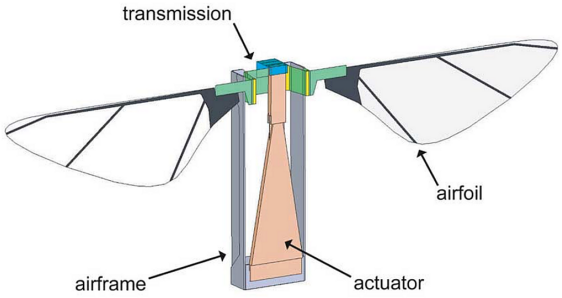

Figure 2.2: Conceptual drawing highlighting the main components of FWMAV utilizing piezoelectric oscillator [37]

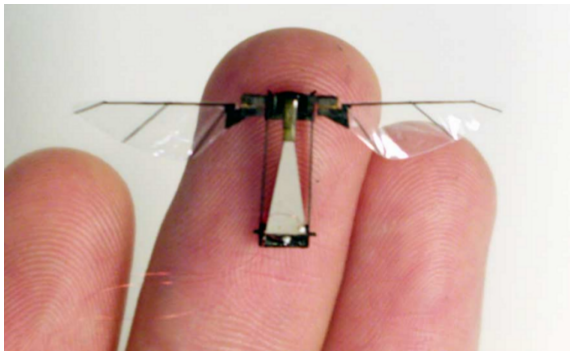

Figure 2.3: An assembled Robobee [37]. Note the piezoelectric oscillator taking up most of the inner airframe.

offset voltage shifts the middle position of the oscillator. Frequency modulation of the signal then changes the frequency of the oscillation [34]. All three effects are important for control as will be shown later. Properties and design of piezoelectric bimorph oscillators were studied in great detail, we would refer the reader for example to [38] and [37]. An example of a FWMAV utilizing this type of actuator is shown in Figures 2.2 and 2.3.

Flapping-wing flight is energetically costly [9] as the inertia of the constantly oscillating wings has to be overcome in addition to the high aerodynamic drag [39], so it is essential to have as efficient power system as possible. One way, which was observed at fruit flies and blowflies, of achieving high efficiency and minimizing the additional inertia of the wings is to drive the wings at their mechanical resonant

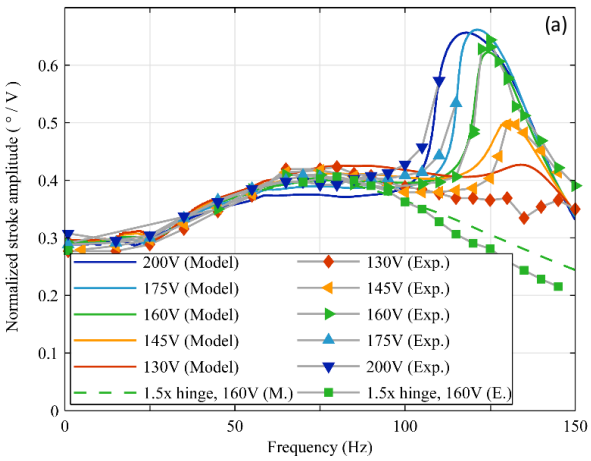

Figure 2.4: Data and model fit for normalized stroke amplitude for various voltages applied to a piezoelectric oscillator. Note that for all applied voltages the resonance peak occurs between 110 and 130 Hz. Details of the experiment in [40]

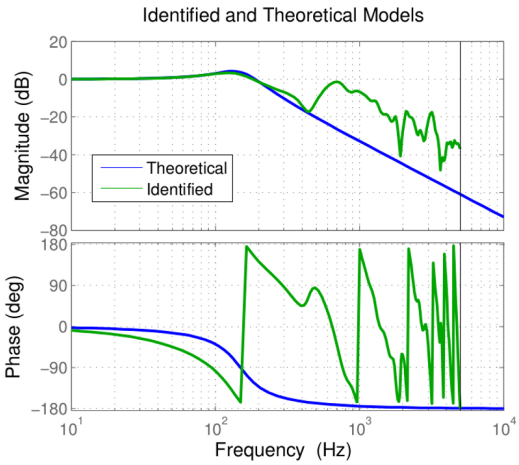

Figure 2.5: A bode plot of linearized theoretical model and identified model of Robobee [33]. Note that in both models the resonance indeed occurs between 100 and 150 Hz which is consistent with observations.

frequency, [39]. Existing systems, such as [11] were indeed designed to drive the wings at their resonant frequency.

The effect of resonant frequency is most obvious at the smallest scale (i.e. fly-scale robots), because the ratio of wings inertia to actuator inertia is large. A good example is a Harvard Robobee [11] for which was this effect both theoretically predicted [23] and experimentally measured (see in Figure 2.4). A linearized identified system model of the Robobee shows identical resonance frequency, as shown in Figure 2.5 As a result, a practical use of piezoelectric oscillators is challenging, because the oscillating frequency of the actuator has to match the resonance frequency of wings. Changing the oscillating frequency of the actuator is non trivial and depends on the size, shape, thickness and material of the oscillator [37]. Oscillators also require a special driving circuitry with matched impedance for correct function [40]. Finally the transmission is directly linked, so it changes torque ratios between the actuator and the wings, but can't change the frequency ratio (which is always 1:1).

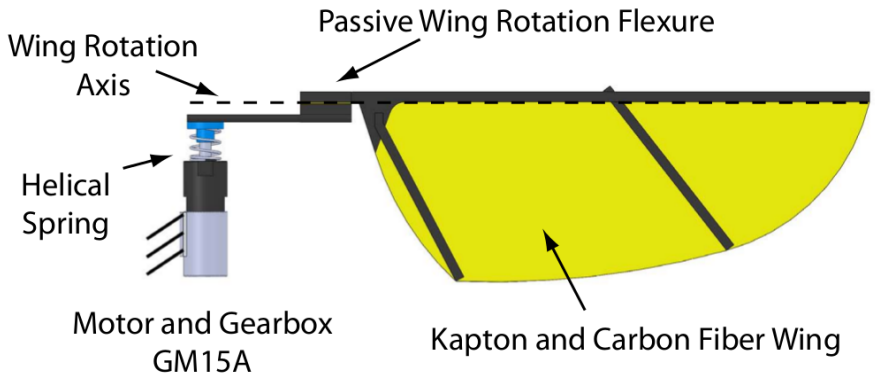

Figure 2.6: An example of a motor driven wing: The motor drives the wing flapping angle through an attached gearbox. The wing passively rotates about a polyimide film and carbon fiber flexure. A spring is attached in parallel at the gearbox and wing spar connection. [39]

2.1.2 DC Motors

Using DC motors (both brushed and brushless) for wing actuation posses fewer challenges (in comparsion with the oscillators), at the cost of larger size and weight. They are also cheaper and easier to obtain, so as a result multiple FWMAVs with DC motors are available. The motor is connected to the wing via a transmission (typically with planetary gears) with various transmission ratio (from 25:1 to 4:1 depending on the size and type of the vehicle), as shown in Figure 2.6. Robots from different research groups differ mostly in how the wing is attached to the transmission, and whether additional gearing is used.

Probably the simplest approach is having a direct sliding crank going from the transmission and attached to both wings via flexible polyimide joints. That way only one actuator is required, but the control authority is limited and additional actuators are required for steering. This solution is also similar to the *Dipteran* flight muscles of insects - an example is shown in Figures 2.7 and 2.8. Needles to say, the mechanism shown in Figure 2.8 doesn't produce enough lift for take-off and was for build for demonstration purposes only.

More advanced version of the slider/crank mechanism also uses flexible polyimide

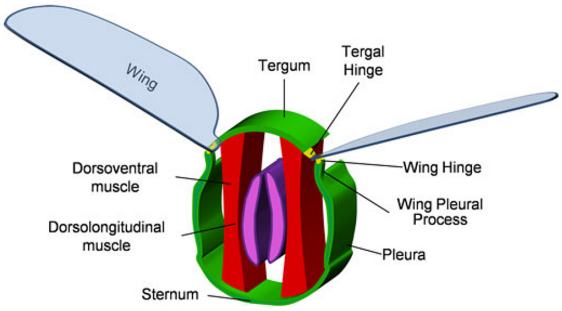

Figure 2.7: Dipteran insect's flight thorax [41]

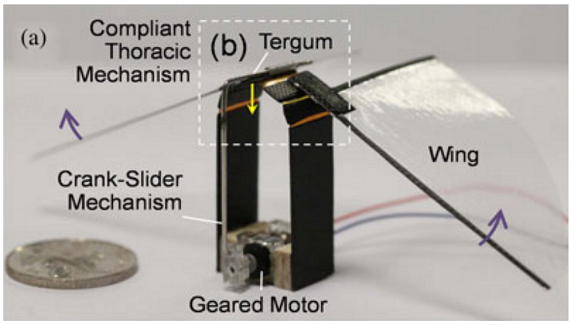

Figure 2.8: Compliant thoracic mechanism with integrated polyimide film hinges for elastic energy storage. As the tergum of the thorax is depressed, its wings beat upwards [41]

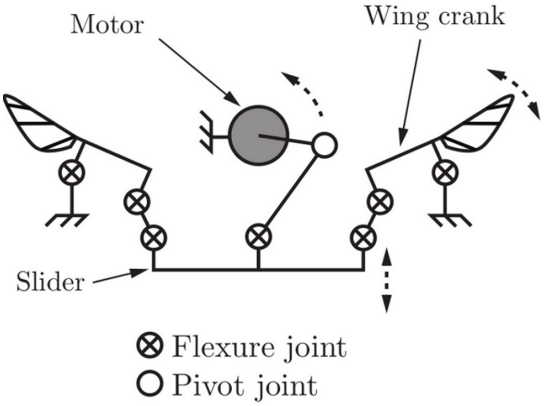

Figure 2.9: Crank-slider/slider-crank transmission [35]

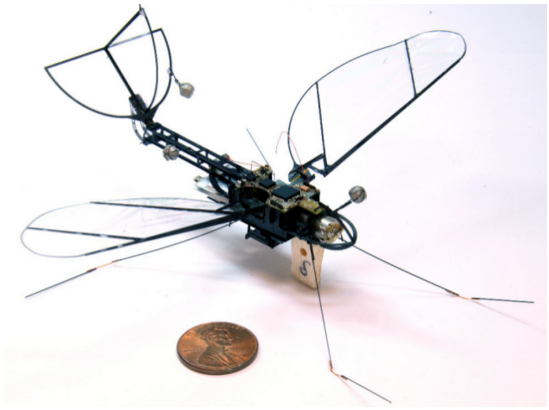

Figure 2.10: 3.2 g untethered flapping-wing micro-air vehicle for flight energetics and control experiments [35]

joints, but is more compact as shown in Figure 2.9. This mechanism was developed for the previously mentioned 3.2 gram locust-size FWMAV (see Figure 2.10) that is capable of free flights.

Although planetary gears are prevalent for adjusting the motor output torque, some researchers employed multi stage gear reducer consisting of spur gears as the transmission [42]. The advantage of this solution is a relative low-cost, since the gears can be 3D printed (unlike high-precision planetary gears), but the system is more susceptible to wear (since the gears are made form plastic) and is also more exposed to the environment. Figure 2.11 shows a detail of the gear reducer, and Figure 2.12

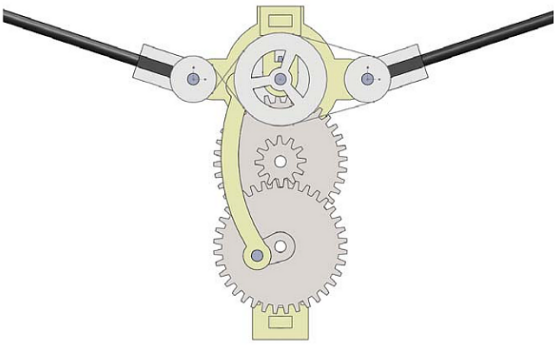

Figure 2.11: CAD model of the flapping-wing mechanism. [42] Note the gear reducer, as well as additional linkages and levers.

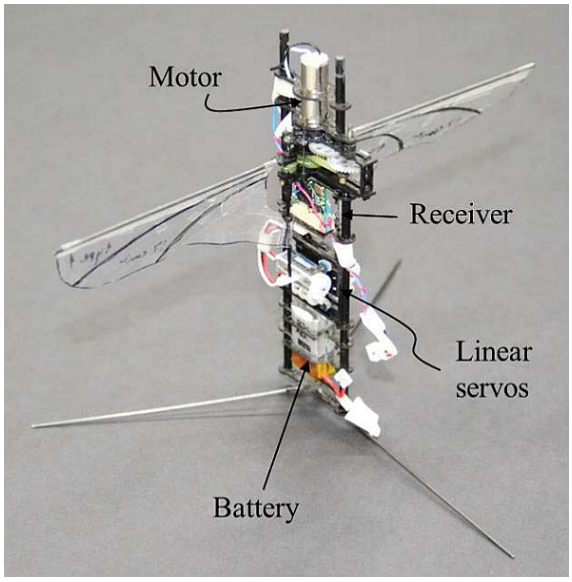

Figure 2.12: Flapping-wing mechanism integrated control system and battery. [42] This robot produces enough lift to carry its own weight.

shows the assembled FWMAV.

A system proposed by [18] utilizes a planetary gear transmission to match output torque of the DC motor (in this case a brushless), and rigid linkages to transfer rotary motion of the motor to wings. The advantage is that the system is more resilient than the one shown in Figure 2.11 (no need for small plastic gears), while still being relatively low-cost since all parts except the motor and the transmission are 3D printed. The CAD model of the assembly is shown in Figure 2.13 while the whole assembly is shown in Figure 2.14.

As can be seen, a variety of FWMAVs exist, differing in the type of actuators (piezoelectric oscillators or DC motors) with different mechanical configuration. The multitude of designs suggests that different applications prefer unique configuration of FWMAV - depending on size, weight, required lift, price, manufacturability and maintainability, and many other factors. FWMAVs capable of free flight already exist, which is important because the results of our research can be used by other teams

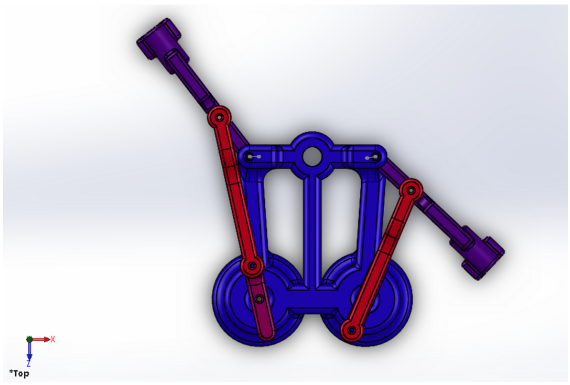

Figure 2.13: CAD model of the flapping wing mechanism [43]. Motors are connected to a planetary gear transmission (4:1 ratio), the output shaft of the transmission is then connected to rigid linkages that move the wings.

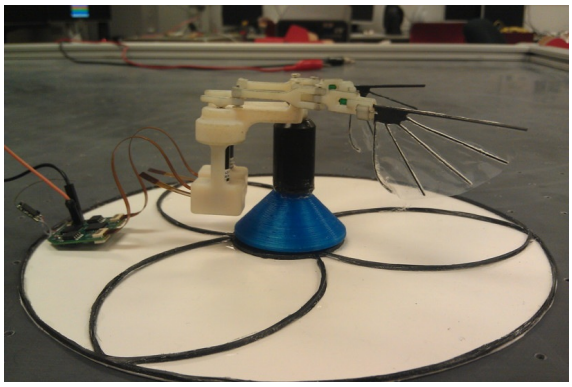

Figure 2.14: Assembled flapping wing mechanism during tests. [43]

on their FWMAVs – to add autonomy and fault recovery functionalities. In the next section we will review on-board sensors available for FWMAVs.

2.1.3 Sensors

The vast majority of experiments with FWMAVs is conducted indoors with the help of external vision system – popular is for example a VICON vision system [11] [35] [39]. External vision sensors are great help during development and testing, but to make FWMAV truly autonomous onboard sensors are necessary, so the robots can explore and navigate in unknown environment. Since the application area expects operations in GPS-denied environments (such as inside buildings), GPS receivers cannot be used to determine position. GPS has also limited resolution (1-10 meters), which is insufficient However, sufficiently small and light GPS modules are already available, for example Telit SE880 [44] weight only 80 mg (slightly more than the Hardvard Robobee) and is 4.7 x 4.7 x 1.4 mm (shown in Figure 2.15).

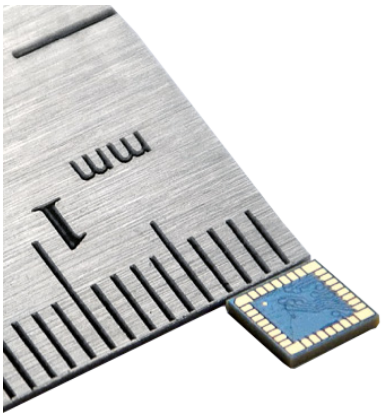

Figure 2.15: Telit Jupiter SE880 GPS Module [44]

Figure 2.16: InvenSense MPU-9250 on a quarter dollar coin [45]

2.1.4 Inertional Measurement Unit

Micro-Electro-Mechanical Systems (MEMS) gyroscopes and accelerometers are a good choice for FWMAVs. MEMS technology allows the sensors to be sufficiently small [35] - a good example is InvenSense MPU-9250 [45] which combines 3-axis gyroscope, 3-axis accelerometer and 3-axis magnetometer in 3 x 3 x 1 mm package (see Figure 2.16). Together these sensors form an Inertial Measurement Unit (IMU) anbd their readings can be combined together to obtain attitude (and in some cases position) information. As the FWMAVs are improving their flight capability, the flight area they can cover is getting larger, which cases problem for the external camera systems that are dependent on conveniently placed markers - because the tracked robots are so small (compared to the area they flight in), the markers are also very small and are hard to distinguish in the camera (because of the finite camera resolution). Even in a very simple example of straight flight, raw gyroscope readings provided more insight into the FWMAV flight dynamics than the VICON system [35].

Gyroscopes measure rotational rates, and are reliable at higher frequencies, but

suffer from a drift (a random walk caused by sensor noise). Accelerometers measure inertial acceleration, and work well at lower frequencies, but have bias that causes errors at higher frequencies [46]. Magnetometers measure the strength and direction of the local magnetic field. The magnetic field measured will be a combination of both the earth's magnetic field and any magnetic field created by nearby objects [47]. Magnetomers are used to improve heading (yaw) estimate.

Several sensor fusion algorithms are available. A simple, yet efficient, is a complementary filter - because gyroscopes and accelerometers are reliable at different frequencies, the complementary filter eliminates the unreliable frequencies for each sensor and then combines their output [46]. A Kalman filer (which is in fact an optimal linear estimator), an extended Kalman filter (which captures some nonlinear dynamics) and a particle filter, are popular algorithms of choice for sensor fusion [48]. A good overview of available sensor fusion techniques is provided in [49].

2.1.5 Vision Based Sensors

Although gyroscopes, accelerometers and even magnetometers have their equivalents in natural world, vision based navigation is perhaps the most familiar to us, because vision is our primary sense (as is for many insects). The simplest approach is using optical flow for autonomous navigation, as was demonstrated on a 20-gram FWMAV with its own stereo vision system [50]. Monocular optical flow was used for autonomous obstacle avoidance at high-velocities [51]. Camera inputs can be used to not only navigate the FWMAV in an unknown environment, it can be also used to reconstruct a map of the environment in real time to create a model of the world around the FWMAV (in a process called Visual Simultaneous Localization And Mapping, or V-SLAM) [52], and to display it to the operators so they have a better understanding of the explored area [53]. It was also demonstrated that a miniature optical sensor mimicking function of ocelli (a type of a simple eye common to insects) can be carried

onboard an FWMAV and used for stabilization and control [10].

2.2 Cyber Physical Systems

The concepts presented here appeared in [54]. Interested readers should consult that paper for more detailed information. We begin with the definition of an embedded system. Definitions vary, but essentially it is an information processing system where the end user is not aware a computer is present. Examples include photocopiers, microwave ovens, engine control in automobiles and price scanners in markets and department stores. More formally, an embedded system is *an information processing system embedded into an enclosing product.*

In general purpose computing performance, such as speed or virtually unlimited memory are major selling factors. Conversely, in embedded systems correctness is most important. Embedded systems often perform safety-critical operations where incorrect behavior can have dire consequences.

Embedded systems are ubiquitous. Applications include automobiles, commercial and military aircraft, weapon systems, medical equipment, smart power grids and transportation systems. They are becoming increasingly complex often including multiple processors, sophisticated communication networks and elaborate sensor and actuator systems. Sales of low-end microcontrollers suited for embedded applications exceed that of PC microprocessor sales and have done so for nearly 15 years.

So what exactly is a "cyber-physical system"? Is it just another term for an embedded system? The short answer is no. The term CPS came into popular use as early as 2006 in large part via the efforts of Helen Gill at the U.S. National Science Foundation. A CPS is not a traditional embedded system or sensor net. The term CPS emphasizes the fact that computer resources (the cyber portion) are tightly integrated with a physical system (the physical portion). Cyber capabilities could be incorporated

into every physical component. A CPS could have elaborate networks and may be reconfigurable. Control loops can be continuous or discrete. Cyber-physical systems exist at all scales from hand-held devices to power grids spanning large geographical areas. The commonly accepted definition of a CPS is as follows: *A cyber-physical system is the integration of computation and physical processes.*

Figure 2.17 shows the abstract architecture of a CPS. Using the term "cyber-physical" emphasizes the strong link between the cyber and the physical worlds. In a CPS the cyber portion affects the physical system and the physical system affects the cyber portion. The integration of the cyber with the physical is extremely tight. In fact, this integration is so tight it may be impossible to identify whether the system behavior is due to computing or physical laws! For example, it may not be possible to tell if an unmanned aerial vehicle maneuver was caused by computer commands or resulted from the natural governing dynamics of the vehicle's airframe. A CPS is not the union of the cyber with the physical but rather the intersection of the two.

FWMAV is a CPS because it contains both the *cyber* portion - control loops, communication interface, path planning algorithms etc. as well as the *physical* portion - motors to be rotated at a precise speed, wings to be controlled and flapped to produce light, and environment constraining the movement of the robot.

2.3 Subsumption Architecture

Readers familiar with the subsumption architecture may skip this section. A subsumption architecture [55] is a very useful concept when we are dealing with:

- multiple goals
- multiple sensor inputs
- multiple actuators

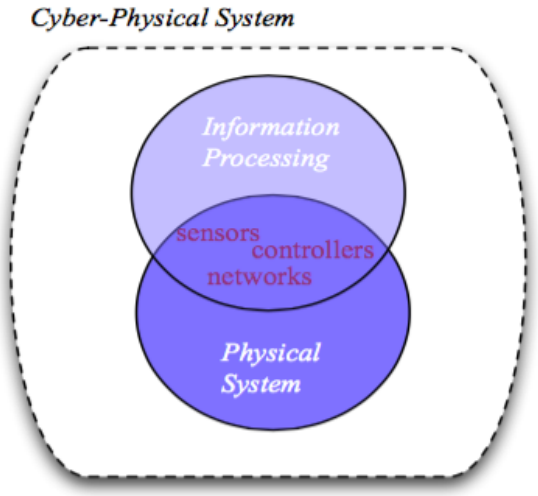

Figure 2.17: An abstract view of the CPS architecture. The information processing system typically consists of one or more low-end microcontrollers. Sensors observe the physical system state while controllers provide inputs that alter the physical system state. Networks interconnect the physical and the cyber portions. The physical system can be electronic, mechanical or electromechanical.

- requirements for robust and easily extensible solution

The subsumption architecture was first used to control an autonomous ground robot, capable of independent exploration and navigation in presence of obstacles in office space. The main idea behind the subsumption architecture is to decompose the problem (i.e. navigation in presence of obstacles) into independent layers, and then assign priority to each of these layers. The layers are shown in Figure 2.18, note that the lower number, the higher priority of the layer. The benefit of the subsumption architecture is that it can model a complex behavior (such as trajectory following or navigation) by using simple rules that are easy to develop, modify and extend.

A *dynamic subsumption architecture* is an extension of the concept, which allows the layers to be dynamically changed, more specifically: a change in the priority of the layers; add/remove layers; modify the consequents of the layers.

Such change can be event triggered (e.g. a sudden change in the environment), or action triggered (i.e. the robot decides to change its behavior based on some internal state). Dynamic subsumption architecture allows the robot to react on changes in the

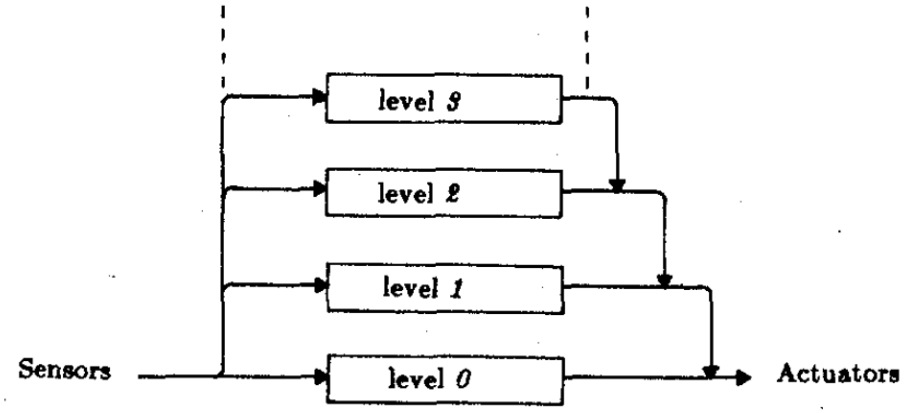

Figure 2.18: Control is layered with higher level layers subsuming the roles of lower level layers when they wish to take control. [55]

environment (e.g. strong wind), in the robot itself (e.g. a damage to actuators, or low fuel), or in the mission (e.g. the mission was terminated by an operator); and its use for autonomous robots was already proven.

Although the subsumption model was used for navigation and control of ground robots [56, 57], it hasn't been used for flapping wing vehicles. Utilizing this powerful concept, we can easily develop and adapt the desired robot behavior. Note that the subsumption architecture was successfully applied in commercial products such as Roomba robot, scientific devices such as Sojuner Mars Rover and in military bomb disposal robots.

2.4 Evolutionary Algorithms

In *Evolutionary Algorithms* (EA) the new solutions to a problem are evolved from existing solutions by emulating Neo-Darwinistic evolution found in nature. Highly fit solutions – i.e. those providing the best solution for the problem – are preserved and further evolved, while solutions with low fitness are removed from the population. There are two types of evolution relevant for CPS – *intrinsic* and *extrinsic* [54]. The difference between *intrinsic* and *extrinsic* evolution is in how the fitness is determined.

In *extrinsic* evolution a computer model of the CPS is used, while in case of *intrinsic* evolution the solution is downloaded to the CPS and physical tests are conducted. As a result, *extrinsic* evolution is suitable for rapid evolution (because it can be run faster than real-time), but its accuracy depends on the model. *Intrinsic* evolution on the other hand is more precise, but slower, because it has to be conducted on a physical device.

An EA consists of a population of individuals, where each individual represents a particular solution to a given problem. New individuals are created from existing individuals via random mutation and recombination, but only the highly fit ones will survive. The fitness formula is dependent on the problem being solved. An EA runs for a fixed number of generations, at the end of the run the fittest individual is the final solution. The EA steps are shown in Algorithm 1.

Algorithm 1: A basic evolutionary algorithm

1. Randomly generate the initial population;
2. Evaluate the fitness of the initial population;
while *max number of generation not reached:* **do**
   i. Select the best individuals for reproduction;
   ii. Generate new individuals via random mutation and recombination;
   iii. Evaluate the fitness of new individuals;
   iv. Discard the least fit individuals;
end

To better show the idea, we present an example of a CPS with an EA: an evolvable hardware – a circuit that can be reconfigured to perform certain tasks, for example band pass filtering. In this case the fitness is determined by the filter performance (i.e. how precise is the band pass). The solution is encoded as a bitstring - representing configuration of the circuit. In case of *extrinsic* evolution, a simulation is used to evaluate the fitness, while in case of *intrinsic* evolution the response of the physical circuit is measured. Figure 2.19 shows the concept in detail.

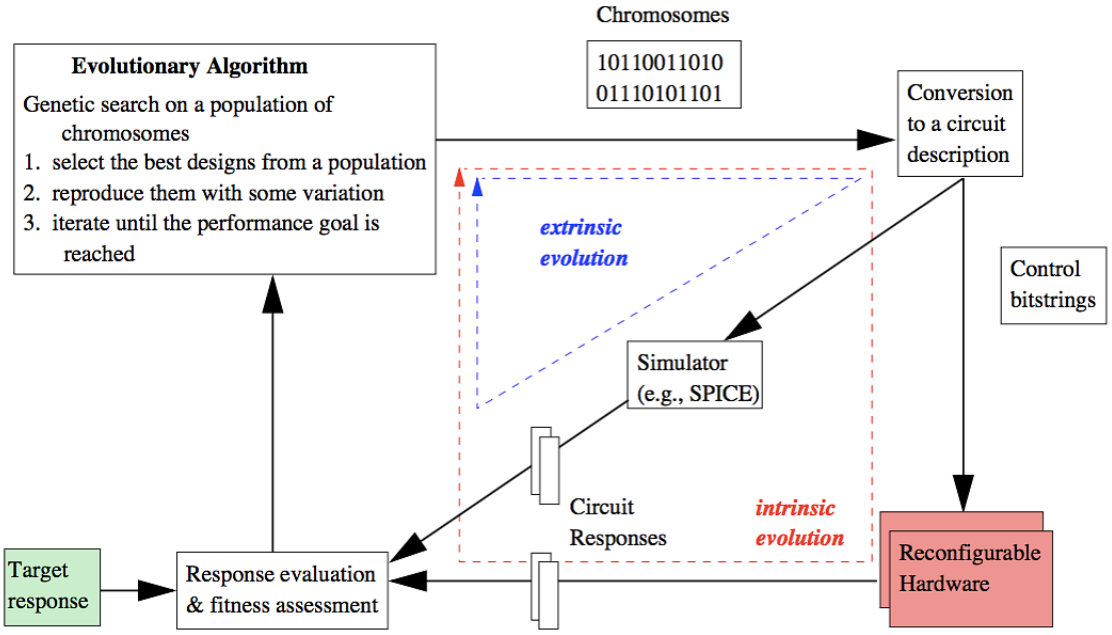

Figure 2.19: Evolutionary Algorithm used in a CPS: evolving a bandpass filter on a reconfigurable hardware [54].

2.5 Multi Agent Systems

To give the reader a better understanding of MAS, we define an agent as follows [58]: *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.* An agent receives *percepts* from the environment, and generates *actions* that might or might not depend on the *percepts*. Figure 2.20 illustrates this idea. The agents can have different types, but for the purpose of our research we considered only *Reactive agents*. These agents react to changes in the environment in a stimulus-response fashion by executing the simple routines corresponding to a specific sensor stimulation. A famous reactive agent architecture is the previously mentioned subsumption architecture [58].

Since multiple agents are used, they interact in some way. For our purposes we use *cooperative* agents – i.e. agents that work together to achieve some common goal (such as moving a robot to a certain place). Other interactions are also possible (some

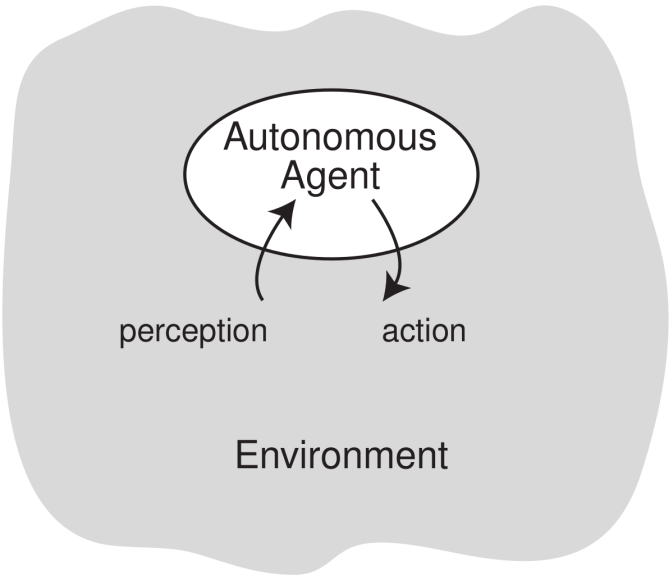

Figure 2.20: An autonomous agent and its environment [58].

agents can be *competitive* against each other), but not suitable for our research. The agents can communicate in multiple ways. The basic communication paradigms are these four [58] (Figure 2.21 illustrates the idea):

- *Peer-to-peer* communication: messages are sent directly to specific agents. This is usually done by identifying the partners, for instance with an email-like address (message-passing-like communication). It is also possible that an intermediate channel takes in charge the transmission of the data, and that the partners of communication do not know from each other.

- *Broadcast* communication: a message is sent to everybody in the MAS. Interested agents can evaluate the received data or ignore it.

- *Multicast* communication: A message is sent to a specific group of agents.

- *Generative* communication: communication is realized through a black-board: agents generate persistent objects (messages) on the black-board, which are read by other agents. The reading can be done independently of the time of the message generation; thus the communication is fully uncoupled.

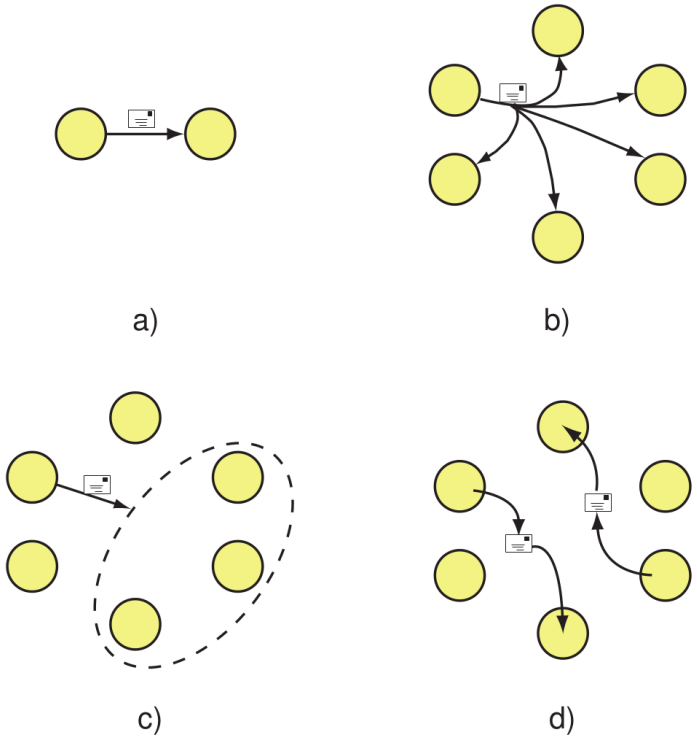

Figure 2.21: Basic communication paradigms: a) peer-to-peer, b) broadcast, c) multicast and d) generative communication. [58].

In summary, a MAS relies upon a number of agents (actors) that communicate with each other and typically cooperate in order to complete a certain task, such as target tracking [59] or mapping [60]. Although a MAS were successfully used for path planning of UAVs [61] [62], convoy protection [63] or flood monitoring [64], it was never used for a control of a FWMAV.

This section showed the reader the bigger picture and research related to our work. In short, FWMAVs are more and more popular and there are many designs available, all utilizing two types of actuators - either a piezoelectric oscillator or a DC motor. Various modelling techniques exist, so it is possible to simulate the performance of a FWMAV before it is built, assuming we have enough information about individual subsystems. A multitude of sensors exist, as well as a wide range of sensor fusion algorithms for position and attitude estimation – which can be applied on FWMAVs in a similar way as on larger UAVs (once sensors with sufficiently small footprints become

available). A subsumption architecture has been used in robots before, achieving relative sophisticated intelligence with a simple set of rules. A MAS has never been used for a control of a FWMAV, although a MAS were also used in robotic applications. The next chapter formally defines the problem our research is solving, and provides details about the developed FWMAV and our experimental setup.

Chapter 3

Problem Definition

This chapter formulates more precisely the problem we are solving, and provides information about the design of our FWMAV, as well as the experimental setup. The intended mission of the vehicle is to continuously follow an arbitrary trajectory. The trajectory is represented as a set of waypoints connected in straight lines. The trajectory is pre-defined and static during the mission. The vehicle will determine what is the best way towards the next waypoint and what control action needs to be taken to get there. The mission ends after reaching the last waypoint. In case obstacles are present, the vehicle is required to avoid them autonomously and resume the desired trajectory. The biggest challenge is the allocation of the control inputs, i.e. finding out what control action is needed. The vehicle is also required to automatically recover from faults that could occur during the mission, such as wing damage and control system malfunction.

3.1 Vehicle Configuration

A conceptual vehicle closely related to those described by Wood [11], [12] and Doman [13], [14], is presented in [18]. Our vehicle is an upgraded version of a FWMAV described in [15] [16] [17] and has two wings providing all propulsion and control forces.

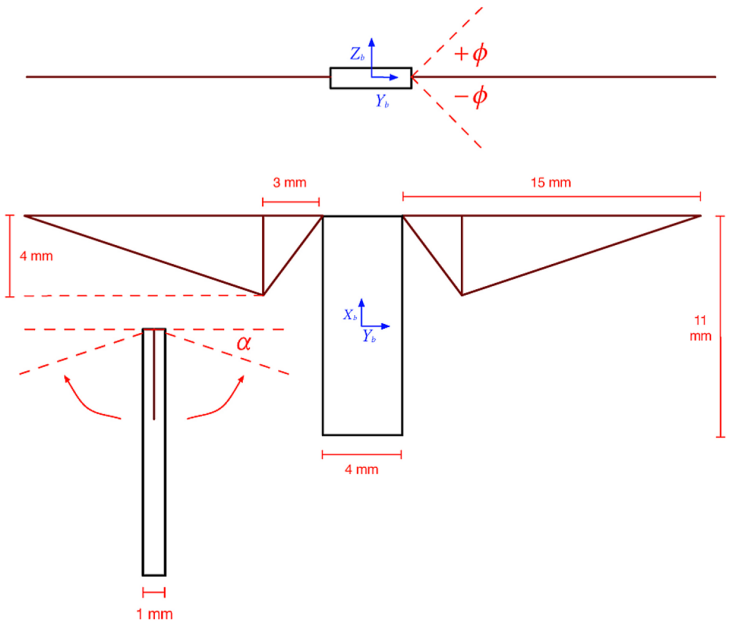

Figure 3.1: Orthographic view of a FWMAV [18]. Both wing spars are restricted to rotational motion about their joints with the body and in the Y_b–Z_b plane. The range of those rotations is $[-1 \ldots 1]$ radians, α is between $\pi/6$ and $\pi/2$ radians. Note that the dimensions are for orientation purposes only, and differ on the actual vehicle which is larger.

It approximates a passively upright-stable version of the vehicle from [18] operating near its hover wing flapping frequency. The wings are mounted in the Y_b–Z_b body plane (see Figure 3.1). These wings are actively actuated within the range of $\pm\phi$ radians. As the spars rotate, dynamic air pressure lifts the triangular wing platforms (membranes) up to an angle of α radians under a base vector embedded in the Y_b–Z_b plane. Individual wing flaps produce independent lift and drag forces at each of the two wing roots (points of attachment of the wings to the body). These can be resolved into body frame forces and torques and cause changes in the whole vehicle's position and pose in three-dimensional space.

3.2 Cycle Averaged / Split Cycle Control

The cycle averaged control is based on estimates of what forces a wing would produce, on average, over a single wing beat. For example, a cycle-averaged altitude controller might compute the error between current and desired altitude, use an error feedback control law to compute the desired force to apply to the body, and finally use a model of the vehicle's wings to compute the parameters of a single wing beat that, when adopted by both wings, produces the desired force (on average) during the whole wing beat. Cycle-averaged control wraps a feedback control law around the whole wing beat as an atomic constructs rather than around finer-scaled micro-motions of wings. The desired wing motions are "communicated" to the wings once per wing beat as a small number of shape parameters that define how the wing will move during that wing beat.

Split-cycle control is a special case of cycle-averaged control in which wing beats are composed of two half-cosine waves, one each to govern the wing's upstroke (front to back) and downstroke (back to front). The shape parameters communicated to each wing are a flapping frequency (ω [rad/s]) and an upstroke/downstroke transition parameter (δ [rad/s]). Advancing the upstroke (and consequently impeding the downstroke) produces a forward force while keeping the wing beat frequency constant. Formally, $\phi_U = \cos((\omega - \delta)t)$ and $\phi_D = \cos((\omega + \sigma)t)$ where σ is dependent on δ. From [14] we know that $\delta \in [-\infty \ldots \omega/2]$ although certain value ranges are particularly important. If $\delta = 0$ the upstroke is symmetrical to the downstroke and a regular wingbeat occurs. However, if $\delta > 0$ the upstroke is impeded and the downstroke is advanced, as shown in Figure 3.2. As a result, a force is generated in the direction of the downstroke. Conversely, if $\delta < 0$ then the downstroke is impeded and the upstroke is advanced, as shown in Figure 3.3, resulting in a force in the direction of the upstroke. These lateral forces act on the vehicle's body via a moment arm producing an angular momentum. Put simply, by applying the split cycle the vehicle can turn. See [14] for

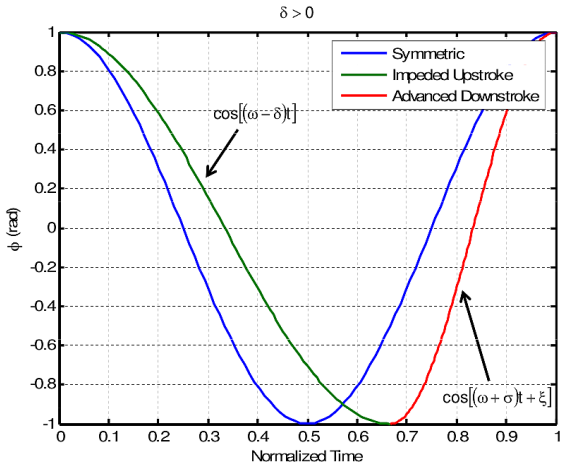

Figure 3.2: Split-cycle results for $\delta > 0$

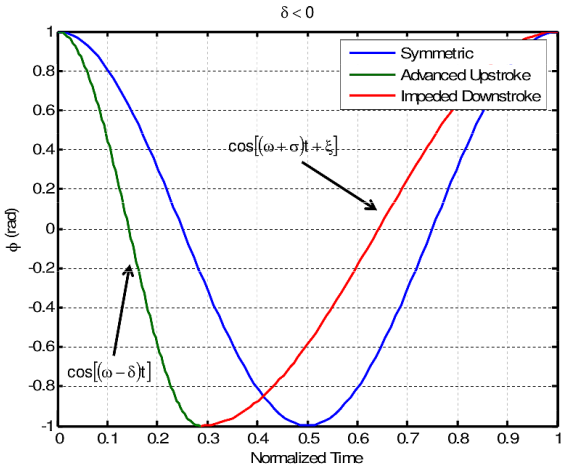

Figure 3.3: Split-cycle results for $\delta < 0$

a derivation and a proof of split-cycle operation.

A conventional application of a split-cycle control to our vehicle might entail an "outer loop" of multiple body axis controllers (e.g., one that computes altitude error and determines a flapping frequency for the wings, one that computes a roll axis angle error and computes antagonistic shifts to produce a roll moment, etc.), and an allocator that harmonizes all of the flapping frequency and shift commands made by the various axis controllers for presentation to an "inner loop" controller that would ensure the wings follow the correct cycle averaged trajectories. Control would then consist of an outer loop that, based on vehicle state, provides ωs and δs that should produce forces required to effectively correct position and pose errors and an inner loop that, receiving those δs and ωs, would ensure the wings moved as required.

3.3 Experimental Setup and Environment

All experiments are to be conducted in a large ($5' \times 5'$) water tank. The vehicle is restricted move on a two-dimensional plane and rotate around its X_b axis, which simulates operation around hover. The vehicle consists of a pair of wings (shown in Figure 3.4 and is equipped with a pair of Lithium Polymer batteries, a power

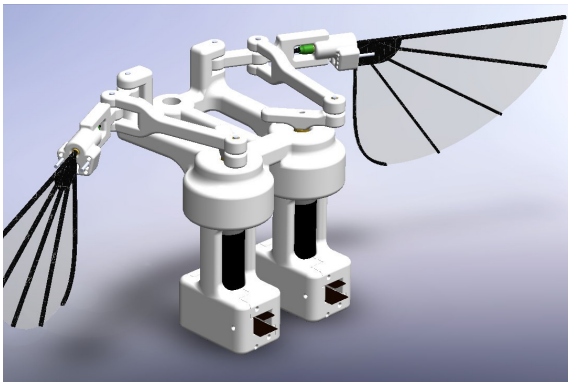

Figure 3.4: 3D model of wings with linkages and motors. Video showing moving wings is available from [43]

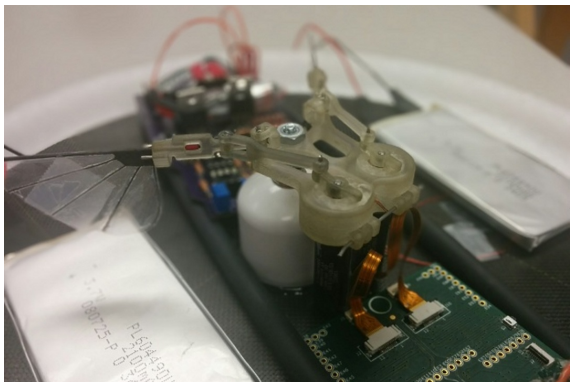

Figure 3.5: Assembled vehicle—note wings in the middle, LiPo batteries on sides, the power distribution board in the back and the control board in front.

distribution board and a main computer, as shown in Figure 3.5. All hardware is mounted on a carbon-fiber platform, attached to a floating Styrofoam puck, which keeps the robot on the surface. The water surface acts as a mechanical low-pass filter, slowing down the vehicle movement and dampening disturbances, which allows us to test our control system without the need for expensive high-speed cameras. A camera is placed above the water tank, so its field-of-view encompasses the entire water tank. The camera locates and records the vehicle position. A sample capture view from the camera is shown Figure 3.6.

During the development of the image processing pipeline that reliably tracks the vehicle in the whole area of the water tank we examined a number of different approaches:

1. **Color Markers** - using color markers of specific color and dimensions is a very common method, but suffers from changing light conditions. Typically the image is converted into HUV (Hue-Saturation-Value) color space because it is more robust than RGB color space, and then filtered so only the markers are left in the image. The position and orientation is then calculated using a simple geometry. Figure 3.7 shows the robot with color markers.

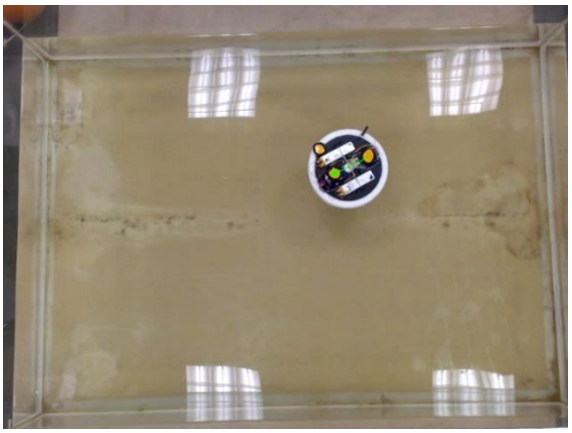

Figure 3.6: Vehicle during an experiment in the water tank (top-view). Note the color markers used for machine vision pose estimation.

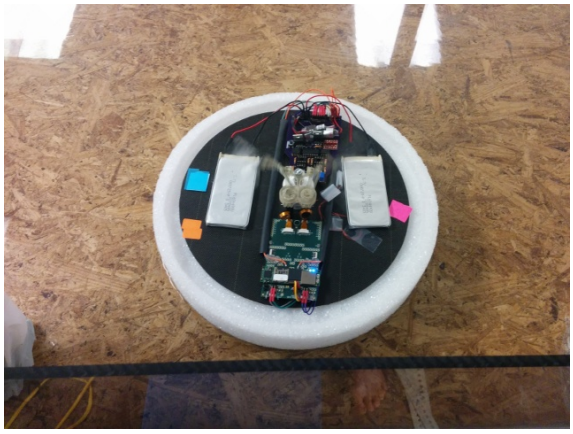

Figure 3.7: Vehicle during an experiment in the water tank (close-up view). Note the color markers used for machine vision pose estimation.

2. **Lukas Kanade Tracker (LKT)** [65] – LKT uses optical flow (mentioned in Section 2.1.5) to track an object within the image. LKT needs a template of the object to start with (for example an image of the robot), but can track changing object (i.e. when the lighting changes), which would be suitable for our application. Unfortunately LKT by itself cannot determine the orientation of the object, which means we would know only position of the robot. To obtain orientation, additional methods need to be used.

3. **SURF feature matching** – SURF detector [66] finds suitable features in the image, that can subsequently be matched by FLANN based matcher [67]. Once the match between the original object and the template is established, homography can be calculated using RANSAC algorithm [68], which is then used to calculate the pose of the robot - an example of this process is shown in Figure 3.8. The drawback is higher computational complexity because of the matrix transformations involved.

We combined the LKT with SURF detector and subsequent feature matching to provide a tracker capable of following the robot across the whole water tank (even when

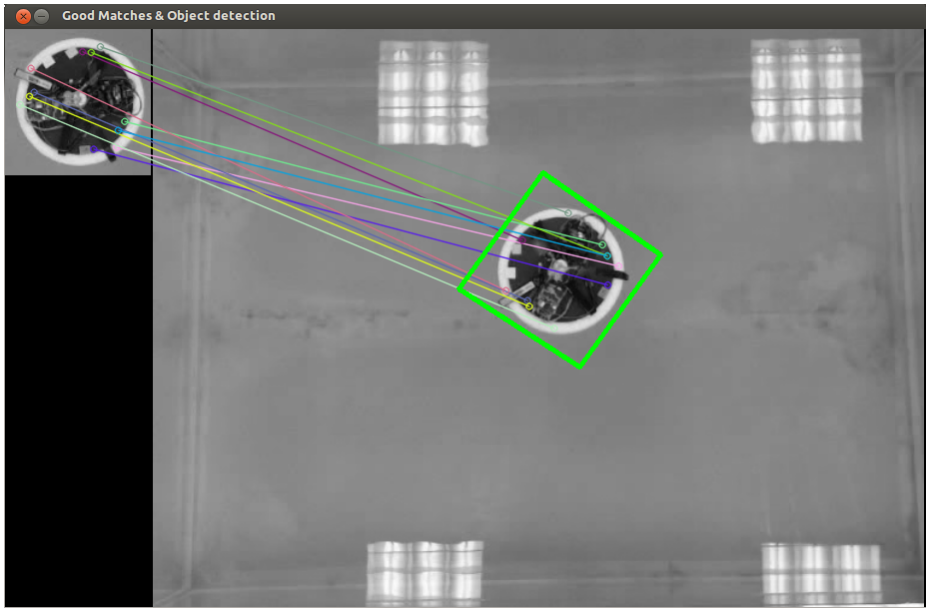

Figure 3.8: An example of feature matching between a template picture (top left) and recorder camera image. The features found in both images are circled in color and connected by colored lines. The green square represents the calculated orientation and position of the robot (in this case roughly 60 deg counter clockwise)

the vehicle was partially occluded), but because the pose was estimated incrementally (i.e. we were looking at the pose difference from the last estimated pose) the algorithm suffered from integration errors and as a result drifted in orientation estimate. Running algorithm is shown in Figures 3.9 and 3.10. These issues could have been indeed resolved, but since the computational complexity of the algorithm was already fairly high (around 100 ms to process one camera frame on a regular laptop), it wouldn't be viable for real-time tracking at 30 Hz and we opted out for a simpler approach using color markers.

After a careful calibration the markers were recognizable in most of the water tank, but in some places the light reflection leads to intermittent loss of tracking. Nonetheless the color markers worked reliable despite for our experiments despite this disadvantage. The processed video is recorded for reference, and the estimated pose is sent to the onboard MAS via a WiFi link.

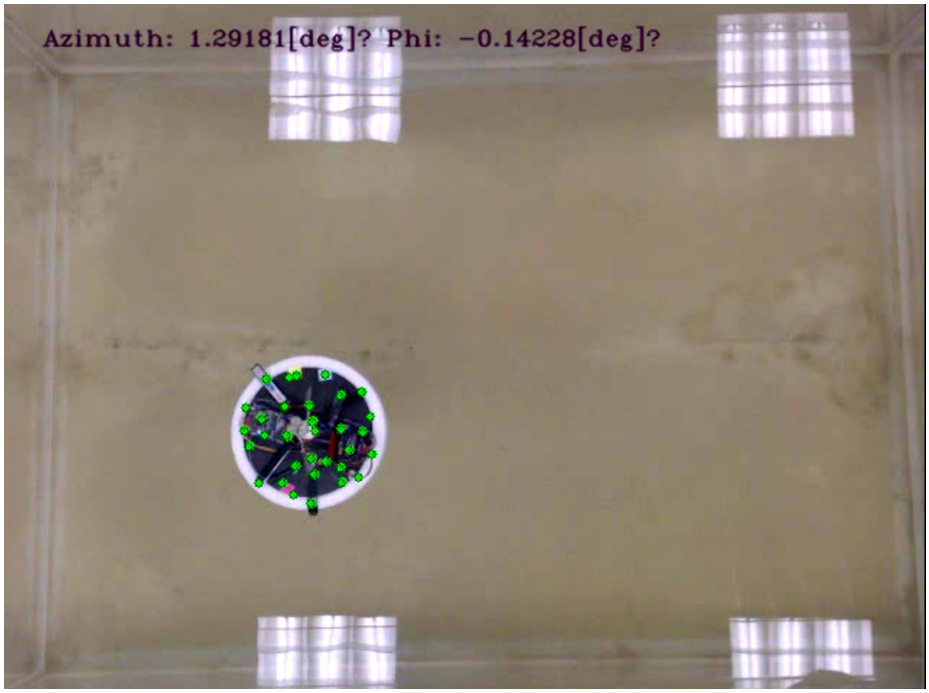

Figure 3.9: An example run of the tracking algorithm based on LKT and SURF: Shortly after the initialization the azimuth estimate is precise (0 deg means the robot is facing directly towards the right)

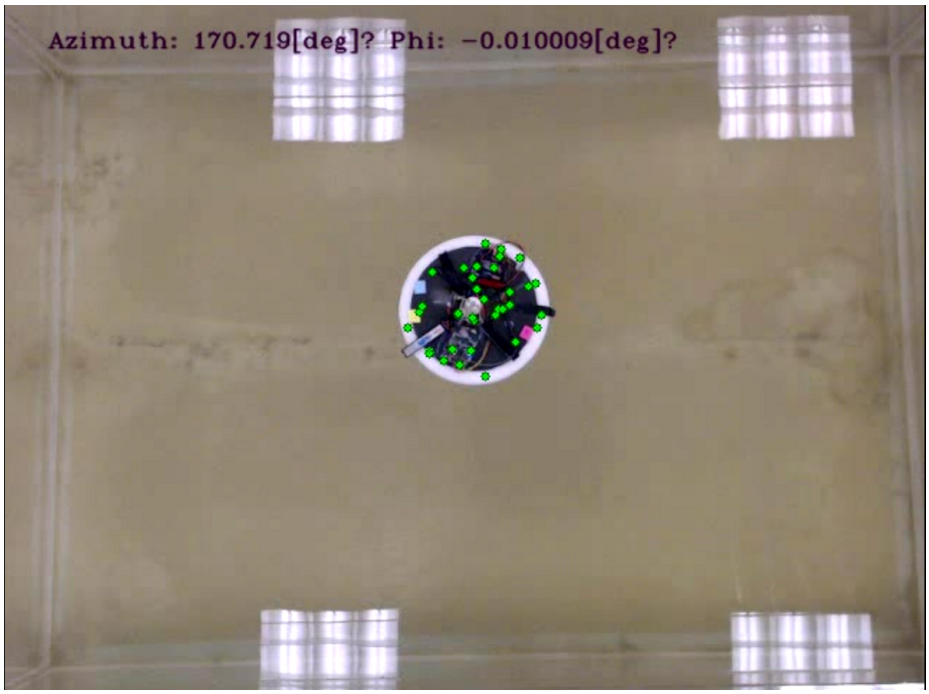

Figure 3.10: An example run of the tracking algorithm based on LKT and SURF: As the integration error accumulates, the estimated azimuth quickly drifts. In this case the estimate is around 170 deg, while the true orientation is around 80 deg)

Figure 3.11: 3D printed parts. Notice the couplers on the left, and the left and right rockers on the right. Left and right rockers are identical.

Figure 3.12: 3D printed parts after cleaning.

3.4 3D Printing & Vehicle Assembly

The original parts for the vehicle were printed using a high-end industrial 3D printer. During the assembly and necessary repairs of the vehicle, it was needed to print additional spare parts. The 3D printing technology advanced very rapidly since the first prototype of the vehicle was built, and we were able to use successfully a consumer grade 3D printer (*Mojo 3D*) to print all necessary linkages. Detailed 3D CAD models of the whole assembly are available, so the whole vehicle can be easily reproduced.

In batch we can print enough parts for three pairs of wings. The price for one batch is around $6. Figure 3.11 shows the printed parts right after they were printed. The printing took around two hours. The printed parts have to be cleaned in a solution (to remove the support material), and in hand (to remove print imperfections). The cleaned parts are shown in Figure 3.12.

After the parts are cleaned, the wing linkage can be carefully assembled. It is a very tedious process, taking up a few hours. The individual parts have to be sanded off for a precise fit; sometimes the holes have to be cleaned with a hand drill and brass bushings have to be inserted to reduce friction between parts. Fortunately, the assembly and repair process is well documented, including a video with a detailed description of the process.

We have shown that new parts can be printed very easily, and a multitude of wing assemblies can be put together relatively quickly, which is important because the ABS plastic isn't very rigid, quickly wears off and bends under stress, hence repair-ability is a crucial aspect of the design. The complete vehicle assembly, however, requires special motors, encoders and custom printed circuit boards and neither of those can be obtained quickly or cheap. More details about the vehicle, including part numbers, mechanical drawings and software architecture can be found in Appendix A.

3.5 Simulation

A simulator of the vehicle is used for extrinsic evolution during initial learning (see Section 4.3.1 for details). The simulator contains a simplified model of the vehicle with 3DOF – a two-dimensional movement on the water surface plus a rotation around vehicle's z-axis. The simplified model assumes no external disturbances, such as wind. It assumes a perfect control over wing position (i.e. no slip in the linkages). It assumes a perfectly balanced vehicle, moving on a frictionless plane.

The simulator calculates cycle averaged lift and drag forces, meaning the produced lift and drag forces are averaged over one full wingbeat. The produced forces are then propagated to the body model, creating moments and change in orientation and position of the vehicle. Cycle averaged forces are a good approximation, because the low level controller cannot change the δ and ω parameters more often than at the

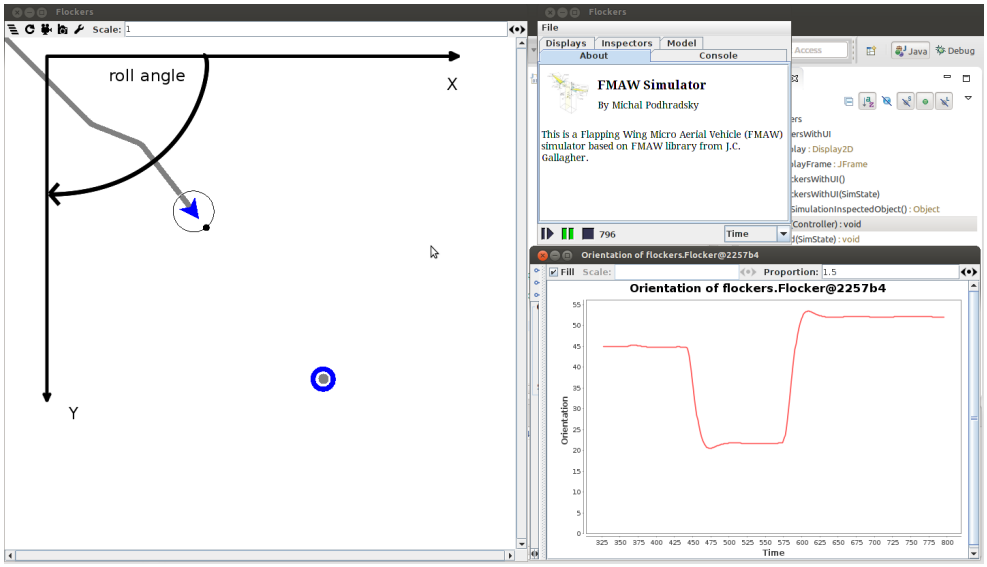

Figure 3.13: FWMAV Simulator. *Left:* main window representing the flight area (in our case the water tank), the blue arrow is the FWMAV, the gray trace is the flight path, the blue circle is the desired waypoint (can be repositioned during the flight), the axis labels are only for reference. *Top right:* control window with settings and start/stop/pause button. *Bottom left:* Live plot of simulation data, in this case the orientation of the vehicle.

beginning of the wing beat. As mentioned previously, the simulator is not intended to perfectly model the vehicle, but to provide a reasonably accurate initial values for the learning algorithms and their verification.

The core of the simulator is a Java library containing vehicle dynamics model, and performing all necessary lift and drag calculations. The library was developed by the research group participating in this project. The agents are implemented using the MASON toolkit [69], which is also used as a visualization front end. The MASON toolkit is used to verify the MAS architecture—e.g. to make sure that the correct rules from the rulebase are firing in right order, and to check the correctness of the scheduler by moving the vehicle along predefined test trajectories. A screenshot of the running simulator is shown in Figure 3.13.

The next chapter describes how we approached the problem and the MAS we designed, together with the fault recovery mechanism.

Chapter 4

Approach

4.1 Multi-Agent Architecture

In this section we describe the multi agent control architecture and the various agents involved in the vehicle control. The vehicle is assumed to be moving in an enclosed, wind-free environment with no obstacles (other than the area boundaries). It is required to follow a set of predefined trajectories specified by a sequence of waypoints. This architecture was first proposed in [70].

4.1.1 Agent Description

The MAS consists of five agents. A *collection agent* receives pose information x, y, ψ from the camera (or simulator), and runs smoothing and averaging algorithms to compute the estimated pose $x', y,' \psi'$. A *monitor agent* observes vehicle behavior and requests vehicle diagnostics if the behavior has deteriorated too much. The *strategy agent* keeps a list of desired waypoints, and provides them upon request to a *controller agent*. The controller agent determines the split-cycle oscillator control inputs (a δ and ω for each wing) based on the vehicle pose. Finally, a *diagnostic agent* runs vehicle diagnostics and determines if a fault occurred and ultimately decides whether

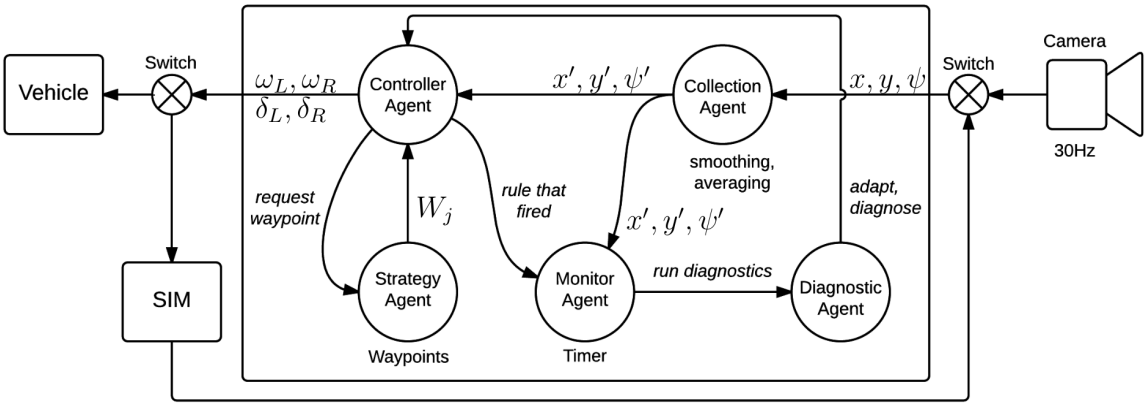

Figure 4.1: Diagram of Agent-based control architecture. Detailed information about each agent is provided in Section 4.1.1. More info about the simulator (SIM) can be found in Section 3.5. Note that the arrows between the agents do not reflect inter-agent messages—the communication is done by event processing, as described in Section 4.1.2

the controller agent's rule base has to be adapted. The MAS diagram is shown in Figure 4.1. Each agent is further described below.

4.1.1.1 Collection Agent

Precepts: x, y, ψ

Outputs: x', y', ψ'

Tasks: Collects high-speed data x, y, ψ from either simulator or the camera, and computes smoothing and averaging of the data. Outputs estimated pose data x', y', ψ' at a lower rate. There are several filtering options under consideration, e.g. an exponential moving average filter.

4.1.1.2 Monitor Agent

Precepts: x', y', ψ', active rule from rulebase

Outputs: issue diagnostics command

Tasks: Receives estimated position data from the collection agent, and movement being executed from the controller agent. Monitors performance of the vehicle, and in the case of unsatisfactory results, directs diagnostics agent to run diagnostics.

4.1.1.3 Strategy Agent

Precepts: none

Outputs: W_j

Tasks: Stores list of waypoints. Sends next waypoint W_j upon request.

4.1.1.4 Controller Agent

Precepts: x', y', ψ', W_j

Outputs: $\delta_L, \delta_R, \omega_L, \omega_R$

Functions: Consists of one or more agents. Responsible for determining control inputs $\delta_L, \delta_R, \omega_L, \omega_R$ based on the vehicle pose x', y', ψ' and the desired position W_j. When the vehicle reaches a waypoint, the agent requests new waypoint coordinates from the strategy agent. Runs maneuvers for diagnostics testing.

4.1.1.5 Diagnostic Agent

Precepts: x', y', ψ', W_j

Outputs: computes likelihood of fault

Functions: Initiates diagnostics testing. Computes likelihood of failure. Determines if behavior must be adapted to compensate for faults.

4.1.2 Agent Scheduling

The scheduler is responsible for specifying when individual agents execute their assigned tasks. In this application scheduling is event-oriented rather than time oriented. Our scheduler is loosely based on the scheduler used in the RePast agent-based toolkit [71].

The scheduling process can be abstractly thought of as a sequence of hooks on a wall (see Fig 4.2). An agent is "hung" on a hook if it is supposed to execute some task at some future time. However, time is relative, not absolute–i.e., hooks are not associated with some specific time nor does hook spacing reflect time intervals; hooks merely establish a partial ordering of agent tasks. For example, if agents x, y, and z are hung on hooks three, four and seven respectively, this simply says agent x performs some task before agent y which performs some task before agent z. Hooks therefore just order agent behaviors with respect to each other. Only one agent can be hung on a particular hook because agents do not execute tasks concurrently. We say an agent is *stepped* if it is directed to perform some action or task.

The entire scheduling process is event driven. A first-in-first-out (FIFO) buffer is used as an event queue. As agents perform assigned tasks—i.e., they are stepped by the scheduler—they may post events in the FIFO, which might cause other agents to be stepped at a later time. Events have a header and a body. The header tells which agent posted the event, the event type and possibly a timestamp. The body contains attributes unique to the event.

The scheduler unloads the buffer, analyzes the events, and processes events by stepping a specific agent. Events are pulled from the FIFO as quickly as possible, but stepping agent is deferred while the FIFO is not empty. All the scheduler does at this time is to decide upon which hook to hang the event. Some shuffling of agents already hung on hooks may occur depending on the priority of the event. When the FIFO is empty, the scheduler starts pulling agents off of hooks in a "first-hung-first-pulled" order and steps them. A stepped agent may be provided information from the body

Figure 4.2: An example showing agents hanging on hooks waiting to be stepped by the scheduler. The left-to-right order reflects the relative ordering of the agent stepping.

of the event that prompted it being stepped.

A simple example will help fix ideas. Suppose the controller agent has just output new δ and ω values. This agent would post a "new δ/ω output" event in the event queue. The body of this event would identify which rule in the rule base fired, so the commanded vehicle movement is recorded. When offloaded from the FIFO the scheduler would hang a monitor agent tag on a hook along with the rule ID extracted from the event body. Once the FIFO is empty, the scheduler pulls the monitor agent tag off of the hook and steps the monitor agent to commence observing the vehicle movement. The monitor agent would be informed at that time which rule fired.

4.2 Agent Implementation

This section describes each of the agents mentioned above in more depth, including the implementation details.

4.2.1 Controller Agent

This agent computes the δ and ω values needed to move the vehicle between waypoints. The vehicle will typically have to adjust its course as it moves along a trajectory.

However, that does not mean new δ values are needed at the beginning of each wing beat.

There are four control inputs to the vehicle available (specifically $\delta_L, \delta_R, \omega_L, \omega_R$), but only two actuators (the left and right wing) thus the control inputs are not independent. This situation can be described as *under-actuated* vehicle, which means we cannot control the position and course independently, and that poses a greater challenge for the controller agent. For example, if the vehicle is moving forward and is heading slightly left off the desired course, it can't turn right without affecting the forward motion.

An arbitrary position and orientation can be achieved by a combination of linear motion (forward/backward movement) and rotation of the robot. Forward motion is done by increasing both δ_L and δ_R while rotation is achieved by increasing δ_L and decreasing δ_R and vice versa. Note that although backward movement is possible, it is not used in this case. Required movements are summarized in Table 4.1.

The vehicle is required to make two distinct turns – a "hard" turn – a rotation of 90 deg that is used for evasive maneuvers, and a "partial" turn, used for slight course corrections. Each turn requires different values of δ_L, δ_R, but the direction of change in δ is the same for both turns. The values of δ will be stored in a look-up table. Increasing ω_L or ω_R (while keeping the same relative value of δ) creates stronger moments and forces, which might be necessary for faster movement, especially rotation. Suitable values for ω_L and ω_R have yet to be determined.

The vehicle must make specific movements to follow a trajectory and a rule base determines which movements will be needed. These rules are organized in a *subsumption architecture* [55] (see Section 2.3 for details). Under this architecture, all control rules have an IF-THEN syntax. The rule base consists of the rules shown in Table 4.2. Each movement indicated in a rule's consequent has an associated set of δ and ω values, which are extracted via a table lookup.

Movement	Control Input
Move Forward	$\delta_L \uparrow$ & $\delta_R \uparrow$ (identical)
Left Turn	$\delta_L \downarrow$ & $\delta_R \uparrow$ (opposite)
Right Turn	$\delta_L \uparrow$ & $\delta_R \downarrow$ (opposite)
Idle	$\delta_L = \delta_R = 0$
Increase velocity	$\omega \uparrow$
Decrease velocity	$\omega \downarrow$

Table 4.1: Required vehicle movements. $\uparrow, \downarrow$ indicate direction of change, not its magnitude.

A subsumption architecture consists of a series of layers where lower layer rules produce simple, critical behavior such as avoiding obstacles while higher levels produce more sophisticated behavior needed for trajectory following. Higher level behavior subsumes lower level behavior. A subsumption architecture is ideal for navigation control in dynamic physical environments. It permits reactive behavior without resorting to prior path planning because there is no world model required.

Layer one has the highest priority. If the vehicle reaches the borders of the perimeter (in this case walls of the water tank), it will turn right to avoid the collision. The second highest priority detects whether the vehicle reached the desired waypoint W_j[1]. At that time, the controller agent acquires the coordinates of the next waypoint on the trajectory W_{j+1} from the strategy agent.

A new waypoint should be requested promptly to prevent the vehicle from unnecessary course corrections and large control actions. The third layer ensures that if the vehicle is pointing at the waypoint, it will move towards the waypoint. If it is not pointing in the right direction, layers four and five will turn the vehicle until it is pointing right at the waypoint, so the third layer (i.e. "move forward") takes control.

[1]The vehicle reaches waypoint W_j if it is within distance ϵ of that waypoint.

Layer	Behavior
6	**if** true **then** *Idle*
5	**if** heading left **then** *Partial right turn*
4	**if** heading right **then** *Partial left turn*
3	**if** heading at waypoint **then** *Move forward*
2	**if** at waypoint W_j **then** Get new waypoint W_{j+1}
1	**if** outside perimeter **then** *Hard right turn*

Table 4.2: Scheme of Controller agent subsumption architecture, Layer 1 has the highest priority.

Finally, if there is nothing better to do, the vehicle idles at its current location until it gets new commands.

4.2.2 Monitor Agent

This agent is responsible for monitoring the vehicle's performance. During normal vehicle operation, the controller agent posts an event every time a new δ and/or ω value is sent to the split-cycle oscillator. That event identifies the specific rule that fired, so the monitor agent knows the expected movement (e.g., *hard right turn*) and, when stepped by the scheduler, begins tracking the movement. If the vehicle's movement doesn't match the expected movement, the monitor agent will post a "poor performance" event in the event queue. No event is posted if the behavior is okay. The scheduler will process this poor performance event by stepping the diagnostic agent to run diagnostics.

4.2.3 Diagnostic Agent

This agent assesses the vehicle's reliability. The vehicle has no specific fault detection and isolation capability. It is worth noting that under such circumstances there is, from

a behavioral standpoint, no real difference between a vehicle mechanical failure or a sensor failure since both produce the same outcome—an inability to follow the desired trajectory. Nevertheless, rather simple diagnostic routines can identify degrading behavior even if the exact cause is not known.

Diagnostics could be performed at regular intervals. For example, every 5 minutes of flight time the vehicle could temporarily idle and then quickly run the diagnostics. However, a more practical approach is to exploit the online learning performed by the monitor agent. The monitor agent will trigger the diagnostics if and only if a degraded performance is observed.

Faults are detected using a Bayesian type of behavior monitor where *likelihood functions* give a qualitative measure of maneuver capability. The idea behind diagnostics is simple: command the vehicle to perform some maneuver and see if it can do it within a prescribed time frame. A rotation through some angle—e.g. $\pi/2$ radians—is a simple and non-trivial maneuver since it requires $\delta_L \neq \delta_R$. The precise δ values are stored in a table as described previously. Note that a complete diagnostic would require the vehicle to rotate in both directions. Pose samples $\hat{\Psi}_n = (\hat{\psi}_1, \hat{\psi}_2, \ldots, \hat{\psi}_n)$ can be recorded by diagnostic agents over some time window and the associated sample mean and sample variance are easily computed. This information is sufficient to construct a Gaussian *pose density function*:

$$f(\hat{\psi}) = \frac{1}{\sigma_\psi \sqrt{2\pi}} \exp\left(-\frac{(\hat{\psi} - \overline{\psi})^2}{2\sigma_\psi^2}\right) \tag{4.1}$$

where $\overline{\psi}$ is the sample mean and σ_ψ^2 is the sample variance.

The control agent outputs a specific δ_L and a δ_R, which are expected to produce some change in pose. Thus, the control agent has some expected pose movement $E[\psi]$ in mind. A diagnostic agent uses the pose density function $f(\hat{\psi})$ to compute the likelihood of $E[\psi]$ given the pose data samples $\hat{\Psi}_n$. This concept is illustrated in Figure 4.3.

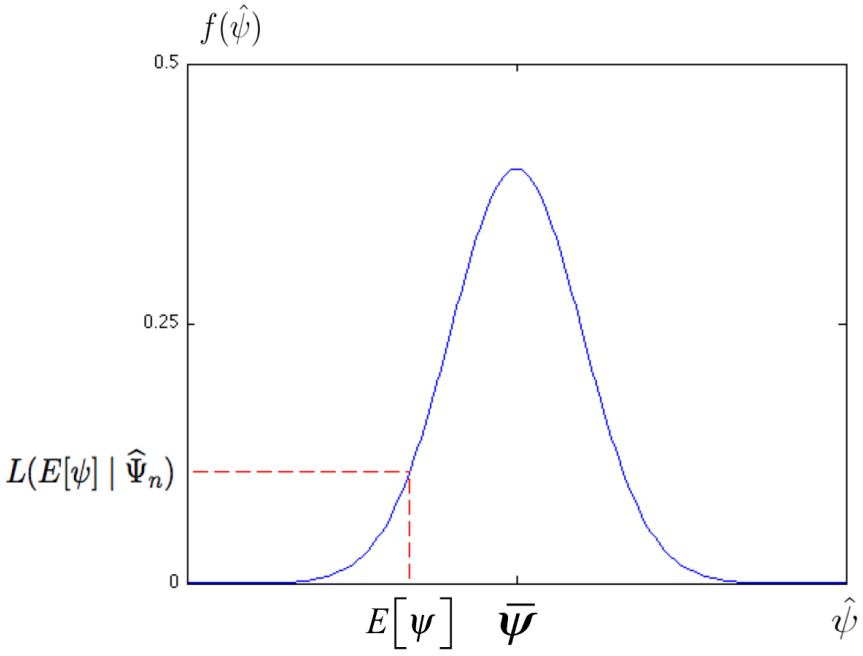

Figure 4.3: Calculation of likelihood of getting the expected pose from a pose density function centered at the sample mean of the n rotation estimate data samples $\hat{\Psi}_n$

A high likelihood suggests the vehicle mechanical hardware and sensor hardware are normally operating while a low likelihood indicates something is wrong. Dividing the likelihood function value by the sample mean (or equivalently just ignoring the $1/(\sigma_\psi\sqrt{2\pi})$ coefficient in Eq. 4.1) makes $L(E[\psi]|\hat{\Psi}) \in [0, 1]$. Then 'high' and 'low' likelihoods are defined by a threshold parameter λ on the unit interval. That is, $L(\cdot) < \lambda$ indicates a low likelihood the vehicle can maneuver properly, and some corrective action is required. The only possible recovery mechanism is to adapt the vehicle's behavior by modifying the rules in the rule base. The next section describes how this adaption is done.

4.3 Agent Online Learning

There are two times when online learning is required. Learning is used to identify appropriate δ and ω values needed for control of the vehicle. This section discusses the details of the learning process.

4.3.1 Initial Learning

Every vehicle is slightly different due to inherent non-linearities such as slip between linkages and manufacturing imperfections. Thus, the same δ and ω values cannot be used for every vehicle; they must be learned. First the vehicle learns values needed to execute the basic movements in Table 4.1. These values are then linked to rule consequents from Table 4.2. The δ and ω values will be determined during this initial learning phase using a combination of extrinsic and intrinsic evolution. The needed parameters will be evolved using a (1,10)-ES. The genotype is

$$\{\delta_L, \delta_R, \omega_L, \omega_R \ ; \ \sigma_1, \sigma_2, \sigma_3, \sigma_4\}$$

where the first 4 parameters are object parameters and the second 4 parameters, are strategy parameters used to control the mutation step size. The object parameters are mutated independently using a normal distribution and the appropriate strategy parameter. The equation for production of a new individual y from a single parent x is given as:

$$\delta_{L_y} = \delta_{L_x} + N(0, \sigma_{\delta_L}) \tag{4.2}$$

where $N(0, \sigma_{\delta_L})$ is a normally distributed random variable with zero mean and a standard deviation of σ_{δ_L}. The other *object parameters* with their respective *strategy parameters* are mutated the same way as described in the Equation 4.2. Most likely a linear reduction schedule will be sufficient for adapting the strategy parameters.

The vehicle will be placed in its operational environment – i.e. a water tank – and object parameter values will be intrinsically evolved for each movement. The initial object parameter values will be evolved extrinsically using an in-house developed simulator.

The monitor agent observes the behavior of each evolved object parameter set and

terminates the evolutionary algorithm for a particular rule when acceptable behavior is achieved. The goal here is not to achieve optimal movements but rather smooth and repeatable correct movements. Thus, fitness will be computed from the average behavior over a small number of trials. After learning is completed, the evolved values will be stored in a library (i.e. a look-up table). Once all rules are evolved the vehicle is ready for trajectory following.

4.3.2 In-Flight Learning

Rules learned during the initial learning phase perform well during normal flight, but in the presence of faults, it will be necessary to adapt them. This online learning phase is performed continuously. Each time a rule fires the monitor agent is informed, so it knows what maneuver was commanded. The monitor agent observes the $x', y, ' \psi'$ pose parameters and determines if the performance is within limits or is degrading. Thus, the monitor agent continuously learns about how the vehicle is performing. If the observed behavior deviates too much from the expected behavior, then diagnostics are run. If the diagnostics confirm the behavior has degraded below some threshold, then the rule base is adapted. Adaption, described below, entails modifying rules' consequents to restore vehicle's functionality.

4.3.3 Rule-base Adaptation

Given the size and weight restrictions, which don't allow us to use conventional fault recovery methods such as redundant hardware, it is not possible to recover from every possible fault. We, therefore, restrict the adaption to cover only a small subset of predefined faults. The recovery mechanism relies on adapting new consequents for rules from Table 4.2 so that the desired motion (i.e. *Partial left turn*) is still achievable. These new δ and ω values will be intrinsically evolved just like the initial online learning was done. The question is how do we intrinsically evolve new parameters for

specific faults?

We will borrow concepts used in conventional *failure modes & effects testing* (FMET). In this type of testing, a set of predefined faults is inserted into the system under test one at a time, and their effect is observed. This testing is always conducted in a laboratory environment where the effects are closely monitored and controlled to prevent damage to the system. In this particular research effort, "faults" such as using different wing sizes or a stuck linkage will be intentionally put into the vehicle, and an evolutionary algorithm will then evolve new parameter values. As before, this evolutionary algorithm will run using onboard hardware resources. The evolved values will be added to the rule base library.

Under normal operation, a single set of $\delta_L, \delta_R, \omega_L$ and ω_R values is sufficient to perform the maneuvers shown in Table 4.1. However, under faults most likely a sequence of δ and ω values will be required, with a new set generated every few hundred wingbeats (because of the slow vehicle dynamics). This sequence of values can be intrinsically evolved too. In this case, the genotype described earlier must be expanded to handle multiple δ and ω values for each maneuver.

Of course prior to initiating fault recovery operations *fault detection & isolation* (FDI) must be done. This process detects if a fault exists and tries to isolate it to a specific subsystem or component. Since the ability to recover from faults on the vehicle is severely limited, isolation is unnecessary. Fortunately, detection is rather straightforward. As stated above, the monitor agent is continuously learning about the vehicle capabilities; it can, therefore, detect poor performance by comparing observed behavior against expected behavior. If the behavior is poor, it will post an event in the event queue. The scheduler would then step the diagnostic agent to start diagnostics. Diagnostics should be able to confirm both degraded performances and identify which of the pre-defined faults is present. The abstract sequence of commands used during FDI is summarized in Algorithm 2.

Algorithm 2: The abstract sequence of commands used during FDI

1. Monitor Agent (MA) detects unsatisfactory performance and posts an event in event queue;
2. Diagnostics Agent (DA) is scheduled to perform diagnostics;
3. DA posts a diagnostic event. Controller Agent (CA) stops waypoint following and enters diagnostics mode;
4. CA starts the test maneuver;
5. MA monitors test progress and passes results to DA;
6. DA computes the likelihood of a fault;
7. If DA detects no problem, it posts a regular operation event. CA resumes waypoint following;
8. If DA detects a problem, it posts a faulty operation event. CA extracts new rule base from the library;

Essentially we will build a library of rules for both the fault-free and the faulty vehicle. Once FDI is finished fault recovery begins. Recovery only requires replacing the current controller agent's rule base with the appropriate pre-stored rule base associated with the identified fault from the library. Thus, fault recovery can be done very quickly.

In this chapter we presented the MAS and explained how we use it to solve the problem of autonomous waypoint following and fault recovery. In next chapter we will describe the experimental results of our work.

Chapter 5

Results

This section presents the findings and results of our research. First, the MAS as described in Chapter 4 Figure 4.1 was implemented. After testing the MAS in the simulator (described in Section 3.5), it was deployed on the actual robot. In order to do that, we had to check the functionality of the hardware. After several tests, it turned out that the moments of inertia of the robot were too large given the size of wings and magnitude of forces and moments they generated. As a result, the robot wasn't agile enough for our size of the water tank – in other words, the experimental area we had was too small for testing. Since increasing the size of the test area was not feasible, we resolved the problem by making a seconds version of the robot, and significantly reducing its weight and size. We were able to decrease the weight of the robot by 55% (from 406 g to 180 g), and the diameter by 30%, all by optimizing the mechanical and electrical components. The actuators and the wigs were unchanged. The second version of the robot had significantly lower moments of inertia, and as a result was much swifter. For more details about the robots we would refer the reader to Appendix A.

The next step was to find good δ and ω values for maneuvers described in Table 4.1. The robot's hardware limits max value of ω to 30 [rad/s] (minimum is 1 rad/s), and

$\delta \in (-10, 10)$ [rad/s], while the split-cycle control requires $|\delta| \leq \omega/2$ (see Section 3.2 for more details), so these values were used as lower and upper bounds for randomly initialized individuals in the (1,10)-ES algorithm (described in Section 4.3.1). The strategy parameters were empirically determined to be $\sigma_i \in (3, 6)$ where $i = 1, 2, 3, 4$.

As mentioned before, in this phase of the research effort, we used an evolutionary algorithm to search for the optimal values of δ and ω. Other optimization algorithms might be useful but whether or not that is true would require a detailed analysis of the solution space morphology which we did not do[1].

Our choice of an evolutionary algorithm to conduct the search was two-fold. First, evolutionary algorithms are usually considered optimization algorithms but basically they are search algorithms. Evolutionary algorithms can search any solution space regardless of morphology. Thus evolutionary algorithms allow us to optimize without conducting a solution space analysis. Secondly, and more importantly, every vehicle is slightly different due to inherent nonlinearities in the linkages and other manufacturing differences (such as a slightly different size of wings, etc.). As a result, *optimal* values for one vehicle will not be optimal for another. The goal here is not to achieve generally optimal movements but rather smooth and repeatable correct movements. Consequently, we needed a search method rather than an optimization method. Evolutionary algorithms allow us to search for good solutions by evaluating actual vehicle behavior, which cannot be accomplished using classical optimization algorithms. Two types of evolution - *extrinsic* and *intrinsic* were used, as described below.

[1]However, our experiments did show small perturbations in δ/ω had no observed behavioral changes which suggests gradient-based optimization algorithms would not be very effective.

5.1 Extrinsic Evolution

Because the lifespan of linkages at the vehicle is limited, it is reasonable to first execute extrinsic evolution of control parameters δ and ω to 1) verify the correctness of the evolution algorithm; 2) get the initial estimate of optimal control parameters. The more precise model of the vehicle is available, the better is the estimate. However, obtaining a precise (first-principle) model of the flapping wing system is very complicated, especially because of the small forces and torques that would have to be measured to correctly identify the model [33]. Modelling non-linearities such as linkage slip also poses a significant challenge [36]. However a simplified model that treats the vehicle as a point mass body and aggregates the generated forces and torques is a sufficient approximation, because it will exhibit similar behavior albeit on a different time scale.

For the purpose of the extrinsic evolution, we started with the following assumption. The faster the wings beat (i.e. higher ω), the more force is generated (because the wing acceleration is higher). The higher split cycle shift (higher $|\delta|$), the more force is generated (because the difference between upstroke and downstroke is higher). The higher force results into faster movement.

The optimal solution completes the basic movement in shorter time, and is within the imposed constraints. Thus in our simple model we use to evaluate fitness of candidate solution we employ the following equation:

$$\text{fit}(x) = K_L \cdot \delta_{L_x} \cdot \omega_{L_x} + K_R \cdot \delta_{R_x} \cdot \omega_{R_x} \tag{5.1}$$

where K_1, K_2 are adjustable weights, in the simplest case:

- $K_L = K_L = 1$ for *Move forward*
- $K_L = 1; K_2 = -1$ for *Turn right*
- $K_L = -1; K_2 = 1$ for *Turn left*

We ran the EA for 20 generations in each run, for 20 runs total. The expected optimal solution would converge to maximal ω for both wings and maximal values for δ but with opposite signs in case of turns. The results are shown in Figure 5.1. Notice in all cases the runs converged to (or at least very close to) the global optimum. The best evolved values for our simple model are summarized in Table 5.1, and they are consistent with our expectations.

5.2 Intrinsic Evolution

During the intrinsic evolution we used the actual vehicle for evaluating fitness of the candidates. The major difference was that for *Turn left* and *Turn right* fitness is defined as $f = 1/T$ where T is the time needed for the vehicle to turn by 360 degrees from its initial position. For *Move forward* the fitness is defined as $f = 1/(K_1 \cdot T_{wp} + K_2 \cdot d_{wp})$, where T_{wp} is time needed to reach x-coordinate of the waypoint p (located approximately 30 cm in front of the vehicle), d_{wp} is the distance from the y-coordinate of the waypoint p when its x-coordinate is reached. $K_1 = 100$ and $K_2 = 1$ are weights to scale the different units (seconds and pixels).

Because the hardware has a limited lifespan, we only ran the EA once and for only 20 generations. This is limiting in the sense that we can reach suboptimal results, but if we were to run more runs as was the case for extrinsic evolution, the linkages could wear out prematurely and would have to be replaced, in which case the learning would have to be done again from the beginning.

The incremental improvements in turn times for evolved control parameters are shown in Figure 5.2. For the forward motion, the vehicle actually was not able to reach the desired waypoint (its y-coordinate) in vast majority of tries. In such case the experiment was stopped after 2 minutes and the fitness of given individual was marked as zero. Effectively this reduced the EA to a random search, until a viable solution

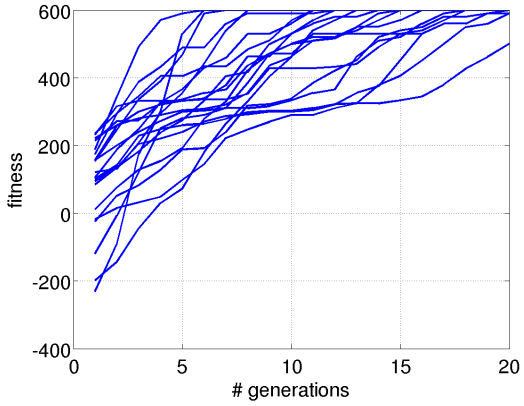

Figure 5.1: Extrinsic evolution run for *Turn left* move. Notice that the algorithm in almost all cases reaches global optimum $f = 600$

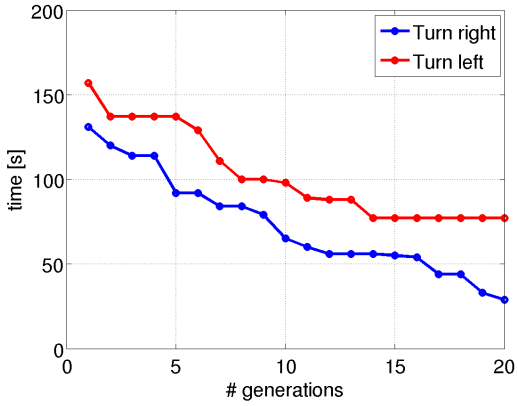

Figure 5.2: Time needed to complete *Turn left* and *Turn right* moves during intrinsic evolution. Notice that left turn takes longer to finish, which is caused by non-linearities in the hardware.

was found. The best solution after 20 generations (and the only one found that had non-zero fitness) is shown in Figure 5.3. The best values of control parameters found for our vehicle are summarized in Table 5.2. The best intrinsically evolved values are very similar to the values found by extrinsic evolution (compare with Table 5.1). This validates the model used for extrinsic evolution, and makes it usable for the initial estimate. The differences in control parameters are caused by imperfections and non-linearities in real hardware, and indeed, those were not included in our model. The intrinsically evolved values, recorded in Table 5.2, were used for further experiments.

5.3 Waypoint Following

Once the control parameters for the basic moves were learned, it was possible to proceed towards waypoint following. Two waypoints were placed to the opposite sides of the water tank, and the vehicle was expected to go back and forth between them. Two waypoints were sufficient because following them required all basic maneuvers. The vehicle was able to follow successfully the waypoints, as can be seen in Figures 5.4

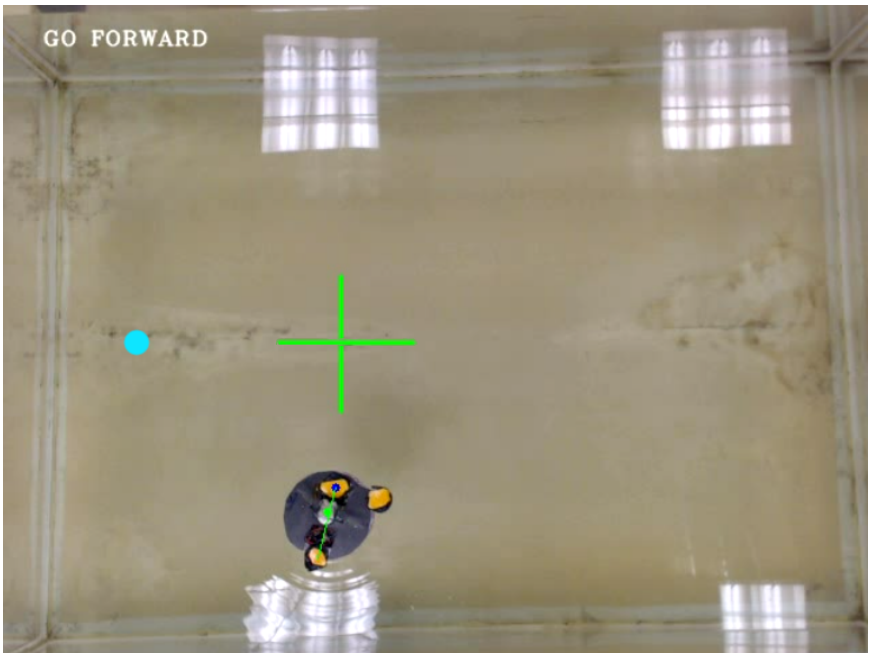

Figure 5.3: Best found solution for forward movement *Blue:* Initial position of the vehicle, *Green cross:* waypoint the vehicle was commanded to reach. The experiment was stopped once the center of vehicle crossed the y-axis of the waypoint.

Movement	δ_L	δ_R	ω_L	ω_R
	[rad/s]			
Move Forward	**10**	**10**	**30**	30
Left Turn	**-10**	10	**30**	30
Right Turn	10	-10	30	30
Idle	0	0	0	0

Table 5.1: Control parameters for the basic movements obtained from extrinsic evolution. The values that differ from those obtained by *intrinsic evolution* are bold. Values for *Idle* (which simply stops the actuators) were not evolved.

Movement	δ_L	δ_R	ω_L	ω_R
	[rad/s]			
Move Forward	**12**	**14**	**25**	30
Left Turn	**0**	**12**	**12**	30
Right Turn	**12**	**0**	**25**	**12**
Idle	0	0	12	12

Table 5.2: Control parameters for the basic movements obtained from intrinsic evolution. Values for *Idle* were determined empirically and based on the hardware initialization procedure (default values). The values for *Left Turn*, *Right Turn*, and *Move Forward* differ from those obtained by *extrinsic evolution* are bold.

and 5.5. The vehicle is as large as the dotted circle shown around waypoints in Figure 5.5, so once the center of the vehicle reaches the dotted circle, the waypoint is considered to be reached. Figure 5.6 shows control inputs and rules that fired during the experiment. We can see that all three rules (*Turn Left*, *Turn Right*, *Go Forward*) are used a similar amount of time. Figure 5.7 shows the heading of the vehicle (there is a wrap-around at 180 and -180 deg). We can see that the heading is relatively steady, changing between ~ 10 degrees and ~ -170 degrees as the vehicle goes back and forth between the waypoints. The spikes indicate turn-arounds after reaching the waypoints. The rate of change is within ± 10 [deg/s] during the whole experiment, as shown in Figure 5.8. This indicates that the vehicle is moving as fast as possible and that its dynamics is relatively slow.

To get a better estimate about the time required to reach a waypoint and to turn left or right, the waypoint following experiment was repeated ten times. The times for each run as well as averaged times are summarized in Table 5.3. We can see that the two waypoints can be reached on average in 5 minutes. The large range (from 2 min 31 seconds to 7 min 2 seconds) is mostly due to varying initial conditions.

We also measured intervals needed for *Turn left* and *Turn right* in order to have a baseline of the robot's performance. Both turns were by 360 deg and were measured ten times from zero initial conditions (i.e. the vehicle wasn't in motion). The results are in Table 5.3 which provides data necessary for construction of the pose density function.

5.4 Fault recovery

Fault detection and recovery is an important feature, as was mentioned in Chapter 4.3. Without a loss of generality we set up a fault detection and recovery mechanism for one fault - a damage of the left wing. In a similar manner a damage of the right wing

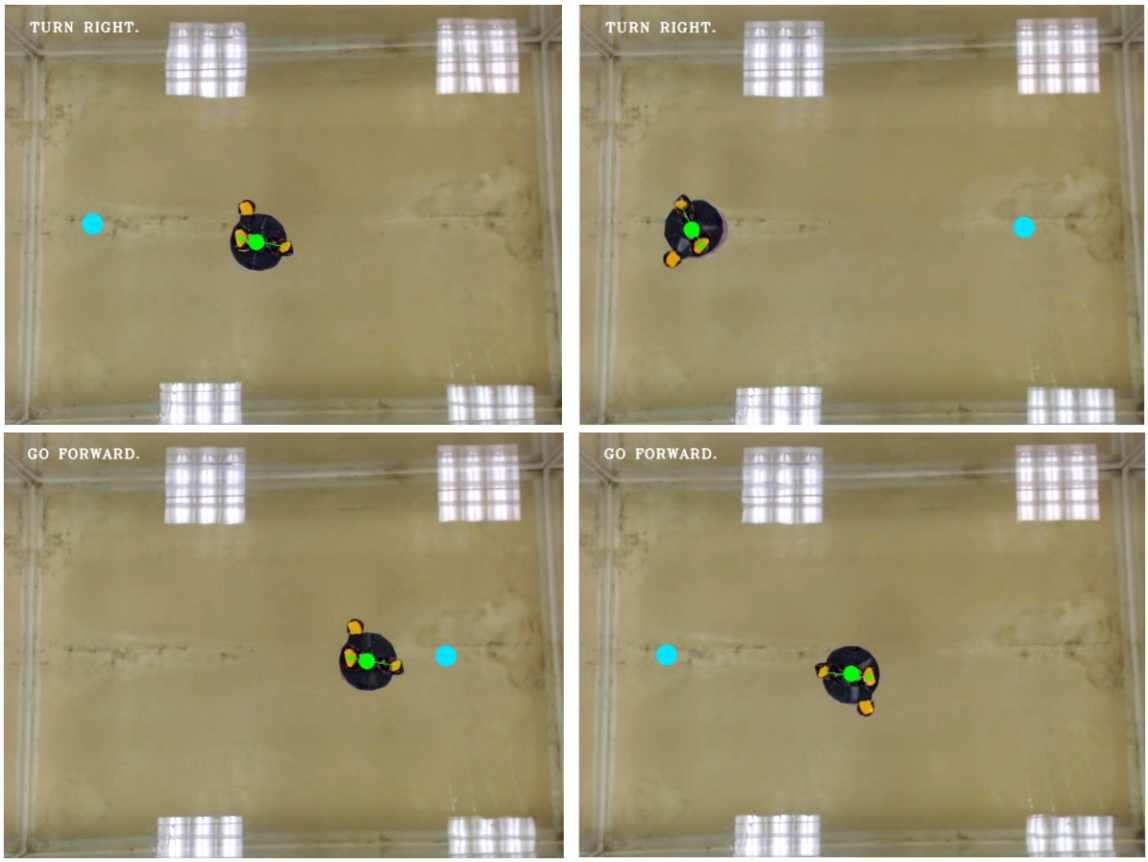

Figure 5.4: Autonomous waypoint following. The blue dot is the desired waypoint; the vehicle is marked with a bright green dot and a green line pointing towards the front of the vehicle. In the top left corner of the screen is shown the rule that fired. Top left: Initial position of the vehicle; Top right: First waypoint achieved, the vehicle is turning around; Bottom left: Approaching the second waypoint; Bottom right: Second waypoint achieved, moving back to the first waypoint.

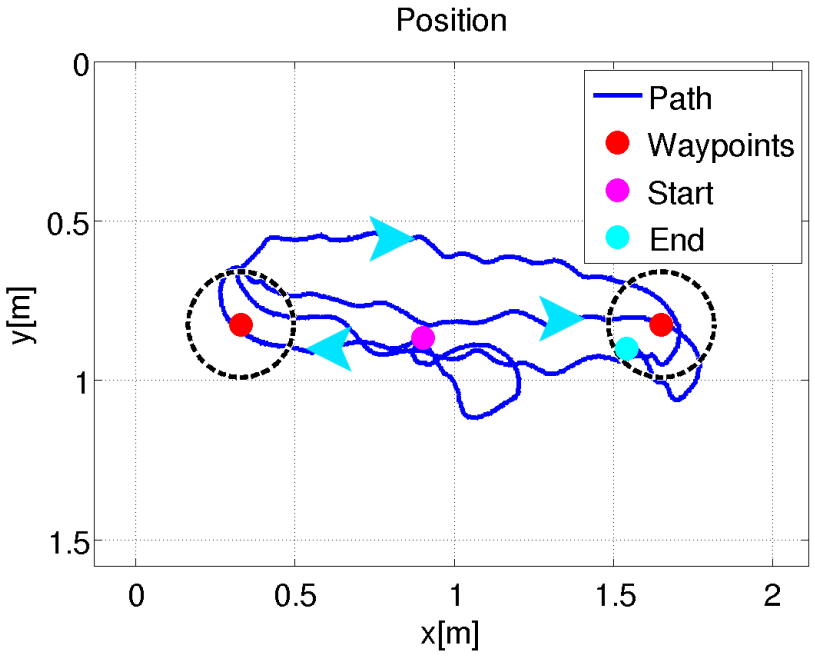

Figure 5.5: Vehicle trajectory (blue) during waypoint following. Two waypoints being followed are marked with red dots, with distance threshold pictured around them. The vehicle itself is as large as the circle around the waypoints. *Purple:* start position, *Light blue:* end position, *Blue arrows:* indicate orientation of the vehicle

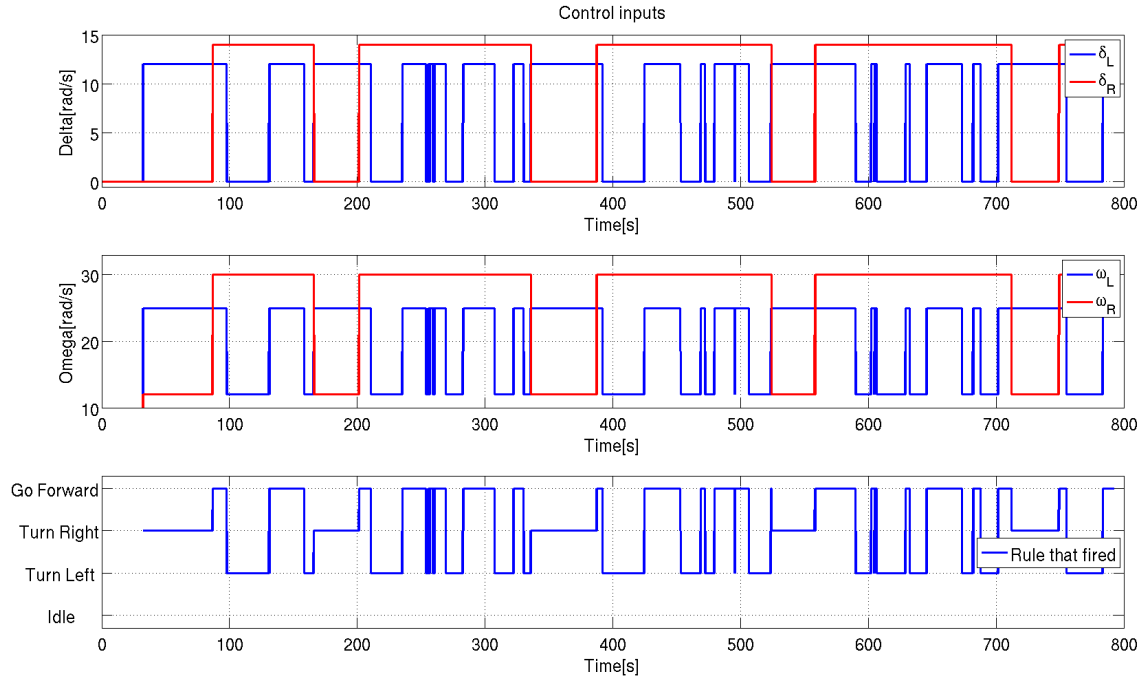

Figure 5.6: Control inputs during waypoint following. *Top:* δ values, *Middle:* ω values, *Bottom:* rule that fired.

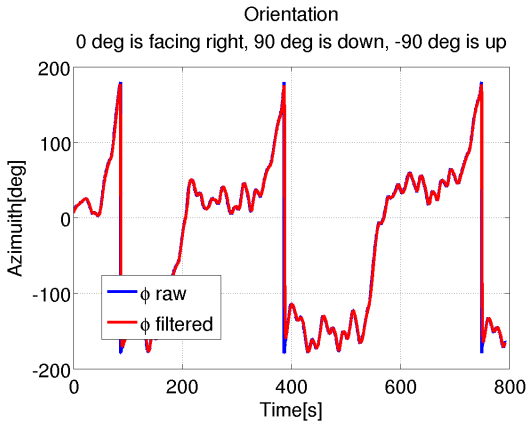

Figure 5.7: Heading of the vehicle during waypoint following. Note the wrap-around at ±180 deg. 0 deg is in the direction of positive x-axis, ±180 deg is in the direction of negative x-axis. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$

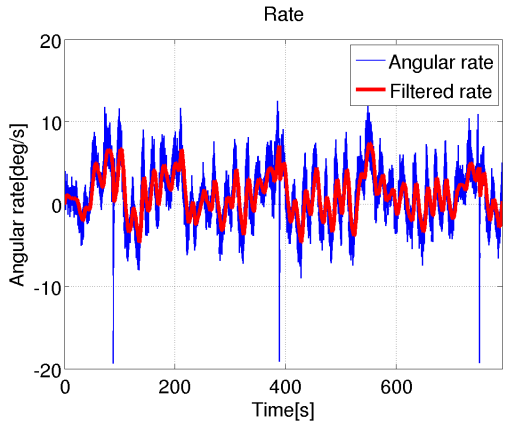

Figure 5.8: Angular rate of the vehicle's heading during waypoint following. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$ The peaks are residuals from wrap-arounds at ±180 deg.

#	Turn Left [mm:ss.ss]	Turn Right [mm:ss.ss]	Waypoints [mm:ss.ss]
1	00:52.00	01:53.00	05:01.00
2	00:52.00	01:36.00	07:02.00
3	00:42.00	01:25.00	*N/A*
4	00:46.00	01:07.00	*N/A*
5	00:39.00	01:02.00	06:18.00
6	01:02.00	01:15.00	05:46.00
7	00:38.00	00:56.00	04:33.00
8	00:45.00	01:18.00	03:55.00
9	00:39.00	01:13.00	06:12.00
10	01:01.00	01:14.00	02:31.00
Mean:	**00:47.00**	**01:17.90**	**05:09.75**
Max:	01:02.00	01:53.00	07:02.00
Min:	00:38.00	00:56.00	02:31.00

Table 5.3: Basic maneuvers performed under nominal conditions with control parameters intrinsically evolved. *Turn Left* and *Turn Right* times are for a 360 deg rotation from zero initial conditions. *Waypoints* are two at the opposite sides of the water tank, the vehicle starts at waypoint 1 and the time is running until it reaches Waypoint 2

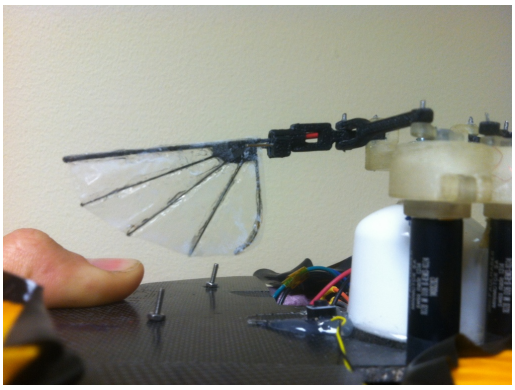

Figure 5.9: Detail of the original left wing

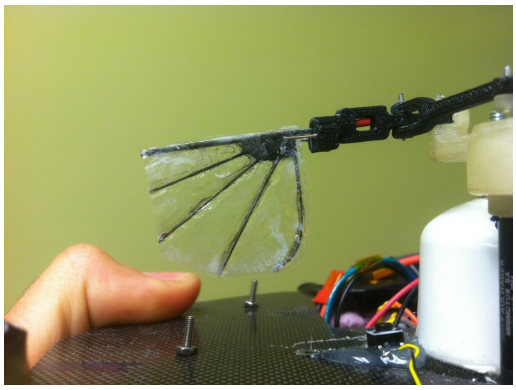

Figure 5.10: Detail of the damaged left wing, roughly 30% of the surface area was removed

can be detected and consequently recovered. In practice, the damage could occur after a collision with an obstacle. In the experimental setup the wing was cut down by around 30% (see Figures 5.9 and 5.10). Because the tip of the wing produces most of the aerodynamic forces, we can expect an even higher loss of performance. After damaging the wing, identical maneuvers as with undamaged wing were performed (i.e. no adaptation of δ and ω parameters occurred). The times are recorded in Table 5.4. Since the left wing was damaged, we expect *Turn Right* turn to be mostly impacted, and as a result waypoint following ability will be affected too.

For *Turn Right* we observe 44% increase in the average time to perform the turn (from 1:17 to 1:51), while *Turn Left* is impacted only minimally (increase by 6% from 00:47 to 00:50 on average). Interestingly, the vehicle is still able to follow the waypoints albeit with worse performance. Note that the waypoint following was executed only once to avoid excessive strain on linkages and premature wear of the vehicle.

In order to detect the fault, the vehicle - after it initializes - performs a 360 deg *Turn Right* and measures the time T it takes. If $T > T_{max}$ where $T_{max} = 90$[s] as was calculated from Tables 5.3 and 5.4, then the *Left Wing Damaged* fault is triggered. To recover from the fault, we had to again intrinsically evolve the δ and ω parameters. Note that the initial assumption was that this fault is recoverable - i.e. with a proper

#	Turn Left [mm:ss.ss]	Turn Right [mm:ss.ss]	Waypoints [mm:ss.ss]
1	00:45.00	02:12.00	06:15.00
2	00:51.00	01:42.00	*N/A*
3	01:14.00	01:39.00	*N/A*
4	00:48.00	01:25.00	*N/A*
5	00:44.00	01:18.00	*N/A*
6	00:55.00	01:55.00	*N/A*
7	00:42.00	01:43.00	*N/A*
8	00:42.00	02:12.00	*N/A*
9	00:49.00	02:24.00	*N/A*
10	00:49.00	02:03.00	*N/A*
Mean:	**00:50.00**	**01:51.30**	**06:15.00**
Max:	01:14.00	02:24.00	*N/A*
Min:	00:42.00	01:18.00	*N/A*

Table 5.4: Basic maneuvers performed after left wing damage (control parameters unchanged).

control input full function of the vehicle is possible. Other faults, for example a broken linkage or a stuck motor would not be recoverable. After running the intrinsic evolution for 20 generations (same settings as described in Section 5.2) the evolved δ and ω values were stored in Table 5.5.

Once the recovery values were found, it was possible to test their effect. The average times as well as individual trial times for all the basic maneuvers are shown in Table 5.6. With the updated control values the times for *Turn Left* and *Turn Right* are comparable with the undamaged wings, and so is the waypoint following. Once again, to conserve the hardware only two iterations of waypoint following were performed. Figure 5.11 shows the fault recovery experiment, Figure 5.12 shows the tracked path of the vehicle, and Figures 5.14 and 5.15 show the orientation and angular rates of the vehicle. The control inputs are shown in Figure 5.13, the long period of *Turn Right* at the beginning of the experiment is when the diagnostics agent is running the test for presence of a fault. In a similar fashion *Right Wing Damaged* fault could be detected and recovered from, assuming only one fault occurs at a time.

Movement	δ_L	δ_R	ω_L	ω_R
	[rad/s]			
Move Forward	**14**	14	**30**	30
Left Turn	0	**14**	12	30
Right Turn	**14**	0	**30**	12
Idle	0	0	12	12

Table 5.5: Evolved fault recovery control parameters. Values for *Idle* were determined empirically and based on the hardware initialization procedure (default values). The values for *Left Turn*, *Right Turn*, and *Move Forward* differ from the undamaged values are bold.

#	Turn Left [mm:ss.ss]	Turn Right [mm:ss.ss]	Waypoints [mm:ss.ss]
1	01:22.00	01:32.00	04:33.00
2	00:57.00	01:32.00	03:20.00
3	01:00.00	01:26.00	*N/A*
4	00:52.00	01:22.00	*N/A*
5	*N/A*	01:19.00	*N/A*
6	*N/A*	01:24.00	*N/A*
7	*N/A*	01:20.00	*N/A*
8	*N/A*	01:38.00	*N/A*
9	*N/A*	01:22.00	*N/A*
10	*N/A*	01:11.00	*N/A*
Mean:	**01:02.75**	**01:24.60**	**03:56.00**
Max:	01:22.00	01:38.00	*N/A*
Min:	00:52.00	01:11.00	*N/A*

Table 5.6: Basic maneuvers performed after recovery left wing damage with updated control parameters.

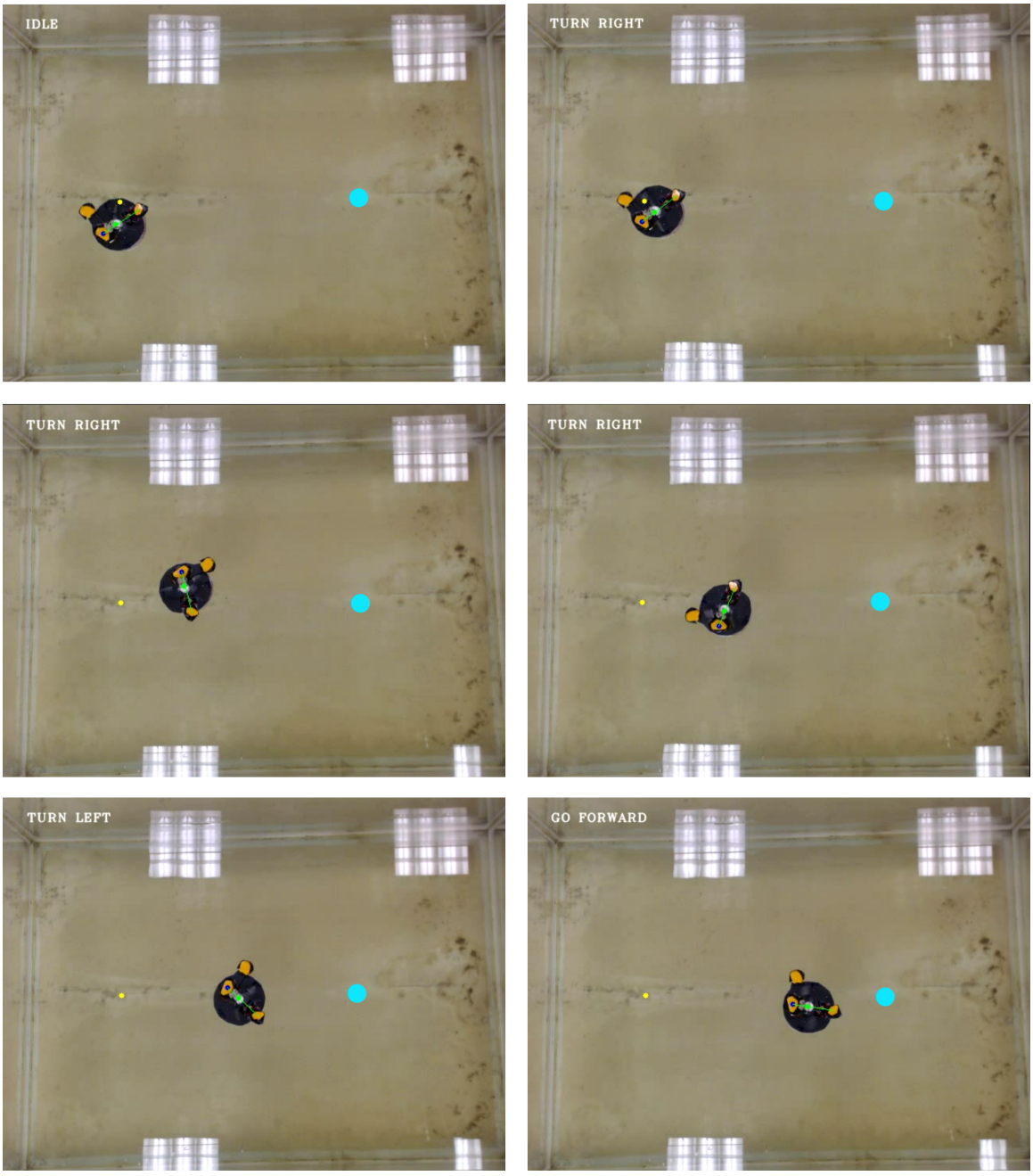

Figure 5.11: Fault recovery example. The obstacle is the green line located in the middle of the water tank, the waypoint the wehicle follows is marked light blue. *Top left:* the vehicle is initializing its wings (takes around 30 sec at the beginning of each experiment), *Top right:* the diagnostic agent initiates the diagnostics and starts a 360 deg right turn, *Middle left:* diagnostics in progress, *Middle right:* diagnostics was finished, the diagnostics agent found a presence of a fault and loaded the evolved fault-recovery control values, *Bottom left:* the vehicle is progressing towards the second waypoint, *Bottom right:* Final approach towards the waypoint

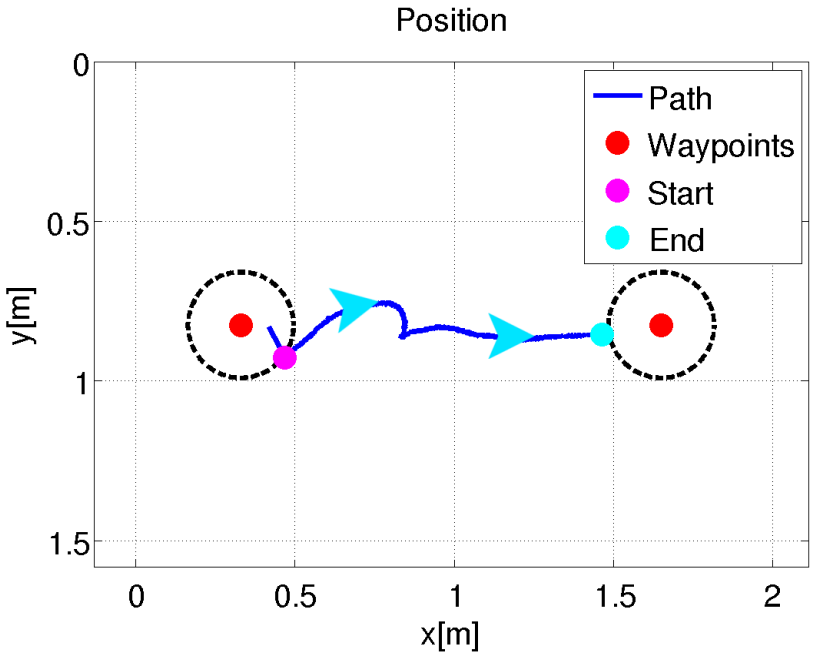

Figure 5.12: Path travelled during fault recovery maneuver from Figure 5.16. *Purple:* start position, *Light blue:* end position, *Green line:* the obstacle, *Blue arrows:* indicate the orientation of the vehicle

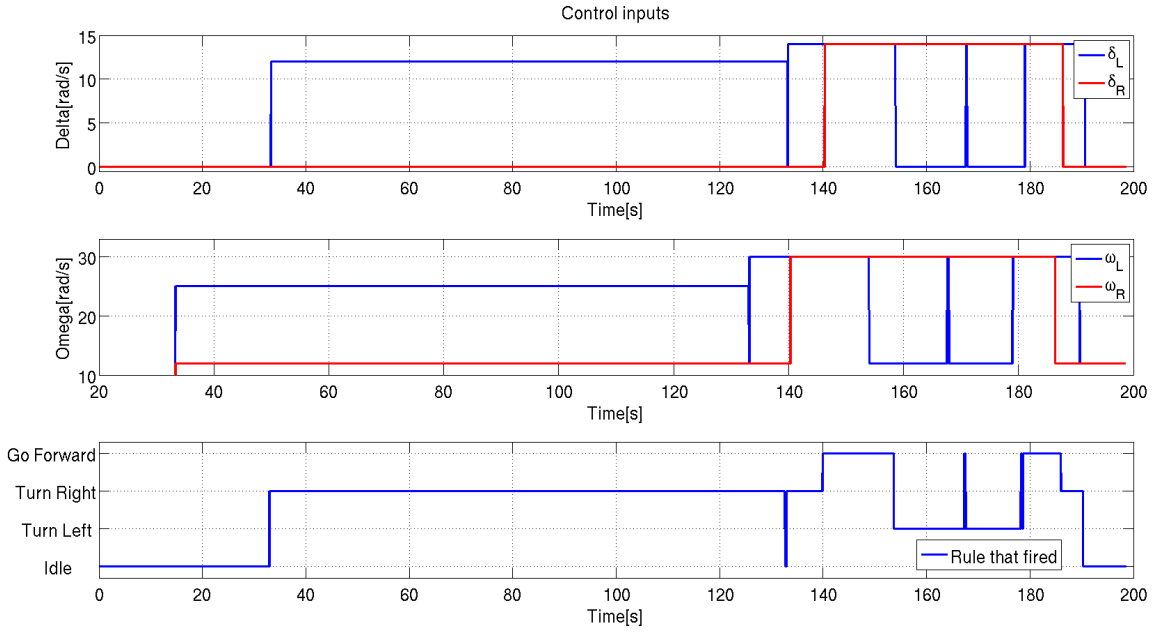

Figure 5.13: Control inputs during fault recovery. *Top:* δ values, *Middle:* ω values, *Bottom:* rule that fired. Note the 30 second initialization period at the beginning of the experiment (rule: *Idle*), and the diagnostics phase with *Turn Right* command.

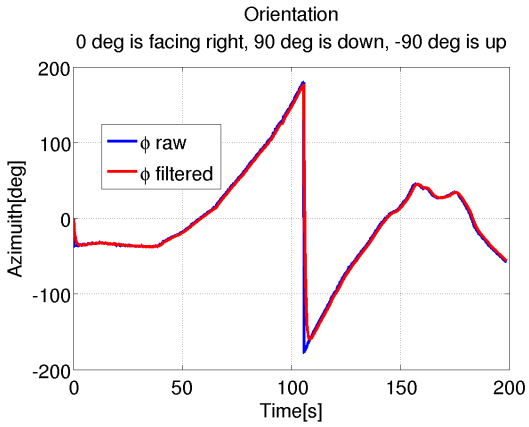

Figure 5.14: Heading of the vehicle during fault recovery. Note the wrap-around at ±180 deg. 0 deg is in the direction of positive x-axis, ±180 deg is in the direction of negative x-axis. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$

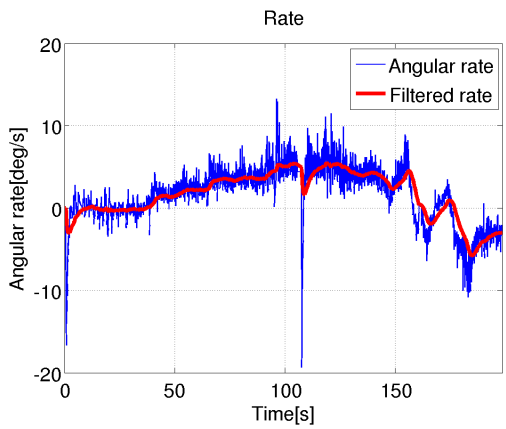

Figure 5.15: Angular rate of the vehicle's heading during fault recovery. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$. The large peak is a residual from a wrap-around at ±180 deg.
.

5.5 Obstacle Avoidance

During the experiments it became apparent that a modification of the rule base from Table 4.2 is necessary. Specifically, the rule number 1: **if** outside perimeter **then** *Hard right turn* never occurs, because the robot cannot physically get outside the perimeter (there are walls in the water tank). Limiting the size of the operating area, so the robot is not allowed to hit the walls, is impractical, because of the extra space limitation. More practical response would be *avoid hitting the wall* or more generally *avoid obstacle*. *Hard right turn* is also not distinguishable from a *Partial right turn*, because the vehicle is already operating at its maximal capability, and is turning as fast as possible. A modified rule base that contains these changes is shown in Table 5.7.

The *Avoid Obstacle* routine needs to be more complex than the other rule consequents. In this case the position of the obstacle is explicitly known, which is sufficient if we want the vehicle to avoid walls of the water tank and/or obstacles that are

Layer	Behavior
6	**if** true **then** *Idle*
5	**if** heading left **then** *Partial right turn*
4	**if** heading right **then** *Partial left turn*
3	**if** heading at waypoint **then** *Move forward*
2	**if** at waypoint W_j **then** Get new waypoint W_{j+1}
1	**if** obstacle ahead **then** *Avoid Obstacle*

Table 5.7: Modified Scheme of Controller agent subsumption architecture with updated rule for Layer 1 (Layer 1 has the highest priority).

imposed in the water. The routine goes is summarized in Algorithm 3

Algorithm 3: Updated Obstacle Avoidance routine

```
if WP_2 behind the obstacle then
 | move WP_tmp in front of the obstacle on the |WP_1, WP_2| line;
end
while !wp_reached(WP_tmp) do
 | wait;
end
shift WP_tmp right (vehicle side) until after the obstacle ends;
while !wp_reached(WP_tmp) do
 | wait;
end
shift WP_tmp forward (vehicle side) behind the obstacle;
while !wp_reached(WP_tmp) do
 | wait;
end
shift WP_tmp left (vehicle side) on the |WP_1, WP_2| line;
```

where WP_1 is the initial waypoint, WP_2 is the final waypoint, and WP_{tmp} is an intermittent waypoint that the vehicle follows while avoiding the obstacle. An example will better explain this - a run with an obstacle in the middle of the water tank is described in Figure 5.16, and Figure 5.17 shows the path the vehicle travelled. The most common rule is *Turn Right* and *Go Forward*. Figures 5.19 and 5.20 show the heading and the angular rate of the vehicle during the experiment. As can be seen,

the vehicle was able to successfully avoid the obstacle.

This obstacle avoidance routine can be improved, because at the moment it covers only cases where a waypoint is behind the obstacle (so going straight ahead is not possible). However, our experience from conducting the experiments has shown that in order to avoid occasionally hitting the walls of the water tank, a more nimble vehicle with more control authority is needed. For example, in several occasions the control system was issuing the correct commands to avoid hitting the wall (e.g. *Partial right turn*), but the vehicle already had too much momentum from the previous movement that it couldn't be turned and stopped in time before the collision. Other example included improper initialization of the vehicle, and consequent malfunction of the control system. In both cases a better obstacle avoidance routine wouldn't have helped.

5.6 Summary

In this chapter we demonstrated a successful implementation of the MAS from Chapter 4 and the EA for determining the values of control parameters δ and ω. We have shown that the vehicle is capable of autonomous waypoint following, which was repeated multiple times, as well as fault detection and recovery - which was demonstrated with a damaged left wing. On top of that, we updated the rule base (see Tables 4.2 and 5.7) to better suite our needs, and implemented obstacle avoidance routine. The implications of this work and a conclusion of our efforts is discussed in the next chapter.

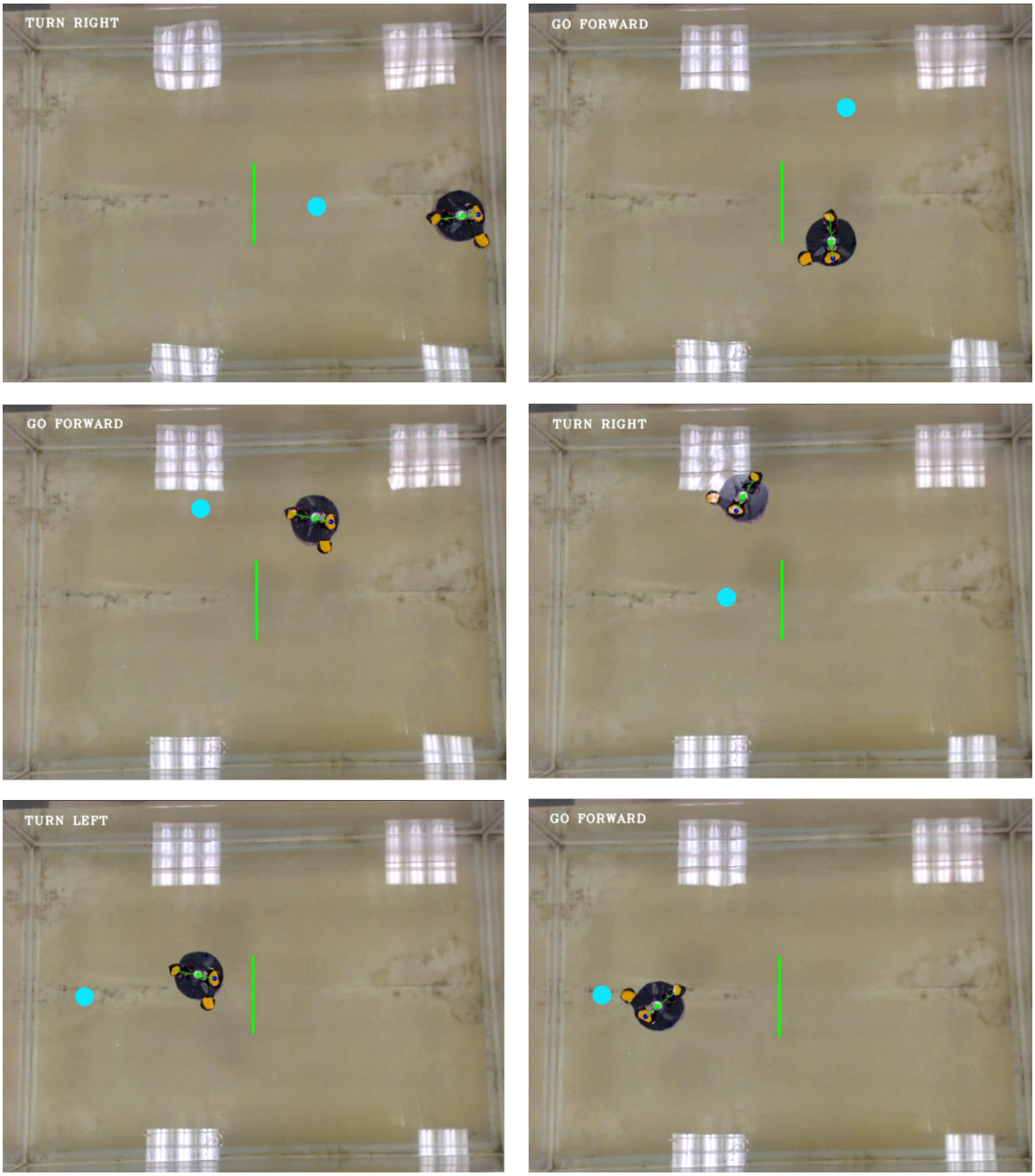

Figure 5.16: Obstacle avoidance example. The obstacle is the green line located in the middle of the water tank, the waypoint the wehicle follows is marked light blue. *Top left:* vehicle starting at WP_1, moving towards the obstacle. *Top right:* vehicle reached WP_{tmp} in front of the obstacle and moved it above the obstacle. *Middle left:* WP_{tmp} is behind the obstacle. *Middle right:* WP_{tmp} is on the $|WP_1, WP_2|$ line. *Bottom left:* end of *Avoid Obstacle* routine, the vehicle follows WP_2. *Bottom right:* the vehicle is about to reach WP_2

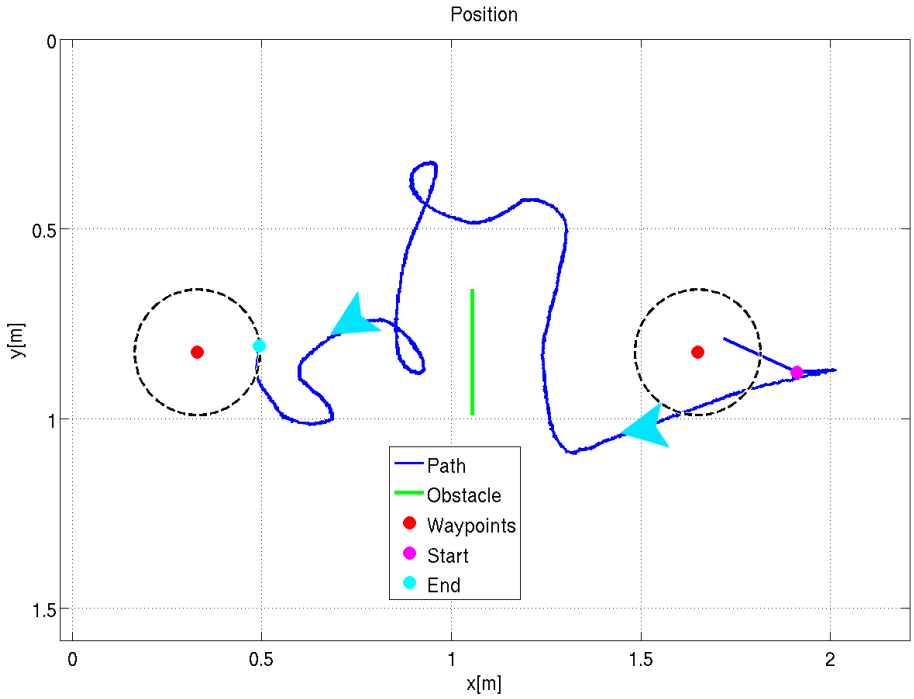

Figure 5.17: Path travelled during obstacle avoidance maneuver from Figure 5.16. *Purple:* start position, *Light blue:* end position, *Green line:* the obstacle, *Blue arrows:* indicate the orientation of the vehicle

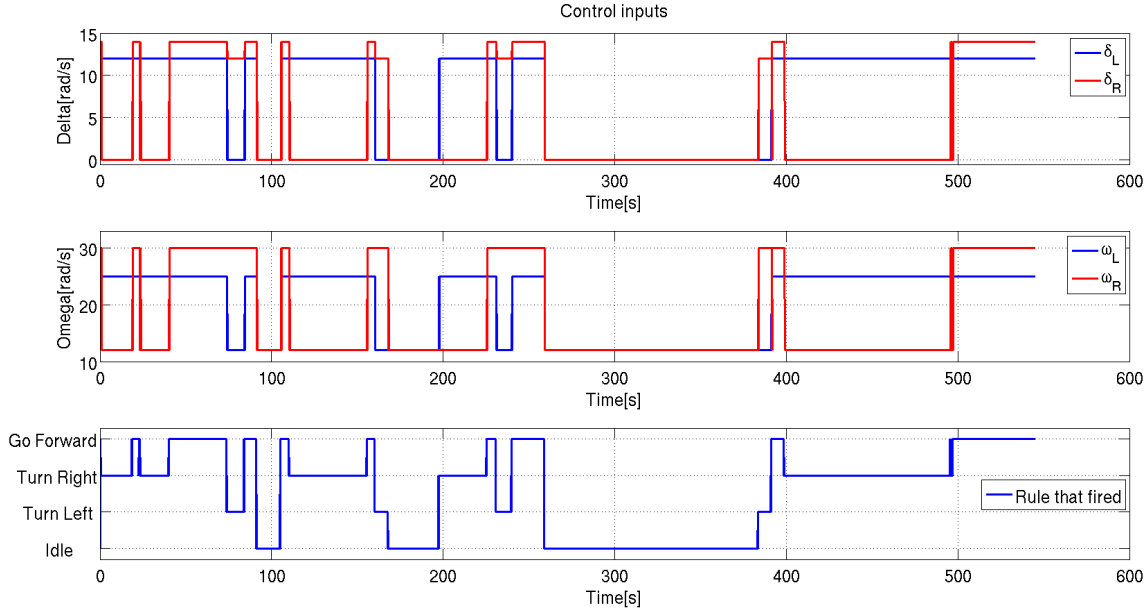

Figure 5.18: Control inputs during obstacle avoidance. *Top:* δ values, *Middle:* ω values, *Bottom:* rule that fired. The large segment of *Idle* in the middle of the experiment is caused by a momentary loss of tracking of the algorithm.

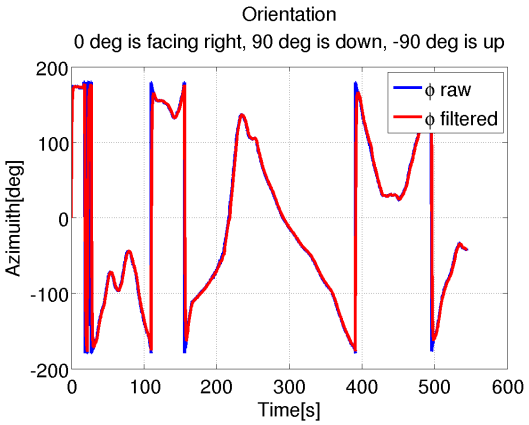

Figure 5.19: Heading of the vehicle during obstacle avoidance. Note the wrap-around at ±180 deg. 0 deg is in the direction of positive x-axis, ±180 deg is in the direction of negative x-axis. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$

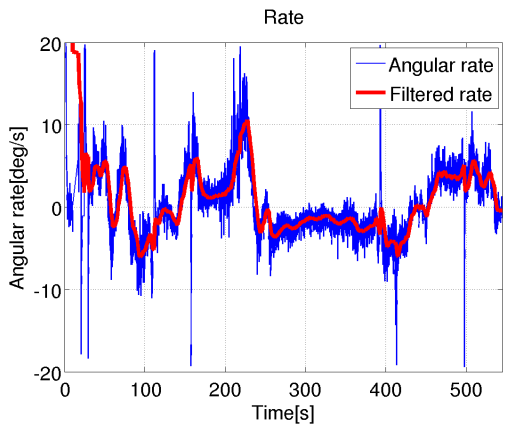

Figure 5.20: Angular rate of the vehicle's heading during obstacle avoidance. The values were filtered with an exponential moving average low-pass filter with $\alpha = 0.05$ The peaks are residuals from wrap-arounds at ±180 deg.

Chapter 6

Conclusion & Future Work

In Chapter 1 we have shown that Flapping-Wing Micro Aerial Vehicles (FWMAVs) have potential for many applications, ranging from military reconnaissance in the battlefield, through search & rescue for mapping dangerous environments and helping during disasters, to artificial plant pollination. In order to fully explore their potential, autonomous operation and fault tolerance is required. Chapter 2 provided theoretical background needed for understanding the concept of FWMAVs and their design, testing and control. The expected outcomes of our research were defined in Chapter 1:

1. understanding the viability of multi-agent system (MAS) for control of flapping wing vehicles

    - Based on the state-of-the-art research summarized in Chapter 2, we showed that a MAS is suitable for the control of a FWMAV with many advantages over conventional control systems.

2. developing a multi-agent control system allowing the vehicle to follow trajectory in experimental settings

    - a MAS has never been used for the control of a FWMAV before. We developed a MAS capable of control, navigation and fault recovery of our

FWMAV. The MAS is decribed in detail in Chapter 4. This system can be used on other FWMAVs, for example those mentioned in Chapter 2 or others not developed yet.

3. developing fault detection and fault recovery mechanisms based on a combination of extrinsic and intrinsic evolution

   - The fault detection and recovery system mechanism is a part of the developed MAS. Extrinsic evolution is used for initial estimate of the control values of the FWMAV, while intrinsic evolution is used to fine-tune those values for each individual vehicle.

4. developing high degree of autonomy of the vehicle, including trajectory following and fault recovery procedures

   - The high degree of autonomy was achieved by using subsumption architecture in conjunction with a MAS. On top of trajectory following and fault recovery our system includes an obstacle avoidance routine, which allows the vehicle to operate in presence of obstacles.

In summary we were able to meet and exceed all expected results, and deliver a working prototype of an autonomous fault tolerant vehicle. The approach we pioneered can be used for new types of FWMAVs and other research projects can use our research as a jump start for their own implementation.

6.1 Future Work

One area that could be explored in deeper details is the evolution of control parameters. We used Evolution Strategy, but the evolutionary algorithm used in this work is only one of many existing evolutionary algorithms. More sophisticated algorithms, such

as *Artificial Bee Colony* [72] or *Particle Swarm Optimization* [73] can be used and compared and evaluated.

The most natural next step would be to focus on hardware of the robot, and remove the restriction to 2 dimensional movement. Flapping wing platforms of comparable size that are able to carry their own weight already exist (for example [39]) so research in this direction would be promising. Another option is to aim for free-flying platforms with on-board sensors, such as [35], and integrate attitude & position estimation algorithms to establish a truly autonomous vehicle.

We can expect many more flapping-wing insect-like robots in the coming years, and we are very happy that we were able to contribute to knowledge in this area.

References

[1] Proxdynamics, "PD-100 Black Hornet," 2016. [Online]. Available: http://www.proxdynamics.com/

[2] Reconrobotics, "Throwbot XT," 2016. [Online]. Available: http://www.reconrobotics.com/products/throwbot-xt/

[3] Chicago Tribune, "Drone helped Gurnee firefighters assess propane-storage blaze," 2016. [Online]. Available: http://www.chicagotribune.com/suburbs/lake-county-news-sun/news/ct-lns-gurnee-fire-update-st-0730-20160729-story.html

[4] Business Insider, "Tiny Flying Robots Are Being Built To Pollinate Crops Instead Of Real Bees," 2014. [Online]. Available: http://www.businessinsider.com/harvard-robobees-closer-to-pollinating-crops-2014-6

[5] R. a. Brooks and A. M. Flynn, "Fast, Cheap and Out of Control: a Robot Invasion of the Solar System," *Journal of The British Interplanetary Society*, vol. 42, pp. 478–485, 1989.

[6] Daily Mail, "Death from a swarm of tiny drones: U.S. Air Force releases terrifying video of tiny flybots that can can hover, stalk and even kill targets," 2013. [Online]. Available: http://www.dailymail.co.uk/news/article-2281403/U-S-Air-Force-developing-terrifying-swarms-tiny-unmanned-drones{%}5C-hover-crawl-kill-targets.html

[7] M. Özartan, S. Akgül, and B. Akça, "A different view to future use of unmanned aerial vehicles," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, may 2013, pp. 167–172.

[8] H. Syse, "Military Robots: Mapping the Moral Landscape," *Journal of Military Ethics*, vol. 14, no. 3-4, pp. 287–288, 2015.

[9] M. Buck, S. Razavi, R. Derose, T. Inoue, P. a. Silver, P. Subsoontorn, D. Endy, Y. Gerchman, C. H. Collins, F. H. Arnold, R. Weiss, M. C. Jensen, C. D. Smolke, L. Wroblewska, L. Prochazka, Y. Benenson, J. J. Tabor, C. a. Voigt, C. Lou, A. Tamsir, B. C. Stanton, M. Wieland, M. Fussenegger, I. Deese, N. Publishing, G. New, S. K. Lee, J. D. Keasling, a. P. Arkin, D. D. Vecchio, and W. Brattain, "Controlled Flight of a Biologically Inspired, Insect-Scale Robot," *Science*, no. May, pp. 603–607, 2013.

[10] S. B. Fuller, M. Karpelson, A. Censi, K. Y. Ma, and R. J. Wood, "Controlling free flight of a robotic fly using an onboard vision sensor inspired by insect ocelli." *Journal of the Royal Society, Interface / the Royal Society*, vol. 11, no. 97, 2014. [Online]. Available: http://rsif.royalsocietypublishing.org/content/11/97/20140281

[11] R. Wood, "The First Takeoff of a Biologically Inspired At-Scale Robotic Insect," *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 341–347, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4457871

[12] Z. E. Teoh, S. B. Fuller, P. Chirarattananon, N. O. Prez-Arancibia, J. D. Greenberg, and R. J. Wood, "A hovering flapping-wing microrobot with altitude control and passive upright stability," in *International Conference on Intelligent Robots and Systems (IROS)*, oct 2012, pp. 3209–3216.

[13] D. B. Doman, M. W. Oppenheimer, and D. Sigthorsson, "Dynamics and control of a minimally actuated biomimetic vehicle: Part I-aerodynamic model," *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, pp. 10–13, 2009.

[14] D. B. Doman, M. W. Oppenheimer, and D. O. Sigthorsson, "Dynamics and control of a minimally actuated biomimetic vehicle: Part II-control," in *Proceedings of AIAA Guidance Navigation Control Conference*, 2009, pp. 10–13.

[15] B. Perseghetti, J. Roll, and J. Gallagher, "Design Constraints of a Minimally Actuated Four Bar Linkage Flapping-Wing Micro Air Vehicle," in *Robot Intelligence Technology and Applications 2*, ser. Advances in Intelligent Systems and Computing, J.-H. Kim, E. T. Matson, H. Myung, P. Xu, and F. Karray, Eds. Springer International Publishing, 2014, vol. 274, pp. 545–555. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05582-4{_}47

[16] S. Boddhu, H. Botha, B. Perseghetti, and J. Gallagher, "Improved Control System for Analyzing and Validating Motion Controllers for Flapping Wing Vehicles," in *Robot Intelligence Technology and Applications 2*, ser. Advances in Intelligent Systems and Computing, J.-H. Kim, E. T. Matson, H. Myung, P. Xu, and F. Karray, Eds. Springer International Publishing, 2014, vol. 274, pp. 557–567. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05582-4{_}48

[17] H. Botha, S. Boddhu, H. McCurdy, J. Gallagher, E. Matson, and Y. Kim, "A Research Platform for Flapping Wing Micro Air Vehicle Control Study," in *Robot Intelligence Technology and Applications 3*, ser. Advances in Intelligent Systems and Computing, J.-H. Kim, W. Yang, J. Jo, P. Sincak, and H. Myung, Eds. Springer International Publishing, 2015, vol. 345, pp. 135–150. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16841-8{_}14

[18] J. C. Gallagher, D. B. Doman, and M. W. Oppenheimer, "The Technology of the Gaps: An Evolvable Hardware Synthesized Oscillator for the Control

of a Flapping-Wing Micro Air Vehicle," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 6, pp. 753–768, dec 2012.

[19] J. C. Gallagher and M. W. Oppenheimer, "An improved evolvable oscillator and basis function set for control of an insect-scale flapping-wing micro air vehicle," *Journal of Computer Science and Technology*, vol. 27, no. 5, pp. 966–978, 2012.

[20] J. Gallagher, L. Humphrey, and E. Matson, "Maintaining Model Consistency during In-Flight Adaptation in a Flapping-Wing Micro Air Vehicle," in *Robot Intelligence Technology and Applications 2*, ser. Advances in Intelligent Systems and Computing, J.-H. Kim, E. T. Matson, H. Myung, P. Xu, and F. Karray, Eds. Springer International Publishing, 2014, vol. 274, pp. 517–530. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05582-4{_}45

[21] G. Greenwood and A. M. Tyrrell, *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems.* Wiley-IEEE Press, 2006.

[22] E. Shimada, J. Thompson, J. Yan, R. Wood, and R. Fearing, "Prototyping millirobots using dextrous microassembly and folding," in *Symposium on Microrobotics ASME Int. Mechanical Engineering Cong. and Exp*, 2000, pp. 1–8. [Online]. Available: http://robotics.eecs.berkeley.edu/{~}ronf/PAPERS/mumanip.pdf

[23] J. Yan, R. Wood, S. Avadhanula, M. Sitti, and R. Fearing, "Towards flapping wing control for a micromechanical flying insect," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 4, 2001, pp. 3901–3908. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=933225

[24] R. J. J. Wood and R. S. S. Fearing, "Flight Force Measurements for a Micromechanical Flying Insect," *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*, pp. 355–362, 2001. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=973383

[25] J. Yan, S. Ayadhanula, and M. Sitti, "Thorax Design and Wing Control for a Micromechanical Flying Insect," Tech. Rep. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract{%}5Cnhttp://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Thorax+Design+and+Wing+Control+for+a+Micromechanical+Flying+Insect{#}6{%}5Cnhttp://scholar.google.com/scholar?hl=en{&}btnG=Search

[26] R. Stevenson, K. Corbo, L. Baca, and Q. Le, "Cage size and flight speed of the tobacco hawkmoth Manduca sexta," *The Journal of experimental biology*, vol. 198, no. Pt 8, pp. 1665–72, 1995. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/9319572

[27] S. N. Fry, "The Aerodynamics of Free-Flight Maneuvers in Drosophila," *Science*, vol. 300, no. 5618, pp. 495–498, 2003. [Online]. Available: http://www.sciencemag.org/cgi/doi/10.1126/science.1081944

[28] M. H. Dickinson and K. G. Götz, "The wake dynamics and flight forces of the fruit fly Drosophila melanogaster," *The Journal of experimental biology*, vol. 199, no. Pt 9, pp. 2085–2104, 1996.

[29] J. M. Birch and M. H. Dickinson, "Spanwise flow and the attachment of the leading-edge vortex on insect wings." *Nature*, vol. 412, no. 6848, pp. 729–733, 2001. [Online]. Available: http://www.nature.com/nature/journal/v412/n6848/full/412729a0.html

[30] S. Sunada and C. P. Ellington, "A new method for explaining the generation of aerodynamic forces in flapping flight," *Mathematical Methods in the Applied Sciences*, vol. 24, no. 17-18, pp. 1377–1386, 2001.

[31] N. V. Hoffer, C. Coopmans, R. R. Fullmer, and Y. Chen, "Small low-cost unmanned aerial vehicle System identification by Error Filtering Online Learning (EFOL) enhanced least squares method," in *International Conference on Unmanned Aircraft Systems, ICUAS 2015*, 2015, pp. 1355–1363.

[32] G. K. Taylor and A. L. R. Thomas, "Dynamic flight stability in the desert locust Schistocerca gregaria." *The Journal of experimental biology*, vol. 206, no. Pt 16, pp. 2803–2829, 2003.

[33] B. M. Finio, N. O. Pérez-Arancibia, and R. J. Wood, "System identification and linear time-invariant modeling of an insect-sized flapping-wing micro air vehicle," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1107–1114, 2011.

[34] P. Chirarattananon, K. Y. Ma, and R. J. Wood, "Adaptive control of a millimeter-scale flapping-wing robot," *Bioinspiration & Biomimetics*, vol. 9, no. 2, 2014.

[35] M. H. Rosen, G. Pivain, R. Sahai, N. T. Jafferis, and R. J. Wood, "Development of a 3.2g Untethered Flapping-Wing Platform for Flight Energetics and Control Experiments," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 3227–3233.

[36] V. M. Mwongera, "A Review of Flapping Wing MAV Modelling," *International Journal of Aeronautical Science & Aerospace Research (IJASAR)*, vol. 2, no. 2, pp. 17–26, 2015. [Online]. Available: http://scidoc.org/IJASAR-2470-4415-02-201.php

[37] R. Wood, E. Steltz, and R. Fearing, "Optimal energy density piezoelectric bending actuators," *Sensors and Actuators*, vol. 119, no. October 2004, pp. 476–488, 2005.

[38] M. Karpelson, G. Y. Wei, and R. J. Wood, "Driving high voltage piezoelectric actuators in microrobotic applications," *Sensors and Actuators, A: Physical*, vol. 176, pp. 78–89, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.sna.2011.11.035

[39] L. Hines, D. Colmenares, and M. Sitti, "Platform design and tethered flight of a motor-driven flapping-wing system," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 5838–5845. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7140016

[40] N. T. Jafferis, M. A. Graule, and R. J. Wood, "Non-linear resonance modeling and system design improvements for underactuated flapping-wing vehicles," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 3234–3241.

[41] G. K. Lau, Y. W. Chin, J. T. W. Goh, and R. J. Wood, "Dipteran-Insect-Inspired Thoracic Mechanism With Nonlinear Stiffness to Save Inertial Power of Flapping-Wing Flight," *IEEE Transactions on Robotics*, vol. 30, no. 5, pp. 1187–1197, 2014.

[42] H. V. Phan and H. C. Park, "Remotely controlled flight of an insect-like tailless Flapping-wing Micro Air Vehicle," in *12th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, oct 2015, pp. 315–317.

[43] ECliPSE, "The CyberPhysical Systems Engineering Group," 2016. [Online]. Available: http://cps.wright.edu/

[44] TelIt, "Jupiter SE880 GPS module," 2016. [Online]. Available: http://www.sequoia.co.uk/telit-jupiter-se880-embedded-gps-module{_}p{%_}1761.php

[45] Inven Sense, "MPU-9250," 2016. [Online]. Available: https://www.invensense.com/products/motion-tracking/9-axis/mpu-9250/

[46] A. Jensen, C. Coopmans, and Y. Chen, "Basics and guidelines of complementary filters for small UAS navigation," in *International Conference on Unmanned Aircraft Systems, ICUAS 2013 - Conference Proceedings*, no. August, 2013, pp. 500–507.

[47] Vectornav, "Embedded Navigation Solution," 2016. [Online]. Available: http://www.vectornav.com/support/library

[48] D. Simon, *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley, 2006. [Online]. Available: http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471708585.html

[49] J. D. Barton, "Fundamentals of Small Unmanned Aircraft Flight," *Johns Hopkins APL Technical Digest*, vol. 31, no. 2, pp. 132–149, 2012. [Online]. Available: http://www.jhuapl.edu/techdigest/TD/td3102/31{%}7B{_}{%}7D02-Barton.pdf

[50] C. De Wagter, S. Tijmons, B. D. W. Remes, and G. C. H. E. De Croon, "Autonomous flight of a 20-gram Flapping Wing MAV with a 4-gram onboard stereo vision system," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2014, pp. 4982–4987.

[51] A. J. Barry and R. Tedrake, "Pushbroom Stereo for High-Speed Navigation in Cluttered Environments," *3rd Workshop on Robots in Clutter: Perception and Interaction*, pp. 2–8, 2014. [Online]. Available: http://arxiv.org/abs/1407.7091

[52] M. Bibuli, M. Caccia, and L. Lapierre, "Monocular-SLAM–Based Navigation for Autonomous Micro Helicopters in GPS-Denied Environments," in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 7, no. 1, 2007, pp. 81–86.

[53] ——, "Autonomous, Vision-based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle," in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 7, no. 1, 2007, pp. 81–86.

[54] G. Greenwood, J. Gallagher, and E. Matson, *Cyber-Physical Systems: The Next Generation of Evolvable Hardware Research and Applications*. Springer International Publishing, 2015, pp. 285–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-13359-1{_}23

[55] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, mar 1986.

[56] F. Nagata, A. Otsuka, and K. Watanabe, "Network-based subsumption architecture for multiple mobile robots system," in *Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS)*, nov 2012, pp. 187–192.

[57] S. Kefi, I. Kallel, and A. M. Alimi, "Hybrid planning approaches for multirobot systems: A review and a proposal of a MultiAgent subsumption simulation," in *4th International Conference on Hybrid Intelligent Systems (HIS)*, dec 2014, pp. 285–290.

[58] J. Siekmann, J. Hartmanis, and J. V. Leeuwen, *Lecture Notes in Artificial Intelligence*, 1814.

[59] J. Han, C. H. Wang, and G. X. Yi, "Cooperative control of UAV based on Multi-Agent System," in *Proceedings of the IEEE 8th Conference on Industrial Electronics and Applications, ICIEA*, 2013, pp. 96–101.

[60] M. A. Kovacinal, D. Palmer, G. Yang, and R. Vaidyanathan, "Multi-Agent Control Algorithms for Chemical Cloud Detection and Mapping Using Unmanned Air Vehicles," in *Prcceedings of the IEEURSJ Intl. Conference on Intelligent Robots and Systems*, no. October, 2002, pp. 2782–2788.

[61] T. Shima, S. Rasmussen, and A. Sparks, "UAV cooperative multiple task assignments using genetic algorithms," *Proceedings of the American Control Conference*, pp. 2989–2994, 2005.

[62] X. Chen, W. He, and Z. Wu, "Research of UAV's multiple routes planning based on Multi-Agent Particle Swarm Optimization," in *Proceedings of the International Conference on Intelligent Control and Information Processing, ICICIP*, 2013, pp. 765–769.

[63] X. C. Ding, A. R. Rahmani, and M. Egerstedt, "Multi-UAV convoy protection: An optimal approach to path planning and coordination," *IEEE Transactions on Robotics*, vol. 26, no. 2, pp. 256–268, 2010.

[64] M. Abdelkader, M. Shaqura, M. Ghommem, N. Collier, V. Calo, and C. Claudel, "Optimal multi-agent path planning for fast inverse modeling in UAV-based flood sensing applications," in *Proceedings of the International Conference on Unmanned Aircraft Systems, ICUAS*, 2014, pp. 64–71.

[65] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," pp. 674–679, 1981. [Online]. Available: http://www.ri.cmu.edu/pub{_}files/pub3/lucas{_}bruce{_}d{_}1981{_}1/lucas{_}bruce{_}d{_}1981{_}1.ps.gz

[66] H. Bay and A. Ess, "Speeded-Up Robust Features (SURF)," Tech. Rep. September, 2008.

[67] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *In VISAPP International Conference on Computer Vision Theory and Applications*, 2009, pp. 331–340.

[68] M. A. Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, jun 1981. [Online]. Available: http://doi.acm.org/10.1145/358669.358692

[69] L. Sean, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Ballan, "MASON: A Multi-Agent Simulation Environment," in *Transactions of the society for Modeling and Simulation International*, vol. 7, no. 82, 2005, pp. 517–527.

[70] M. Podhradsky, G. Greenwood, J. Gallagher, and E. Matson, "A Multi-Agent System for Autonomous Adaptive Control of a Flapping-Wing Micro Air Vehicle," in *IEEE Symposium Series on Computational Intelligence*, dec 2015, pp. 1073–1080.

[71] M. North, N. Collier, and J. Vos, "Experiences Creating Three Implementations of the REPAST Agent Modeling Toolkit," *ACM Transactions on Modeling and Computer Simulation*, vol. 16, no. 1, pp. 1–25, 2006.

[72] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm," *Journal of Global Optimization*, vol. 39, no. 3, pp. 459–471, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10898-007-9149-x

[73] K. Uosaki and T. Hatanaka, "Evolution Strategies Based Particle Filters for Fault Detection," in *IEEE Symposium on Computational Intelligence in Image and Signal Processing, CIISP*, apr 2007, pp. 58–65.

Appendix A

Mechanical design

Mechanical drawings of our FWMAV are shown in Figures A.1 – A.6. Pictures of the assembled vehicle are shown in Figures A.8 and A.9, showing top and side view of the vehicle. Figure A.10 shows the sealed and marked shaft, which was a necessary measure to prevent the shaft from slipping under higher loads (such as during change from δ to $-\delta$). Figure A.11 shows the correct initial position of the wings that is necessary because there is no absolute position sensor of the motor angle. The user has to place the wings into the initial position at the beginning of each experiment.

The first iteration of the vehicle was built at Wright State University (WSU). The second (modified) version was build at Portland State University (PSU). Properties of both versions are compared in Table A.1. The second version is lighter and more agile. It can follow the trajectory faster than the first version. The custom PCB boards for power distribution and control could be further improved in future iterations.

Both versions are using two brushless DC motors Faulhaber series 1028_B, product code 1028S006BIEM3-1024. The motor is 10 mm in diameter, 6.0 V coil and 1.2 mm diameter output shaft. The mechanical drawing of the motor is shown in Figure A.7. The motor assembly includes series IEM3, integrated 3 channel magnetic incremental encoder with 1024 Counts-Per-Revolution (CPR) resolution. The motor uses planetary gearhead 10/1 with 4:1 ratio.

Property	1st iteration	2nd iteration	Comments
Weight	406 g (14.3 oz)	180 g (6.3 oz)	55% decrease
Diameter	TBD cm (TBD in)	TBD cm (TBD in)	30% decrease
Current cons.	$\leq$ 1A	At max speed of	40 rad/s (6.2 Hz)
Peak cur. cons.	3 A	Immediately	after start
Battery	2 cell 3Ah	2 cell 300mAh	3hrs $\rightarrow$ 20 min flight

Table A.1: Electrical and Mechanical Characteristics of the vehicle, for both 1st and 2nd iteration

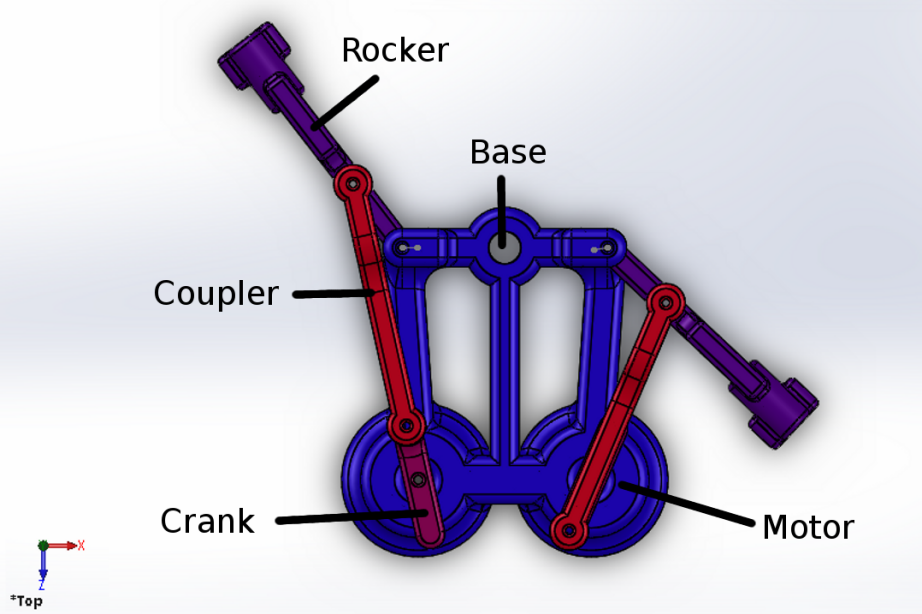

Figure A.1: Top view of the actuator assembly (actual size)

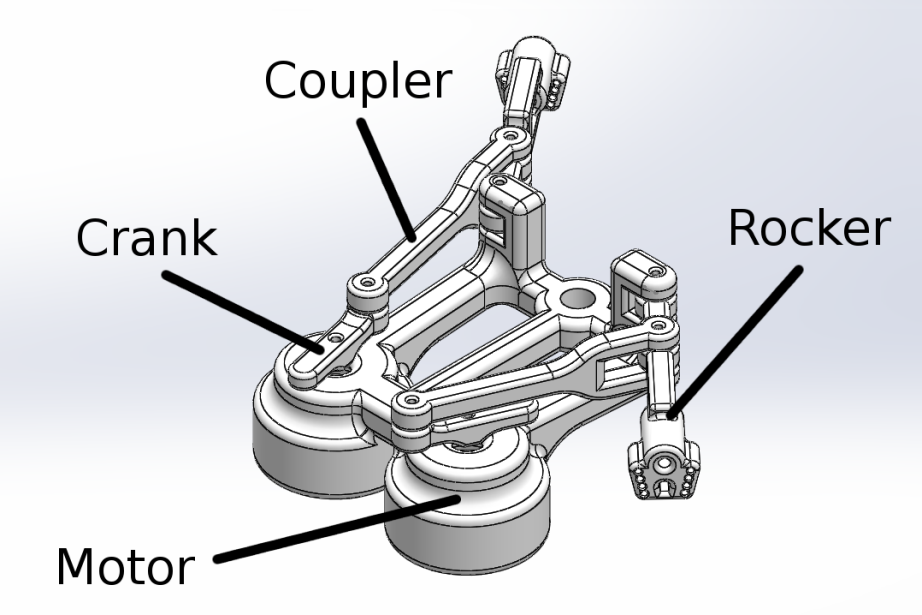

Figure A.2: Isometric view view of the actuator assembly (actual size)

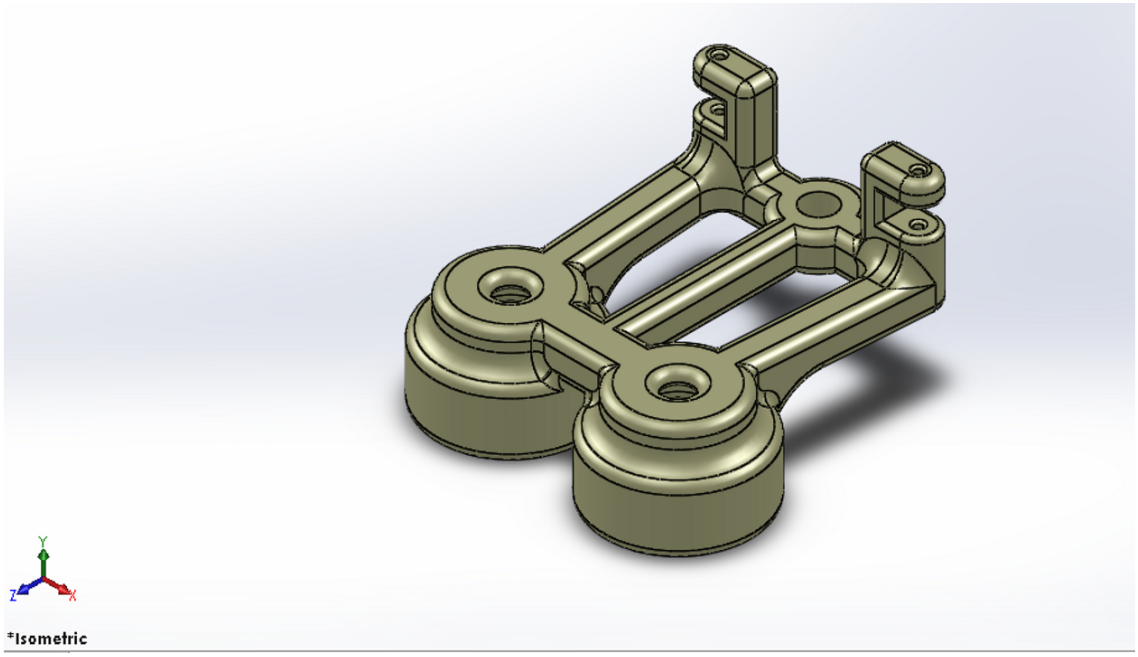

Figure A.3: A base (isometric view, not to scale)

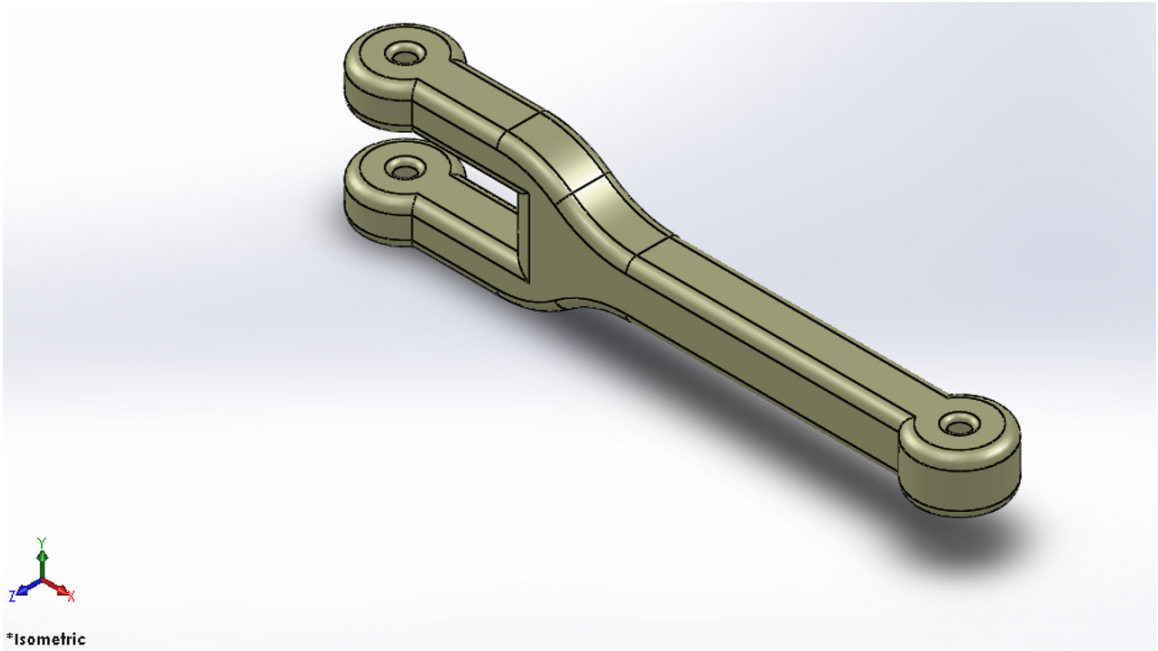

Figure A.4: A coupler (isometric view, not to scale)

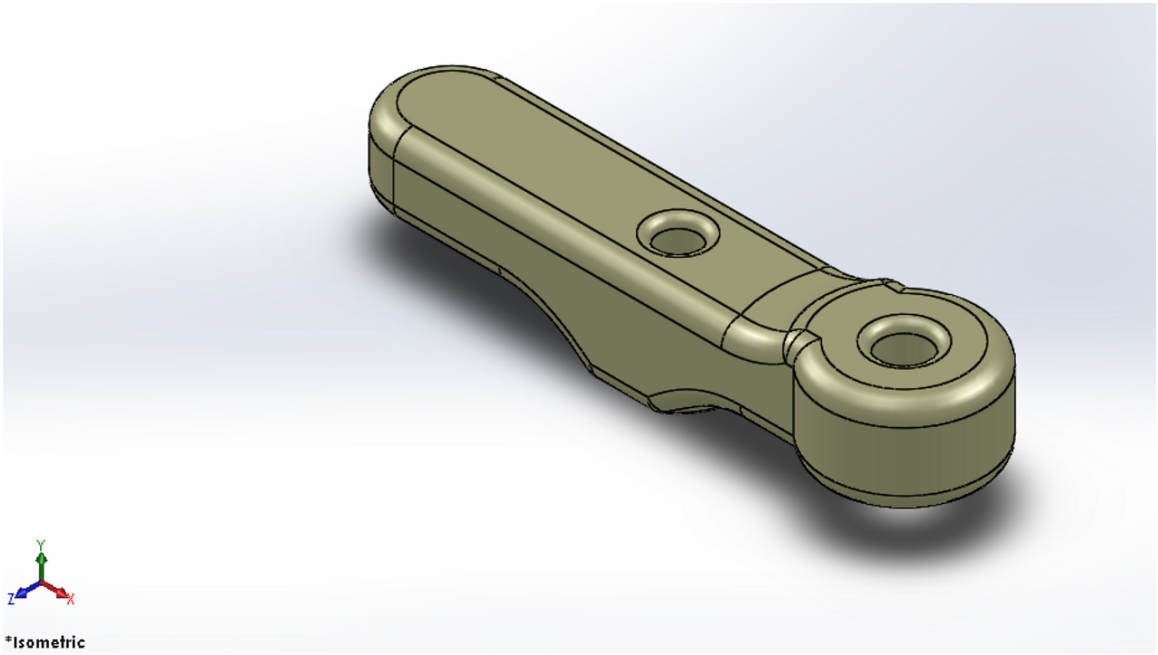

Figure A.5: A crank (isometric view, not to scale)

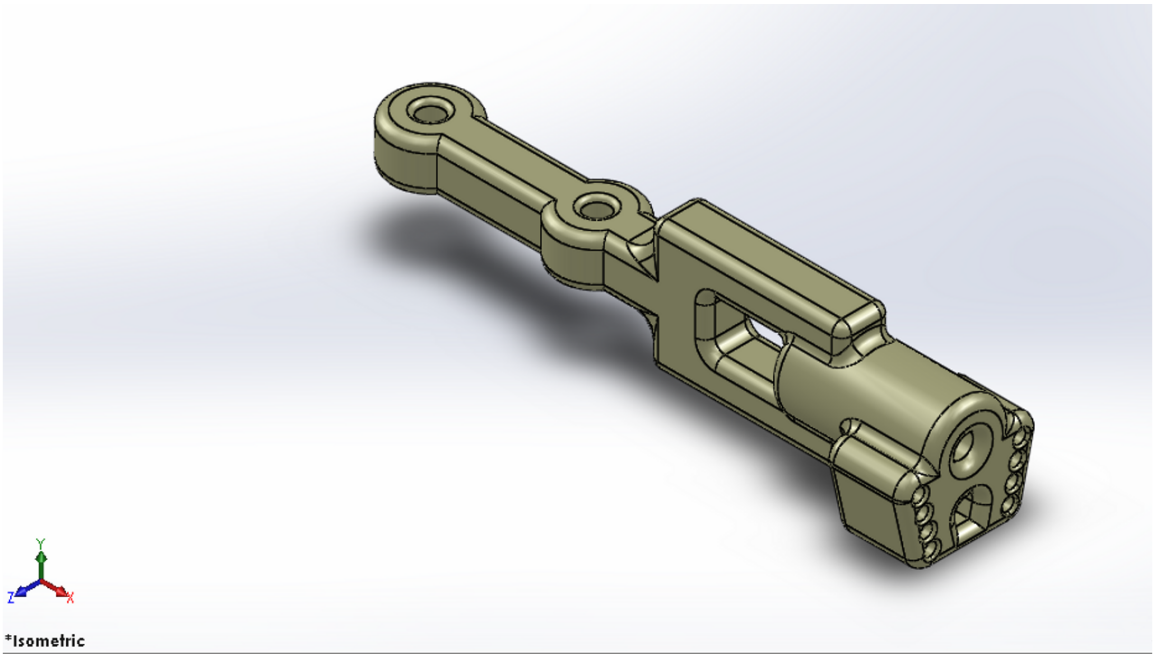

Figure A.6: A rocker (isometric view, not to scale)

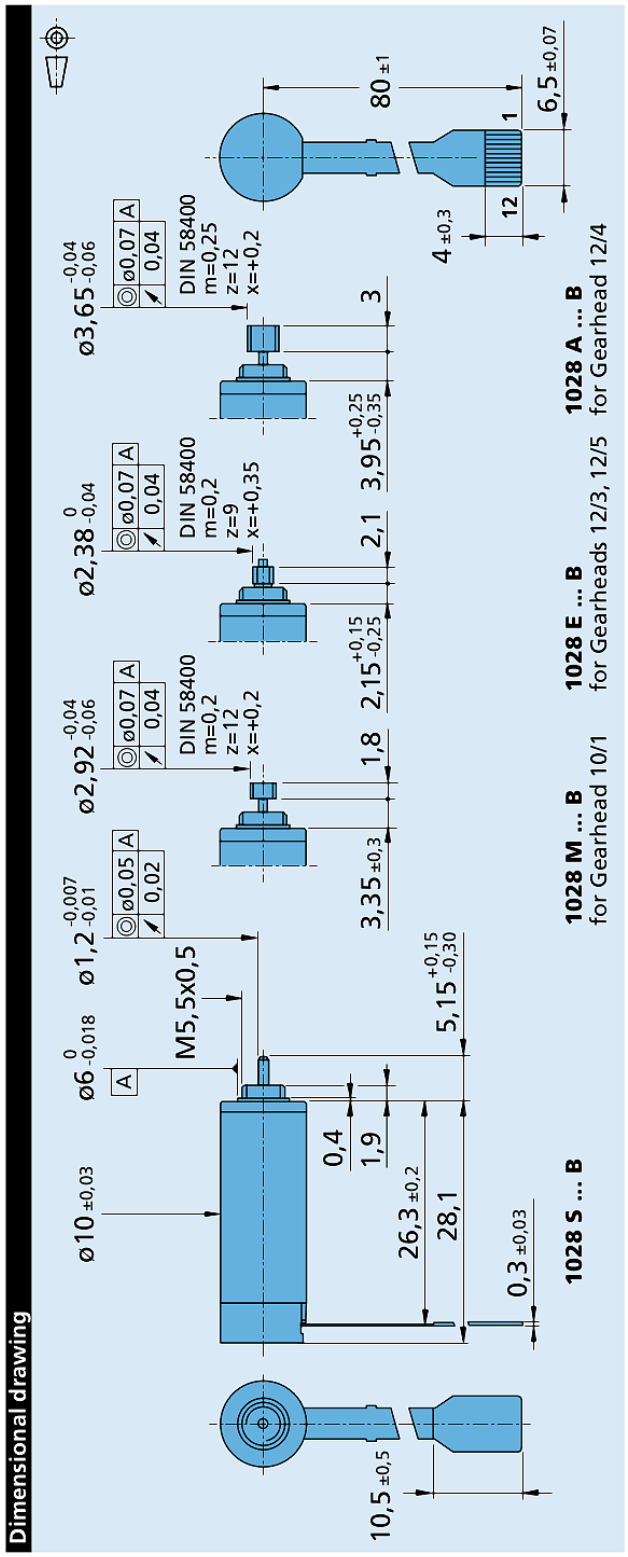

Figure A.7: Faulhaber series 1028_B brushless DC motor, product code 1028S006BIEM3-1024; The motor can be ordered at `http://www.micromo.com/1028s006biem3-1024.html`

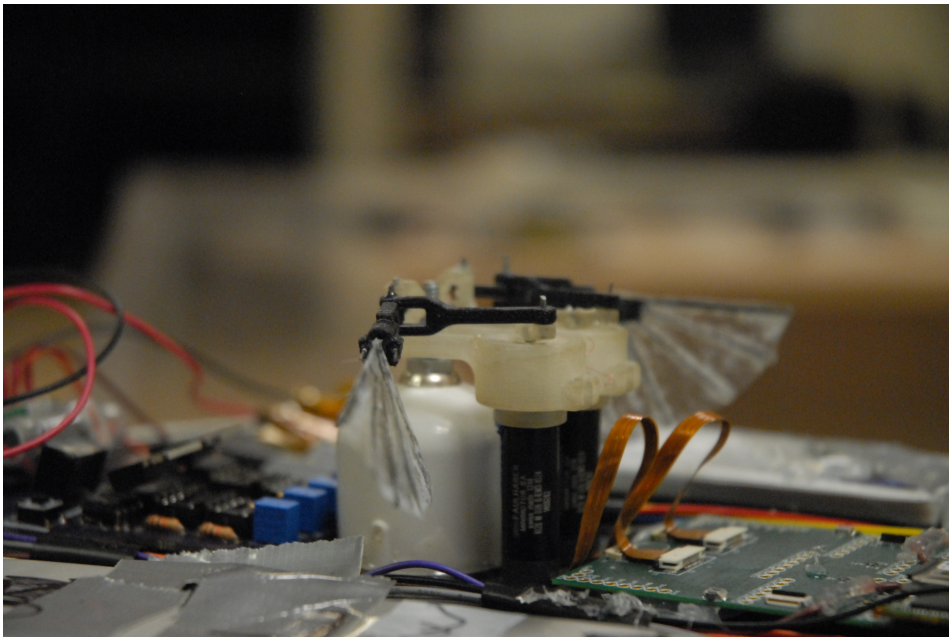

Figure A.8: Side view.

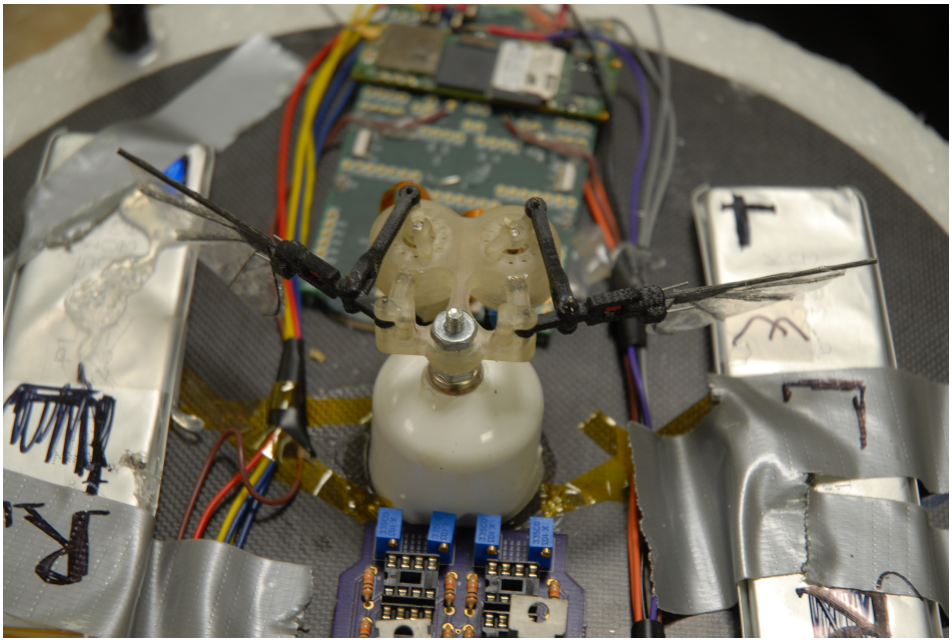

Figure A.9: Top view.

Figure A.10: Sealed shaft with markers to prevent shaft slippage

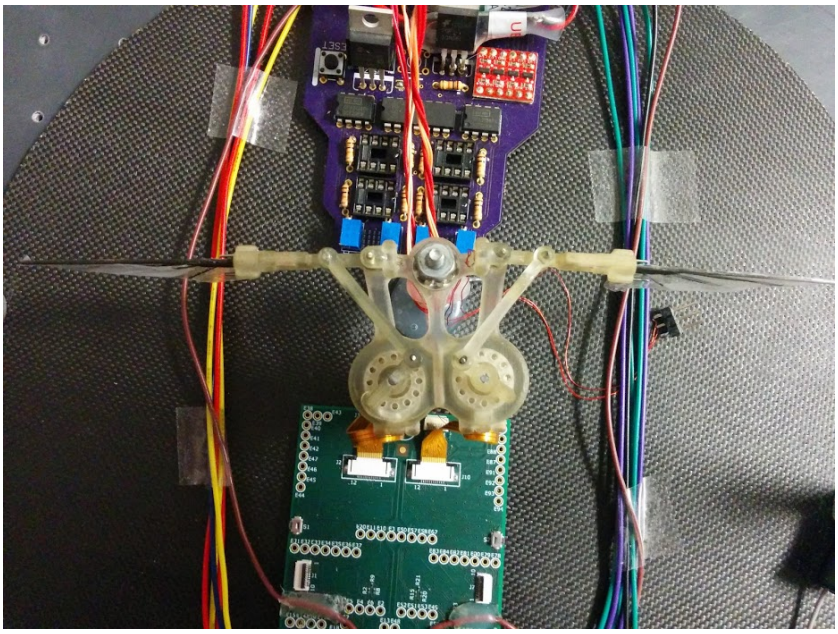

Figure A.11: Starting position of the wings

Appendix B

Software Architecture

The code is split into four main layers. Currently different programs contain separate layers, but in the future all layers can be used in one embedded program while keeping the functionality. The code has these layers:

1. Motor firmware – directly controls the motors by driving FET transistors
2. Multi Agent System – controls the vehicle (see Section 4
3. Pose estimation System – captures raw image from a video camera and applies clever image processing algorithms on it to find out the vehicle's pose (see Section 3.3)
4. Graphical User Interface (GUI) – to let user command (start, stop, restart) the vehicle and to show telemetry from the vehicle

Each layer is further described below.

B.1 Motor Firmware

This is the low level code that directly gives commands to the motors and reads the outputs of the encoders. This code runs on PIC micro controllers (MCU) – two PICs for each motor – and typically the user don't interact with it. The firmware was developed and is maintained by Hermanus Botha[1].

B.2 Multi Agent System

Is the implemented MAS from Chapter 4. It determines desired control inputs δ and ω based on the state of the vehicle and its pose. It uses two sub-layers: *Application Programming Interface (API)* to connect with the motor firmware and *UDP communication* to connect with the Pose Estimation System.

[1] wkjid10t@gmail.com

B.2.1 API

The API is running on the embedded Linux on Gumstix Overo, and is setting wing beating frequency and delta values. API gives commands to the PICs over SPI, the PICs then translate these commands into signals for the motors. The API, developed by Hermanus Botha, consists of several functions that can be called from user application. The most important ones are:

- *set_frequency()*, *set_delta()* which sets ω, δ for corresponding wings
- *init()* initializes the motors before they can be used
- *close()* closes the communication with motors in an orderly manner

B.2.2 UDP Communication

The highest layer of communication takes place over WiFi network, and can be fully defined by the user. It requires a server (typically running on a laptop), and a client program (running on the robot). The server waits for an incoming connection from a client, and once it is established it keeps communication with the client alive. The clients tries to connect with the server, and if successful it initializes the wings and waits for commands from the server. In the most basic example there are two UDP packets:

- Server $\rightarrow$ Client: four float numbers with desired values of $\delta_L, \delta_R, \omega_L, \omega_R$ in rad/s
- Client $\rightarrow$ Server: four float numbers with actual[2] values of $\delta_L, \delta_R, \omega_L, \omega_R$ in rad/s

The desired values of δ, ω can be passed to the server either from a user application running on the same machine (typically a laptop), or directly from the user - for example from the console.

B.3 Pose Estimation System

This system is written in C++ and uses OpenCV library for image processing. The algorithm is in detail described in Section 3.3. Although developed separately (mostly for the ease of testing), it is now integrated into the GUI. That way the user can easily control the pose estimation system (for example start/stop recording the video), and the whole program is more compact (no need to pass data between two separate processes).

B.4 Graphical User Interface

Currently we have two user programs that can communicate with the vehicle. One is a sample server with console, so the user can set the δ, ω values manually by typing

[2] i.e. the values that API gets from PICs

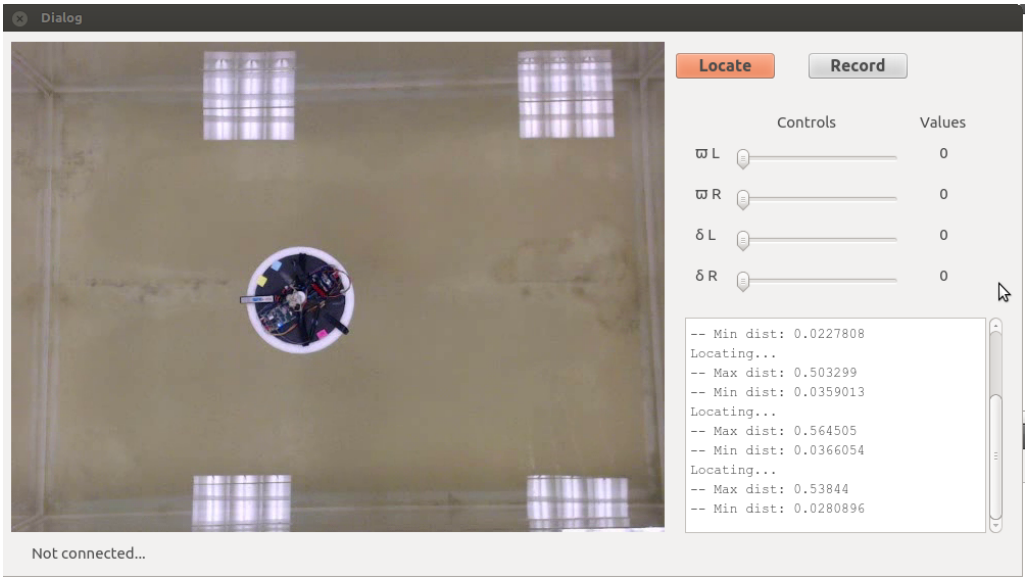

Figure B.1: Qt GUI with camera feed on the left, user controls in the top right and console on the bottom right.

commands. The other program is build using Qt framework, and allows the user to set the desired values using sliders (in the manual mode), or to control the MAS and the pose estimation system (in the autonomous mode).. This Qt application also reads data from the camera, localizes the vehicle in the watertank and will run the basic evolution algorithms to determine the basic movements. The GUI is shown in Figure B.1.

B.5 Application Notes

One might ask why we are not using Matlab, since it makes developing learning algorithms very simple. The main problem is that server example is running in Linux (Ubuntu 12.04) and Matlab available at the university doesn't support using webcams in Linux [3]. We are using Qt GUI.

[3] `http://unix.stackexchange.com/questions/\87001/connection-error-for-linux\-webcam-driver-for-matlab`

B.5.1 Initialization procedure

The procedure to initialize the vehicle needs to be done before the experiments can be conducted, and goes as follows:

1. Turn on the WiFi router
2. Connect laptop to WiFi
3. Turn on the vehicle
4. Establish SSH connection
5. Loop through:
   (a) Reset PICs using reset switch (vehicle)
   (b) Align wings to their apexes (vehicle)
   (c) Start the server (laptop)
   (d) Start the client (vehicle)
   (e) Wait for initialization (vehicle)
   (f) Send commands (laptop)
   (g) When done close the server
6. Disconnect everything

Steps 1 – 4 are needed only once per session, steps 5 – 6 have to be done before each new measurement.

Appendix C

Supplementary Material

Supplementary material such as videos, datasets and the code is available at `http://podhrmic.github.io/`