

Automatic Design of Controllers for Miniature Vehicles through Automatic Modelling

RENZO DE NARDI

A thesis submitted for the degree of Ph.D. in Computer Science

School of Computer Science and Electronic Engineering

University of Essex

September 2010

Contents

1	Introduction	1
1.1	Motivations	2
1.2	Approach	4
1.3	Research Objectives	7
1.4	Thesis Outline	8
1.4.1	Publications	9
2	Background and Related Work	11
2.1	The Automatic Control Task	11
2.1.1	Transferability	13
2.1.2	Incremental Evolution	15
2.1.3	Modularity	17
2.1.4	Car Control	19
2.1.5	Single Rotor Helicopter Control	20
2.1.6	Quadrotor Helicopter Control	22
2.2	The Modelling Task	24
2.2.1	Model Representation	25
2.2.2	Model Inputs	34
2.2.3	Excitation Signals and Data Collection	35
2.2.4	Model Estimation	39
2.2.5	Model Validation	46
2.3	Previous Work in Vehicles Modelling	50
2.4	Summary	53

3 Platforms and Data Acquisition	55
3.1 Flying Arena and Motion Capture System	55
3.1.1 Reference Frames	58
3.1.2 Accuracy	60
3.1.3 Delay Estimation and Compensation	61
3.2 Data Preprocessing	62
3.2.1 Differentiation Noise	65
3.3 The Platforms	66
3.3.1 The X3D Quadrotor	66
3.3.2 The Owl Quadrotor	73
3.3.3 The Toy Car	76
3.3.4 The ATTAS Aircraft	80
3.3.5 The Autopilot Helicopter Simulator	84
3.4 Summary	87
4 Non-Evolutionary techniques	89
4.1 Parameterised First Principles Models	89
4.1.1 ATTAS Lateral-Directional Model	91
4.1.2 Quadrotor 6DoF Model	100
4.2 Black-Box Models	110
4.2.1 Nearest Neighbour (NN)	112
4.2.2 Multi Layer Perceptrons Neural Networks (MLP)	123
4.3 Discussion	142
4.4 Summary	147
5 Coevolutionary Modelling	149
5.1 Fundamental Considerations	149
5.2 The Idea	150
5.3 The Algorithm	158
5.3.1 Design	158
5.3.2 Coevolutionary Loop	164
5.3.3 Evolution of the Models	167
5.3.4 Evolution of the Tests	175

5.4	Analysis of a Typical Coevolution Run	178
5.5	Automatic Modelling of the ATTAS Aircraft	184
5.5.1	Convergence	187
5.5.2	Performance Analysis	190
5.5.3	Analysis of the Equations	195
5.5.4	Conclusion	197
5.6	Automatic Modelling of the Toy Car	198
5.6.1	Convergence	198
5.6.2	Performance Analysis	202
5.6.3	Analysis of the Equations	209
5.6.4	Conclusions	212
5.7	Automatic Modelling of the X3D Quadrotor	213
5.7.1	Convergence	213
5.7.2	Performance Analysis	217
5.7.3	Analysis of the Equations	224
5.7.4	Conclusions	231
5.8	Automatic Modelling of the Owl Quadrotor	232
5.8.1	Convergence	232
5.8.2	Performance Analysis	235
5.8.3	Analysis of the Equations	242
5.8.4	Conclusions	248
5.9	Further Algorithm Investigation	249
5.9.1	Effect of Models' Population Size	249
5.9.2	Effect of Length of Training Windows	253
5.10	Summary	260
6	Beyond Deterministic Modelling	263
6.1	A Stochastic Model	264
6.2	The Algorithm	268
6.3	Optimization of Parameters	272
6.4	Experiments	273
6.4.1	Toy Car	274

6.4.2	X3D	277
6.5	Summary	280
7	Automatic controller design	283
7.1	The Toy Car	284
7.1.1	Task	284
7.1.2	Controllers	287
7.1.3	Evolutionary Training	292
7.1.4	Training and Testing	298
7.1.5	Real Car Testing	306
7.1.6	Different Types of Tasks	322
7.2	The X3D Quadrotor	327
7.2.1	Task	327
7.2.2	Controllers	330
7.2.3	Training and Testing	333
7.2.4	Real X3D Testing	338
7.3	The Autopilot Helicopter Simulator	344
7.3.1	Task and Fitness	345
7.3.2	Controllers	347
7.3.3	Controller Training	350
7.3.4	A Partial Overview of Preliminary Non-Working Approaches	350
7.3.5	First Working Approach: Incrementally Substituting a PID with Neural Networks	353
7.3.6	Second Working Approach: Incremental/Simultaneous Evolution of Modular Networks	354
7.3.7	Performance Analysis	356
7.3.8	Considerations	360
7.4	Summary	362
8	Conclusions and Future Work	365
8.1	Research Achievements	365
8.2	Methodological Critique	373
8.3	Future Work	376

A Derivations of the Parametric Quadrotor Model	379
A.1 Rotor Forces	380
A.1.1 Blade Parameters	382
A.2 Forces and Moments	384
A.3 Propulsion Group Model	387
A.4 Stability Augmentation System	392
A.5 Full Model	394
A.6 Model Parameters	396
B Algorithms and Definitions	397
B.1 List of Symbols	397
B.2 Mathematical Definitions	398
B.3 Parameters	400
B.4 Algorithms	400
B.4.1 Numerical Differentiation	400
B.4.2 Maximum Likelihood Estimation	401

Acknowledgements

I would first and foremost like to thank my supervisor Owen E. Holland. Without his continuous guidance, encouragement and outlook this work simply would have not been possible.

I would like also to thank the people whom I have collaborated with during these years and who have influenced me as a researcher. These are Julian Togelius, Richard Newcombe, Hugo Gravato Marques and Alberto Moraglio.

I am grateful to Simon Julier and Steve Hailes at UCL for their understanding and for allowing me the time to complete the writing of this thesis.

I am thankful to my family and all my friends for being supportive and understanding during these years. A warm thanks goes to Fabiola, Aino and Georgia for being so caring. As always I am grateful to my parents Annamaria and Valeriano for their unconditional love and continuous encouragement.

Finally, I am grateful for the scholarship from the School of Computer Science and Electronic Engineering, and for financial support from Swarm Systems Ltd.

Abstract

This thesis investigates the problem of automatically designing controllers for vehicles that can be represented as a rigid body. The approach is based on the idea of automatically obtaining a dynamic model of the system of interest, and using it to design controllers automatically. A novel aspect of our approach is that of not requiring any form of platform specific knowledge, and being as a consequence both hands-off and very generic.

The acquisition of models is based on data logged when a human pilot was controlling the vehicle, and is carried out by an evolutionary algorithm based on competitive coevolution. Models in the form of symbolic expressions are coevolved along with the portions of the training data that are used to compute their fitness. This results in an effective and computationally efficient way of constructing models.

The modelling method is applied to a small toy car, a full sized aeroplane and two different types of small quadrotor helicopters. For comparison, models of the same vehicles are also derived using standard modelling techniques that exploit platform knowledge. The models produced by our technique are shown to be as accurate or better than those produced manually. Importantly after a limited amount of rearrangement of terms, the models also prove to be interpretable.

A method is presented for reproducing in the models the noise and uncertainties that characterize real world platforms. The evolved deterministic models produced are augmented with a simple yet computationally efficient Gaussian noise model, and a principled method based on unscented Kalman filtering is used to estimate the noise parameters. The augmented models are demonstrated to reproduce most of the variability shown by real vehicles.

The automatic design of controllers considers both monolithic and modular structures based on recurrent neural networks. Conventional steady state evolution is used to evolve monolithic controllers, and cooperative coevolution is applied to modular controllers. Manually designed controllers are also developed for purposes of comparison. Controllers are mainly evolved for path-following tasks, but other tasks like imitating game players' abilities are also considered.

In general monolithic controllers are shown to be very effective in controlling the toy car, but have limitations when applied to the helicopters. Modular networks show a better ability to scale to more demanding platforms, and in simulation reach levels of performance comparable to or better than controllers designed manually.

Tests show that for both the toy car and quadrotor helicopters, the evolved controllers successfully transfer to the real vehicles, although a certain amount of mismatch exists between the performances predicted in simulation and those on the real platforms.

Chapter 1

Introduction

One of the most challenging research questions in the field of robotics is how to provide systems with the ability to perform tasks entirely on their own, or in other words how to be autonomous. This is such a broad question that it is fair to say that most of the research in robotics deals with at least some aspects of autonomy. Such important topics as localization and mapping, vision and scene understanding, planning and control, learning and adaptation all have the aim of extending the boundaries of what a robot is capable of doing without human intervention.

In this thesis we deal with the topics of the modelling and control of robotic platforms, but while models and control algorithms are often developed by roboticists and control engineers as means of automatically performing a task, we instead consider the problem of autonomously learning such models and control algorithms, and doing so in a platform agnostic way, that is without using any platform knowledge at all.

Consider for example the situation in which a new platform has been built (e.g. a flying machine) and, the experts who built it have also devised a way to control such a machine as part of the design process. This is a reasonable assumption since producing a new platform is a cyclic process of design, test and refinement which demands some form of control; often this control is provided by a skilled human expert. This example includes all the vehicles and robots that were first built to be controlled by a human, that we would now like to make autonomous. In this setting, we can control the platform (at least to a certain extent) and in this way collect data on its behaviour in response to the control signals.

Importantly, we focus on platforms on which we cannot develop controllers directly

since any inappropriate control input could lead to unsafe operation or to damaging the platform; this is a common problem in most interesting real world systems.

Given this situation, we would like to use some automatic methodology to produce a description of the behaviour of the platform (i.e. a model) that matches the experimental data sufficiently accurately, and we would also like to synthesize a controller in order to carry out an arbitrary but feasible task, a task that is not necessarily something we can already do with the platform through the normal means of control.

The automatic modelling of systems and the automatic design of controllers are certainly not unexplored topics, but in this thesis we have chosen an unusual approach: we have deliberately decided to *avoid the use of any knowledge about the platform* and to aim at a platform independent methodology for developing both models and controllers.

Our central research questions is: how far can we get in automatically building models and controllers without using any platform specific knowledge?

This is a great starting point for an interesting, and potentially useful research journey.

1.1 Motivations

Our interest in automatic modelling and control sprang out of the work carried out in the School of Computer Science and Electronic Engineering in the domain of miniature unmanned aerial vehicles (UAV) [101, 57, 58].

In such a context it is often the case that small electric aerial platforms (both fixed and rotary wing) are readily available on the market in the form of ready to fly toy vehicles (e.g. [135, 78]) or components that can be used to assemble custom platforms (e.g. [156]). Batteries with higher energy density (i.e. Lithium-Polymer cells), cheaper sensors (i.e. MEMS¹ accelerometers and gyros) and more efficient electric motors (i.e. brushless motors) are the technologies that in the last few years have played a primary role in making such vehicles available.

Once retrofitted with the necessary sensors, computation and communication modules such vehicles become very interesting robotics platforms. In recent years, the availability of such inexpensive and safe platforms has increased the number of research endeavours that make use of miniature UAVs (e.g. [132, 149, 195, 86, 8]). This is in contrast to only a decade ago when the research was instead limited to purpose built and much larger,

¹Micro-electro-mechanical systems.

expensive and dangerous single rotor autonomous helicopters suitable only for outdoor operations (e.g. [152, 115]).

In COTS (commercial off-the-shelf) vehicles, cost competition ensures simplicity of design, but also limits the overall quality standard, and differences in flying qualities are therefore common even between vehicles of the same type. In the case of the model toy market, the design of these helicopters is often largely the result of the experimental work of a single committed designer (e.g. [166, 240, 167]), rather than of a conventional engineering effort; this means that no adequate dynamic model of these novel flying machines is generally available. Under these circumstances, it is clear that the methodologies of control system design traditionally used in the aircraft control community become very difficult to apply, or are simply unsuitable since new design concepts often require completely new control solutions.

With this premise, our interest in a methodology that automatically produces models and controllers for this type of platform is easily explained. Ideally such an approach would allow the production of models that are (at least to a certain extent) understandable, and controllers that can be customized to achieve different tasks. Changes in the platform's design or payload could then easily be handled simply by repeating the modelling and controller design processes, following which the form of the resulting models could give some insight into the nature of the changes in the platform.

Although the specific challenge of miniature autonomous rotorcraft is what first interested us in automatic control and modelling, our research has developed, and has applications beyond this domain.

The questions of modelling and controller design are crucial ones in robotics. For a robotic system, a highly desirable contribution towards autonomy is the ability to learn how to behave in its environment; this in turn entails both understanding and controlling its own body, and also making sense of its surroundings. The work in this thesis contributes to addressing the first of these two challenges for a well defined but general class of platforms.

In this thesis we will limit ourselves to experiments that are directly relevant to our goal, but we can easily foresee that our ideas could be part of a more complex autonomous system where they could provide a way of bootstrapping the learning process away from a set of safely executed manoeuvres towards more uncertain territory.

The motivations for the present work also derive from the engineering aspects of the

challenges of modelling and controller design. Modelling, or system identification as it is often defined in the control engineering field, is a topic of active research, since producing models that match the behaviour of real dynamic systems is both necessary and far from simple. It is therefore of obvious interest to study methods that can deal with non trivial systems distinguished by coupling and non-linearity like some of those we will consider.

Our results will be relevant in the context of system identification because we will make specific choices in order to produce models that are as far as possible transparent and interpretable. In addition we will also choose for our models the same state space formulation commonly used in the control system community for models based on conventional engineering principles. The models obtained should therefore have a value that goes beyond their use for automatic control design.

In the control domain, the evolutionary techniques that we will use do not require any prior knowledge of how to solve a task, and therefore they could be used to provide insight into the solution of novel tasks for which it is not obvious how to manually design a controller. Alternatively, even in the case of a control problem for which a solution is known, better results might be achievable from automatic design, and this could provide insight into how to improve existing controllers. Although we will not address this specific aspect in our work, other authors have shown this to be a viable methodology in the context of UAV controllers based on evolved artificial neural networks ([93]).

1.2 Approach

Let us now sketch the approach followed in this thesis. For the sake of clarity, we will not at this stage address in any depth the motivations that led to our choices; we ask the reader to bear with us until the next chapter when we will address the problems of modelling and control in more detail, and our choices will be supported and related to the literature in the field.

In answering our main question of how far we can get in designing controllers in a completely platform agnostic way, we take what is very much an experimental approach which starts with collecting data, and ends by testing the controllers on the real world platforms.

In this thesis we will limit ourselves to platforms that can be represented as a 6DoF

rigid body; we will use a small toy car (details in Section 3.3.3) two quadrotors (details in Sections 3.3.2 and 3.3.1) a full sized aircraft (details in Section 3.3.4) and a simulation of a single rotor helicopter. All of these platforms have two facts in common: they can be flown or driven to collect data about their dynamic behaviour; and, very importantly, they can be damaged and/or unsafe if driven by inappropriate control inputs (e.g. random inputs). For example in the case of a helicopter it is not generally a good idea to try out control manoeuvres at random, since almost inevitably this would lead to crashing the helicopter into the ground, or a wall, or would flip the machine upside down, a situation from which recovery may be impossible. In what follows we will call a machine with these characteristics *non-recoverable* using the definition proposed by Togelius ([231]). Non recoverability is a characteristic of many real world systems, and therefore a general methodology like ours aims to be, should take it into account.

The obvious consequence of this is that any control design methodology that requires testing the real platform with poorly defined control inputs, or with potentially poor controllers, cannot be applied in the present context. For this fundamental reason, we decided to tackle the problem of automatic controller design offline, through the use of a simulator, a methodology widely used in the evolutionary robotics community² ([243]). In general, evolution carried out in simulation is also considerably faster than evolution carried out on the real platform since a dynamic model can run at least few orders of magnitude faster than real time. It is also more practical since a simulated vehicle does not need the setup and maintenance required for a real vehicle.

Since the need to avoid the use of platform specific knowledge is fundamental to our approach, we must build the dynamic model of the platform automatically, relying solely on experimental data.

In the thesis the automatic design of models and the validation of the models obtained are approached as follows:

- coevolution and genetic programming are used to infer both the structure and the parameters of the models of the platforms under study;
- the obtained dynamic models are extended with a stochastic model to represent the variability that characterizes real world platforms;

²Benefits and shortcoming of evolutionary techniques are discussed in detail in section 5.2.

- to provide grounds for comparison we also produce (or attempt to produce³) models based on more conventional system identification methods;
- to demonstrate that the approach is platform independent, our modelling methodology is applied to various different platforms without any major changes in the algorithms.

Once we have a model of a system, we can focus on the control side of the problem. Again, our radical choice of not using platform knowledge prevents us from using standard control theory design techniques, since these require platform knowledge at the level of the structure, inputs and parameters of the controller.

While we deliberately avoid the use of platform knowledge, we do allow the use of some knowledge of the control task to be solved, in particular to develop well specified measures of the controller's performance and of the types of behaviour we want to obtain. We will also often adopt metrics used in control systems engineering. We believe this is an important step in obtaining controllers that are not only interesting⁴ but also useful to the point that they can be integrated with other system components designed using more traditional engineering methods.

In our work we will mostly address the automatic design of controllers tackling the problem of path following, which involves dealing with the problem of translating a medium level platform independent description of a task (i.e. a path to follow) into low level and very platform dependent control commands. However, we will also examine some examples of different types of tasks.

We approach the design of controllers and their validation as follows:

- we rely on evolutionary computation to automatically make the design choices that are usually made by an expert engineer i.e. different sizes of the platform state, different numbers of inputs, and very different dynamics;
- we make in depth comparisons with handcrafted controllers in order to understand advantages and shortcoming of our evolutionary approach;
- we repeatedly test the obtained controllers on the real platforms to study the accuracy

³Not all of our platforms are amenable to the standard techniques that we implemented.

⁴By interesting we mean able to solve a task for which an obvious solution does not exists or solve the task in a novel way.

and consistency of their performance, and the agreement with the results obtained in simulation.

1.3 Research Objectives

We have sketched the way we have planned our research, and we identified our main aim. However there are several objectives that we plan to fulfil in order to reach our target.

Following the order in which we will address them throughout the thesis, we aim at:

- proposing a generic model structure that can cope with 6DoF rigid body models with different levels of complexity, different numbers of inputs and both linear and nonlinear dynamics;
- devising a general training method for determining the structure and parameters of a model, a method that relies solely on data collected from the real platform and, more importantly, relies on training data that has not been examined and selected by a human expert;
- ideally obtaining performance at least as good as the models produced by experts, and by using more conventional identification techniques;
- generating models that are transparent and understandable by an expert engineer who could use them to gain insight into the dynamic behaviour of the platform;
- generating models that are physically consistent i.e. that produce meaningful accelerations and velocities making them usable in place of the equations devised by a human expert;
- encoding in the model the uncertainty usually present on the behaviour of real world systems;
- devising a way of automatically determining the uncertainty characteristics of the model from real data;
- proposing a generic controller structure (or set of structures) that can cope with the number of control inputs that a typical rigid body system could have;
- devising training techniques for controllers that can scale up to the complexity posed by different platforms and tasks;

- ideally obtaining controllers with performances as good or better than those produced by experts;
- generating controllers in simulation that can transfer to the real platforms and control them successfully ideally with the accuracy shown in simulation.

Obviously we might not be able to achieve all of these objectives to the extent we would like, but establishing the limits of the approach is indeed a vital and integral part of the research programme.

In all our work we have devoted a great deal of effort to comparing our results with other more conventional techniques, and reporting all the details needed to understand, make use of, and replicate our experiments. We believe that this ensures that many of the technical details explored in this work will be relevant to researchers and practitioners in the disciplines this work touches upon.

1.4 Thesis Outline

The work reported in this thesis naturally falls into two distinct parts, modelling of the chosen platform and automatic design of controllers based on the obtained models, before those:

In Chapter 2 we introduce the many aspects of the modelling and control tasks and bring into contest the most relevant results from the literature. In this chapter we also put in place the foundations that underpin the novel contributions presented in Chapters 4-7.

Chapter 3 describes the various platforms used in this work, the settings in which our experiments were conducted, and the ways in which the data were pre-processed. This chapter also discusses the many technical details that inevitably have to be addressed when working with real world platforms.

The first part deals with automatic modelling:

- Chapter 4 presents standard approaches to modelling, some of which are not in line with our minimal platform knowledge concept but which constitute a very useful point of comparison;
- Chapter 5 presents our coevolutionary approach, in which we apply very recent findings in the evolutionary computation field to the problem of modelling, showing both

their advantages and their shortcomings;

- Chapter 6 adopts a different perspective on the problem of modelling, shifting from a deterministic to a probabilistic paradigm;

The second part addresses the problem of automatically designing controllers based on the models built in chapters Chapter 5 and Chapter 6:

- Chapter 7 tackles the challenging problem of automatically designing controllers for two real platforms the car and quadrotor.

Chapter 8 sets out the conclusions and identifies the strong and weak points of the thesis, also identifies interesting directions that future research is likely to take.

1.4.1 Publications

Some of the work presented in this thesis has already appeared in refereed publications. The chapters in which this material appears are listed below; all of the publications can be downloaded from the author's web page⁵.

- Renzo De Nardi and Owen E. Holland *Coevolutionary modelling of a miniature rotorcraft*. Proceedings of the 10th Intelligent and Autonomous Systems Conference IAS10, 2008. Extensively reported in Chapter 5.
- Julian Togelius, Renzo De Nardi, Hugo Marques, Richard Newcombe, Simon M. Lucas and Owen Holland. *Nonlinear dynamics modelling for controller evolution*. Proceedings of Gecco 2007. Extensively reported in Chapter 5
- Renzo De Nardi, Julian Togelius, Owen E. Holland and Simon M. Lucas. *Evolution of Neural Networks for Helicopter Control: Why Modularity Matters*. Proceeding of the IEEE Congress on Evolutionary Computation, CEC06. Extensively reported in Chapter 7.
- Julian Togelius, Renzo De Nardi and Simon M. Lucas. *Towards automatic personalised content creation for racing games*. Proceedings of IEEE CIG 2007. Briefly reported in Chapter 7.

⁵<http://www.cs.ucl.ac.uk/staff/R.DeNardi/phd/>

- Julian Togelius, Renzo De Nardi and Simon M. Lucas. *Making racing fun through player modeling and track evolution.* Proceedings of the SAB Workshop on Adaptive Approaches to Optimizing Player Satisfaction, SAB06. Briefly reported in Chapter 7.

Other collaborative publications in related technical areas, but not directly involved in the main reported text, are listed below. These can all be downloaded from the author's web page.

- Renzo De Nardi and Owen Holland. *UltraSwarm: A Further Step Towards a Flock of Miniature Helicopters.* Proceedings of the SAB Workshop on Swarm Robotics, 2006.
- Renzo De Nardi, Owen Holland, John Woods, and Adrian Clark. *SwarMAV: A swarm of miniature aerial vehicles.* Proceedings of the 21st Bristol International UAV Systems Conference, 2006.
- Owen Holland, John Woods, Renzo De Nardi, Adrian Clark. *Beyond swarm intelligence: The UltraSwarm.* Proceedings of the IEEE Swarm Intelligence Symposium (SIS2005), 2005.
- Julian Togelius, Renzo De Nardi and Alberto Moraglio. Geometric PSO + GP = Particle Swarm Programming. Proceeding of the IEEE Congress on Evolutionary Computation (CEC08), 2008.
- Julian Togelius, Simon M. Lucas and Renzo De Nardi. *Computational Intelligence in Racing Games.* In Norio Baba *et al.* (eds.) Advanced Intelligent Paradigms in Computer Games, 2007.

Chapter 2

Background and Related Work

In the introduction we described the setting in which this work developed, and also identified our research targets. In presenting the targets, we had to be brief about the multifaceted nature of the problem that we are tackling both in the modelling and control domain; this chapter will provide much more detail.

We aim to provide a comprehensive background description of the processes of automatic modelling and automatic design of controllers, and provide references to the relevant literature. The novel contributions to be discussed in the following chapters will then fall naturally into the appropriate context of theory and practice.

Although in our experimental work we will first address the problem of modelling, in this chapter we start by looking at the problem of control, since this allows us to better situate our work in the wider context of current robotics research.

2.1 The Automatic Control Task

The Reinforcement Learning (RL) problem is defined as the problem of deciding which action a (virtual or real) agent should take in a given environmental state in order to maximize its long-term reward¹. This definition encompasses a variety of problems, including those of automatically designing a robot controller.

The most popular approaches used to solve RL problems are temporal difference (TD) learning and related algorithms ([218]), and evolutionary methods.

¹A more rigorous definition in terms of the optimal control of Markov decision processes can be found in [218].

Empirical evidence has shown that neuroevolution² can outperform TD methods in domains with large state and action spaces, especially if they involve sensor and/or actuator noise ([83, 225]). This is the situation for all of our platforms since their state is continuous (i.e., there are an infinite number of states) and they are distinguished by sensor and actuator noise, therefore in our work we decided to use evolutionary methods, and in particular we have focussed on neuroevolution. However founded on evidence from the published literature, our choice remains a pragmatic one, and it would be certainly interesting to see if TD learning and related techniques would yield similar results.

In our context, where avoiding the use of platform knowledge is of primary importance, evolution is a particularly suitable technique because in order to train controllers it requires only a simulation of the platform, and a fitness function; no knowledge of how the task should be solved is needed.

We can produce a simulation of the platform by using automatically produced models (see Section 2.2), and determining a fitness function is generally much easier than determining a solution to the problem. Given these two ingredients, an evolutionary algorithm can then be used to search the space of controllers allowed by the chosen representations. This is of course the methodology used in the field of evolutionary robotics ([175]), and our approach shares many of the tools and techniques commonly used in this discipline. However, here we focus only on the automatic engineering aspects of the approach; we are not interested in the open questions in biology concerning evolutionary, developmental, and brain dynamics that have been investigated in connection with evolutionary robotics ([91, 47]). We see evolution as an optimization technique that can be used to search for solutions to complex control problems, and which makes the most of the information present in the experimental data, potentially avoiding the biases that designers inevitably introduce when using more conventional optimization techniques.

Designing controllers using evolution is not devoid of problems and limitations, and in the following sections we will discuss those directly related to our work: transferability, incremental evolution and modularity. In the discussion we will tend to focus on controllers based on artificial neural networks since in the experiments reported in Chapter 7 we will primarily use such representations.

²Neuroevolution refers to the practice of evolving the weights or the weights and structure of artificial neural networks.

2.1.1 Transferability

Since we are evolving our controllers in simulation, we have to ensure that they will transfer successfully to the real platform³. During the training evolution can potentially learn strategies that exploit behaviours produced by the simulator but that are not expressed by the real platform; transferability is therefore directly related to the fidelity of the simulator used for evolution.

In practice transferability is a common problem in evolutionary robotics, and some of the earliest research devoted to it was carried out by Jakobi ([110, 111]). Jakobi proposed dividing the aspects of the robot and its environment into a *base set* and an *implementation set*. The base set includes all the aspects deemed to be essential for evolving a good controller, and should be reproduced by the simulator with the same variability that such aspects have in the real world. The implementation set instead consists of all the remaining aspects of the simulation, and these need to be represented with variable levels of noise with the aim of preventing evolution from relying on them. While sound, the approach of Jakobi is not that useful since in practice the line between the base and implementation sets is very fuzzy, and ultimately the choice of how to define the two sets and how to decide on the level of noise to use is down to the designer ([247]).

In [154] and [155] Miglino and co-workers proposed to sample the responses of the sensors and actuators of a Khepera[162] robot in order to build look-up tables to be used as sensor and actuator models⁴. In the process the choice of which elements in the systems were to be modelled was left to the designer, and as the authors comment, due to the need to carefully sample the response of the real robot in a variety of situations, this technique becomes infeasible for complex robots and environments.

To improve transferability, instead of focussing on more realistic models one could design controllers with the inherent ability to adapt to changes in the robot and in the environment (i.e. with plasticity). Floreano and Mondada ([71]) propose a method that, instead of evolving the weights of a neural network controller, evolves the type of rule used to learn each of the synaptic weights, along with learning whether a weight is excitatory

³In the evolutionary robotics community, the concept of transferability is defined in a rather loose way. The term is mostly used when a controller evolved in simulation is able to perform the same task it was trained for on the real system (e.g. balancing a pendulum [80] or performing a memory task [110]). In this work we will adopt this qualitative definition.

⁴We use the terms sensor and actuator models, as these are in line with the probabilistic robotic terminology that we prefer to adopt in this work, Miglino *et al.* simply called them “a simulation of the infrared sensors and of the electric motors”.

or inhibitory. The controllers obtained are then tested on the real robot. At run time the synaptic weights are initialized randomly, and adapted to the stimuli from the environment according to the learning rules.

A similar but more sophisticated approach is proposed in [63] which evolves the properties of the diffusion and reaction neuromodulators for each of the neurons of a network. Diffusion neuromodulators are used to determine how a neuron can influence its neighbours, while reaction properties determine the effect of the incoming neuromodulators on the properties of a neuron. More specifically, neuromodulators can affect the threshold of a neuron, change a synapse from inhibitory to excitatory, or change the rules that govern learning (i.e. Hebbian, anti-Hebbian, non-learning). The concept of neuromodulators shares many similarities with the diffusion gases proposed by Husbands for GasNet networks ([105]).

Although interesting, these approaches based on plasticity have only been tested on recoverable robots (i.e. a Khepera and a robotic gantry) that are robust and stable enough to tolerate the fact that the controller is learned while performing the task. It is not clear how such methods would fare on the types of platform we are interested in, which require very proficient controllers as soon as the task commences.

In the literature, the idea of evolving controllers on the real platform, or of performing part of the evolution in simulation and part of the evolution on the real hardware, has been proposed as a method for dealing with the problem of transferability. Among these ideas we single out the seminal approach of Bongard and Lipson ([33]) who proposed the idea of evolving a proficient controller with the minimum number of trials on the real hardware. Interestingly, to achieve this goal they coevolved the controllers and the robot simulator. The simulator is used to evolve both controllers, and actions that when tested on the real platform will allow effective discrimination between poor and good models. However, we note how, even in the case of Bongard and colleagues, the four legged robot they used was recoverable.

Approaches based on performing the full evolution or even a very limited part of it on the real platform (like in the work of Bongard), rely heavily on having a recoverable platform. As a consequence, in our settings they cannot be directly applied.

2.1.2 Incremental Evolution

Evolution is essentially a process of search, and therefore its performance is directly connected to the smoothness and multimodality of the fitness landscape, which in turn depends on the platform, the environment and the fitness function used.

As a result of this complex interaction of factors, the part of the search space that contains a solution can be very small in comparison to the size of the search space. Consequently it is not uncommon to have a situation in which evolution is not able to find controllers capable of solving the task in hand even if the controller representation is sufficiently powerful. Nolfi [172] named this phenomenon 'the bootstrapping problem'. Enlarging the population will increase the chance of sampling a good region of the search space, but for many interesting problems (e.g. evolving neural controllers) the search space is so large that simply changing the population size is not sufficient.

Incremental evolution is an established technique for evolving solutions to problems that are too difficult to be solved directly, it is based on the idea of creating a sequence of tasks of increasing difficulty which leads eventually to the full task. To increase the difficulty of the task, modifications can be made to the environment, or to the fitness function, by for instance including additional terms.

Harvey and colleagues [92] were among the first to propose this idea in the context of developing controllers able to use minimal visual input to reach and follow a target using a robotic gantry. Initially large fixed visual targets were used to bootstrap evolution, and these were then substituted by smaller targets, and finally by moving targets.

Gomez *et al.* [81] explored the idea of incremental evolution in the setting of evolving neural network controllers for a double pendulum balancing problem. By fixing the length of the first pole (l) and progressively increasing the length of the second one from $l/2$ to l , they created a series of tasks that were progressively harder⁵. In this setting they showed that, for tasks that can be solved with direct evolution, incremental evolution required a lower number of function evaluation. In addition incremental evolution was also able to produce successful controller for tasks that were too hard to solve for direct evolution.

Barlow *et al.* [19] showed similar benefit from the use of incremental evolution. In their work they looked at the problem of using a UAV to locate an intermittently emitting mobile

⁵For the double pole balancing problem it is known ([95]) that the task gets increasingly difficult as the lengths of the two pendulums become more similar.

radar source. The environment is made progressively more challenging by starting with a continuously emitting fixed source, then moving to a continuously emitting mobile source, and finally considering the case of an intermittently emitting mobile source. Incrementality was also introduced at the level of the fitness function by adding additional objectives. In the initial 200 generations of each of the tasks, only the distance to the radar source was used as fitness, but subsequently another three metrics were used with the aims of minimizing the number of turns, maximizing the time spent in level flight, and getting the UAV to loiter around the target.

When the aim is to evolve not only one but a set of behaviours, the idea of scaffolding can be used, that is, gradually restructuring the robot's environment so that selection pressure favours the addition of a new behaviour. In a recent publication [26] Bongard showed how this approach can be used to evolve controllers for simulated quadruped and a hexapod robots that are able to approach, grasp, lift and hold an object. Bongard defined a way to make the task easier (i.e. by allowing more time) and a way to make the task more difficult (i.e. by moving the target object further away). During the evolutionary run, the difficulty of the task was changed in response to the ability of the current best controller. This mechanism allowed the controller to learn successfully all the four behaviours of approaching, grasping, lifting and holding the target object.

In all the reported examples, the understanding of the problem by the designer was crucial for defining how to construct an incremental evolutionary path. In [73] Fukunaga and Kahng looked in detail at the correlation between the way tasks are incrementally decomposed and the performance of the evolved controllers. In their analysis of controllers evolved for a two-agent pursuit-evasion game, and also for the ant trail following task, they argue that the effectiveness of the incremental approach may be dependent on the similarity between the incremental tasks, rather than on their increasing difficulty. In fact, in their experiment they also show how even intermediate tasks that are more difficult than the target task can lead to effective incremental evolution.

In summary, while it is clear that incrementality can be a very effective tool, for scaling up evolution to more complex and useful tasks, is very much the case that the human input is a key component in identifying effective ways to do so.

2.1.3 Modularity

Instead of subdividing the task or the fitness function one could also subdivide the controller and make it modular. Many reasons have been put forward to explain why a modular structure is in practice often beneficial.

In the case of neural controllers, a modular architecture allows for a reduction in search space dimensionality due to the reduction in inter-modular connections. In a fully connected neural network with n neurons, we have a number of connections roughly of the order of $O(n^2)$; in a network consisting of several modules with only limited interconnections between modules, this number will in general be much smaller.

Calabretta *et al.* [39] introduced the concept of *neural interference* to express the situation in which the learning of a mechanism (i.e. a behaviour) interferes with the learning of additional mechanisms. This is based on the idea that if the same neural connections in a network can potentially be used by more than one mechanism, whichever mechanism is learned first might exploit those connections in such a way that a second mechanism cannot be learned without disrupting the first. Their arguments are based on experiments attempting to evolve artificial neural networks for an abstract neuroscientifically-inspired task, in which the network has to perform two related but different tasks using the same input. Calabretta *et al.* and others have shown that modularity can eliminate this effect by dedicating a module to each one of the tasks.

In the same paper Calabretta *et al.* also propose a second mechanism that plays an important role when learning more than one behaviour using evolution: *genetic interference*, which is a consequence of genetic linkage. With any reasonably high mutation rate, several mutations are made in every generation at different positions in every genome. This means that a beneficial mutation in one part of the genome is likely to be accompanied by a disadvantageous mutation in another part of the genome, meaning that the individual with this genome gets a low fitness, and that the beneficial mutation is likely to be lost. Calabretta and coworkers managed to alleviate this problem to some extent by coevolving the two separate neural modules that constitute their controller.

Several researchers have investigated modularity in conjunction with cooperative co-evolution, since such an approach allows one to exploit the advantages of a modular decomposition while taking into account the complex interdependencies that might arise between subcomponents ([187]).

In the literature different levels of modularity have been considered for controllers, from the macroscopic level where a module is represented by several neurons, to the microscopic, in which a single neuron is considered a module.

The work of De Garis [55] is of the first type; he evolved the modules of an artificial neural network for controlling simulated creatures. At this level there is also the more recent work of Garcia-Pedrajas and Ortiz-Boyer who coevolved network submodels along with the way they interconnect in order to solve a pattern classification task [75]. Interestingly the latter placed the number of submodels as well as their topology under evolutionary control.

The microscopic type includes a family of well known neuro-evolutionary systems; SANE [165] and ESP [80], which automatically evolve subpopulation of neurons from which individual neurons are selected to participate in the network that represent the controller. In this technique the number of subpopulations is also under evolutionary control. Both approaches have shown very good abilities in evolving networks for complex control tasks such as double pole balancing.

In the last two sections we have seen that there are ample indications that the division of tasks and networks into subtasks and subnetworks improves the ability to evolve good controllers. However, in the cases in which the human designer imposed this subdivision, this might at the same time prevent evolution from finding solutions outside the boundary fixed by the designer, perhaps eventually even decreasing the final fitness. In this respect it has been argued that as many decisions as possible about the structure of the controller should be left to the evolutionary process [173].

In practice we have seen that, while more progress has been made in the cooperative coevolution scenario, the decomposition in incremental evolution is mostly down to the designer. Inevitably we are therefore left with a trade-off between the ease of development on the one hand, and the theoretical maximum performance of a controller when developed by an evolutionary algorithm on the other. This trade-off echoes a similar one encountered when designing controllers manually. For example, in the case of a MIMO (multi-input multi-output) system it is often possible to split the controller into several SISO (single-input single-output) loops to ease the design process.

In the next two sections we take a more practical look at the published work on the problems of learning car and helicopter control, which is therefore directly relevant to our investigation.

2.1.4 Car Control

The problem of designing a controller for a car like vehicle has received a significant amount of attention in the literature; solutions achieving various degrees of effectiveness have been proposed. Approaches vary from those considering a linearized model of the car ([38]), to versions that learn a more sophisticated nonlinear car model ([87]), or are model independent or that look ahead on the path ([221]) in order to improve the tracking abilities.

In [228] and [99] the problem of car vehicle control has been revisited to produce a controller able to drive a car through rough terrain as required for the 2005 DARPA Grand Challenge ([54]). In these settings robustness was considered the main requirement, and this led to the design of a particularly effective controller that contributed to the outstanding performance of the Stanford Racing Team's vehicle Stanley⁶.

The approach of Tanev *et al.* [223], uses evolution with the aim of generating driving rules that are path independent and can drive a car following a set of pre-defined waypoints. Rules are expressed in the form of basic parameters of the car's behaviour, which are straight-line velocity, turning velocity, and throttle lift-off zone⁷, and genetic programming is used to generate functions that allow the calculation of such basic parameters at each point of the track. The inputs available to the functions generated by genetic programming are features of the track: the distance from the previous to the current waypoint, the distance from the current to the next waypoint and the angle between two consecutive segments that defines the track midline. During evolution the fitness of a controller is computed as the average of the fitnesses obtained on four different circuits. The controllers obtained at the end of the evolutionary run are capable of good performances on all the four sample circuits on which they were evolved.

Finally let us review a domain in which the automatic learning of car controllers has received significant attention, that of car racing games. Fostered by the competitions organized during recent computational intelligence conferences [235, 137] several approaches to learning car controllers have been developed, ranging from approaches based on artificial neural networks involving training weights, to those in which the topology is also evolved, as well as approaches based on genetic programming and fuzzy control. Since the compe-

⁶We will comment more on this controller in Section 7.1.2.

⁷The distance from the apex of a corner at which the car begins slowing down from the straight-line velocity to the turning velocity.

tition rules do not explicitly constrain the techniques or the amount of knowledge used⁸, competitors often used their experience in playing the car racing game to obtain more proficient controllers, either with a well engineered learning strategy or by using specific controller structures.

The first competition was based on a relatively simple simulation developed by Togelius during his doctoral work ([231]), while the second featured a much more complex environment based on the freely available TORCS simulator ([237]). Although many of the qualities reproduced in such simulators are derived from physical phenomena defining the dynamics of racing cars, it is difficult to gauge to what extent such simulators reproduce the dynamics and uncertainty of a real vehicle since their main aim is to ensure a challenging game for the player.

It has to be said however that the flexibility of many of the learning schemes and representations used in this domain makes them potentially very suitable for learning controllers that could transfer to real vehicles if used in conjunction with an appropriate simulator.

2.1.5 Single Rotor Helicopter Control

In the last decade the design of control systems for single model helicopters has received a lot of attention from the aeronautics and control communities because due to their inherent instability and complex nonlinear dynamics, such platforms are ideal test benches for advanced control techniques.

The vast majority of the literature considers the problem of the manual design of controllers for single rotor helicopters. Initially work concentrated on controlling the machine in the near-hover condition where, thanks to the reduced dynamic envelope of the flying machine, the problem of control is greatly simplified. The approaches proposed ranged from simple multi-loop PID schemes [37, 209] to techniques from robust control [21, 248].

The research later extended the envelope of the controllers to more advanced manoeuvres including forward flight and coordinated turns ([128]). The most advanced abilities were demonstrated in [76] with a small set of aerobatic manoeuvres for which they recorded piloted demonstrations, and then hand-engineered a sequence of desired angular rates that the controller has to track. Independently of the specific techniques used, a fundamental

⁸Even manually written controllers are allowed.

ingredient in all the approaches is the availability of a very good nonlinear dynamic model of the helicopter, able to capture both its complex dynamics and the coupling between the helicopter's dynamic modes. We will comment further on such models in Section 2.3.

The techniques used for this type of advanced control are often extensions of those already mentioned in the case of near-hover flight. La Civita *et al.* propose an approach based on gain-scheduled \mathcal{H}_∞ control in which a linearized model of the helicopter is used to design a series of \mathcal{H}_∞ controllers that are then scheduled in accordance with the flight mode. Kim *et al.* [120] instead implemented a scheme based on model predictive control which at every control step chooses the control outputs that will ensure that the reference trajectory is tracked as well as possible.

Among the approaches based on learning, the most successful ones come from the reinforcement learning literature. Bagnell *et al.* [13] and also Ng *et al.* [170, 171] used policy search methods to learn the weights of a custom designed neural network forming a small module for each of the control inputs. In the case of Ng, specific connections between modules were also added to take into account the dynamic coupling between axes. Hover and inverted flight on a real model helicopter have been successfully achieved with this approach.

Abbeel *et al.* [4] proposed a reinforcement learning technique called apprenticeship learning in which an expert demonstrates the task and the data collected is then used to train a linear model of the system, and also to establish the reward functions for learning the controllers. Abbeel and colleagues were able to demonstrate sophisticated aerobatics on a real helicopter, but this required a lot of expert input; different manoeuvres required different costs (e.g. a cost that penalizes the change in inputs over consecutive time steps to stop the controller from switching rapidly between low and high values) and also modifications to the controller (e.g. introducing an integral term).

The concept explored by Abbeel was extended by Coates *et al.*, in [48] where multiple demonstrations from a pilot were aligned in time and used to provide a very accurate local model of the helicopter by using LWLR (locally weighted linear regression). The model was then used online in a model predictive control scheme where at each time step a LQR was efficiently computed using differential dynamic programming. Using this approach the authors were able to repeat a series of impressive aerobatics manoeuvres previously demonstrated by a proficient pilot. To achieve better trajectories, constraints

were manually added to the pilot’s demonstration. Although the result is remarkable, the method is restricted to trajectories previously demonstrated.

More limited success, predominantly in simulation, has been achieved using evolutionary computation. In [185] a GA was used to optimize the gains of a predesigned pitch controller for a simulated helicopter. A floating point representation of the genes appeared to facilitate the evolution process. However, limiting the study to the longitudinal dynamics alone is a drastic simplification of the general problem. Evolutionary computation has also been employed to refine the parameters of a predesigned fuzzy rule controller for a simulated helicopter ([97]). Separate functional modules (longitudinal, lateral, altitude) were used in the controller. The simultaneous learning of the weights of all the rules turned out to be possible only if a suitable initial set of rules was provided. Giving evolution the freedom to increase the rule set from a simple reduced subset demonstrated the ability to produce an adequate rule set from scratch, although much of the controller was still hard-wired.

Very recently ([123]) neuroevolution has been applied to train the weights of a network with custom-designed topology in order to stabilise a simulation of a single rotor helicopter in near hover⁹. It is interesting to note how the authors also attempted to evolve the structure of the controller as well as its weights using well known neuroevolution methods (i.e. NEAT [216]); however, none of the evolved networks performed better than the handcrafted one. The authors attribute the poor performance to the large search space which makes it difficult for evolution to search.

2.1.6 Quadrotor Helicopter Control

Although the quadrotor was one of the first successful VTOL (vertical take off and landing) vehicles [131], it is only in the last few years that the availability of batteries with higher energy density (i.e. Lithium-Polymer) and MEMS (Micro Electro Mechanical systems) gyros has made it possible to build miniature flying machines based on this concept.

Given its simple construction, its robustness and its good payload capabilities, the quadrotor concept has gained popularity as a platform base for research [85, 94, 245] and for commercial applications [78, 119, 79].

A large number of authors have addressed the problem of manually designing control

⁹The simulation is based on the linear model developed by Abbeel *et al.* [5].

systems for quadrotors using techniques from control theory, and they have considered the problem of stabilization as well as the problem of position and attitude control.

Techniques proposed in the literature range from PID control [35, 224, 89], to Lyapunov theory [62, 138, 181], backstepping [144, 145] and LQR techniques [138, 35]. In various ways these standard techniques require either a model of the system (generally based on first principles), or a good understanding of quadrotor dynamics, in order to design the controller structure. Platform knowledge is also often exploited to make simplifying assumptions (e.g. decoupling). For some of the techniques (e.g. PID), it is also possible to determine the controller parameters empirically by means of careful testing.

All the cited work has focussed on controlling the helicopter in near-hover and indoors, with the notable exception of [100] which examined outdoor trajectory tracking using carrier phase GPS as a primary sensor.

More recently the focus has started to move towards the idea of using learning as part of the control system design process. In [245] a mixed methodology was adopted; an integral LQR controller was designed for the attitude control, while the altitude control was optimized using reinforcement learning. LWLR was used on data from flight tests to produce the dynamic model necessary for the learning. Policy iteration was then used to determine the constants for a hand designed controller based on altitude, velocity and acceleration feedback.

In [140] and [190] the focus moved to extending the autonomous flight capabilities of quadrotors to perform aerobatics¹⁰.

In [190] Purwin and D'Andrea looked at extending the control of quadrotors to more aggressive manoeuvres using iterative learning control. At first a simplified model of the system (in 2D) is used to generate a feasible trajectory that is tested on the helicopter. The difference between the real and planned trajectories is used to update the control sequence used during the manoeuvre. Such a procedure can be reiterated to get better and better control results for the chosen manoeuvre.

Lupashin *et al.* [140] devised an algorithm to learn the parameters of an aerobatic backflip¹¹ manoeuvre. Using an in-depth understanding of the platform and its constraints,

¹⁰It is worth noting that, due to the fact that the blades of a quadrotor have fixed pitch, they are not able to produce lift when the machine is flying upside down, therefore is not possible to compensate for any altitude loss during manoeuvres; this makes aerobatics particularly tricky. In contrary, in a single rotor helicopter it is possible to reverse the pitch of the main rotor to produce sustained inverted flight [170].

¹¹A manoeuvre in which the quadrotor is flipped rapidly about its lateral axis.

the manoeuvre is manually split into phases. Each phase is then parameterised by a simple set of values like its duration or for instance, in the case of a constant acceleration phase, the value of the acceleration to be maintained. An initial parameter set is optimized offline using a simplified model of the system, and then the results of testing the manoeuvre on the real system are used to improve the parameters. Several tens of iterations are needed for the parameters to converge but the results are good, with the quadrotor able to fly up to three consecutive backflips.

Finally another recent approach that is similar to our work [117], considered the case of learning the parameters of a multi-loop PID controller for a simulated quadrotor. A model based on first principles is used to simulate the system dynamics, and zero mean Gaussian noise is added to the model's predictions to simulate flight disturbances. The CMA-ES¹² algorithm is used to optimize the controller parameters, and the various control loops are evolved sequentially to simplify the learning. When a new loop is added, the fitness function is modified to add the error term corresponding to the newly added control loop; this ensures that the new behaviour is maintained while not disrupting the abilities already learned. The controller obtained shows good tracking abilities, although only in a simulation.

2.2 The Modelling Task

This section addresses the concept of vehicular modelling in-depth, covering all the aspects related to the work described in this thesis, and dedicating particular attention to the implications of the concept of limited platform knowledge. In the sections that follow we will address each of the fundamental steps of producing a model, in turn; these are:

- choice of representation;
- choice of model inputs;
- choice of excitation signals and data collection;
- model estimation;
- model validation.

¹²A derandomized extension of evolution strategies [90].

2.2.1 Model Representation

This thesis will focus on a very specific subset of models, that is relevant to the automatic development of control laws for robotic vehicles that can be described as 6DoF rigid bodies.

More precisely we will restrict ourselves to 6DoF dynamic systems for which the relationships that define the development over time of the system's state do not change with time (time invariant). Additionally we consider only lumped systems that can be described as rigid bodies, neglecting any change within the vehicle structure (e.g. structural deformations).

We will not make any restriction on the linearity of the systems under study, since in general, in our domain of interest, both linear and nonlinear systems are relevant. Both deterministic and probabilistic representations will be used. Our concerns will be directed towards continuous systems, but since all our algorithms and simulations run on digital computers, we will in practice only deal with the discrete (sampled) counterparts of continuous models.

At first one might think the class of systems chosen to be rather narrow, but in practice, many interesting autonomous systems from wheeled robots, to aerial vehicles, underwater and surface vehicles can be usefully approximated as 6DoF rigid bodies. While we already explained how we will look at the use of such models for control, their domain of application in robotics extends from state estimation, to mapping, fault detection, planning and practically any other situation in which it would or could be useful to have a prediction about the development over time of the system's state.

Phenomenological vs. Behavioural

Now that we are clear about the type of systems we will be dealing with, it is important to address a major distinction concerning the type of representation that can be chosen for our models. Specifically, one can choose between a *phenomenological* model, which is derived from basic principles and the underlying theoretical formulation of the physical phenomena involved, or a *behavioural* model, that is limited to reproducing the input-output relationships of the system. Both types of representation present advantages and disadvantages as summarised in Table 2.1, and ultimately the choice of which one is to be preferred depends on the intended purpose of the derived model.

A *phenomenological* model entails the principled formulation of every part present in

	Phenomenological models	Behavioural models
<i>A priori information</i>	fundamental	not necessary
<i>Structure</i>	physical interpretation	no concrete interpretation
<i>Parameters</i>	physical interpretation	no concrete interpretation
<i>Validity</i>	potentially larger than the envelope covered by the data	limited to the envelope covered by the data

Table 2.1: Phenomenological versus behavioural models (adapted from [113]).

the system, which leads therefore to the definition of a clear structure for the model, the parameters of which will have a clear physical interpretation. As a consequence the process of model identification is limited to the determination of the free parameters of the model based on the evidence provided by the experimental data. Since such a description is based on in-depth prior knowledge, it is often appealing because of its explanatory abilities, as it allows the assessment of the obtained parameters on physical grounds. However, depending on the specific system, a *phenomenological* model can become very complex (e.g. due to the number of parts that compose it); in extreme situations, it might not even be possible to define accurately the underlying physical phenomena taking place¹³. Once their parameters have been identified from experimental data, these models can theoretically predict the behaviour of the system even outside the envelope sampled during data collection; ultimately however, only further experimental data collection will be able to confirm or contradict the model's expected validity.

Phenomenological models constitute the standard modelling practice for the majority of engineering disciplines, including aeronautics and robotics.

Behavioural models focus instead on replicating the input-output response of the system under analysis, and are based solely on the evidence provided by the collected data. In this context, the problem of modelling can become almost field and platform agnostic, concerned only with finding a suitable methodology to learn and reproduce the system's

¹³Examples of such phenomena that are particularly relevant in our work are the turbulent flow effects involved in propellers and aircraft wings (well known for exhibiting chaotic properties) and the anisotropic friction between the wheels of our toy car and the floor surface (also known to have very local and nonlinear properties).

behaviour. Since no knowledge about the system is required *a priori*, a very rich spectrum of possible model descriptions must be handled: linear, non-linear, with or without internal states, with multiple inputs and multiple output variables, etc. Behavioural models do not require any in-depth knowledge of the system, but at the cost (in general) of not being able to provide a physically meaningful model structure, nor physically meaningful parameters. Some techniques such as symbolic regression, can produce models amenable to partial physical interpretation, although this cannot always be guaranteed. This clearly limits the type of applications that the model can be used for. For example, for simulation purposes, the form of the model is immaterial, as long as the simulation replicates the input-output of the actual process; if instead if we want to apply techniques from classical control theory, a representation in terms of a transfer function (or its equivalent) might be needed. Fortunately, if a faithful simulation of the system is available, the problem of control design can be turned into a problem of the optimisation of the structure and parameters of the controller as we explained in Section 2.1.

Behavioural models are not only restricted to simulation or automatic controller design, as it is often possible to extract parameters with conventional meaning from a learned behavioural model; a classic example in the domain of aircraft modelling is the numerical computation of the aerodynamics derivatives¹⁴ from a behavioural model (e.g. an artificial neural network [67][191]).

Overall, in this thesis, we have to reconcile the choice of description with the principle of minimal domain knowledge that we identified in the previous chapter as being the distinctive trait of this work. On this basis it is clear that a *phenomenological* description is not at all appropriate to our aim, since it is based on in-depth knowledge of the system, and of its internal components, as well as on specific knowledge of the scientific or engineering disciplines that apply to such a system. Our approach to automatic modelling will therefore be based exclusively on non-platform specific non-linear behavioural models.

In the system identification field, a slightly different model taxonomy denominated by colours is used; the classification is based on the amount of *a priori* knowledge that a model assumes ([136]). The term *white-box* refers to models for which both the structure and the parameters are known *a priori*; both are usually determined from first principles and static measurements. At the opposite end of the spectrum are *black-box* models for

¹⁴A brief explanation of the aerodynamics derivatives and their importance can be found in Section 4.1.1.

which no direct *a priori* knowledge is assumed, and only the input and the output signals are well defined¹⁵. Models in which some knowledge about the internal structure of the system is used are called *gray-box* models. To this class belong phenomenological models which are based on first principles, but in which the parameters are estimated to provide the best match to the experimental data.

State and Noise Definition

Since we are dealing with time invariant systems, and in particular with vehicles, their dynamics will be defined simply by their development over time. Among other possibilities, the most intuitive and also the most common way of expressing the dynamic equation for vehicular systems is the state space formulation, in which the state development over time is expressed in the form of differential equations:

$$\dot{\mathbf{x}}_t = f(\mathbf{x}_{t:0}, \mathbf{u}_{t:0}, \boldsymbol{\omega}_{t:0}) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.1)$$

$$\mathbf{y}_t = g(\mathbf{x}_t, \boldsymbol{\nu}_t), \quad (2.2)$$

where \mathbf{x} is the state vector, \mathbf{u} is the input, $\boldsymbol{\omega}$ and $\boldsymbol{\nu}$ are respectively the system noise and observation noise and \mathbf{x}_0 is the state initial condition.

In equation 2.1 we use $\mathbf{x}_{t:0}$ as a shorthand for the sequence of states from 0 to t , more precisely:

$$\mathbf{x}_{t:0} = [\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_0];$$

the same subscript notation is also used for \mathbf{u} and \mathbf{w} . Is worth noting that the functions f and g do not necessarily need to be deterministic.

To understand equations 2.1 and 2.2 we have to first define the concept of *state* (\mathbf{x}).

In [227] Thrun *et al.* write:

“...it will be convenient to think of the state as the collection of all the aspects of the robot and its environment that can impact the future.”

and they also introduce the concept of complete state:

“A state \mathbf{x}_t will be called complete if it is the best predictor of the future.

¹⁵Since as we will see in Section 2.2.2 we consider the choice of model inputs a form of platform specific knowledge, a black-box model might or might not fit into our definition of not using platform specific domain knowledge, depending on how the inputs are chosen.

Put differently, completeness entails that knowledge of past states, measurements, or controls carry no additional information that would help us predict the future more accurately.”

Is worth noting that this definition does not force the state development over time to be deterministic. Systems whose dynamics fulfil this condition are commonly known as first order Markov chains, and the Markovian property is formally written as:

$$p(\mathbf{x}_{t+1}|\mathbf{x}_{t:0}, \mathbf{u}_{t:0}) = p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t). \quad (2.3)$$

Clearly, the notion of complete state is only of theoretical importance, as in practice it is impossible to specify a complete state for any realistic and useful system since this would include an extremely large number of variables. In the case of a model helicopter for example, this would include air temperature and humidity, any deformation of the blades or structure, the internal resistance of the battery, the temperature of the coils of the electric motors etc. just to name a few. In practice only a small subset of the complete state variables can be considered, leading to an incomplete state. If the state is incomplete, then strictly speaking the Markovian property is not ensured. A crucial task is therefore to define a set of state variables that is sufficient to predict the development over time of the system to the level of accuracy required.

In this work we wish to deal with generic 6DoF vehicles that can be represented as rigid bodies, so we choose as state variables the same quantities that are normally used in physics to describe this type of system. Given a body-fixed reference frame attached to the vehicle, we will use as state variables the linear velocity in body coordinates (u, v, w) , the rotational velocity about its principal axes (p, q, r) and the angles that define its orientation with reference to a world-fixed reference frame¹⁶ (ϕ, θ, ψ) :

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T. \quad (2.4)$$

Since the kinematic transformations that relate accelerations, velocities and orientation are all well defined and platform independent¹⁷, the function f could predict linear and

¹⁶For some of our vehicles, a reduced state vector will be used; in Chapter 3 the state variables chosen for each platform will be described in detail.

¹⁷We report those standard transforms for each of our platforms in Chapter 3.

angular velocities:

$$\mathbf{v}_t = f(\mathbf{x}_t, \mathbf{u}_t, \boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.5)$$

$$\dot{\mathbf{x}}_t = H_2(\mathbf{v}_t, \phi_t, \theta_t, \psi_t) \quad (2.6)$$

$$\mathbf{y}_t = g(\mathbf{x}_t, \boldsymbol{\nu}_t), \quad (2.7)$$

where $\mathbf{v}_t = [u, v, w, p, q, r]^T$ is constituted by the concatenation of the three linear and three angular velocities in body coordinates. Or equivalently, f could predict accelerations:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_t, \boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.8)$$

$$\dot{\mathbf{v}}_t = H_1(\mathbf{a}_t, p_t, q_t, r_t) \quad (2.9)$$

$$\dot{\mathbf{x}}_t = H_2(\mathbf{v}_t, \phi_t, \theta_t, \psi_t) \quad (2.10)$$

$$\mathbf{y}_t = g(\mathbf{x}_t, \boldsymbol{\nu}_t), \quad (2.11)$$

where $\mathbf{a}_t = [a_x, a_y, a_z, \alpha_x, \alpha_y, \alpha_z]^T$ is constituted by the concatenation of the three linear and three angular accelerations in body coordinates. H_1 and H_2 represent the computations that perform the necessary coordinate transformation and for simplicity we have omitted the integration in time. The computations H_1 and H_2 and the integration are well known from basic physics and will be described in detail in Chapter 3. In this work often we will call this two approaches respectively modelling in acceleration space and modelling in velocity space. In principle the two formulations are equivalent, however the first one can potentially give origin to more complex models ([5]) since the model needs to represent the nonlinear transform H_1 which instead is performed explicitly in the acceleration formulation. The formulation based on predicting acceleration is very common in the case of models based on first principles since forces torques and moments are naturally expressed in terms of accelerations.

Without loss of generality in the remaining part of this chapter we will refer to the case of predicting the system's accelerations; the extension of our considerations to the case of predicting velocities is straightforward. In all the thesis we will almost exclusively use velocities and accelerations wrt the body frame, therefore for the remaining we will take the freedom of not mentioning explicitly the body frame of reference. We will remind the reader of the appropriate frame of reference whenever a misunderstanding could rise.

Before analysing in detail equation 2.8, it is worth clarifying that, while the Markovian assumption prescribes \mathbf{x}_{t+1} to be a function of \mathbf{x}_t and \mathbf{u}_t , in practice delays might be present in the inputs of a real system (alternatively, delays within the system might be approximated by lumped delays at the system inputs). Since delays affect the timeliness of the system response, they are particularly important when the model is used for control system design. In this work we are only dealing with discrete time models and so input delays can be implemented quite easily by a time shift of the control inputs. To take this possibility into account equations 2.8 and 2.11 can be rewritten as:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_{t-D}, \boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.12)$$

$$\mathbf{y}_t = g(\mathbf{x}_t, \boldsymbol{\nu}_t), \quad (2.13)$$

where D indicates an array of time delays¹⁸; the array \mathbf{u}_{t-D} becomes the new array of inputs, the components of which are the input signals shifted by the delays specified in the corresponding element of D ¹⁹

With a clear definition of the concept of state and inputs in place, we can now examine equation 2.12. In such equation is easy to recognise the Markovian assumption at work: the acceleration vector (\mathbf{a}_t) is expressed as a function (in general nonlinear) of the inputs \mathbf{u}_{t-D} and of the state \mathbf{x}_t . Or equivalently, the state vector at time $t+1$ (\mathbf{x}_{t+1}) is a function only of the state and input at time t (respectively \mathbf{x}_t and \mathbf{u}_{t-D})²⁰.

In addition to the deterministic input \mathbf{u}_{t-D} , the general formulation of equation 2.12, considers the system also to be excited by a stochastic input $\boldsymbol{\omega}_t$. In the estimation literature $\boldsymbol{\omega}_t$ is usually called process noise; in what follows we will use the two terminologies interchangeably. In the evolutionary computation literature, in the context of improving controller transferability the process noise has sometimes also been called *trajectory noise* ([82]). The process noise is a stochastic formulation of those contributions to the system dynamics that are not captured due to the specific representation chosen for the model; this might be because they are not measurable, or are not possible to model, or are not modelled on purpose in order to reduce model complexity.

¹⁸When referring to a single control input we will simplify the notation for example we will write u_{th-D} instead of $u_{th,t-D}$.

¹⁹To simplify the notation, from now on we will not report explicitly the delay array D unless relevant to our discussion.

²⁰This obviously follows from the fact that the integration that computes the new state from the accelerations and the previous state is deterministic.

In general, it cannot be assumed that state variables are directly measurable, but only that a set of output variables \mathbf{y}_t is observable. We need therefore a measurement function g that expresses how \mathbf{y}_t is related to \mathbf{x}_t . Any real world observation of the output variables will also inevitably be affected by noise ($\boldsymbol{\nu}_t$).

Equation 2.12 defines the system state update equation, but to fully specify the development over time of the system, we also need to define its initial conditions \mathbf{x}_0 . Identifying the initial conditions \mathbf{x}_0 should be part of the modelling however in practice it seems that this is seldom done ([113]) because only the initial segment of recorded data can be expected to provide useful information to identify \mathbf{x}_0 , and in the case of a 6DoF rigid body model this information is often not sufficient to provide a good estimate ([147]). A pragmatic approach is simply to set the initial conditions to the first data point in the time series, or to compute it as the average of the first few samples in the dataset ([113]).

The formulation of equations 2.12 and 2.13 is still very general, and to make any practical use of those equations additional assumptions need to be made. In particular we need assumptions about $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$ since these stochastic components are not directly measurable. The various assumptions made about the characteristics of $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$ and about the way those stochastic components relate to \mathbf{a} and \mathbf{y} , or in other words the types of noise models, distinguish different approaches to the modelling problem.

Among other possibilities others we discuss two types of assumptions often made in the domain of aircraft modelling [113]; the first is modelling $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$ as Gaussian noise and the second is neglecting the process noise $\boldsymbol{\omega}_t$. These assumptions are also used in our experimental chapters on modelling (Chapters 4-6).

Formulating the contribution of $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$ as additive zero mean Gaussian noise components added to the deterministic functions f and g is a relatively simple assumption and has the additional benefit of not requiring any specific knowledge about the vehicle or system being modelled.

By making such assumptions we are not arguing that real world disturbances or sensor measurements do in fact have a Gaussian distribution, but rather that the Gaussian assumption is a simple and computationally tractable way of expressing these stochastic components.

Obviously if we know exactly what sensors are used to collect the data or in what form the stochastic inputs manifest themselves, more complex and effective noise models could

be used. In the sensor fusion literature a wealth of information and techniques can be found on modelling sensors and processes ([227]). However, this is ultimately domain knowledge about the platform at hand and including it would not be in line with the philosophy of this work.

Inputs and sensor measurements are also likely to be affected by systematic error (bias); such terms are made explicit in the case of models based on first principles but have to be learned in the case of automatically produced models.

Under these assumptions, equations 2.12 and 2.13 can be rewritten as:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_t - \mathbf{b}_{\mathbf{u}}) + \boldsymbol{\omega}_t \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.14)$$

$$\mathbf{y}_t = g(\mathbf{x}_t) - \mathbf{b}_{\mathbf{z}} + \boldsymbol{\nu}_t, \quad (2.15)$$

where $\mathbf{b}_{\mathbf{u}}$ and $\mathbf{b}_{\mathbf{z}}$ are respectively the input and observation bias²¹. With a little abuse of notation we are reusing the symbols $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$ to mean a zero mean multidimensional Gaussian noise with a diagonal covariance matrix²² instead of a general stochastic noise. For the filter error (FE) methods, the modelling task therefore involves identifying the functions f and g as well as the parameters of $\boldsymbol{\omega}_t$ and $\boldsymbol{\nu}_t$.

In the aeronautics community it is accepted (and has been experimentally established) that, if the data used for the estimation have been collected from flight tests in a steady atmosphere, the contributions of the stochastic inputs can be often neglected [113].

The assumption is also of relevance because is often made (explicitly or implicitly) by many of the works in the literature ([199, 31, 176])

Assuming negligible process noise, equations 2.12 and 2.13 simplify to:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_t - \mathbf{b}_{\mathbf{u}}) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (2.16)$$

$$\mathbf{y}_t = g(\mathbf{x}_t) - \mathbf{b}_{\mathbf{z}} + \boldsymbol{\nu}_t, \quad (2.17)$$

and the modelling task is in this case to identify the functions f and g .

In all the above formulations, the function f predicts the accelerations in body frame coordinates which, as explained, need to be integrated forward in time to produce the state

²¹To simplify the notation from now on we will not explicitly write the bias terms unless relevant to our discussion.

²²By having a diagonal covariance matrix we assume that the noise components added to the state vector are independent.

variable time series. Since the process of integration is always needed, in the rest of the thesis we will often not mention explicitly that it is being used.

2.2.2 Model Inputs

An important step in the modelling task is to define the inputs to the dynamic model. The equations in state space form described in the previous section, allow with absolute generality for the accelerations to be a function of both the current state \mathbf{x}_t and the (possibly delayed) control inputs \mathbf{u}_{t-D} ; this leaves open the issue of which of those variables are useful and which are irrelevant, as well as not specifying how the delays D should be determined.

The use of domain knowledge is a straightforward answer to both these problems; when this option is available it is often the most sensible, but unfortunately, given our overall aims, this is also an answer that is inadmissible for the main target of the thesis.

Using all the inputs is a way of avoiding the issue, but leads to a high dimensional approximation problem. The number of inputs will grow rapidly especially if all the possible delayed versions of a signal are presented as inputs. For instance, in order to account for the delay in input u_{th} , all the inputs $u_{th}, \dots, u_{th-d_{max}}$ (with d_{max} the maximum delay) would need to be presented as inputs²³. Large amounts of data, long training times and more importantly, a complex model that might be impossible to learn, are the risks faced if this strategy is adopted.

Two other ideas could be applied to the problem of choosing relevant inputs and suitable delays: unsupervised and supervised input selection. The first makes use of the input data distribution to select which inputs are relevant. The typical tool for supervised input selection is principal component analysis (PCA)[169]. The weak point of this approach is that only the input data is considered for the selection, rather than the effect that the selection would have on the modelling. In principle potentially useful inputs could therefore be discarded. The supervised selection scheme addresses exactly this problem; it looks at selecting the set of inputs that delivers the best identification performance. Although more computationally expensive, these methodologies deliver the best results since they are driven by the overall modelling objective. Examples in this class are stepwise regression ([113]), pruning methods for artificial neural networks ([53]), or the use of global

²³Strictly speaking, presenting the model with the whole set of past inputs leads to a model that is not Markovian.

search techniques (e.g. evolutionary computation [45]) to produce the correct input set. For identification techniques that do not rely on a predefined model structure, the input selection becomes an integral part of the modelling task. Well known methodologies in this class are genetic programming (e.g. [31]), neuroevolution of network structure (e.g. [215]), and combinations of fuzzy logic and genetic algorithm (see [51] for a comprehensive list).

2.2.3 Excitation Signals and Data Collection

Once we have found an appropriate formulation for our model, we need to gather the experimental data to train and successively test our model. Data gathering is a very important stage of the modelling process; as Jategaonkar writes in [113]:

“if it is not in the data, it cannot be modelled”.

In the aircraft and helicopter domains, a wealth of understanding about the physical principles of flight is available and the main dynamic modes of the systems in question can be evaluated *a priori*, based on first principles and wind tunnel tests. Therefore, in practice, a mixture of theoretical results and practical experience have been used by the competent aviation authorities to rigorously define control manoeuvres and flight test conditions that are currently employed for both identification and model performance evaluation. In addition to using well engineered control inputs, it is common practice in aircraft system identification to manually inspect the data from flight tests. Only the parts of the dataset containing well executed control manoeuvres are used for identification, leading to improved identification results. The experience and skills of the engineer are decisive in making such choices and guaranteeing satisfactory results.

In the last few decades, the system identification community has been very active in addressing the problem of identification for control, and as a consequence much attention has been put into the intimately connected problem of optimal experimental design. This is a very active research area (see [112, 49]) in which the typical focus is on designing inputs that optimize the asymptotic properties of the covariance matrix²⁴ of the model parameters while fulfilling specific constraints on the signal input.

This turns out to be a classic chicken or egg problem, since computing the optimal input requires knowledge of the system model. In practice it has to be addressed in an

²⁴Typical criteria are minimizing the matrix trace, its determinant or the largest of its eigenvalues ([112]).

iterative fashion ([77]). At first, the system is bootstrapped using a pseudo-random input and the data obtained is used to generate an initial approximate model. Subsequently, the optimal control input for the approximate model is computed and is then fed to the real system. Thanks to the tailor-made input, the newly obtained output provides more information about the system under test and is used to refine the model.

In this we see the fundamental way in which the new methods in identification for control differ from classical system identification; this is the idea of iteratively testing and refining models, as explained in [184]. A second important (and in some ways intuitive) concept that has been found useful in this field, and that is very relevant to our work, is that the identification of the model should be conducted under the experimental conditions in which the controller to be designed will operate. The advantages of this idea are clear; the model in this restricted envelope is potentially simpler than a more general model, making the identification easier. More importantly, this should avoid focusing resources on reproducing system effects that are not relevant for control. As a consequence, much of the ongoing research in identification for control focuses towards the closed loop identification of the system when performing the required task [96]. A more in-depth analysis of the result obtained in this field is beyond the scope of this section, but we refer the interested reader to [77] for a very good overview.

Ideas very similar to those just presented have also been explored in the computational intelligence community within the field of evolutionary system identification²⁵. The formalism is generally not the same used by the identification for control community, and the implementation is even more different, but the ideas of the active generation of tests and of iterating test and identification phases remain.

A representative example in this category is the work of Bongard and Lipson [28, 30], who proposed to approach the identification problem with a two stage Estimation-Exploration Algorithm (EEA). The estimation stage is responsible for formulating plausible models of the system, while the exploration stage looks at producing test inputs for the system to deliver an output that aims to improve identification. Two evolutionary algorithms are used, the first evolving a population of models and the second evolving a population of tests.

²⁵The concepts of active learning [207, 141] and artificial curiosity [202], respectively from the machine learning and artificial intelligence fields, are also related to this issues but we concentrate here on efforts explicitly directed towards system identification.

The procedure starts with a random test being generated and fed to the target system (which must therefore be accessible) in order to produce the first sensor data representing the platform behaviour.

During the first iteration of the algorithm, a population of models is created and evolved by the first evolutionary algorithm for several generations in order to produce a behaviour that matches the data recorded from the real system. The second evolutionary algorithm then evolves a newly created set of tests by selecting for their ability to elicit disagreement within the population of evolved models. The test that induces the highest variance between the time series of simulated sensor data produced by the current models is chosen and tested on the real robot, concluding the first iteration.

The iterations that follow proceed as the first with the exception that the new sensor dataset is used along with the previously acquired one, guaranteeing that the evolved models are able to explain the data resulting from all tests produced so far (the test history). This is essential to avoid cycling (i.e. the recurrence of previously visited states of the population), a known problem in coevolutionary methods [174]. During evolution, the tests are initialized randomly at every iteration, whereas for the models the best individuals from one iteration are used to seed the population of models in the next.

Bongard and Lipson in [31] and Schmidt and Lipson in [204] and later in [205] introduced a variation to the modelling scheme just described. Instead of generating inputs to actively test the target system, a pre-collected dataset is used. Once a test is generated, the dataset is searched for inputs matching the test, and the corresponding outputs are then used for identification, without the need for explicitly testing the system. In this variation, the algorithm is still bootstrapped with a random test, this being nothing more than a randomly selected portion of pre-collected data. The algorithm then actively searches for the portions of the dataset that are most useful for discriminating between models. As the algorithm proceeds, therefore, parts of the dataset that are already well predicted by the models are disregarded, as are the parts of the data that are badly predicted by all models in the dataset. The active search for discriminating tests guarantees that if certain parts of the dynamic envelope happen to have been sampled more often during data collection, they will not be given more importance than the others, since they will not be more discriminative than tests that are already present in the test history. Similarly, chunks of data that are not easy to model, for example as a consequence of unmeasured external inputs,

will not be used for identification, greatly increasing the resilience of the methodology to temporary disturbances.

Given that the methodology does not enforce any restrictions on the model and test representations, nor over the genetic operators used for evolution, a variety of model structures can be accommodated.

This is a characteristic signature of the work done in the computational intelligence field, which generally uses limited domain knowledge because of the type of algorithms employed. The estimation exploration method has been shown to require less experimentation and to achieve better identification results than similar methods based on randomly generated tests. Good identification results have been obtained for different types of target systems, from gene regulatory networks to dynamical systems, biological models, and even a four legged robot ([30]). In the case of the four legged robot, the tests are not represented in terms of the input given to the robot but instead by the specification of neural network controllers. During the first 16 cycles of the algorithm, during the exploration step the controllers are evolved to maximize disagreement between models; after that they are evolved towards achieving a task (i.e. moving the robot forward). The controllers will therefore preferentially feed the target robot with commands that will prompt it to move forward, and therefore the identification will focus on this part of the robot's dynamic envelope. Again we see some similarity to the concepts coming from the field of identification for control.

The doctoral work of Abbeel ([2]), which is concerned with learning control from expert demonstration, is also well aligned with the understanding developed within the identification for control community. In particular Abbeel shows that, in the case of a single rotor model helicopter, it is possible to devise a model predictive control scheme that perform as well as the expert aerobatic flight manoeuvres that have been repeatedly demonstrated ([5]). More importantly Abbeel shows that to do so it is sufficient to learn a dynamics model based on nothing more than the data collected when the expert was performing the desired types of aerobatics.

Differently from what proposed in this work, both the model and the controller used by Abbeel and colleagues are based on in-depth knowledge about the platform. In their work in the case of the model the structure is predefined and the learning consists in obtaining the model parameters. Similarly the type of controller is decided a priori while

its parameters are obtained in real time solving at every timestep a finite horizon optimal control problem.

The results obtained in the standard aircraft and helicopter literature are clearly adequate, sound and well developed, but our explicit choice of using only limited domain knowledge about our platforms does not provide us with sufficient information to apply those methods.

In this work we are interested in modelling vehicles that are not recoverable and therefore we cannot rely on random control inputs during the identification as suggested by Bongard and colleagues. However, we can manually control our vehicles, and our data will therefore be collected with the pilot trying to control the vehicle in an envelope of velocities and accelerations similar to that of the control task.

2.2.4 Model Estimation

In the system identification literature, and in particular in the subset dealing with aircraft and helicopter identification, the algorithms used are organised into classes depending on the assumptions made about the noise (i.e. the noise models) and about the availability of direct measurements of the state variables. A number of broad classes of identification methodologies can be defined, and here we refer to the classification proposed by Jategaonkar ([113]) who leverages the terminology in common use in the system identification field and proposes four main classes: *filter error* methods (FE), *equation error* methods (EE), *recursive estimation* methods (RE) and (probably the most common ones) *output error* methods (OE).

- Filter error methods (Figure 2.1) are among the most general methodologies, in that both measurement and process noise are assumed to affect the system, and the function g is used to relate the measured response to the system state. Due to the presence of process noise the system becomes stochastic, and as a consequence, it requires a suitable state estimator to propagate the state²⁶. A noise model needs to be defined for the process, and it will need to be estimated along with the process model. Generally speaking, the error between the estimated measurements and the experimental data is used to drive the model's optimisation; the form of the error

²⁶Due to the presence of noise, simply integrating the noisy state forward in time will produce a state trajectory that drifts apart from the experimental ones.

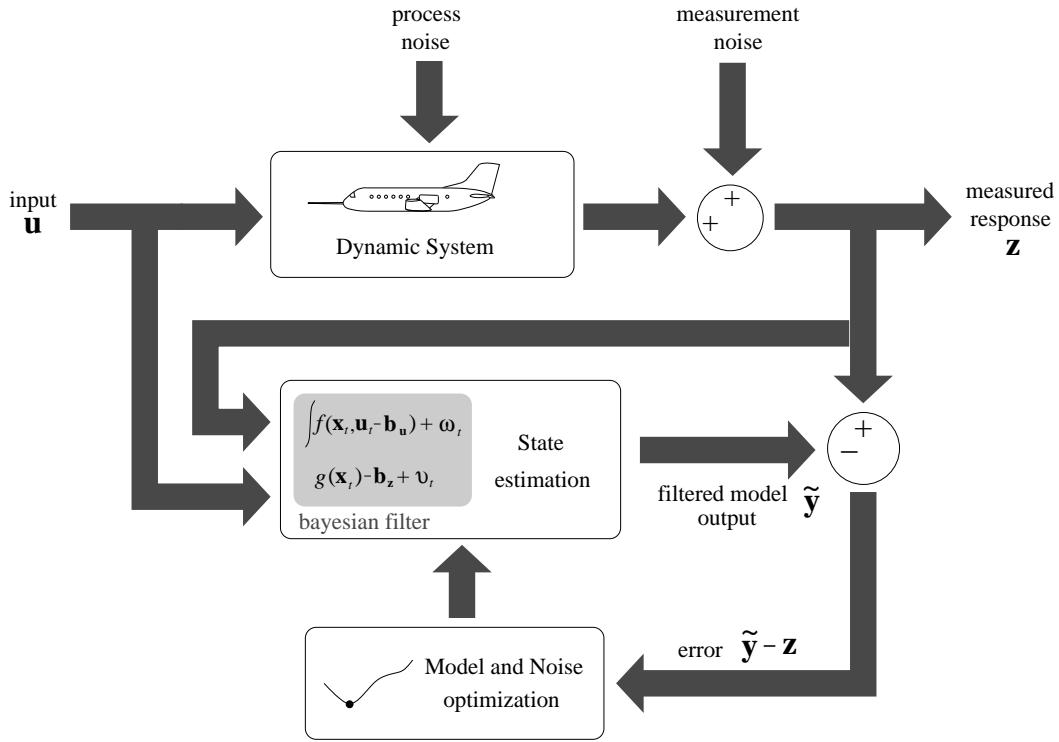


Figure 2.1: Diagram of filter error (FE) methodologies: since both process and measurement noise are considered to be present, the state variables are estimated using a Bayesian filter.

function is however implementation specific.

Methodologies based on the FE concept were first proposed by Balakrishnan [14] and were then extended by several other authors ([151, 106]). Assumptions need to be made about the process noise in order to be able to implement such a framework, and the works mentioned so far assume zero mean Gaussian additive noise for both the process and the measurements. Under these assumptions the model takes the form of equations 2.14 and 2.15, for which a suitable and efficient state estimator is the extended Kalman filter (EKF²⁷). To further simplify the problem, the process noise components in the various dimensions of the space are assumed to be independent and therefore the process noise estimation reduces to estimating one variance parameter for each of the model dimensions. These assumptions might appear simplistic, but are common practice in estimation when knowledge about the system is not available to further constrain the noise characteristics [113].

It needs to be emphasized that, since the state equations are propagated forward in

²⁷The Extended Kalman filter is an efficient recursive filter that estimates the internal state of a nonlinear dynamic system from a series of noisy measurements. The reader not familiar with this type of technique is referred to [227] for a principled and in depth treatment.

time inside the filter (this is in effect the counterpart of the process of integration done with deterministic models), any model error will build up and propagate through the whole system due to dynamic coupling. The estimated noise parameters will need to account for all those effects (we expand on this point in Section 2.2.4).

The idea of learning filter noise parameters has also been investigated in the adaptive filtering literature and more recently in the robotics domain [3]. In the latter, although it starts from the perspective of tuning a Kalman filter to improve estimation, what is derived is essentially a time-varying implementation of the FE method.

- Output error (OE) methods address the case in which under the same considerations made in Section 2.2.1 the process noise is neglected (obtaining therefore equations 2.16 and 2.17). With no process noise, the model equations can be integrated forward in time, and the states generated can be used via the measurement function g to provide a prediction of the measurements (see Figure 2.2). The error between the predicted measurements the experimentally derived measurements, is indicative of the performance of the model, and it is therefore an appropriate metric for optimization. Since it does not involve a filtering algorithm, nor the estimation of any noise parameters, the OE approach is comparatively less complicated than the FE

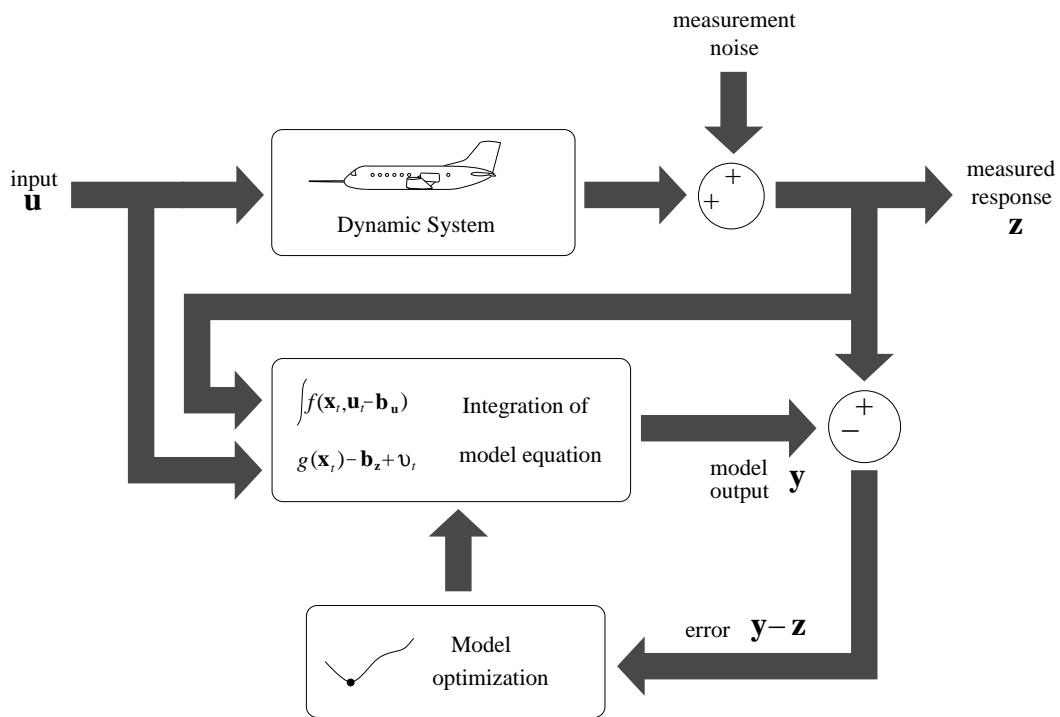


Figure 2.2: Diagram of output error (OE) methodologies: to compute the error the model is integrated forward in time but is assumed that only measurement noise is present.

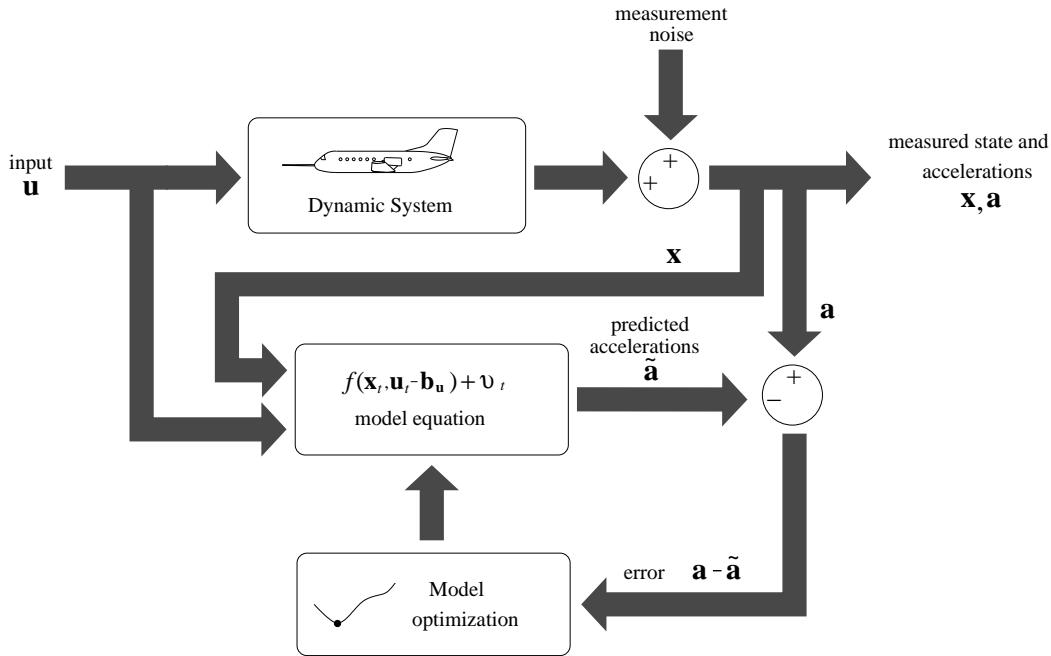


Figure 2.3: Diagram of equation error (EE) methodologies: state derivatives are directly measured, and the modelling becomes a function approximation problem.

approach. In principle, any representation can be used for the model function f however since the error metric is not directly computed in acceleration space but is instead a function of the model prediction integrated forward in time, any training methodology that assumes that a measured value for accelerations (\mathbf{a}) is directly available will not be suitable for the OE method. Again, even for the OE method, the optimisation metric is based on the propagated state and therefore any error will be accumulated forward in time reflecting the contribution of any long range effects²⁸ neglected by the Markovian assumption.

The OE method is the workhorse of the aircraft system identification field, where it is very often used in conjunction with predefined models based on first principles [113].

- Equation error (EE) methodologies take a different standpoint on the problem of modelling: they assume that both the state and its derivatives are actually available because they are measured (see Figure 2.3). Under this hypothesis, and with the Markovian assumption, we simply have to find a suitable function approximator that, given the measured state \mathbf{x}_t and input \mathbf{u}_t , produces the measured accelerations at time t ; we are therefore in the domain of supervised learning.

In this methodology the model equations are not propagated forward, and therefore

²⁸Influence of state or inputs older than $t - 1$.

any effects of correlated errors or cross coupling between axes will not build up. In the presence of correlated errors, noise is of course underestimated and therefore the model will be biased. EE methodologies are entirely based on the Markovian assumption and therefore they explicitly neglect any long range interactions in the state development over time, with potentially detrimental consequences.

Equation error methods certainly look appealing due to their simplicity and their reduced computational requirements. However, along with the problem of correlated noise, their requirements concerning the availability of measurements of the state and of its derivatives are in practice often difficult to fulfil. In the case of aircraft modelling, for example, several of these quantities are simply not measurable (e.g. no sensor is usually available to measure the aircraft angular acceleration), or very noisy (e.g. a standard GPS receiver [238] has typically a CEP²⁹ of 2.5m).

- Recursive estimation (RE) methods are the fourth and last type of methodologies identified by Jategaonkar. In contrast to the three methodologies already discussed, which are batch processes, recursive estimation involves the type of algorithms that can be used on line to obtain immediate knowledge about a vehicle model. Those methodologies are generally approximations to the batch methodologies that we have already analysed and since we do not have online requirements they do not offer us any significant advantage; therefore they will not be considered further.

All the training methods presented (OE, EE and FE) are loosely based on the idea of training a model by minimizing the error between the predicted and measured state variables. We deliberately did not specify exactly what sort of loss function should be chosen to compute the error, and we did this for two reasons. The first reason is that we wanted to make clear that all the considerations made in this section are independent of the choice of loss function. The second is that loss functions usually depend on the specific modelling technique; in later chapters we will therefore specify the loss function used in this work along with each of the chosen modelling techniques.

²⁹A circular error probable (CEP) of n meters means that 50% of the measurements have a distance of less than n from the true value.

Long Range Effects and Training Windows

In presenting the four classes of training methodologies, we suggested that the ones based on integration of the model (i.e. OE and FE) have the capability to capture some of the long range dependency in the dynamics that are neglected by the Markovian assumption, a suggestion that is worth expanding.

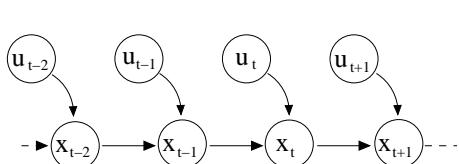
When discussing the concept of state, we introduced the Markovian assumption and explained how it is in practice only an approximation. For most real world systems, the state at time $t+1$ does not depend only on the state and input at time t (see Figure 2.4(a)), but to a lesser extent will also depend on other previous states (e.g. on the previous two, as shown in Figure 2.4(b)). While not doubting the usefulness of the simplification introduced by the Markovian assumption, it is relevant to ask ourselves if we can modify our training algorithms so as to compensate for higher-order dependencies, and therefore to improve the overall quality of our model.

Abbeel and Ng consider this question in [6] and propose an algorithm that instead of carrying out the model training using only the first-order transition probabilities (as would be optimal for a truly Markovian system), also explicitly considers higher order dependencies. More formally, considering a parametric model defined by its parameter array Θ , instead of finding the $\hat{\Theta}$ that optimises

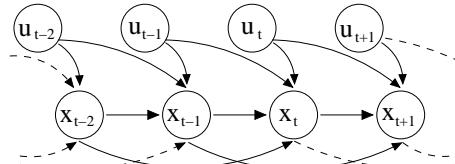
$$\hat{\Theta} = \arg \max_{\Theta} \sum_{t=0}^{T-1} \log p_{\Theta}(\mathbf{x}_{t+1} | \mathbf{x}_t, \mathbf{u}_t), \quad (2.18)$$

they propose to also take into account all the long range probabilities $p_{\Theta}(\mathbf{x}_{t+h} | \mathbf{x}_t, \mathbf{u}_t)$ up to a horizon H , each one being weighted by a discount factor γ^h

$$\hat{\Theta} = \arg \max_{\Theta} \sum_{t=0}^{T-H} \sum_{h=1}^H \gamma^h \log p_{\Theta}(\mathbf{x}_{t+h} | \mathbf{x}_t, \mathbf{u}_t), \quad (2.19)$$



(a) First order Markov system. Each transition depends only on the current state and input.



(b) Higher order (2nd) Markov system. Each transition depends also on past states.

Figure 2.4: Transition probabilities graphs.

where $\gamma \in [0, 1)$. They named this the *lagged error criterion*, and for systems with discrete state and action representations they also provided a training algorithm based on the EM (Expectation Maximization) procedure. As the same authors comment, an exact implementation of the algorithm, is prohibitively expensive in the case of continuous state and action spaces. However, for the specific case of learning the linear dynamics of a single rotor helicopter they suggest (in [5]) an approximate algorithm that makes the lagged error criterion effectively usable. In their example, since the model is linear, the identification boils down to identifying the model transition matrices³⁰ which are iteratively approximated by optimising the model prediction error over a series of consecutive time steps up to the horizon H . The model is effectively integrated forward over a time window so that the prediction error at time t will have effects also at future time steps. In their implementation Abbeel and Ng integrate the model forward H time steps for each of the data points, effectively evaluating the model error over a set of training windows covering the whole dataset³¹. The prediction horizon was empirically set to 20 time steps, equivalent to 2 seconds.

We can easily see that what the lagged error criterion is effectively doing: it is transforming the identification from an instance of the EE methodology³² to an instance of the OE method. The arguments made in Section 2.2.4 in favour of methods that compute the model error by propagating the model predictions are therefore shown to be not only intuitively but also theoretically sound.

The concept of training windows has also a second important role in the OE modelling setup. In reality no model can be expected to be perfect, therefore if during training it is integrated forward in time over a very long horizon, the accumulated error will tend to push the measured and predicted data series apart, potentially making them uncorrelated. When this happens, any error function based on the difference between the two time series will start to lose its value as an indicator to guide model optimization.

³⁰In the setting of Abbeel and Ng, the helicopter model is considered as being linear, therefore its state can be written in matrix form as:

$$\mathbf{a}_t = A\mathbf{x}_t + B\mathbf{u}_t + \boldsymbol{\omega}_t.$$

The generic non linear function f that we considered in equation 2.14 is replaced by the linear combination of the state and input variables with the coefficients given respectively by A and B . The model is therefore completely identified by the coefficients of the matrices A and B . In their case all the state variables and accelerations are assumed to be directly observable, and therefore the observation function g becomes the identity and does not appear as an additional equation.

³¹Each window overlaps with the following one on $H - 1$ points.

³²The lagged error criterion with a horizon $H = 1$ reduces to estimating the model parameters by least squares linear regression, which is possibly the most used of the EE methods.

Bongard and Lipson recognized the same problem in [27] when trying to identify the parameters of a four legged robot solely from sensor measurements. As a solution they devised a metric (the rolling mean) that in essence aims to compare sets of average sensor³³ activations over a short initial time period from a specific point in the sensor time series. In the same publication they showed experimentally how, even in situations like theirs where the initial models are very poor (i.e. random), such metrics allows the search to converge eventually to good models.

In the case of the FE method, since the state is propagated forward in time using a filter, and the collected experimental data are used within the filtering loop as measurements, there will be no significant divergence between measured and predicted data. Therefore for this type of methodology there is no practical limitation to the amount of consecutive data that can be used by the algorithm.

In the EE case, since no integration takes place, the concepts of horizon and window are not meaningful, and the whole dataset can be used during training.

2.2.5 Model Validation

Having described how to make the most of our data during the model training stage, we now look at a very important question in the field of modelling: how can we validate and measure the results of our efforts?

The value of the loss function³⁴ used for the optimization is a measure of the ability of the model to fit the training data. However, using the value of the loss function as a measure of the model's ability presents two problems. The first is that, as we explained above, different loss functions are appropriate for different training techniques, and therefore the loss obtained at the end of the training cannot be used directly to compare the modelling approaches. The second is that the model fitting process ensures that the accuracy of the model over the training data is as high as possible. Therefore using the value of the loss function as an indication of the model's likely performance on unseen data will probably give an overly optimistic estimate of the model's accuracy. A common way to get a more

³³In the work of Bongard and Lipson, the real and predicted quantities are sensor measurements but the same concept can be applied to any simulated variable for which there exists a real data counterpart.

³⁴To avoid confusion we want to state that the problem we are addressing in this section is model validation, which entails determining an appropriate error metric. This is different from the concept of a loss function, which is the objective that is optimized during the training. The same function (e.g. mean square error) could be used to fulfil both roles, potentially leading to confusion. In this work we will strive to keep the two concepts separated to avoid any misunderstanding.

realistic estimate of how the model would perform with unseen data ([159]), is to test the model's performance on a second dataset - a part of the original data that is set aside and not used during the training process. Such a dataset is commonly known as a validation dataset, while the portion of data used for training constitutes the training dataset.

Such a method of creating a validation dataset has the obvious drawback that a part of the data, and the information it contains, is never exploited in the learning process. As a consequence it might be necessary to collect more experimental data to ensure that enough data is available for both training and validation.

A standard way of making better use of the data ([159]) is to split the dataset in k parts called folds (hence the name k -fold), and to train the model k times, each training trial using $(k - 1)$ folds as the training dataset and the remaining fold as the validation dataset. The performance of the algorithm is then assessed as the average performance of the k models. This approach makes better use of the available data but requires repeating the training k times, which can be problematic for an algorithm in which the training is a particularly expensive operation.

Which of the two strategies is the most favourable depends in practice very much on the experimental settings and on the training algorithm used; we will discuss these considerations again in Section 3.2 when describing how we collected the experimental data.

The matter of choosing a metric to measure the difference between the state predicted by the model and the actual measured data is not straightforward. State variables differ in meaning, in units, and in the amount of noise they are associated with in the experimental data. For example, it is not immediately clear whether an error of $1m/s$ in linear velocity is as important as an error of $1rad/s$ in angular velocity. In addition errors may have widely different meanings when dealing with different types of platforms. As a consequence, no consistent criteria can be found in the literature that would allow the direct comparison of different models ([113]); however, it is interesting to analyze the most common metrics used for model validation.

In the system identification field, for the case of full scale helicopters, some authors ([230]) have proposed the use of weighting factors derived from practical experience to gauge the importance of the different error contributions. However it is difficult to understand how such coefficients could be equally meaningful for platforms such as the model

quadrotors and the toy car that we consider in this work.

In [113] Jategonkar uses estimation algorithms based on the principle of maximum likelihood and chooses the determinant of the covariance matrix of the residuals R as metrics to validate his models. More specifically

$$|R| = \frac{1}{N} \sum_{t=1}^N [\mathbf{z}_t - \mathbf{y}_t][\mathbf{z}_t - \mathbf{y}_t]^T, \quad (2.20)$$

where as seen in Section 2.2.1 \mathbf{y} is the model prediction, \mathbf{z} is the observation (the experimental data) and N is the number of samples in the dataset. This metric has the drawback that if the error in one of the variables³⁵ is particularly low, it can effectively compensate for the error in other variables and still deliver a low value for the determinant. In the extreme, a zero error for one of the variables would be sufficient to deliver the minimum value for the metric even if the error in the remaining variables is not zero. Fortunately, in practice no model would achieve a zero error, so this remains only a theoretical possibility. Jategonkar recognizes such limitations but pragmatically reports that, in his experience, using such a metric without any platform specific adaptation proved to be effective across different types of systems in the aircraft and rotorcraft identification domain.

A second possibility is to use the mean square error (MSE) a well known metric in the domain of signal fidelity ([244]) and machine learning ([193]) which is defined as:

$$MSE = \frac{1}{N} \sum_{t=0}^N (\mathbf{z}_t - \mathbf{y}_t)^2. \quad (2.21)$$

Since it corresponds to the sum of the square of the Euclidean distance between the model prediction y and the measurement z , its value will be zero only if the error in each of the variables is zero, avoiding the possibility of large errors on some variables being discounted because the prediction on others is particularly good. In the MSE the different variables have the same contribution to the square error, therefore no weighting is applied.

To provide a more detailed comparison of the outputs of a model, the MSE can also be computed for a single variable:

$$MSE_j = \frac{1}{N} \sum_{t=0}^N (\mathbf{z}_t^j - \mathbf{y}_t^j)^2, \quad (2.22)$$

³⁵ \mathbf{z} and \mathbf{y} are vectors constituted by several observed variables.

where j indicates one of the variables predicted by the model.

In order to provide a more intuitive meaning, the square root of the MSE (RMSE) is often considered:

$$RMSE = \sqrt{\frac{1}{N} \sum_{t=0}^N (\mathbf{z}_t^j - \mathbf{y}_t^j)^2}, \quad (2.23)$$

which is of course expressed in the same physical units as the variable in question.

Other possible metrics used to gauge the quality of models ([182]) include the TIC (Theil Inequality Coefficient) [226]. The expression of the TIC for the generic predicted variable j is:

$$U_j = \frac{\sqrt{\frac{1}{N} \sum_{t=1}^N (\mathbf{z}_t^j - \mathbf{y}_t^j)^2}}{\sqrt{\frac{1}{N} \sum_{t=1}^N (\mathbf{z}_t^j)^2} + \sqrt{\frac{1}{N} \sum_{t=1}^N (\mathbf{y}_t^j)^2}}, \quad (2.24)$$

which has value $U_j = 0$ in the case of a perfect fit and $U_j = 1$ in the case of maximum inequality. A single measure for the overall fit could be obtained by combining the values of the TIC from all the variables predicted by the model. This however leaves open the problem of determining which scaling factors are more appropriate for combining the contributions of the many predicted dimensions, something that is hardly possible to deal with in a platform agnostic way.

In Section 4.1.1, we will choose the most appropriate way to validate our models in the light of these considerations.

As previously discussed, when learning the dynamics of a system using the OE or FE methods, the prediction of the state derivatives is propagated forward in time; this means that any errors will accumulate and therefore will have an effect not only on the current prediction but also on the subsequent ones. As a consequence, the error computed on a single prediction step will not be representative of the true performance of the model, even for the purpose of validation. For this reason, it is standard practice in aircraft system identification ([113]), robotics ([121, 5]) and evolutionary system identification ([32], [31]) to integrate the model forward in time for several consecutive time steps in order to improve the computation of the error.

The same considerations that we brought into play when discussing our training windows scheme, about how the predicted and recorded time series tend to become uncorrelated, again apply. Over a very large number of time steps, the difference between the predicted and recorded state trajectories can eventually lose its value as an indicator of

model performance. This is again a problem that has to be addressed pragmatically when computing the validation error.

2.3 Previous Work in Vehicles Modelling

Flight vehicles

In this work we will test our modelling approach on data from two quadrotors and a fixed-wing aircraft (see Sections 3.3.1, 3.3.2 and 3.3.4), and so it is of interest to give a short overview of the field of flight vehicle modelling, from full scale to miniature flying machines.

Modelling, or as it is usually called in the aeronautics community, system identification, finds a highly successful domain of application in aircraft and rotorcraft. Thanks to computers, very accurate sensing devices and well understood underlying basic physical principles, system identification has evolved to become an essential tool for both aircraft design and certification. In fact, a model derived from flight data is most often, the only sure way to match the dynamics of the actual vehicle. When white box models fail to provide the necessary fidelity, system identification usually proves to be more time efficient than actually attempting to correct the physics based model to better match the flight data. This is even more applicable when the effects requiring correction are not easy to model.

Techniques for aircraft and rotorcraft system identification can be divided into two categories, frequency domain techniques and time domain techniques. Frequency domain techniques are essentially restricted to linear models³⁶, while frequency methods have proven their value when the dynamics allows for such a simplification (see [230] and references therein). Linear models have been found to be adequate for both aircraft and rotorcraft in the parts of the flight envelope where nonlinear effects are not predominant. In contrast, time domain techniques are suitable for both linear and nonlinear modelling, and therefore have been applied to a variety of problems in the domain of aircraft system identification. Models based on first principles are by far the most popular type. Since they allow for a complete breakdown of the system response into identifiable components, they deliver in-depth understanding. For example this is the kind of knowledge that is very valuable

³⁶While frequency domain techniques for linear models are a well established methodology ([136]), examples of their extension to nonlinear system have been introduced only recently (e.g. [134]). Such techniques are still a very active topic of research and we are not aware of any applications to aircraft or rotorcraft.

during the design stage.

There are, however, applications for which a model based on first principles is not needed. These include applications such as fault detection, sensor calibration or mission planning, as well as situations in which a model based on first principles would be very complex. All of those are well suited to black-box models. Among the function approximators, artificial neural networks are the most popular (see [113] for a comprehensive list of references), but neuro-fuzzy and symbolic regression approaches have also been used (see [148] and [74] respectively). Black-box techniques are mostly used as partial models dealing with specific dynamic effects or specific flight conditions. Knowledge about which inputs are relevant, or assumptions about the decoupling of dimensions, are often heavily exploited to simplify the training. In general, Black-box techniques are used as a pragmatic way to improve a model's agreement with flight data. A typical example is [148] in which a combination of neural networks and fuzzy rules is used to learn the model for a sideslip angle³⁷ virtual sensor for a full scale aircraft. Such a model reproduces only a specific component of the aircraft, and can be used to simulate a sideslip angle sensor from other sensor sources. The authors use their knowledge of the sensors and of the aircraft (i.e. the type and location of sensors and the aircraft-sensor coupling) to select the input variables and the neuro-fuzzy structure that are most appropriate. The resulting model is then trained to provide good sideslip angle prediction.

Parametric models based on first principles are also the de facto standard for model rotorcraft, which nowadays are widely used as unmanned robotic platforms. Starting with the seminal work of Mettler *et al.* ([152]), the standard system identification techniques used for full scale vehicles have been proved successful when applied to both large and small scale single rotor model helicopters (i.e. 3000mm to 300mm rotor diameter). Both frequency domain (i.e. linear models) and time domain approaches have delivered models with the fidelity necessary for control system design; of course, researchers often found it necessary to refine previously available model structures to match the platform at hand.

A very large number of quadrotor models is available in the literature [34, 98, 188, 15, 88, 239, 42, 183, 161, 52], all of which are derived from first principles. Most of the models mentioned consider only the thrust and torque associated with each rotor, which are the primary contributions that allow the control of the vehicle. [34, 98, 188] also consider

³⁷Angle between the aircraft centerline and the direction of the relative wind.

secondary aerodynamic affects affecting each rotor such as drag, ground effect and blade flapping³⁸; these are modelled using principles from helicopter theory. Estimating the parameters of the rotor dynamic model generally requires dedicated experiments using the motor-blade assembly. Each rotor is effectively treated as a subsystem, and its model is then used within the full quadrotor model. The implicit assumption is that combining the rotor models with the remaining dynamics based on parameters derived from static measurements will deliver a good full model. However, in none of the works mentioned is this assumption actually verified using experimental data and, to the best of our knowledge, the full estimation of the model parameter based on recorded flight data has not been reported.

In the area of miniature flight vehicles, only a few contributions consider the problem of automatically defining the model structure and parameters. In [245] Locally Weighted Linear Regression is used to model the vertical dynamics of a quadrotor as a stochastic Markov process, while in [13] the same technique is demonstrated to be suitable for modelling the decoupled dynamics of a single rotor helicopter. In [256] the transfer function form of a linear helicopter model is identified using a genetic algorithm. Ko *et al.* [122] present an interesting use of Gaussian process regression to learn the dynamic model of a small blimp. The advantage of this regression technique is that a variance expressing the model uncertainty is associated with each of the model predictions (i.e. at each timestep). This uncertainty is learned from the training data and can be used for obtaining a model that is not deterministic³⁹.

While the techniques and some of the concepts we will use in this work have already found application in the aircraft and helicopter domain with various degrees of success, the idea of devising a general method of building a full 6DoF rigid body model without the use of expert knowledge has not been explored, an aspect that makes our investigation both novel and interesting.

Car like vehicles

A further type of vehicle that we will consider in this work is a toy car; since it has a dynamic behaviour very different from that of our aircraft and quadrotors, it represents

³⁸The up and down movement of a rotor blade, which causes dissymmetry of lift (see [115] for more details).

³⁹The predictions from a Gaussian Process are made in terms of mean and variance of a Gaussian distribution from which is possible to sample, thus obtaining a non-deterministic model.

an interesting point of comparison for our algorithms.

Due to its poor manufacturing quality, its size, and the type of propulsion system used (see Section 3.3.3) our toy car is very different from its full-sized counterpart. These differences affect not only the form of the model but also how much of its dynamics is in practice predictable. For instance, in our toy car, friction with the ground (which is notoriously difficult to model), has (in proportion) a much larger effect on the dynamics than it would have in a full size car. Similarly, it is not easy to compare the repeatability of a full size car designed to meet precise manufacturing tolerances with a toy car using plastic gears and a loose steering mechanism essentially designed to be cheap. Consequently, the sort of very detailed models commonly used in the automotive industry are not appropriate for our vehicle, and neither is the in-depth platform knowledge that such models require.

We are clearly not the first to use model cars in the domain of robotics and control, but, since the basic dynamics of a car is well understood, most of the available work is based on parametric models derived from first principles or from well-established engineering formulations (e.g. Ackerman steering [61]). In such models, the expert's understanding of the car's dynamic behaviour as well as of its physical characteristics are heavily exploited, either by measuring the physical parameters of the car ([222]) or by making explicit simplifying assumptions. In [7] for example the lateral dynamics is considered negligible since the specific car used has good grip and does not tend to skid; in addition, the car has been retrofitted with a voltage stabilizer circuit in order to make the performance independent of the battery level.

Again our aim of not using platform knowledge but still delivering models that are physically consistent⁴⁰ constitutes a clear difference from the work reported so far. This opens challenges that have not been addressed before.

2.4 Summary

In this chapter we have provided an in-depth view on the two main research topics tackled in this thesis, namely modelling and automatic controller design.

The techniques and challenges that distinguish each of those tasks were identified and discussed with special emphasis on the aspects related to real world platforms as well as to platform knowledge, both of which are key to our work.

⁴⁰Model that predict meaningful acceleration and follow the laws of rigid body dynamics.

This extensive review of the state of the art in modelling and control of the three type of platforms involved in our study (car, aircraft and helicopters) helps the reader to understand where our work stands in respect of other learning based techniques, as well as in relation to more conventional engineering techniques.

Chapter 3

Platforms and Data Acquisition

In the previous two chapters we presented cardinal idea and the scope of the thesis. In this chapter we present in a unified fashion the platforms that were used in this work: two quadrotors (X3D and Owl), a toy car, an experimental aircraft (the ATTAS) and the Autopilot software simulator; we also give details of the data collection. These aspects are important in order to understand clearly the conditions under which our results are derived; they are reported here to avoid cluttering the presentation of the core parts of the research. We also need to touch on some of the principles of aircraft and rotorcraft; expert readers will understand that for reasons of brevity our treatment needs to be in some respects simplistic.

3.1 Flying Arena and Motion Capture System

All the experiments were carried out in our indoors robotics arena, a large research laboratory in the Computing and Electronics Systems department at the University of Essex. The facility (see Figure 3.1) has a cylindrical shape 12m in diameter and 6m in height and was purpose built for conducting flight experiments. It is equipped with a state of the art motion capture (MOCAP) system that allows tracking, streaming in real time, and recording the position of several tens of infrared passive markers¹. For rigid objects three or more markers rigidly connected can be used to define a reference frame; in this case the system will directly return the position and orientation of the object. In practice, to reduce the chances of hidden markers during data collection it is advisable to use more

¹We used a Vicon MX system (<http://www.vicon.com/products/viconmx.html>) composed of 8 Vicon MX 3+ cameras (240fps @ 0.3 Mpix full frame) and 2 MX ultranet units, one of which has an additional A/D converter module.

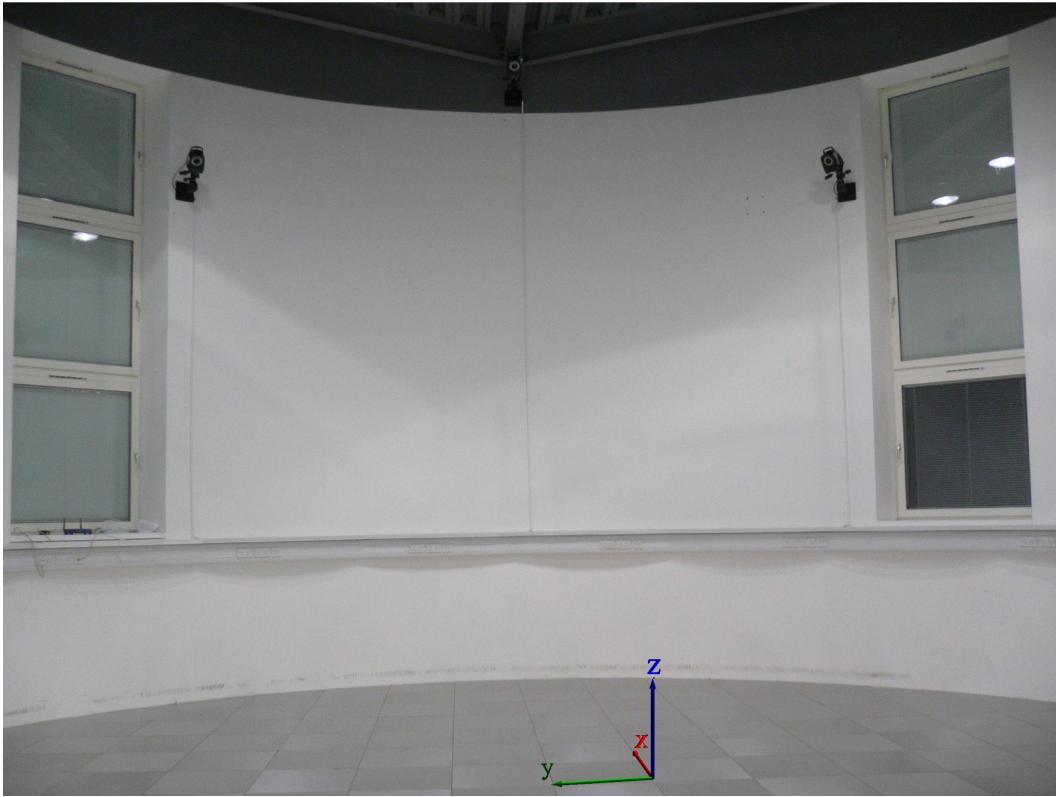


Figure 3.1: Flying arena and MOCAP infrared cameras. Note the superimposed *world* reference frame defined in the centre of the capture area.

than three markers for each of the objects being tracked; in our experience five markers carefully placed to avoid occlusion from the vehicle produce good tracking results.

For the purpose of modelling we need the ability to log the position and orientation of our vehicles, and also the command inputs that the vehicle is receiving from the pilot. For the automatic control experiments, we need the ability to retrieve in real-time the position and orientation of the vehicle, and to feed it to our controller which will produce an appropriate control input that is then sent back to the platform.

The software suite that accompanies the MOCAP system allows for the configuration of the many parameters that control the system², and most importantly provides a TCP port that can be used to stream in real time the position and orientation of the object(s) being tracked. We developed our own software to interface the real-time data stream, as this allowed us to define and use the same data format for both logging and real-time control, making for a seamless transition of our controllers to the real vehicles.

To read and output the pilot's control commands we designed two custom hardware

²In our experiments we used one of the configurations produced by the Vicon engineers at the time of installation of the system.

solutions to interface with the radio remote controllers that we used for our vehicles. In the case of the quadrotors, our interface (see Figure 3.7) connects to the remote through the trainer port³. This port offers a signal that encodes the sticks' positions as a $54Hz$ PPM (pulse position modulation) pulse train; our microcontroller decodes the PPM sequence and transmits it to the host computer using a serial port link. A second signal line present on the same port is directly connected to the remote RF stage so that the inverse route from serial data to a PPM signal can be used to send control commands to our vehicles. We encode the PPM signal as a 10bit digital value, which ensures a resolution smaller than the level of noise normally present on the PPM signal itself. For the toy car, since its remote does not have a trainer port, we built a simpler parallel port interface (see Figure 3.12) that taps into the electronic board of the remote. In this case, since the controls are discrete⁴, the signals from the parallel port can be used to drive the remote directly without the need for an additional microcontroller.

With both these interfaces the host computer sends and receives control data at 100Hz; the control data coming from the remote is logged along with the stream coming from the MOCAP system.

The quadrotors and the toy car have electric motors powered directly from the main battery of the vehicle, therefore as the battery discharges and its voltage changes, the power generated by the motor for a defined throttle setting also changes. Since the battery level directly affects the behaviour of our vehicles, we treat it as an additional input that also needs to be logged. To record the battery level a XBee-PRO RF module [108] was used. This module is a light and low power Zigbee communication solution, capable of delivering serial communication with a maximum speed of 115200bps over an indoor range in excess of 100 meters. This unit can be configured to periodically sample up to 5 analogue channels and send its reading to a remote receiver. The A/D converter on board the XBee-PRO module has a 10bit resolution, an accuracy that easily meets our needs. The module was set up to sample the battery voltage at 50Hz and the serial stream at the receiver end was simply logged along with the control data coming from the remote. Since the control and MOCAP data are logged at 100Hz, each sample of the battery voltage is effectively kept constant for two consecutive time steps. Any transmission delay introduced by the Zigbee

³A port normally used to connect the main remote (called teacher) to a second remote (called student) for training. With this setup, at any point in time the teacher remote retains the ability to override the controls coming from the trainer port.

⁴See Section 3.3.3 for more details.

communication is small compared to the rate at which the battery voltage changes in our vehicles, therefore we can safely neglect it.

Required Data

To simplify the steps of data logging and controller testing we run the MOCAP tracking software and our custom made logging and control software on a ground station computer in all our experiments⁵.

For each platform, the data collection provides the necessary data for both the training and the validation of the dynamic models.

In Section 2.2.5 we described the two possible approaches for conducting the validation of our models, which define how much data we need to collect. In making a choice, we need to consider that some algorithms (those based on evolutionary computation) have a training procedure that is computationally expensive, and as a result the k -fold cross validation method is not advisable. Fortunately, we are in control of the data collection stage, and so the requirement of not using all our data for training can be overcome simply by collecting more data in the first place. In general, since only a few minutes worth of data are necessary for our algorithms, acquiring more data merely requires that we fly our helicopter or drive our car for only a few more minutes. For all platforms we will therefore collect at least two datasets, the first used for training, and the second used for validation. In the case of the ATTAS aircraft, as we will see in Section 3.3.4, two distinct repetitions of the same flight manoeuvre are available, and so one can be used for training while the other is reserved for validation.

3.1.1 Reference Frames

Before looking in detail at the data pre-processing, let us describe more precisely the format in which we encode the data returned by the MOCAP for the purpose of real-time control and logging.

At calibration time⁶ a fixed origin and coordinate frame needs to be set for the capture volume; for our flying arena we have chosen a right handed NWU (North, West, Up) coordinate frame with the origin in the middle of the circular floor (see Figure 3.1). We

⁵As a ground station we use a dual core 2.66GHz Intel Xeon desktop computer with 2GB of RAM running Windows XP. Its performance is more than adequate to run simultaneously the MOCAP software and our custom logging and/or control software.

⁶Generally, on every day in which we run experiments, we start by calibrating the system.

call this the *world*-fixed reference frame. After defining a NWU *body* reference frame for an object⁷, the position (x, y, z) of the object and its orientation in Euler angles (ϕ, θ, ψ) are well defined with respect to the *world*-fixed frame. We prefer to represent orientation by means of Euler angles because their meaning is very intuitive⁸; however several alternative conventions exist for Euler angles. Following common practice in the aeronautics literature, we used the ZYX convention, which defines the following steps to transform the *world*-fixed coordinate frame into the *body*-fixed frame:

- right-handed rotation about the z -axis (positive ψ , see Figure 3.2(a))
- right-handed rotation about the y' -axis (positive θ , see Figure 3.2(b))
- right-handed rotation about the x'' -axis (positive ϕ , see Figure 3.2(c)).

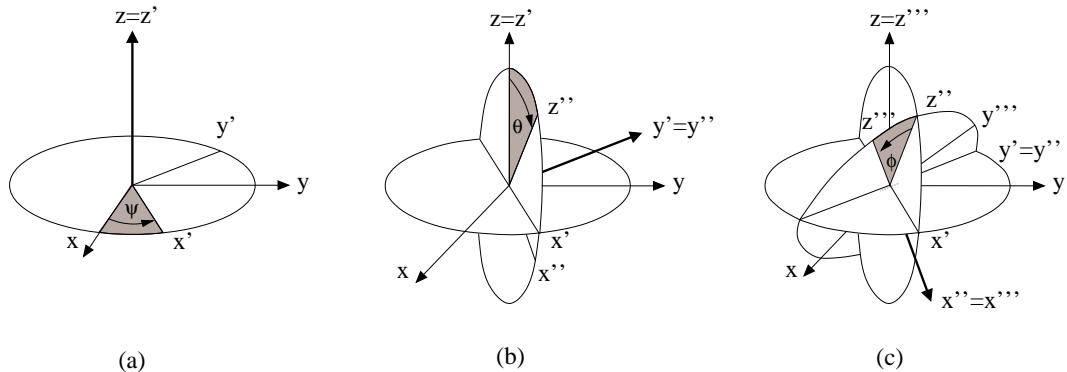


Figure 3.2: Euler ZYX rotations that define the attitude of a vehicle w.r.t. the *world* coordinate frame. The transformation is represented by a sequence of three different rotations. (a) a positive rotation of ψ about the z -axis transforms the reference frame xyz into the frame $x'y'z'$; (b) a positive rotation of θ about the y' -axis transforms the reference frame $x''y''z''$ into the frame $x'''y'''z'''$; (c) finally a positive rotation of ϕ about the x'' -axis transforms the reference frame $x''y''z''$ into the body frame $x'''y'''z'''$.

Each of the rotations can be expressed in the form of an orthogonal matrix; the product of the three defines the DCM (direct cosine matrix) that allows the transformation of coordinates from the *world*-fixed frame to the *body*-fixed frame:

$$\mathbf{C}_b^w = \begin{bmatrix} c(\theta)c(\psi) & c(\theta)s(\psi) & -s(\theta) \\ s(\phi)s(\theta)c(\psi) - c(\phi)s(\psi) & s(\phi)s(\theta)s(\psi) + c(\phi)c(\psi) & s(\phi)c(\theta) \\ c(\phi)s(\theta)c(\psi) + s(\phi)s(\psi) & c(\phi)s(\theta)s(\psi) - s(\phi)c(\psi) & c(\phi)c(\theta) \end{bmatrix}. \quad (3.1)$$

⁷For each of our robots we have defined the reference frames shown respectively in Figures 3.10,3.7 and 3.12

⁸In our experiments the attitude of the platforms never reached the situation of gimbal lock.

Often we use the word 'pose' to mean both the position and orientation of the vehicle, or more precisely the vector:

$$[x, y, z, \phi, \theta, \psi]^T.$$

3.1.2 Accuracy

In the case of tracking a rigid body the accuracy of the MOCAP system varies depending on the number of markers used to define the object, and also on the position of the object within the tracking volume⁹. The attitude of the object is computed geometrically from the positional data of several of the markers; its precision therefore depends not only on the precision of the system but also on the relative positioning of the markers. All our robots are small, and the distance between the markers goes from 50mm to 180mm at most.

To establish the precision of the tracking system, we did some static experiments during which we positioned one of our vehicles (i.e. the toy car) in three different locations within our tracking volume. In Figure 3.3 we see the histograms of the position and angle errors obtained from 5000 data points collected with the toy car positioned on the ground close to the edge of the tracking volume. For all three dimensions the distribution of error is

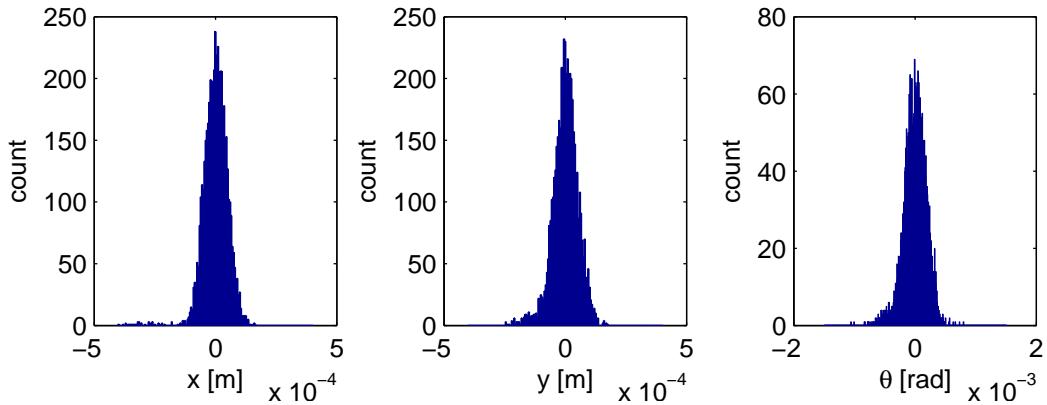


Figure 3.3: Histogram of the position and heading error distribution computed from a static experiment that lasted 50s.

unimodal; very similar histograms were also produced for the remaining locations. Slightly different standard deviations for the errors were obtained at different locations. To produce a conservative estimate of the error, we consider the positions and angles produced by the tracking system to have the maximum standard deviation of these computed at the sampled

⁹The position of the object within the tracking volume determines the number of cameras simultaneously tracking the same marker.

locations. This translates to a positional standard deviation of $0.00012m$ and an angular standard deviation of $0.0003rad$.

3.1.3 Delay Estimation and Compensation

Given the amount of computation involved in resolving the position of the tracked markers, the data stream is inevitably affected by a time delay. For the vehicles we used, we will see that this delay is short enough to allow for good control; however, it has to be correctly accounted for during both data logging and automatic controller design.

To estimate the delay of the incoming tracking data stream (Δt_{MOCAP}), we exploited the fact that in the MOCAP system the data coming from the sampling of the A/D channels is time aligned (in hardware) with the tracking data. To estimate the delay, we conducted a simple experiment in which we simultaneously logged pilot's commands using both our custom interface and the A/D input of the MOCAP system; we then compare the two time series (see Figure 3.4). Our analysis revealed a constant delay $\Delta t_{MOCAP} = 140ms$ between

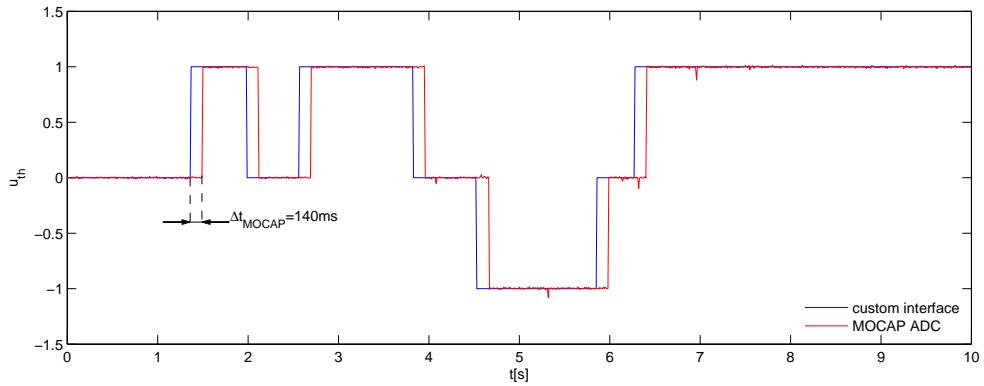


Figure 3.4: Delay between inputs logged by the MOCAP system and our custom electronic board.

the two time series; this delay is compensated for when the data from the MOCAP system and from our custom interface is combined and logged.

It is worth noting that if the control or logging software is not run on the same machine that is running the MOCAP software, network delays (which are often variable) will add to Δt_{MOCAP} significantly affecting the quality of the collected data.

3.2 Data Preprocessing

In Chapters 4 and 5 we will apply OE and EE techniques to model the vehicles, and we will investigate the use of models that predict accelerations (equations 2.8-2.11), and also of models that predict velocities (equations 2.5-2.7). It is therefore necessary to derive the body frame velocities and accelerations¹⁰ from the logged position and attitude information.

The real data obtained from the tracking system is noisy, and in the case of both the helicopters and the toy car, the variables most affected by noise are the angular ones (see Figure 3.5(a, b) and Figure 3.6(a, b)). As is commonly done, ([84]) our first step of pre-processing is to apply a low pass filter to reduce the noise content in the data. This procedure ultimately aims at improving the quality of the acceleration and velocity estimates that we will compute by double differentiation. Inspection of the PSD (power spectral density) of the collected data (see typical examples in Figure 3.5(c) and Figure 3.6(c)) shows that the majority of the power of the signal is at frequencies below 4Hz. This allows us

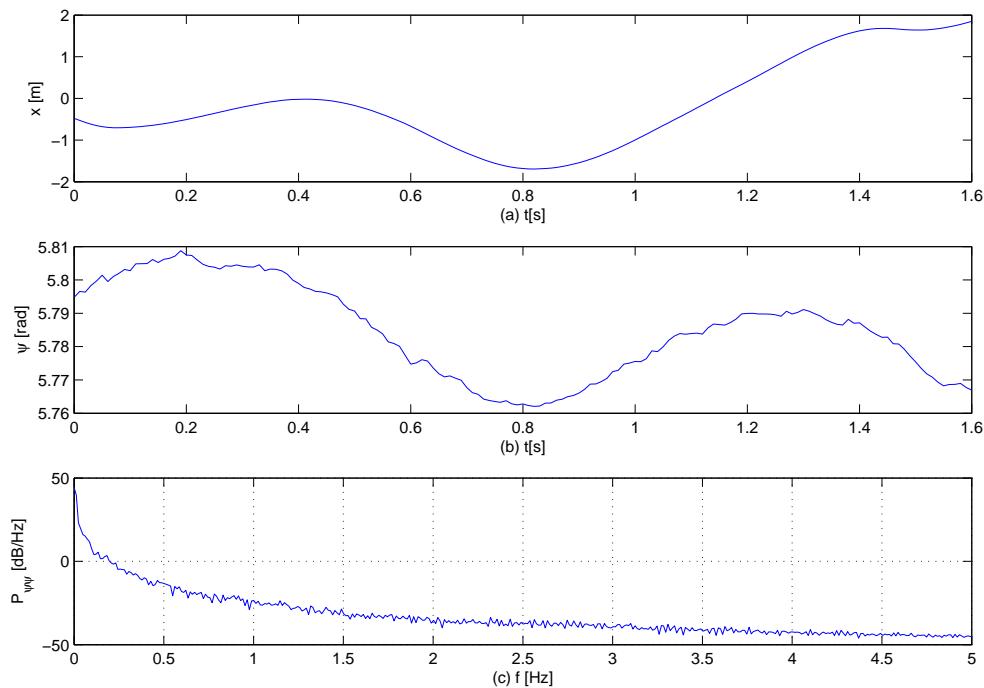


Figure 3.5: Example time series from one of the toy car datasets. (a) a segment of position data in the time domain (x); (b) a segment of heading data in the time domain (ψ); (c) power spectral density for the heading variable ψ computed over the whole dataset (note how the signal power content is concentrated at low frequencies).

¹⁰The way in which the acceleration and velocity data is used during the modelling will be clarified in Chapters 4-6.

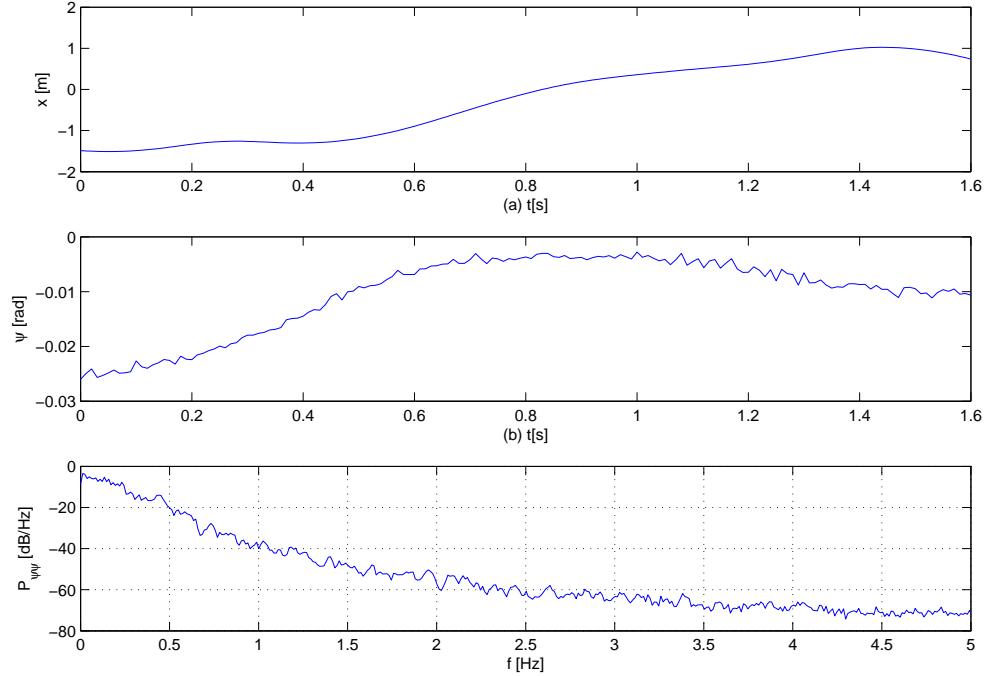


Figure 3.6: Example time series from one of the X3D datasets. (a) a segment of position data in the time domain (x); (b) a segment of heading data in the time domain (ψ); (c) power spectral density for the heading variable ψ computed over the whole dataset (note how the signal power content is concentrated at low frequencies).

to safely low pass filter the collected data without the risk of removing signal components that form part of the dynamics of our vehicles¹¹. To avoid introducing phase distortion, we perform the filtering using a finite impulse response (FIR) filter; in addition, to avoid amplitude distortion, we choose a filter with 201 taps to ensure a flat transfer function in the pass band. The delay introduced by the filter is compensated by shifting the filtered signal¹².

As is common practice in the machine learning community, we also “centre” the input time series (\mathbf{u}) by subtracting the steady state values of the controls (i.e. terms \mathbf{b}_u in equation 2.14)¹³.

Both when training our models, and also when using the models for controller evolution, we need to integrate their state equations forward in time at each time step. This operation accounts for a large majority of the computational cost incurred by our training

¹¹Later on in the section it will be made clear what cut-off frequency was chosen.

¹²For a linear phase filter the delay is equal to $(N - 1)/2$ where N is the number of coefficient of the filter ([178]).

¹³For the X3D and the Owl \mathbf{b}_u corresponds to the value of the controls when hovering. For the toy car these are the controls when the car is at rest. In the case of the ATTAS no processing was applied since the control signals in the available data set are already centred. For the battery input u_{ba} the nominal values of 10V and 4V were used for the helicopters and for the toy car respectively.

algorithms. Given the limited bandwidth of our system, it is possible to reduce the amount of computation required to propagate the models forward simply by increasing the time step used for integration (or equivalently by downsampling the collected data). In this way for a lower number of iterations, and therefore less computation, is needed for propagating the model forward of specified time. In choosing the downsampling rate we make a trade off between reducing computation and maintaining our ability to reconstruct the pose of the vehicle from the computed acceleration data. Since we will compute velocities and accelerations numerically as first order differences, we implicitly assume that the acceleration is constant between successive time steps; therefore, the longer the timestep is, the larger will be the error between the reconstructed and real trajectories. In making this trade off between computation and accuracy, we decided to accept a reconstruction error that was small in comparison to the error of the models; in other words, we wanted to ensure that when evaluating the error of our models, the error introduced by the time discretization was negligible. After conducting a small number of preliminary modelling experiments, we defined as acceptable an error that was lower than 1% of the signal amplitude for all the variables in the pose¹⁴ within a window of 300 time steps. For all of four platforms, a downsampling factor of 4 guarantees an acceptable reconstruction error; this brings the sampling frequency to 25Hz ($\Delta t = 0.04s$). Coincidentally the ATTAS dataset was also already downsampled to 25Hz, so that after the differentiation this dataset also has the same time resolution as those we collected.

Downsampling can introduce aliasing if the bandwidth of the sampled signal is larger than the Nyquist frequency ([178]). To avoid aliasing it is sufficient to ensure that the bandwidth of the low pass filter that we use for denoising is lower than the Nyquist frequency f_N , which in our case is:

$$f_N = \frac{1}{2\Delta t} = 12.5\text{Hz},$$

so we chose a cut-off frequency of 12Hz.

After the downsampling, we compute the first and second order derivatives, as first order differences scaled by the time step Δ_t (see Section B.4.1). During the computation of the derivatives it is necessary to perform changes of coordinates between the various

¹⁴In Chapters 4 and 5 it will be clear that this level of error is certainly negligible in comparison to the accuracy of the models we obtain. Since the error produced due to time discretization increases over time when integrating the accelerations, the acceptable error needs to be defined in reference to a meaningful time window. We have chosen a window of 300 steps since, as we will see, this is the window size used in most of our experiments.

frames of reference, since the body frame changes as the vehicle changes its orientation. For the ATTAS data, we used several datasets from [113] which already provided the linear and angular velocities, and so only one differentiation was needed to obtain the body frame accelerations.

Computing the body frame accelerations only yields to the dynamic acceleration of the platform, so gravity is not included in the obtained acceleration terms. Since gravity is explicitly included in the case of our first principle models (Section 4.1), we must add its contribution¹⁵ to the data computed from the differentiation.

The vehicle's attitude combined with the first order derivatives computed by differentiation constitute the vehicle's state (equation 2.4). Therefore, after the preprocessing, all the state variables are available for every time step of collected data; as a result, the observation function g in equations 2.5, and 2.7 is simply the identity function. The modelling task is therefore reduced to identifying the function f .

The data preprocessing discussed in this section is essentially the same for all the vehicles in this study, so we can safely say that such preprocessing is not biasing our approach toward any specific type of vehicle, in complete accordance with the idea of limited platform knowledge proposed in this work.

3.2.1 Differentiation Noise

An important point must be made about the way in which the velocities and accelerations are computed. Since we are using numerical differentiation to compute the state derivatives, the differentiation will inevitably tend to amplify the noise present in the pose data. If the noise in the original data is high, the differentiation noise can completely overcome the signal (i.e. the derivatives) we are interested in. This is a well known problem and [43] offers an extreme example.

The FIR filtering that we apply for de-noising attempts to alleviate this problem by smoothing the collected pose data. However, the signal to noise ratio in the numerical derivatives ultimately depends on the amplitude of the derivative as well as on the noise, and it is therefore not only platform but also dataset specific. Without the use of domain knowledge or additional data sources¹⁶ it is difficult to ascertain the quality of the computed

¹⁵Given the vehicle attitude, it is simple to re-project the gravity vector into body coordinates that can then be added to the acceleration computed from the dataset.

¹⁶We can imagine that if direct acceleration measurements (for example from an on board accelerometer)

derivatives. Under these circumstances, there is a possibility that any attempt at learning such numerically computed derivatives (e.g. for the purpose of modelling) might not lead to learning the correct dynamics of the system in question, since any acceleration information may have been compromised by the differentiation process.

These considerations apply to both the first and second order derivatives. In our experience, thanks to the accuracy of the MOCAP system, the differentiation noise is not problematical for the first order derivative for the vehicles considered in this work. It would therefore appear sound to prefer a model that predicts velocities (see equation 2.5-2.7) to a model that predicts accelerations. However, this will inevitably neglect the fact that a model that predicts velocities may be more complex as discussed in Section 2.2.1. Proceeding pragmatically, we will experiment with both possibilities. In Chapter 4.2 we will use EE methodologies to learn models that predict accelerations, and also models that predict velocities. The models that predict accelerations will be trained using the pre-computed accelerations, therefore their quality will depend on the quality of the acceleration data. In Chapter 5, we will instead apply the OE methodology to learn models that predict accelerations but that are trained using their error in predicting the system state (i.e. the velocities); these models will therefore depend on the quality of the velocity data.

3.3 The Platforms

In this section we introduce the four platforms that we used in this work, the X3D and Owl miniature quadrotors, the toy car, the ATTAS full sized aeroplane and the Autopilot miniature helicopter simulator. Along with the many technical details of each vehicle, we will also highlight any characteristics that are likely to have an influence on the automatic modelling, and/or on the automatic design of controllers.

3.3.1 The X3D Quadrotor

The X3D quadrotor is the platform on which most of the initial development of the algorithms was carried out.

The X3D (Figure 3.7), commercially available as a kit ([78]), is a light and small quadrotor (details in Table 3.1). Its main structure consists of a central magnesium alloy cage to which are attached the four arms. At the end of each arm is a brushless motor were available, these could be fused with second order derivatives of the pose to provide a better estimate.

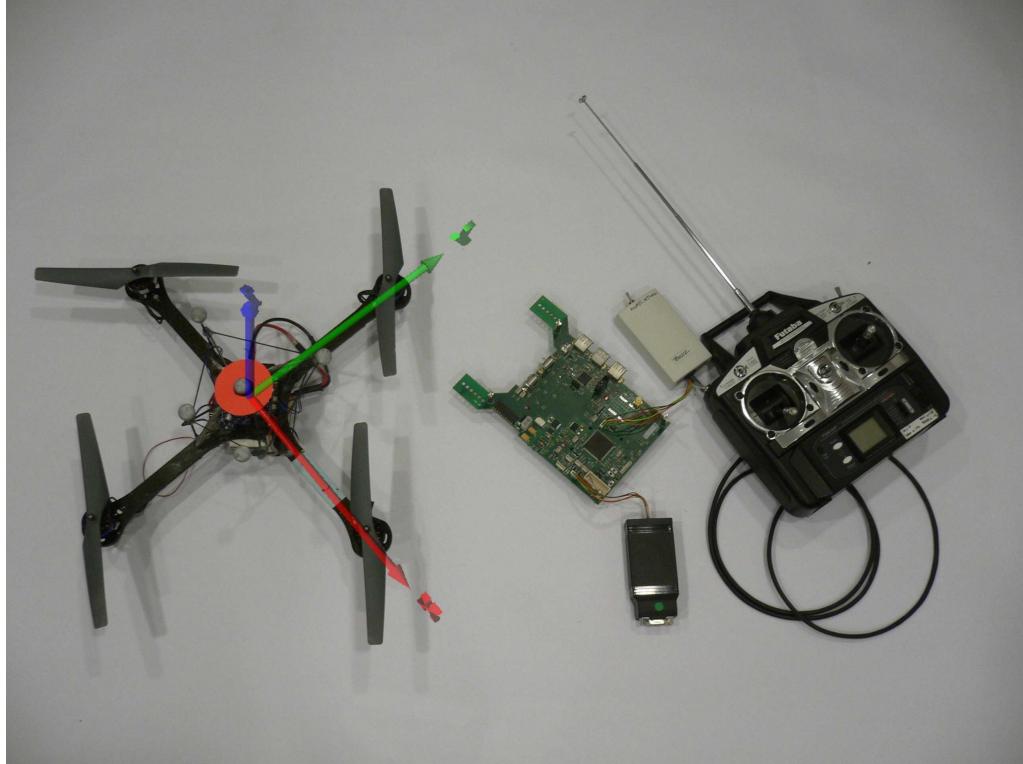


Figure 3.7: Commercial X3D quadrotors with remote controller and custom designed serial port computer interface. On the helicopter are visible the five spherical reflective markers used for tracking. The custom made electronic board is connected to the remote through the trainer port.

size	540x540 mm
weight	380g
propellers	200 mm 2 blades
battery	1300 mAh @ 11.1V
endurance	15 minutes

Table 3.1: X3D main physical characteristics.

on which a fixed pitch propeller is mounted. Both the electronics and the battery are installed inside the central cage. To give a good weight distribution and maintain the CoG (centre of gravity) approximately at the centre of the vehicle¹⁷, the flight control system electronics is placed above the rotor plane, and the battery is placed below it. Brushless motors have been selected in this design for their ability to provide more torque than conventional brushed motors, and as a result no reduction gears are needed, reducing both the mechanical complexity and the amount of energy dissipated as friction. The type of blades used can actually bend and to a limited extent also twist; both types of deformations

¹⁷This is known to improve controllability [188].

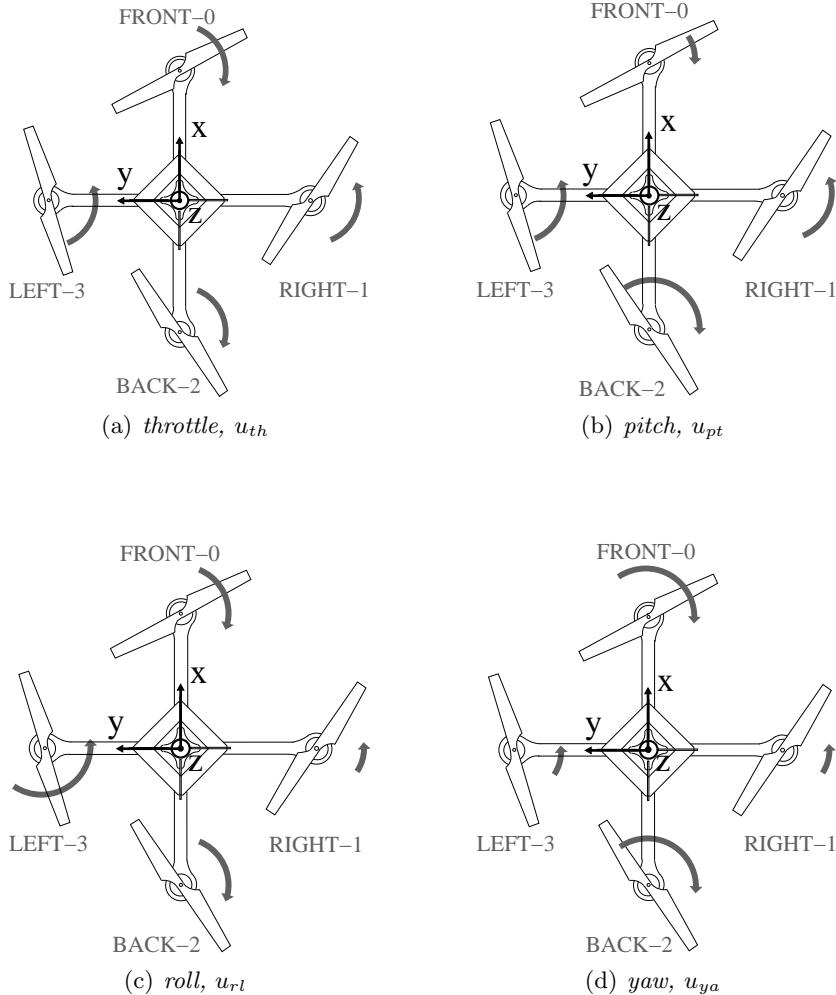


Figure 3.8: Quadrotor control principles: a) u_{th} governs simultaneously the speed of all rotors allowing to control altitude, b) u_{pt} governs the speed of front and back rotors to control the pitch rotation about the y axis, c) u_{rl} governs the speed of left and right rotors to control the roll rotation about the x axis, d) u_{ya} governs simultaneously the speed of the front and back rotors and of the left and right rotors to control the yaw rotation about the z axis; (the length of the arrows is proportional to the motor speed).

change the pitch of the blades, making them less efficient but also making for a complex dependency between rotor speed and the generated thrust and drag.

In their standard embodiment, electrically powered quadrotors are designed with fixed pitch propellers, and changes in thrust are achieved simply by controlling the speeds of the four motors, two of which rotate clockwise while the remaining two rotate anti-clockwise. The principle of operation of a quadrotor is straightforward: in steady state hover the four motors run at the same speed and produce a total thrust force that balances out the gravity force. Simultaneously increasing or decreasing the speed of all rotors (Figure 3.8(a)) produces a resulting force that allows the control of the vertical motion of the flight machine. Any difference in the thrust of two opposing rotors will produce a resulting torque

and therefore a rotation about the x and/or y axis. For example the flying machine can be made to pitch forward by decreasing the speed of the front rotor while increasing that of the rear rotor (Figure 3.8(b)), or to roll by adjusting similarly the left and the right rotors (Figure 3.8(c)).

Each rotor generates not only a thrust force but also a reaction torque as result of the blades' drag. The reaction torque has a direction opposite to the direction of rotation of the blade so in steady state hover those torques normally balance out. To control the yaw motion it is sufficient to speed up the two motors that are running clockwise while slowing down those running counter-clockwise (see Figure 3.8(d)). Although not rigorous, this explanation of the quadrotor flying principles will suffice until Chapter 4 where we will see a more complete description in terms of the physics that governs quadrotor flight.

In a real quadrotor, manufacturing tolerances and aerodynamic turbulence mean that it is impossible to achieve the correct force balance needed for stable flight. To address this problem, quadrotors are commonly provided with some form of stability augmentation. The standard X3D kit includes a circuit board that uses MEMS gyros to provide such stabilisation. The augmentation systems works by employing a simple feedback loop for each of the three principal axes of the machine (x , y , z), and maintains a rotational speed proportional to the respective control command for each axis. In [89] Gurdan *et al.* describe how such a system was designed for the X3D.

Thanks to the stability augmentation system, a trained human pilot finds it not too difficult to keep the X3D airborne, but controlling it in a precise manner turns out to be very challenging. First, since the controls are proportional to the rotational speeds around the principal axes, the helicopter does not return to level flight if the controls are released; instead the machine maintains its current attitude and therefore keeps moving in the current direction, a rather unintuitive behaviour. Second, the effect of any command is lagged due to inertia¹⁸. For example to hover at a precise altitude, the throttle has to be controlled precisely, but since any throttle change has a lagged effect, it is in practice difficult for a human pilot to achieve an accuracy of more than 30cm.

Since the throttle setting is not controlled by the stabilization system, the pilot finds himself having to actively compensate for the fact that the battery is discharging by increasing the throttle as the battery depletes. Any automatic control will obviously need to

¹⁸The inertia of the system is the result of the mass of the platform, its moment of inertia and the fact that due to drag and inertia the blade can not change speed instantly.

be able to replicate this ability in order to keep the flying machine aloft for any prolonged amount of time.

The X3D is light, powerful and extremely robust. We have crashed it against walls or the floor several times without any major damage, making it the perfect choice for our initial development, especially when testing some of our concepts and algorithms before applying them to the more expensive and delicate Owl quadrotor in Section 3.3.2.

Datasets and State Definition

The X3D has been our main workhorse, and during the development of the various parts of our system many datasets have been recorded, but only a selection of them will be used throughout the whole thesis. The datasets selected are not better or significantly different from any others; we simply need to choose a set of data in order to better compare our various methodologies, and support meaningful conclusions.

For the X3D we define \mathbf{x} to be the state vector at time t , which consists of the six velocity variables and the three Euler angles¹⁹:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T.$$

For each entry in our datasets we also have the array of inputs $\mathbf{u} = [u_{th}, u_{ya}, u_{rl}, u_{pt}, u_{ba}]^T$ and the pre-computed linear and angular accelerations $\mathbf{a} = [a_x, a_y, a_z]^T$ and $\boldsymbol{\alpha} = [\alpha_x, \alpha_y, \alpha_z]^T$. Although even a basic understanding of quadrotor flight principles tells us that its dynamic behaviour is independent of the heading angle²⁰, we maintain it in our state since neglecting it would count as a form of platform knowledge.

As explained in Section 3.2.1 we will be looking at two ways of modelling the X3D dynamics, the first based on predicting velocities and the second on predicting accelerations. In the case of modelling in velocity space, the variables of the state vector that are velocities will be directly predicted by the function f , while the remaining variables will be obtained by integration. In this case the general state update equation 2.16 becomes:

$$[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T = f(\mathbf{x}_t, \mathbf{u}_t) \quad (3.2)$$

$$\boldsymbol{\Phi}_{t+1} = \boldsymbol{\Phi}_t + H(\boldsymbol{\Phi}_t)\boldsymbol{\omega}_t\Delta t. \quad (3.3)$$

¹⁹For clarity of notation we are dropping the explicit reference to time.

²⁰Without any difference in the environment is indeed unreasonable to think that the helicopter will respond differently to our control inputs when heading north, then for example when heading east.

where $\mathbf{v} = [u, v, w]^T$, $\boldsymbol{\omega} = [p, q, r]^T$, $\Phi = [\phi, \theta, \psi]^T$, $H(\Phi_t)$ are the Euler kinematics equations defined in equation B.5 and Δt is the integration time step. In this case only equation 3.3 is an integration.

When modelling in acceleration space, the function f will predict linear and angular accelerations in the body frame:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (3.4)$$

and the state update equations become:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{C}_{b_t+1}^{b_t} \mathbf{a}_t \Delta t \quad (3.5)$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \boldsymbol{\alpha}_t \Delta t \quad (3.6)$$

$$\Phi_{t+1} = \Phi_t + H(\Phi_t) \boldsymbol{\omega}_t \Delta t, \quad (3.7)$$

where $\mathbf{C}_{b_t}^{b_{t+1}}$ is the DCM (direct cosine matrix) between the body frame at time t and that at time $t+1$. In this case all the three equations are integrations.

In Chapter 6 we will need to make predictions about the full 6DoF pose of our vehicle, and so we will need to use an *extended* state that also contains the 3D position $\mathbf{p} = []^T$. The *extended* state is defined as:

$$\mathbf{x} = [u, v, w, p, q, r, x, y, z, \phi, \theta, \psi]^T. \quad (3.8)$$

To compute \mathbf{p} we simply integrate the velocities \mathbf{v} after an appropriate change of reference from body frame to world frame²¹:

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \mathbf{C}_{w_t}^{b_t} \mathbf{v}_t \Delta t. \quad (3.9)$$

Two different dataset were collected for the X3D quadrotor (`datax3d_2` and `datax3d_3`); both were collected by manually flying the quadrotor within the capture volume. No specific manoeuvres were flown, but instead the pilot tried to cover the range of speeds and angular orientations that the helicopter could safely achieve within the confined space of our flying arena (see the portion of one of the datasets shown in Figure 3.9). The X3D is

²¹Using the DCM $\mathbf{C}_{w_t}^{b_t}$.

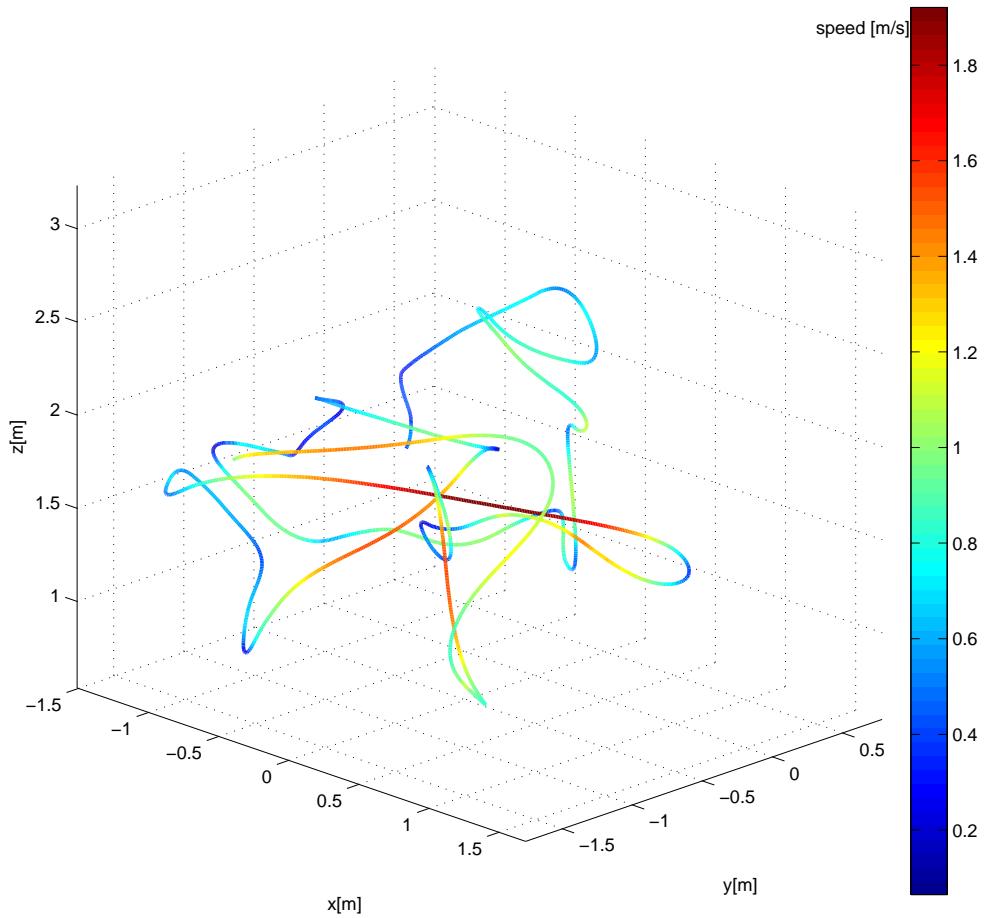


Figure 3.9: Short section (40s) of data from one of the X3D datasets. A variety of manoeuvres were executed, the colour of the path indicates the speed of the X3D.

in fact capable of high flight speed and extreme manoeuvres (e.g. multiple flips about any of its axes), but these are clearly not achievable in our limited space. While many of the manoeuvres flown involve the use of only a subset of the control inputs, those manoeuvres are generally not longer than a few seconds; as a consequence any subset of the data that involves at least a minute of flight can be considered to be representative of the system dynamics. The datasets do not include take-off or landing phases since any interaction with the ground (e.g. through the landing gear) is in effect an external input to the system that with our setup we cannot measure. To ensure good position tracking from the MOCAP system we deliberately avoided flying at the edges of the capture volume²² where only a limited number of cameras would be able to see the IR markers. Keeping a distance of approximately 1m from the walls and from the ground also ensured that any ground effects were minimized. Two flights were recorded: log `datax3d_2` consists of 15616 samples and corresponds to 624.64 seconds worth of continuous flight, while log `datax3d_3` is slightly

²²The walls and ceiling of our robotic lab.

longer with 16516 samples for a total of 660.64 seconds.

3.3.2 The Owl Quadrotor

The Owl is the working name of the second quadrotor platform that we use; in some respects it can be considered as the big brother of the X3D since it shares many of its design ideas. The Owl is not a commercially available vehicle; it is the result of a collaboration between the University of Essex²³, Swarm Systems Ltd ([220]) and the University of Surrey²⁴ aimed at taking part in the Ministry of Defence Grand Challenge ([160]). During the summer of 2008 a swarm of 14 Owls was designed and built for the competition, in which our entry was awarded of the prize for the most innovative idea.

While the X3D is essentially a RC toy, the Owl is designed to be a fully autonomous robot. Looking at Figure 3.10 it is clear that the arms and the central body of the machine are essentially scaled up versions of those one of the X3D. A bigger space is needed in

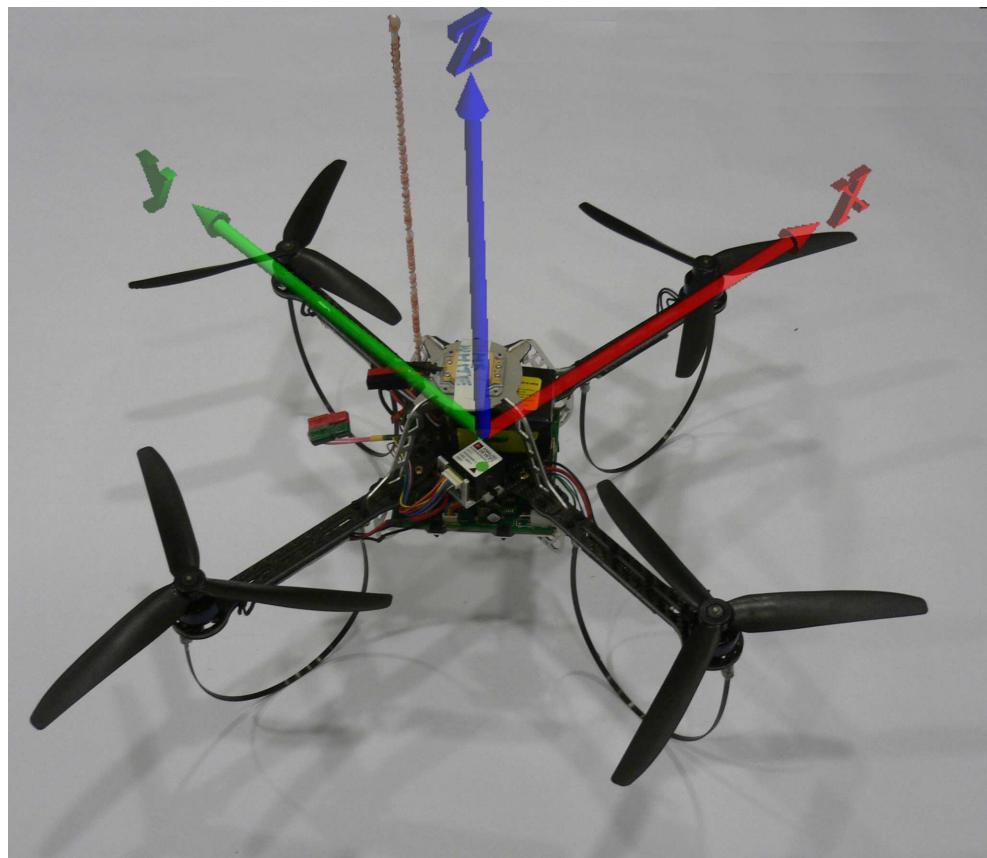


Figure 3.10: The Owl research quadrotor designed and built in collaboration with Swarm Systems Ltd.

²³The Essex team consisted of Prof. Owen Holland, Richard Newcombe and the author.

²⁴The Surrey team was led by Dr Krystian Mikolajczyk.

size	650x650 mm
weight	725g
propellers	230 mm 3 blades
battery	4000 mAh @ 11.1V
endurance	19 minutes

Table 3.2: Owl main physical characteristics.

the central body in order to host the larger and heavier electronics, and longer arms are necessary to mount larger and more efficient propellers. To keep the weight down, cut outs in the form of slots and holes have been designed into all the structural element of the flying machine, resulting in a structure that is not as rigid as that one of the X3D model. Consequently, vibrations from the propellers tend to propagate through the structure. Such high frequency vibrations are not a problem for the autonomous operation of the Owl, but they do affect the quality of the flight data.

In order to have a longer flight endurance, the Owl has a battery with a higher capacity than that in the X3D in addition to the heavier electronics, this brings the total weight of the platform to 725g (see Table 3.2 for other physical characteristics). During the design, the position of the various components was chosen carefully in order to maintain the CoG of the vehicle approximately at the geometric centre to avoid compromising the controllability of the platform²⁵. The propellers used in the Owl are larger (230mm), have three blades, and are made out of very rigid plastic material. These characteristics make such blades more effective²⁶, and as a result they are able to produce a significant change in thrust even with a very small change in rotor speed. Combined with the fact that the Owl has a lower thrust to weight ratio than the X3D, making it perceptibly slower; precise control is even more challenging than what is for the X3D, especially in altitude.

The size and weight of the Owl and the X3D are certainly different, but the it is the Owl's electronics and sensor payload that really separates the two machines. The Owl hosts a large suite of sensors: a tri-axial gyroscope and accelerometer unit, a tri-axial magnetic sensor, a pressure sensor, a GPS module and a downward pointed ultrasonic module. A RISC microcontroller is used to interface all the sensors and, more importantly, to implement the necessary stability augmentation system. For the purpose of this thesis

²⁵On the basis of our 3D CAD model, we estimated the CoG to be on the z axis of the flight machine at 12mm along the positive direction of the z axis.

²⁶The drag resulting from blade flapping and sub-optimal pitch is minimized, and the larger blade surface produces more lift.

we implemented a low level controller that translates the pilot’s inputs into commanded rotational speeds; our implementation is essentially the same as the one described in detail in Section A.4. With the addition of this stabilisation system, the Owl can be flown manually, and the four control inputs act on the platform in the same way as described in Section 3.3.1 for the X3D.

The pilot’s control commands are received by the flying unit thanks to a RF receiver directly interfaced to the on-board microcontroller. The same remote controller and PC interface developed for the X3D were used for the Owl.

During the development phase we experimented with a variety of different settings for the PID stabilization algorithm. Ultimately we settled for fairly high control gains in order to have a fast controller response and better disturbance rejection capabilities²⁷. The stabilization controller uses an integrative feedback based on the gyros’ readings to control the heading of the flight machine; since the gyros are noisy, the integrated error obviously tends to drift with time. High gains in the controller lead to a faster drift in the heading which the pilot (or any automatic controller) has to correct for²⁸.

The Owl also carries a lightweight Linux computer [50] and a 802.11g wireless card. These modules give the helicopter computation and communication abilities that are essential for the outdoor navigation task that the Owl was designed for, but had no use in the experiments referred to here.

For the Owl, we define the state and input vectors in the same way as for the X3D:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T \quad (3.10)$$

$$\mathbf{u} = [u_{th}, u_{ya}, u_{rl}, u_{pt}, u_{ba}]^T, \quad (3.11)$$

and so the same state update and integration equations as for the X3D apply (equations 3.4-3.7).

As for the X3D, no specific manoeuvres were flown when collecting the experimental data, but because the Owl is heavier and has a slower dynamics than the X3D, the range of speeds and angular orientations that can be achieved within the space of our arena is more

²⁷This in turn also allowed the EKF (extended Kalman filter) running on board to produce better attitude estimation; this filter was not used in this work but it is a crucial component for enabling outdoor navigation.

²⁸This effect is also present on the X3D, it is simply more noticeable on the Owl due to the higher feedback gain of the controller.

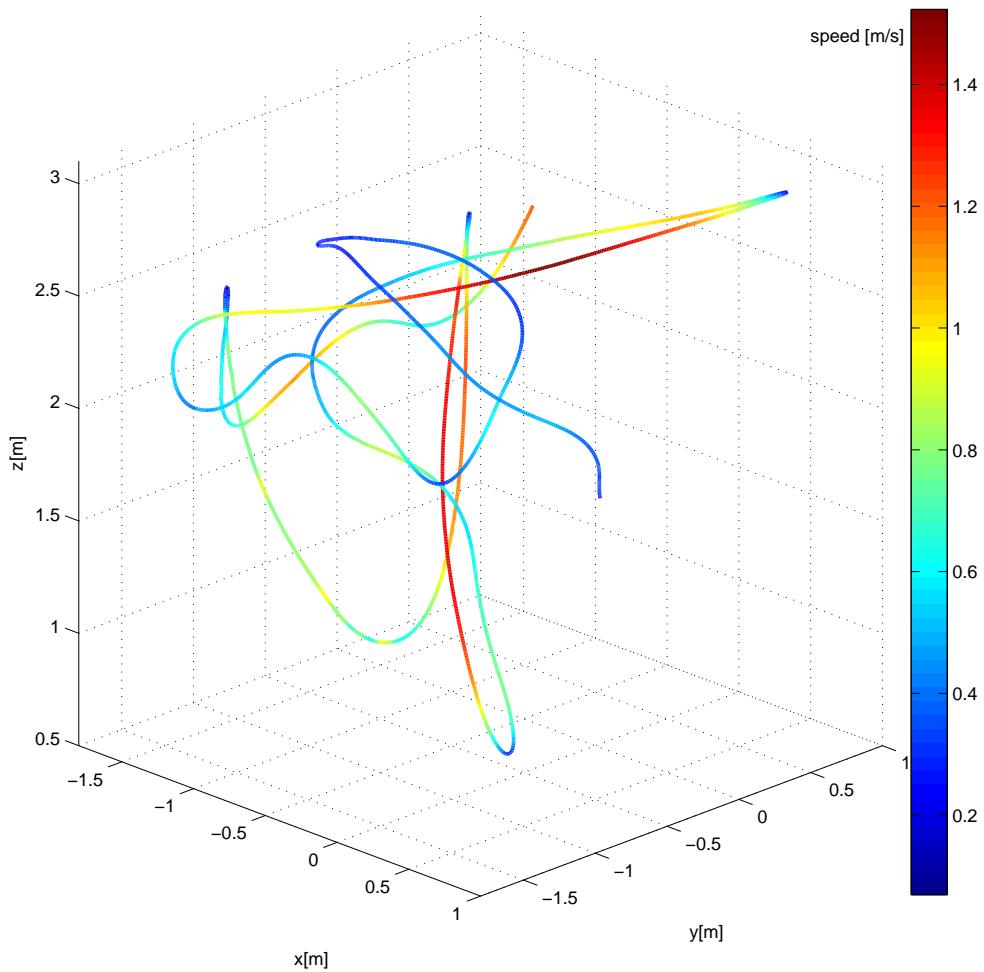


Figure 3.11: Short section (40s) of data from one of the Owl datasets. A variety of manoeuvres were executed; the colour of the path indicates the speed of the Owl.

limited (compare Figure 3.11 with Figure 3.9) . Two distinct sets of data were collected: `dataowl_1` and `dataowl_2`, 484.6 seconds and 571.9 seconds long respectively. After the preprocessing, each entry in our datasets contains the state vector \mathbf{x} , the control inputs \mathbf{u} and the precomputed accelerations \mathbf{a} and $\boldsymbol{\alpha}$.

As usual the datasets do not include take-off or landing phases, and we again tried to minimise the possible sources of turbulence by avoiding flying close to the ground or to the walls.

3.3.3 The Toy Car

Although we are very familiar with other helicopter concepts as can be seen from some of the author’s publications ([101, 58, 57]), we felt it more interesting to test our algorithms and solutions with a platform very different in nature from a flying machine with the aim

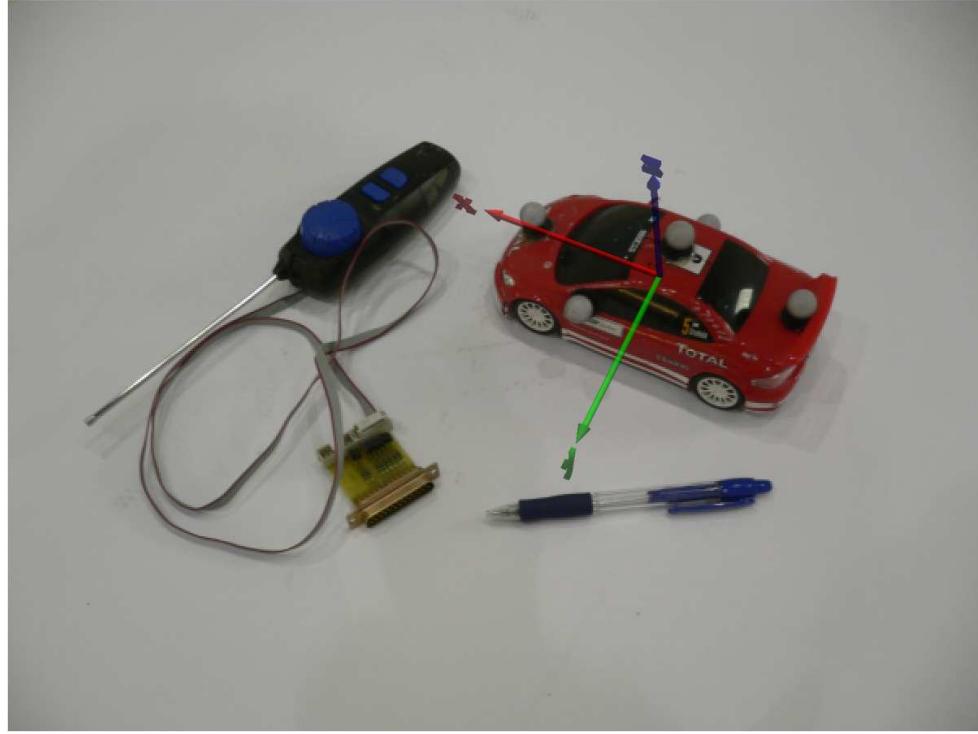


Figure 3.12: Toy car and electronic interface used to connect the remote to the PC parallel port.

size	180x100 mm
weight	290g
wheels diameter	28mm
battery	2300 mAh @ 4.5V
endurance	10 minutes
max speed	$\approx 2\text{m/s}$

Table 3.3: Toy car main physical characteristics.

of testing the generality of our approach.

A toy car (see Figure 3.12) is very suited to this purpose since, given its small size, it can be used directly on the floor of our flying arena. Its small size allows us to reach the top speed within the space available in our test area, allowing us to effectively sample the whole speed envelope of the vehicle. The car is small, light (see Table 3.3) and has small wheels, and so even small irregularities in the floor (i.e. any small gap between the carpet tiles or any variation in the roughness of the carpet) translate into oscillations of the markers. As the car is not new, and is of poor quality its turning radius is asymmetrical and its differential drive is worn out to the point that the wheels have a few millimetres of lateral play. No electrical modifications of any kind were made to the car (i.e. no power stabilization), therefore the battery level has a direct impact on the car's top speed,

responsiveness and turning abilities²⁹. As for the quadrotors, the battery voltage needs to be recorded as an additional input. In a car-like vehicle the turning speed is intimately related to the forward speed since given a fixed steering the car will obviously reach an higher rotational speed when travelling faster; we can already foresee how this might lead to a non linear coupling between the linear and rotational motion.

The car platform is different in several ways from the two quadrotors, but the major difference is possibly in the type of control. The car is provided only with simple discrete control inputs: forwards or backwards at full throttle or zero throttle, and steering right, left or neutral (Figure 3.13). The only way to control the car's speed is therefore to repeatedly turn on and off the throttle, a skill that is not easy to master. Nevertheless it

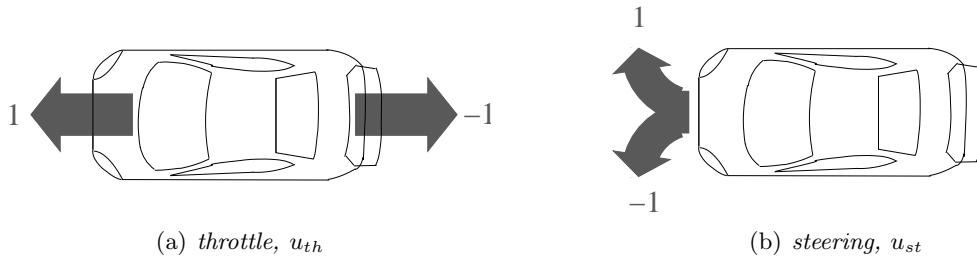


Figure 3.13: Toy car control commands: a) the throttle drives the car forward $u_{th} = 1$ or backward $u_{th} = -1$; b) the steering drives the car left $u_{st} = -1$ or right $u_{st} = 1$.

will be extremely informative to see how our methodologies can cope with a discrete rather than a continuous input.

Once the car has been equipped with infrared markers, the same procedure used to collect data for the Owl or the X3D can be applied without modifications³⁰. Since a car can be seen as a 6DoF rigid body constrained to move on a plane, we could even use the same state vector we used for our quadrotors, but to simplify our notation it is easier to redefine the state vector \mathbf{x} as:

$$\mathbf{x} = [u, v, r]^T,$$

considering only the forward and lateral speeds u and v and the angular speed r . For each entry in our datasets we have in addition to \mathbf{x} the array of inputs $\mathbf{u} = [u_{th}, u_{st}, u_{ba}]^T$ and the computed accelerations $\mathbf{a} = [a_x, a_y]^T$ and $\boldsymbol{\alpha} = [\alpha_z]$.

In the case of the toy car, we will again be experimenting with two different types of

²⁹The car has a steering mechanism implemented using an electro magnet, so the force generated by the magnet, and therefore its ability to steer the car, depends directly on the battery voltage

³⁰In this case the coordinates z , ϕ and θ will be approximately constant since the car is moving on the ground.

models, one in velocity and one in acceleration space. For the first type we will learn the function f that, given the current state, predicts the velocities; this leads us to rewrite equation 2.16 as:

$$[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (3.12)$$

where $\mathbf{v} = [u, v]^T$ and $\boldsymbol{\omega} = [r]$.

In the case of predicting accelerations, the model f can be written as:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (3.13)$$

and the state update equations become:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{C}'_{b_t}^{b_{t+1}} \mathbf{a}_t \Delta t \quad (3.14)$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \boldsymbol{\alpha}_t \Delta t, \quad (3.15)$$

where $\mathbf{C}'_{b_t}^{b_{t+1}}$ is the two by two DCM between the body frame at time t and that at time $t+1$ as defined in equation B.2.

In Chapter 6, when producing a stochastic model of the car dynamics, we will use an *extended* state that includes the car's position and heading (respectively $\mathbf{p} = [x, y]^T$ and $\Phi = [\psi]$). The *extended* state is defined as:

$$\mathbf{x} = [u, v, r, x, y, \psi]^T. \quad (3.16)$$

To compute these quantities we need to transform the linear and angular velocities from the body frame to the world frame, and then integrate:

$$\Phi_{t+1} = \Phi_t + \boldsymbol{\omega}_t \Delta t \quad (3.17)$$

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \mathbf{C}_{w_t}^{b_t} \mathbf{v}_t \Delta t. \quad (3.18)$$

where $\mathbf{C}_{b_t}^{w_t}$ is the two by two DCM between the body frame and the reference frame at time t as defined in equation B.3.

In total, about 25 minutes of driving data was collected, and many different manoeuvres were performed varying from loops, figures of eight, sudden starts and stops manoeuvres, and driving at different speeds forwards and backwards. We also recorded data from

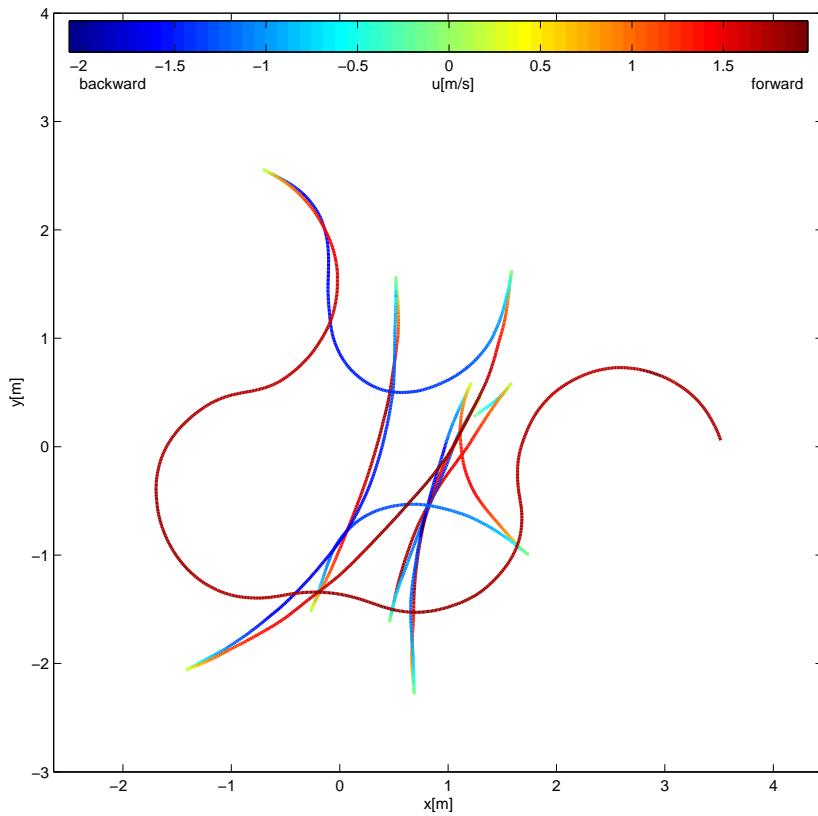


Figure 3.14: Short section (40s) of data from one of the toy car datasets. A variety of manoeuvres were executed driving both forwards (red portion) and backwards (blue portion).

periods of time in which no control inputs were given and the car was sitting still, as well as the situation in which the car was driven to maximum speed and left to slow down until stopping as a result of friction. Figure 3.14 shows a typical trajectory from one of the datasets). Three different datasets were collected under similar battery conditions³¹ (`datasetcar_6`, `datasetcar_7` and `datasetcar_8`) about 420, 400 and 450 seconds long respectively.

3.3.4 The ATTAS Aircraft

Our primary research interest and expertise is in miniature rotorcraft, but the work we are presenting here is not by any means restricted to small vehicles; and so it is of interest to extend our experimentation to larger platforms. For decades, aircraft have been at the center of a body of research in control engineering, which has led to the development of some of the most advanced techniques of control and system identification. Flight datasets

³¹A set of fully charged batteries were put on the car at the beginning of each run, and the run was terminated when the battery was depleted (i.e. at a battery voltage of 3.4V).



Figure 3.15: Experimental aircraft VFW 614 ATTAS.

length	20.6 m
wingspan	21.5 m
empty weight	12.179 Kg
power plant	2 turbofan engines
cruise speed	722 Km/h
range	1195 Km

Table 3.4: ATTAS main physical characteristics.

specifically collected for system identification are readily available for full size aircraft, as are a full range of standard identification tools these constitute an excellent ground for comparison of our methodologies.

We will be using datasets collected on-board the experimental aircraft VFW-614 ATTAS (Figure 3.15) at the Deutsche Zentrum für Luft- und Raumfahrt (DLR). Originally this was developed as a 40 seat civil transport aircraft (see main characteristics in Table 3.4), but it was then equipped with a variety of measurement and actuation system to be used as in-flight-simulator and demonstrator.

The ATTAS shares the design principles common to many modern aircraft and is flown by means of three different sets of control surfaces, elevators, ailerons and rudder. By changing the orientation of the elevators on the trailing edge of the horizontal tail stabiliser, the pilot can produce a moment that pitches the aircraft. As a result the angle of attack of the wings changes, with a consequent change in the magnitude of the lift force; the aircraft

therefore ascends or descends (Figure 3.16(a)). When the ailerons on the trailing edge of the outer part of the wings are deflected in opposite directions the difference in lift between the left and right wing produces a torque that rolls the aircraft about its longitudinal axis (Figure 3.16(b)). Consequently the aircraft banks, and the resultant of the difference between the aerodynamic lift and the gravitational force will drive the aircraft into a turn. When deflected, the rudder on the trailing edge of the vertical tail produces a yaw motion (Figure 3.16(c))), commonly used in coordinated turns to avoid side slipping. The dynamic behaviour of an aircraft like the ATTAS is well understood, and in the normal flight regime

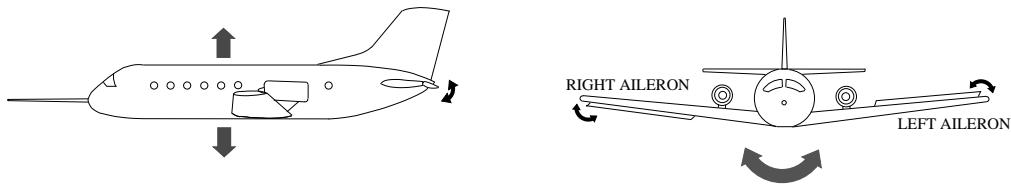
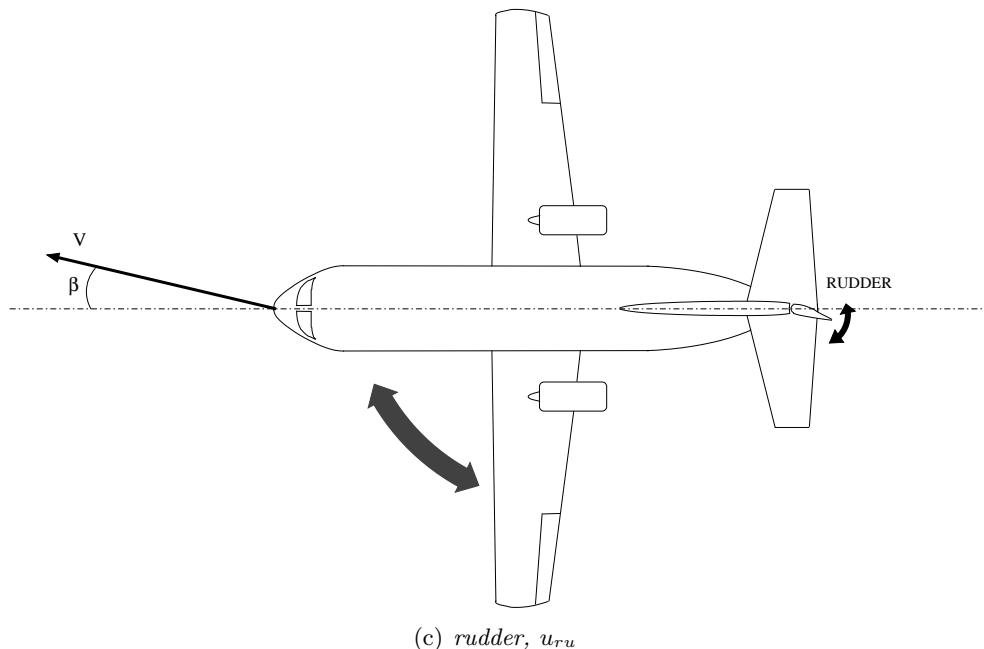
(a) elevators, u_{el} (b) ailerons, u_{ai} (c) rudder, u_{ru}

Figure 3.16: Aircraft controls: a) the elevators control ascent and descent, b) the ailerons control banking, c) the rudder controls the yaw motion. In Figure c) the side-slip angle β is also visible.

(i.e. with a low angle of attack) the longitudinal and lateral dynamics are often treated as being decoupled as follows: first the sideslip angle β is defined between the axis of the aircraft and the direction of motion identified by the velocity $\mathbf{v} = [u, v, w]^T$. More specifically,

$$\beta = \sin^{-1} \left(\frac{v}{V} \right), \quad (3.19)$$

where V is the airspeed³². β is then treated as an additional input along with u_{ai} and u_{ru} . The introduction of β as an additional input allows the decoupling of the yaw motion from the lateral and longitudinal motion.

Since for aircraft like ATTAS the principal dynamic modes of the system are known, it is possible to choose specific input manoeuvres during flight testing to improve the results of identification as explained in Section 2.2.2. In general, for each of the control inputs several seconds of steady state³³ flight are recorded, then a specific control input is applied, and finally the aircraft is allowed to return to the steady state before initiating the next manoeuvre. A typical input for system identification is a two sided pulse input (doublet) where the duration of the pulse is chosen accordingly to the time constant of the specific dynamic mode to be excited.

For the ATTAS we will investigate only the lateral-directional dynamics (see details in Section 4.1.1), and for this purpose the state vector can be defined as:

$$\mathbf{x} = [v, \theta, \psi, p, r]^T.$$

For the input vector we will make use only of the ailerons, the rudder controls, and the side-slip angle β

$$\mathbf{u} = [u_{ai}, u_{ru}, \beta]^T.$$

The datasets that we will be using were obtained from [113], and are the logs of two separate doublet manoeuvres executed with the ailerons and rudder controls (datasets `fATTASAilRud1`, `fATTASAilRud2`); both have a duration of 58.8 seconds. For each time step in the datasets we have the values of the state vector, the pre-computed accelerations $\mathbf{a} = [a_y]$ and $\boldsymbol{\alpha} = [\alpha_x, \alpha_z]^T$, and the controls $\mathbf{u} = [u_{ai}, u_{ru}, \beta]^T$ ³⁴.

³² $V = \sqrt{u^2 + v^2 + w^2}$.

³³Level flight at a defined cruise speed

³⁴The sideslip angle β among our inputs however for simplicity this will be calculated online from the linear velocities as per equation 3.19, so it is not included in the dataset.

For the ATTAS aircraft we will consider only models in acceleration space which take the form:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (3.20)$$

The state update equations then become:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{C}''_{b_t+1} \mathbf{a}_t \Delta t \quad (3.21)$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \boldsymbol{\alpha}_t \Delta t \quad (3.22)$$

$$\boldsymbol{\Phi}_{t+1} = \boldsymbol{\Phi}_t + H''(\boldsymbol{\Phi}_t) \boldsymbol{\omega}_t \Delta t, \quad (3.23)$$

where $\mathbf{v} = [v]$, $\boldsymbol{\omega} = [p, q]^T$, $\boldsymbol{\Phi} = [\theta, \psi]$, \mathbf{C}''_{b_t+1} is the two by two DCM between the body frame at time t and that at time $t+1$ as in equations B.4, and $H''(\boldsymbol{\Phi}_t)$ are the two dimensional Euler kinematic equations as in B.6.

3.3.5 The Autopilot Helicopter Simulator

Some of our experiments on automatic control design were carried out before any of our helicopter platforms was ready, so a software simulator was used as an interim measure to test our methodologies. The helicopter simulator included in the Autopilot software suite ([12]) was selected for being realistic³⁵ and freely available.

The dynamic model of this simulator is based on the principles of helicopter aerodynamics ([179],[189]) and although it is not based on real data, it aims at reproducing the dynamics of the X-Cell Fury 60 single rotor model helicopter ([109]); its main parameters are shown in Table 3.5.

main rotor radius	2.25 ft
tail rotor radius	0.54 ft
weight	19.5 lbs
total length	2.52 ft

Table 3.5: Autopilot single rotor helicopter simulator main physical characteristics.

Blade element theory is used as the basis for the computation of rotor thrust and drag forces, and the body dynamics and the stabilising bar are modelled as proposed in Mettler *et al.* ([152]). In a single rotor helicopter all the thrust is provided by the single main rotor on which are mounted two or more variable pitch blades. The rotor speed is usually

³⁵In terms of the level of physical detail and fidelity of the simulation.

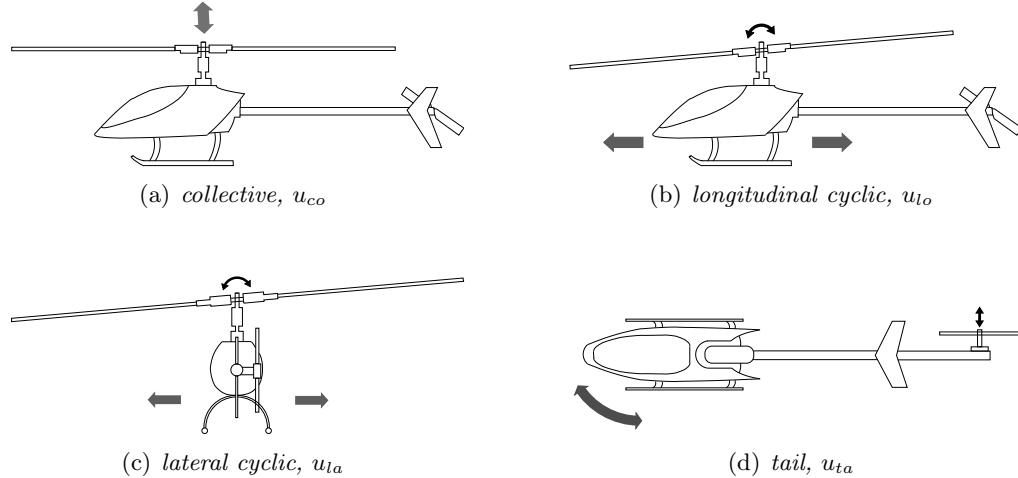


Figure 3.17: Single rotor helicopter controls: a) the collective pitch of the main rotor controls the vertical motion, b) the longitudinal cyclic control the helicopter's pitch, c) the lateral cyclic control its roll, d) the tail rotor pitch governs the yaw motion.

kept constant, and the pilot can control the thrust by varying the collective pitch (u_{co}) of the blades (Figure 3.17(a)). Directional control is achieved by changing the cyclic pitch of the blades. The longitudinal cyclic (u_{lo}) cause the rotor tip path plane³⁶ to pitch, which has the effect of tilting the thrust vector forwards or backwards in the longitudinal plane (Figure 3.17(b)). Similarly the lateral cyclic (u_{la}) rolls the rotor tip path plane laterally, tilting the rotor thrust left or right in the lateral plane (Figure 3.17(c)). In a single rotor configuration the reaction torque produced by the blades' drag needs to be balanced using a tail rotor. Changing the pitch of the tail rotor blades (u_{ta}) controls the magnitude of the balancing torque and therefore allows the control of the yaw motion of the helicopter (Figure 3.17(d)).

We have modified the autopilot software to provide a 3D visualisation (Figure 3.18) and to create a TCP port through which we can query the simulator state, set the control inputs and step the simulator forward in time. One step of the simulator corresponds to 20 ms of simulated time, but the dynamic equations are internally integrated with a time step of 10 ms to improve the precision of the simulation. The state vector has the familiar form that we have seen already in the case of our quadrotors:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T,$$

³⁶The imaginary plane described by the tip of the rotating blades.

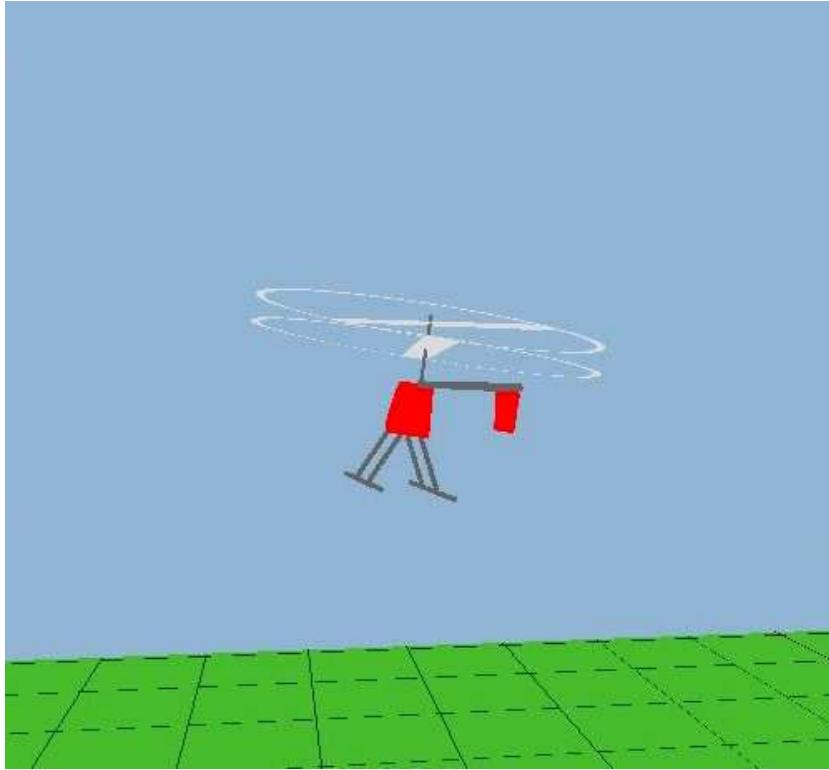


Figure 3.18: Snapshot of the 3D visualization that we developed for the Autopilot helicopter simulator.

while the input vector takes the form:

$$\mathbf{u} = [u_{co}, u_{lo}, u_{la}, u_{ta}]^T$$

In order to test the robustness of our controllers we also extended the simulator to allow us to change the mass of the helicopter and to add aerodynamic disturbances in the form of a wind component with time varying velocity.

As it is derived only from first principles and not from any actual flight data, there is no guarantee about the fidelity of this simulator. However, this model still embeds all the challenging coupling and nonlinearity known to be present in real helicopter platforms; the automatic control results obtained by using this simulator are therefore qualitatively meaningful and will be indicative of the capabilities of our approach.

As a side note, we note that this simulator is extremely difficult to fly even for a pilot who is capable of flying any of the quadrotor helicopters already presented.

3.4 Summary

The chapter deals with many of the practical details that characterize the experimental part of this work. In the first part we looked in detail at how the experimental data used throughout the work were collected and preprocessed in a platform agnostic fashion. We also addressed the issues of the noise and delay that affects the collected data. The second part of the chapter presented in detail the characteristics of each of the platforms used in this work; we focussed in particular on identifying the peculiarities that make each platform challenging from the point of view of automatic modelling and/or automatic controller design. For each of the platforms we also defined and discussed, the forms used for the state vector, and the mathematical transformations needed to propagate the dynamic equations forward in time. These definitions will be used extensively in the next three chapters.

Chapter 4

Non-Evolutionary techniques

In this chapter we start looking at the problem of modelling by applying some of the standard techniques that have been developed within the fields of machine learning, robotics and aerospace engineering. The results of this chapter will form a point of comparison for the material presented in subsequent chapters, and will allow us to gain a firsthand understanding of the advantages and shortcomings of these widely used techniques.

In the first part of the chapter, (Section 4.1) we look at the type of models most commonly used in the aerodynamic engineering domain. These are grey box models, which have a clear and explicit structure, based on an understanding of the underlying physics; the process of fitting such models to data is essentially one of parameter estimation.

In the second part of the chapter (Section 4.2) we will focus on black box techniques using some well known machine learning tools that can be applied to our vehicles with minimal use of domain knowledge.

All the methodologies applied in this chapter belongs to the families of EE or OE methods and therefore are based on the assumption of negligible process noise (see Section 2.2.4 for more details on this assumption).

4.1 Parameterised First Principles Models

The primary physical phenomena governing many conventional vehicle concepts (e.g. standard aircraft, cars, helicopters) are well understood, therefore dynamic models based on these principles can be constructed. Those phenomenological¹ models are however in-

¹The distinction between phenomenological and behavioural models is introduced and discussed in Section 2.2.1.

inevitably derived under specific assumptions and simplifications, and as a result, they can deviate considerably from the actual vehicle at hand. This is especially true when the vehicle is complex, or is subject to effects difficult to capture analytically, like for example friction or aerodynamic effects. Both of these are present in our platforms; the first is present in the interaction between our car and the floor surface on which it is driven, while the second plays a paramount role in the dynamics of our flying machines. As explained in Chapter 2, in the system identification community these discrepancies are usually addressed by refining the analytical model in the light of the evidence produced by the experimental data, in an iterative process that eventually delivers the necessary level of performance. In our examples we are using analytical models derived by other authors both for the ATTAS aircraft and the X3D quadrotors; however, in the case of the X3D quadrotor, it was necessary to extend the available model with some specific equations in order to match the characteristics of our propulsion and stabilization systems.

A model derived from first principles generally has the form of a parameterised set of equations in which some of the parameters are directly measurable (mass, moments of inertia, wing surface area etc.) while others needs to be estimated from experimental data. The most widely used principle for determining the model parameters is that of Maximum Likelihood which is based on the idea of finding the parameters that maximize the probability of obtaining the observed dataset. In the following two examples we will model both of our vehicles in acceleration space, and will assume as explained in Section 2.2.4 that the process noise is negligible, thereby locating ourselves in the OE domain. In other words, the acceleration predictions of our models will be integrated forward in time to compute step by step the complete development of the state variables over the chosen time window. The distribution of the error between the predicted and the measured state variables constitutes the metric that guides the parameter optimization.

In their review of nonlinear black-box modelling, Sjöberg *et al.* [212] argue that the challenging part of system identification is determining the model structure, and that searching for the correct model parameters is often not a difficult problem. In line with this, most of our work will focus on generating the structure (as well as the parameters) for our models; however, in this section it will be informative to start our investigation by looking at parametric models since in practice they are the most widely used type of model in both the aerospace [229, 113] and robotics [152, 7] community.

4.1.1 ATTAS Lateral-Directional Model

A conventional aircraft spends most of its flight time in a wing-level steady-state flight condition; in this condition, and with the additional assumption of a small side-slip angle² β , its longitudinal and lateral motion can be considered decoupled ([217]). This reduces the standard 6DoF rigid body model to two more amenable 3DoF models. The first is a lateral-directional model that deals with the yaw motion, the roll motion and the linear-lateral motion (involving therefore the variables v , p , r , ϕ and ψ), while the longitudinal model covers the vertical, longitudinal and pitch motion (and involves the variables u , w , q and θ). In our experiments we will concentrate on the first model because it is the more challenging of the two since it includes the coupling between yaw and roll motion.

It is standard practice to simplify the dynamics even further by producing a linearization of those models about an equilibrium point³ since in this way a wide range of sound techniques from the theory of linear systems can be applied. Linear models expressed using aerodynamic derivatives have been shown to reproduce effectively the experimental data in many aircraft of conventional design, and as a consequence have been widely used to analytically derive the principal dynamic modes of the aircraft motion, as well as to design control systems based on classical control theory ([217]). Since the aerodynamic derivatives can easily be calculated analytically or numerically (e.g. by perturbation of a model around an equilibrium condition) and can provide good insight into the dynamic characteristics of an aircraft⁴, they constitute a fundamental block in the theory of aircraft dynamics. For all these reasons we selected this model as a very interesting proving ground for comparing our modelling technique with more conventional ones.

We have to remind ourselves however that these simplified models are valid only where the behaviour of the system is approximately linear (e.g. level flight) making them unsuitable for complex regions of the flight envelope (e.g. near stall conditions).

²See Section 3.3.4 for a definition of the side-slip angle β .

³The point at which all of the state derivatives are identically zero, i.e. the system is “at rest”.

⁴The aerodynamic derivatives are an indication of how quickly the model departs from the equilibrium point in response to an input or a change in one of the state variables.

Equations of Motion

For the ATTAS model we will produce a model in acceleration space, and so we need to learn the function f :

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (4.1)$$

where we recall that $\mathbf{a} = [a_y]$, $\boldsymbol{\alpha} = [\alpha_x, \alpha_z]^T$, $\mathbf{x} = [v, \theta, \psi, p, r]^T$ and $\mathbf{u} = [u_{ai}, u_{ru}, \beta]^T$.

From [113] we have that the linear lateral-directional model f of the ATTAS can be written as:

$$a_y = Y_p p + Y_r r + Y_{u_{ai}} u_{ai} + Y_{u_{ru}} u_{ru} + Y_\beta \beta + b_{a_y} \quad (4.2)$$

$$\alpha_x = L_p p + L_r r + L_{u_{ai}} u_{ai} + L_{u_{ru}} u_{ru} + L_\beta \beta + b_{\alpha_x} \quad (4.3)$$

$$\alpha_z = N_p p + N_r r + N_{u_{ai}} u_{ai} + N_{u_{ru}} u_{ru} + N_\beta \beta + b_{\alpha_z}, \quad (4.4)$$

where $L_p, L_r, L_{u_{ai}}, L_{u_{ru}}, L_\beta, N_p, N_r, N_{u_{ai}}, N_{u_{ru}}, N_\beta, Y_p, Y_r, Y_{u_{ai}}, Y_{u_{ru}}, Y_\beta$ are the aerodynamic derivatives which express the linear dependency of each angular or linear acceleration on the relevant state variables. Since the model is linear, the bias parameters $b_{\alpha_x}, b_{\alpha_z}, b_{a_y}$ effectively play the role of control input biases.

This dynamic model of the ATTAS is straightforward and simple, but at the same time it represents a useful and meaningful model and is therefore a perfect test bench with which to start our journey in the estimation of parameters.

Training and Validation

We now have a well defined model of our system and we have specified its state update equations (equations 3.21-3.23). Before proceeding with the fitting of the model, we need to look in details at how the training and the validation will be carried out.

On the basis of the considerations of Section 2.2.4, we train our models over several data windows. To permit meaningful comparisons among techniques, we use windows of the same size across the different OE methods used for the same platform. In deciding on a length for the windows, we have therefore to take into account that the coevolutionary technique that we will discuss in Section 5.5 cannot work at its best with a small number of training windows⁵. As a trade-off between taking long range interactions into account,

⁵One of the fundamental parts of the technique is to search for the most appropriate windows of data; having only a few windows available would make such a search pointless.

limiting the computation, and ensure a reasonable number of windows we pragmatically chose a fixed length of $T = 100$ data points. In order to use our data efficiently we use consecutive windows that cover the whole training dataset.

Since we are working in OE settings, the state variables need to be given an initial value at the beginning of each time window. In our experience the simplest of the strategies discussed in Section 2.2.1, that of setting the state to the value it has in the training dataset, has proved effective. The accelerations predicted by the parametric model are then integrated forward in time to compute the new state of the platform. The prediction and integration steps are repeated for all the remaining time steps to predict the state of the system over the whole time window.

The likelihood is defined as the probability density over the observed data $\mathbf{z}_{0:N} = \{\mathbf{z}_0, \dots, \mathbf{z}_N\}$ given a defined set of model parameters (Θ) and the control inputs (\mathbf{u}_t):

$$p(\mathbf{z}_{0:N} | \Theta, \mathbf{u}_t). \quad (4.5)$$

Assuming that the error has a Gaussian distribution and is statistically independent in time, it is relatively straightforward to show (see Section B.4.2) that maximizing the likelihood is equivalent to minimize the determinant of R , the covariance matrix of the residuals, which is defined as:

$$R = \frac{1}{N} \sum_{t=1}^N (\mathbf{z}_t - \mathbf{y}_t) (\mathbf{z}_t - \mathbf{y}_t)^T. \quad (4.6)$$

In our case \mathbf{z}_t is represented by the data we collected during our experiments, while \mathbf{y}_t is the output predicted by the model.

To minimize $|R|$ and compute the set of model parameters that maximize the likelihood (Θ_{ML}) we use the Levenberg-Marquardt (LMA) algorithm ([133]), a well known iterative minimization method. The computation of the gradient of Θ_{ML} needed by the LMA is done by numerical perturbation of the function $|R|$.

Obtaining Θ_{ML} typically involves the following steps:

1. selecting an initial value for Θ
2. computing the predicted system output \mathbf{y}_t using Θ with the model in question, doing the necessary integration to propagate the state variables forward in time
3. using the LMA algorithm to minimize $|R|$

4. iterating steps 2-3 until the maximum number of iterations⁶ is reached.

Selecting appropriate values for the starting parameters is an important step, as a badly chosen set of parameters can stop the LMA from converging to a solution. Since the models are based on physical principles, an experienced engineer is often able to produce a good guess for the initial parameters; this is another way⁷ in which domain knowledge comes into play in the case of models based on first principles.

Once the training is complete, we need to measure the performance of our model. In Section 2.2.5 we saw that choosing a validation metric is not straightforward; to make a pragmatic and platform independent choice, we decided to avoid any scaling factor, and defined our performance metric using the *RMSE* (root mean square error)⁸. This simple and well known metric is often used in the machine learning literature, and has the advantage of having an intuitive meaning since its value is expressed in the same units as the data⁹.

As seen in Section 2.2.5, windows can also be used to produce a more representative estimate of overall performance. Using a set of data windows we can average the validation metrics across windows with different initial conditions and different levels of model performance. In our tests we will set the initial state vector to its recorded state value, so if we take different windows of data, some will start from initial conditions with less measurement noise than the average, and others from initial conditions with more noise. In addition, the model itself will very probably perform better in certain parts of the dataset, and worse in others. In general, the manoeuvres executed by the pilot have a duration that is much shorter than a data window; several of them will be present in each window. It might of course happen that a manoeuvre is split across consecutive windows, but the fact that we are averaging over many windows will compensate for this.

Data windows can be selected from the dataset in various ways. For example, the dataset could be divided into consecutive windows, or the positions of the windows could be selected randomly, and the windows might or might not be allowed to overlap. For a given data point in a window, the error calculated at that point will depend not only on

⁶Based on a set of exploratory runs in our experiments we set the maximum number of iterations to 50.

⁷The other is obviously writing down the analytical expressions that constitute the model.

⁸In some disciplines the same quantity is called root mean standard deviation.

⁹In Section B.4.2 is shown that the mean square error (*MSE*), and therefore also the *RMSE* are intimately related to the concept of Maximum Likelihood estimation under the assumption of a Gaussian noise model.

the point itself but also on the preceding point in the same window¹⁰. Selecting windows that do not overlap is therefore preferable, since it leads to independent error estimates. To make the most of our data, selecting consecutive windows is the obvious choice, and since in this way we get the maximum number of non-overlapping windows, we can expect a relatively accurate estimation of the model performance.

For the computation of the error of the car model we use the same window sizes that were used during training (i.e. 100 data points) which gives us $W = 15$ windows. For a window w of length T and for the variable i

$$RMSE_{iw} = \sqrt{\frac{1}{T} \sum_{t=t_{w0}}^{t_{w0}+T} (z_t^i - y_t^i)^2}, \quad i \in \mathbf{x}^{11} \quad (4.7)$$

where z^i are the measured values of the variable, y^i are the predicted values, and t_{w0} indicates the starting time of the window. The standard deviation of the $RMSE$ (we will call it $RMSEsd$) over such a set can be used as a measure of how the predictive abilities of the model vary over different parts of the dataset:

$$RMSEM_i = \frac{1}{W} \sum_{w=0}^W RMSE_{iw} \quad (4.8)$$

$$RMSEsd_i = \sqrt{\frac{1}{W} \sum_{w=0}^W (RMSE_{iw} - RMSEM_i)^2}. \quad (4.9)$$

It is often useful to have a metric that instead of being at level of the single variable (i.e. i) is representative of the performance of the model over all the state variables. We call the resulting metric $RMSE_{total}$ and define it as:

$$RMSEM_{total} = \frac{1}{WT} \sum_{w=0}^W \sum_{t=t_{w0}}^{t_{w0}+T} \sum_{i \in \mathbf{x}} (z_t^i - y_t^i)^2, \quad (4.10)$$

where as we have already seen \mathbf{x} is the state vector.

In our discussion we will compare all the various trained models with one another. However it is very often useful to make comparisons against a very simple baseline model, and to have some kind of relative metric that we can use more or less intuitively to understand the quality of our model. The simplest model that one can think of is a *constant*

¹⁰ As a result of the propagation of error, as previously discussed.

¹¹ The $RMSE$ can be defined for any system variable i , but in our experiments we will only compute such a metric for state variables, therefore $i \in \mathbf{x}$.

model that, given a state \mathbf{x}_{t_0} , predicts that for all the following time instants the state will not change (i.e. $\mathbf{x}_t = \mathbf{x}_{t_0} \forall t$). We can then define what is often called the *RAE* (relative absolute error [251]):

$$RAE_{x^i w} = \frac{\sum_{t=1+t_{0w}}^T |z_t^i - y_t^i|}{\sum_{t=1+t_{0w}}^T |y_t^i - y_{t_{0w}}^i|} \quad i \in \mathbf{x}, \quad (4.11)$$

where z^i are the measured values of the variable, y^i are its predicted ones and t_{0w} indicates the starting time of the window w . For consistency with the *RMSE* metric we will use the same data windows. From equation 4.11 it can be seen that the *RAE* is a positive number. Its value is zero for a perfect model, and is lower than one for a model that is better than the *constant model*. Any value higher than one indicates an error larger than would be achieved by not predicting any change, and therefore identifies a poor model.

To have a measure that reflects the capabilities of the model in different state configurations, we are interested in computing such a metric over many windows. The considerations made in the case of the *RMSE* still apply, and so once again we choose the strategy of using consecutive windows:

$$RAEm_i = \frac{\sum_{w=0}^W \sum_{t=1+t_{0w}}^{t_{0w}+T} |z_t^i - y_t^i|}{\sum_{w=0}^W \sum_{t=1+t_{0w}}^{t_{0w}+T} |y_t^i - y_{t_{0w}}^i|} \quad i \in \mathbf{x}. \quad (4.12)$$

The denominator of equation 4.12 will be zero if in each window, each predicted state variable is constant and equal to its starting value $y_{t_{0w}}^i$. For any meaningful model (i.e. one which has some dynamics) this will never be the case, so the *RAEm* is in practice always well defined.

Very often in our examples we will plot the development over time of the predicted state variables over a short time window, along with the experimentally recorded data. We will usually select the time window randomly from the dataset¹², and therefore it will be useful to have an idea of how the error in the selected time window compares to the *RMSEM* of the model over the whole dataset. For this purpose we introduce the comparison ratio *CR* over the window w which is defined as:

$$CR_i = \frac{RMSE_{iw}}{RMSEM_{iw}}. \quad (4.13)$$

¹²This is most easily done by drawing the starting time t_0 from a uniform random distribution $\mathcal{U}[D, N - T]$, where N is the size of the dataset and D is the maximum input delay.

This ratio will be lower than one if, on the specific window used, the error is lower than the $RMSE_{iw}$ and will be higher than one for a $RMSE_{iw}$ that exceeds the $RMSE_{iw}$. This ratio can help us to understand to what extent the qualitative performance depicted in the plot of the chosen window is representative of the average behaviour of the model.

Model Fitting

For the ATTAS aircraft two different datasets are available; dataset `fATTASAilRud1` was used for the training and dataset `fATTASAilRud2` for the validation.

We first tried to run the optimization of all three equations of the ATTAS model (equations 4.2-4.4) setting all of the initial parameters to zero, but the algorithm did not converge within the maximum number of iterations allowed. This was done as a mere confirmation test, since we have already noted that the LMA algorithm is sensitive to the choice of initial parameters. Since in our dataset we have the pre-computed values for all the state derivatives, we can take a pragmatic approach to the issue of initializing the model parameters. If instead of searching jointly for the parameters of all the equations, we only search for the parameters of one equation at a time, we are likely to encounter a much smoother search space which will help the convergence of the LMA to a set of appropriate parameters for that equation. In place of the prediction of the remaining equations, we will use the pre-computed values from the training dataset. For example we can start by finding the initial values for $L_p, L_r, L_{u_{ai}}, L_{u_{ru}}, L_\beta$ using only the first equation of our model and minimizing the square error on v ¹³. Using a similar procedure with the remaining equations 4.3 and 4.4 (using the errors on p and r respectively as metrics) will provide a suitable set of initial values for the remaining model parameters. If the LMA fails to converge for any equation, it indicates clearly that domain knowledge is needed to set the initial parameters to appropriate values.

Once a set of appropriate starting parameters was obtained we proceeded to the joint optimization of all three model equations. The error in all five state variables was used for the cost function, and produced the set of parameter values in Table 4.1.

A first assessment of the result was made by testing the model on a randomly selected window of data from the unseen validation dataset. In the aeronautics community, com-

¹³In the case of a single variable the square error corresponds to $|R|$.

Parameter	Value
L_p	-2.21
L_r	0.88
$L_{u_{ai}}$	-1.33
$L_{u_{ru}}$	0.094
L_β	-3.84
b_{α_x}	0.0008
N_p	-0.166
N_r	-0.16
$N_{u_{ai}}$	-0.04
$N_{u_{ru}}$	-0.117
N_β	2.95
b_{α_z}	0.003
Y_p	3.76
Y_r	-0.82
$Y_{u_{ai}}$	1.36
$Y_{u_{ru}}$	1.25
Y_β	-32.4
b_{a_y}	-0.29

Table 4.1: ATTAS linear lateral-directional model: identified parameter values.

paring the prediction of the model against the experimental data as in Figures 4.1 and 4.2 is often called proof of match (POM [113]).

The predictive capabilities of the model are obvious; with a completely unseen data window and over a time of 4s, the agreement between the predicted and the measured data is very good, and for the velocities (v, p, r) only small differences are noticeable. The largest of the errors is in the yaw angular velocity (r), and this has an immediate impact on the yaw angle ψ . It is interesting to compute the comparative ratio CR (see equation 4.13) to understand how representative of the general performance of the model this randomly selected window is. The low values of CR (Table 4.2) for the variables p and ϕ indicate that the predictive ability of the model for those two variables is better than the average

	RMSE	CR
v	0.28 m/s	0.70
ϕ	0.0014 rad	0.19
ψ	0.012 rad	1.14
p	0.0022 rad/s	0.35
r	0.0059 rad/s	0.83
<i>total</i>	0.28	

Table 4.2: Prediction error of the parametric model computed over the window of data plotted in Figure 4.1.

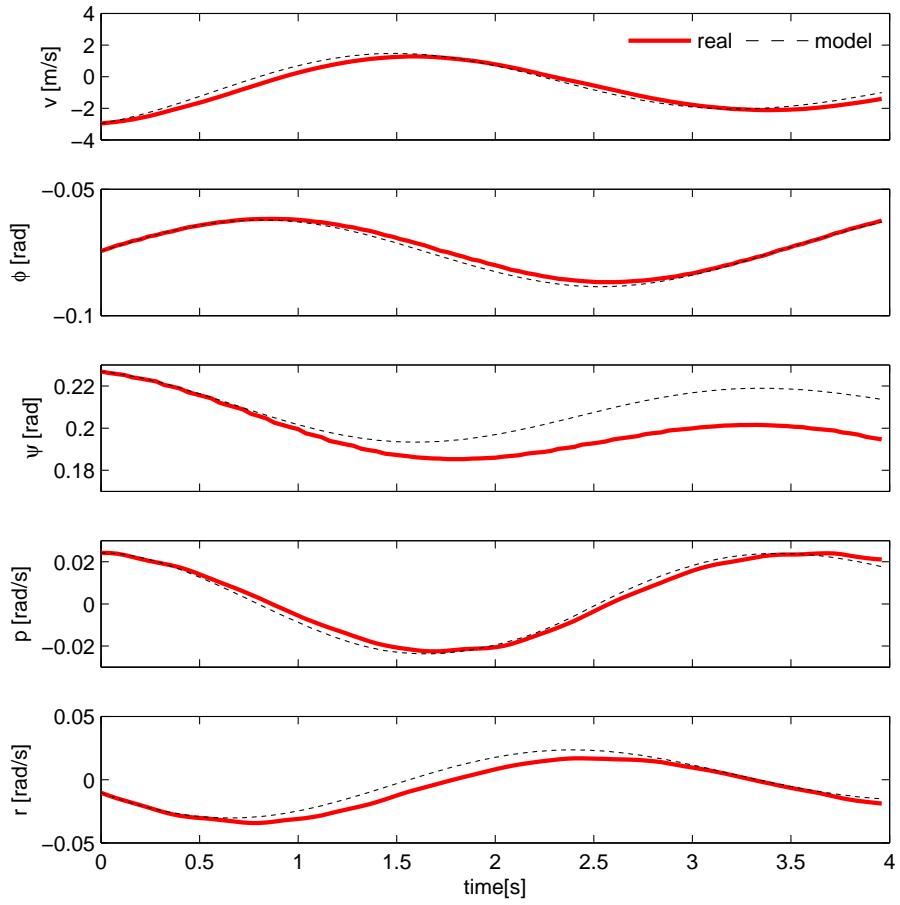


Figure 4.1: Proof of match: the state evolution predicted by the *ATTASAccFP* model, versus the measured real state. The small discrepancies in velocity, especially in r , build up and produce a small angular error.

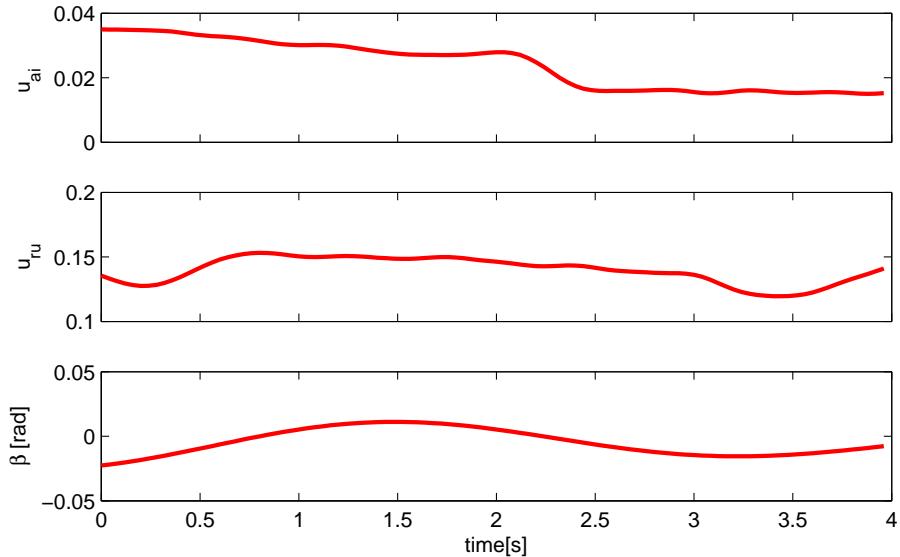


Figure 4.2: Control inputs given to the *ATTASAccFP* model during the time window in Figure 4.1.

	RMSEm	RMSEsd		RAEm
v	0.40	0.39	m/s	0.15
ϕ	0.007	0.008	rad	0.14
ψ	0.01	0.01	rad	0.40
p	0.006	0.007	rad/s	0.06
r	0.007	0.007	rad/s	0.22
<i>total</i>	0.40	0.39		

Table 4.3: Performance of the parametric model on the validation dataset.

model behaviour in this window. For the remaining variables, the behaviour is certainly more representative of the average model abilities.

For a more objective evaluation we then tested the model thoroughly, computing the *RAEm* and the *RMSEm* metrics as explained in the previous section . We also computed the *RMSEsd* across all the windows in the validation dataset so to have a measure of the model’s ability in different windows. The results obtained, shown in Table 4.3, confirm the very good performance of the parametric model with low values of the *RAEm* metric in all five dimensions. The heading angle (ψ) probably pays the price of being obtained by double integration by showing a slightly higher relative absolute error.

While we will be able to say more about the absolute error performances of this model in later chapters by comparing it with other models developed using different techniques, we can see by looking at the *RMSEsd* that the model performance varies to some extent between windows in the dataset. In all the variables, the *RMSEsd* is as large as the *RMSEm* confirming that the variability is common to all the dimensions in the dataset, and is not restricted to certain state variables.

4.1.2 Quadrotor 6DoF Model

We saw in Section 2.3 that, in the case of quadrotors, several analytical models have been presented in the literature; all are derived from first principles, and differ in complexity, the types of aerodynamic effects accounted for, and the way the propulsion system is modelled. Out of all those models, we took most inspiration from that proposed by Bouabdallah ([34]) since it is one of the most comprehensive (and therefore flexible), and is fully nonlinear, and includes all the most important aerodynamic effects.

Some research groups addressed aerodynamic effects that Bouabdallah neglects, for example the effect of blade flapping [98, 188]. For heavy flying machines, the larger size

of the propellers and the magnitude of the aerodynamic forces make this effect relevant, particularly in fast forward flight, but given the relatively small size of our blades (10cm in radius) and the light weight of our platform, we can comfortably neglect these effects.

The full derivation of the dynamic equations of the X3D platform, complete with the detailed modelling and identification of the propulsion group¹⁴ is presented in Appendix A, since its length and level of detail would interfere with the main focus of the thesis. It is not necessary to read it to understand the following sections, but occasional reference to it will be helpful. (And since its length reflects the time taken to develop it, so the author would be grateful to anyone who takes the time to read it through!).

Various authors have used analytical models with statically estimated parameters to aid the design of control systems for quadrotors ([188, 34]), but to the best of our knowledge, the following section describes the first example of applying a fully fledged system identification technique to a miniature quadrotor. For the first time, the aerodynamic and control parameters have been directly estimated from flight test data, and comparisons are presented between predicted and experimental data.

Model fitting

Before undertaking the parameter estimation, we report for convenience the core expressions of the model dynamics obtained in Appendix A that predict the system accelerations¹⁵:

$$a_x = -(H_{0x} + H_{1x} + H_{2x} + H_{3x}) - \frac{1}{2}C_{xy}A_c\rho u|u| - \sin(\theta)g \quad (4.14)$$

$$a_y = -(H_{0y} + H_{1y} + H_{2y} + H_{3y}) - \frac{1}{2}C_{xy}A_c\rho v|v| + \sin(\phi)\cos(\theta)g \quad (4.15)$$

$$a_z = \frac{T_0 + T_1 + T_2 + T_3}{m} + \cos(\phi)\cos(\theta)g \quad (4.16)$$

$$\alpha_x = \frac{qr(I_{yy} - I_{zz}) + J_r q \Omega_r + l(T_3 - T_1) - h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jx}}{I_{xx}} \quad (4.17)$$

$$\alpha_y = \frac{pr(I_{zz} - I_{xx}) + J_r p \Omega_r + l(T_2 - T_0) + h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jy}}{I_{yy}} \quad (4.18)$$

$$\alpha_z = \frac{pq(I_{xx} - I_{yy}) + J_r \dot{\Omega}_r + \sum_{j=0}^3 (-1)^j Q_j + l(H_{0y} - H_{1x} - H_{2y} + H_{3x})}{I_{zz}}. \quad (4.19)$$

¹⁴We call propulsion group the combination of motor, blade and electronic speed controller.

¹⁵Equations 4.14-4.15 express the longitudinal and lateral acceleration (a_x, a_y) as functions of the quadrotor attitude and of the aerodynamic forces. Equation 4.16 computes the vertical acceleration (a_z) as a function of rotor thrusts and platform orientation. Finally 4.17-4.19 predict the angular accelerations ($\alpha_x, \alpha_y, \alpha_z$) as functions of rotor thrusts and torques.

As explained in Section 3.3.1 the PID stabilization system is integral part of the X3D dynamics. Since in this section we will explicitly identify some of its parameters, we report its equations (obtained in Section A.4)¹⁶:

$$u_p = K_{SDpr}(u_{pi} - b_{pt}) - K_{Dpr}q + K_{Ppr} \int q - K_{SPpr} \int (u_{pt} - b_{pt}) \quad (4.20)$$

$$u_r = K_{SDpr}(b_{rl} - u_{rl}) - K_{Dpr}p + K_{Ppr} \int p - K_{SPpr} \int (b_{rl} - u_{rl}) \quad (4.21)$$

$$u_y = K_{SDya}(u_{ya} - b_{ya}) - K_{Dy}r + K_{Py} \int r - K_{SPy} \int (u_{ya} - b_{ya}) \quad (4.22)$$

$$u_t = u_{th}. \quad (4.23)$$

As seen in Appendix A, this analytical model is associated with the extended state vector \mathbf{x} of the form:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r, h_{pt1}, h_{pt2}, h_{rl1}, h_{rl2}, h_{ya1}, h_{ya2}, \Omega_1, \Omega_2, \Omega_3, \Omega_4]^T, \quad (4.24)$$

which in addition to the usual attitude and velocity variables ($u, v, w, \phi, \theta, \psi, p, q, r$), contains also the instantaneous rotor velocities ($\Omega_1, \Omega_2, \Omega_3, \Omega_4$) and the state of the stabilization system ($h_{pt1}, h_{pt2}, h_{rl1}, h_{rl2}, h_{ya1}, h_{ya2}$).

Now that we have a mathematical expression for the dynamics of our quadrotor system, we can say something about the level of coupling between the degrees of freedom in the model, at least under the assumptions made in devising it.

Starting with the angular accelerations ($\alpha_x, \alpha_y, \alpha_z$), we can see from equations 4.17-4.19 that in a situation involving only slow motion, the rotational dynamics in each of the three axes is to a first approximation decoupled¹⁷. For low rotational and linear velocities (i.e. for q, r small and H_{ij} small), the roll motion depends mainly on T_3 and T_1 and therefore ultimately on the roll control u_{rl} . Under the same assumptions the pitch motion depends mostly on the pitch command u_{pt} , and the yaw motion is essentially driven by the yaw control u_{ya} . From the update equation for the angular velocities (equation 3.6) we see that the decoupling between the axes is maintained while equation 3.3 tells us that each of the three Euler angles defining the attitude is coupled with at least two of the angular velocities. Equations 4.14-4.16 reveal that, as a consequence of gravity, even the linear

¹⁶Equations A.58-A.61 calculate the inputs (u_p, u_r, u_y, u_t) to the motor controllers (see Section A.3) as a functions of the control inputs ($u_{pt}, u_{rl}, u_{ya}, u_{th}$).

¹⁷By decoupled we mean that the motion in one axes does not affect the motion of the others and vice versa.

accelerations under moderate dynamic motion (H_{ij} small) are dependent on the platform pose, which in turn obviously depends on the angular velocities.

In summary, we can therefore expect that in the slow dynamic regime when drag and the Coriolis effect can be neglected, an error in modelling one of the angular velocities will have a relatively small impact on the others. However, any error in one or more of the angular velocities will build up over time into an attitude error that will then have a considerable impact on the linear velocities. Under more dynamic conditions, where the effects of both linear and rotational velocities increase, coupling between all six degrees of freedom is to be expected.

The model just examined is a good example of how forces and torques are naturally expressed in terms of accelerations in body coordinates, this supports the argument of Abeel *et. al.* ([5]) who suggests that a model in acceleration space should be easier to learn.

All the mechanical parameters (dimensions, mass, moments of inertia) can be accurately estimated using our CAD model of the X3D (see Section A.2), but unfortunately such a model cannot help in estimating the control and aerodynamic parameters. These consist of all the controller constants ($K_{SDpr}, K_{Dpr}, K_{Ppr}, K_{SPpr}, K_{SDpr}, K_{Dpr}, K_{Ppr}, K_{SPpr}, K_{SDy}, K_{Dy}, K_{Py}, K_{SPy}$) none of which are disclosed by the quadrotor's manufacturer, as well as some of the aerodynamic constants (C_d, C_{xy}) which are not possible to estimate accurately without dedicated measurement facilities¹⁸.

In order to identify the parameters of the model, we will adopt the maximum likelihood procedure that we have already used for the ATTAS aircraft (see Section 4.1.1). For both training and validation we chose a window length of $T = 100$ steps, as a compromise between limiting the computation and taking long range interactions into account; since the training dataset (`datax3d_2`) is large, this window size also guarantees a large number of training windows ($W = 30$), a key requirement for the evolutionary methods that we will use in Section 5.7. For each data window, we again start our state from a pre-recorded value from the dataset; we then predict the system accelerations using equations 4.14-4.19, and integrate the state forward in time using equations A.48-A.57. As for the ATTAS, the errors between the predicted state¹⁹ and its experimentally recorded counterpart are

¹⁸An estimation of C_d can be produced by collecting torque measurements during the static thrust test (e.g. [98]), while for directly estimating C_{xy} wind tunnel measurements are necessary.

¹⁹Although for the X3D we defined an extended state, for the purpose of optimization we use the standard state $\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]$ since the additional variables cannot be measured experimentally.

used to drive the optimization (i.e. the minimization of the covariance of the residuals see Section 4.1.1).

The first step is to produce a suitable set of starting values for all the parameters of our model. Given the complexity of the model and the number of parameters involved, attempting to identify all of them simultaneously is not feasible, and so we will use a step by step procedure in which the different dynamic blocks of the model are identified separately. Although such separate identification might well produce a good model by itself, it would be better to use the separately identified parameters as the starting values for carrying out a full six degree of freedom joint identification.

The way in which we will produce our starting parameter set is clearly important, and we will need to make use of the mathematical formulation of the model as well as of our understanding of quadrotor dynamics in deciding how the model should be decomposed. The procedure therefore depends on our domain knowledge about the quadrotor platform, and the specific set of parameters we achieve will be conditioned by our choices; we consider this as an inherent weakness of parametric models (at least for the type of problem we consider in this thesis), since no other way of defining the starting values of the parameters is available.

We start by considering only equation 4.16 (and therefore a_z); since the propulsion group dynamic has already been identified, the only remaining free parameter in this equation is b_{th} . The remaining accelerations ($a_x, a_y, \alpha_x, \alpha_y$ and α_z) are not computed from their respective equations, but the pre-computed acceleration values are used instead. The various controls contribute additively to the speed of each rotor²⁰, so we can eliminate any influence of the inputs b_{rl} , b_{pt} and b_{ya} simply by setting all the controller constants to zero and optimizing the bias parameter b_{th} alone. Equation 4.16 affects only the vertical velocity in body coordinates w , therefore only the error in this variable is used to compute the likelihood for the optimization of b_{th} . The starting value for b_{th} was set to -0.485 in accordance with the hovering throttle value found from our thrust stand experiments²¹. The LMA reliably converged, and the value obtained for b_{th} is reported in Table 4.4.

We then focussed our attention on the rotational dynamics of the vehicle, starting

²⁰As explained in equation A.40.

²¹From the static thrust experiments Figure A.5 we found that the hovering throttle varies in the range [0.42, 0.55] depending on the battery level. Pragmatically we set the starting value of b_{th} to the middle of the interval -0.485 . The negative sign for the value of b_{th} is needed because at hover b_{th} should cancel out the throttle input.

with equation 4.19 for the yaw motion. In this case the free parameters to be identified are the controller constants ($K_{Pya}, K_{Dya}, K_{SDya}$ and K_{SPya}) and the rotor drag constant C_d ²². The state variables directly affected by the yaw dynamics are ψ and r , and so the error in both these dimensions is used for the optimization of the parameters; for the remaining accelerations ($a_x, a_y, a_z, \alpha_x, \alpha_y$) the pre-computed values are used. In practice we experienced a problem with the convergence of the LMA. This was easily solved by using the r error alone to produce a first set of parameters that was then used to initialize the full optimization based on both ψ and r .

Next we tackled the pitch and roll motion using equations 4.17 and 4.18 to identify the parameters of the lateral and longitudinal controllers ($K_{Ppr}, K_{Dpr}, K_{SDpr}$ and K_{SPpr}). The state variables directly governed by the dynamic equations are p, q, ϕ and θ , all of which will be used to compute the model error. Running a first optimization in which only the rotational velocities p and q were used to compute the error, and then using the resultant parameters to do a full optimization taking into account the angular errors again proved to be a more reliable approach (i.e. the LMA would always converge).

Parameter	Value
K_{th}	1.06
C_d	-0.17
K_{Ppr}	0.0016
K_{Dpr}	-0.19
K_{SDpr}	0.34
K_{SPpr}	-0.00038
K_{Py}	0.0031
K_{Dy}	-0.107
K_{SDy}	0.14
K_{SPy}	-0.00038
C_{xy}	47.7
b_{th}	-0.42
b_{rl}	-0.0098
b_{pt}	-0.0033
b_{ya}	-0.0027

Table 4.4: X3D model based on first principles: values of parameters obtained using ML.

Finally we focussed on the lateral and longitudinal dynamics (equations 4.14 and 4.15) for which the only free parameter to be identified is the drag coefficient C_{xy} . Again, values from the pre-computed dataset were used for these accelerations not involved in the parts of

²²The coefficient C_d was initialized with the value used in [34] while the controller constants were initialized to small random numbers drawn from a normal distribution with mean zero and variance 0.1.

the dynamics under study (i.e. a_z, α_x, α_y and α_z). The forward and lateral linear velocities (u, v) are the only two state variables directly governed by equations 4.14 and 4.15 and so only the error in these is used during optimisation.

At the end of this rather long procedure, we obtained a full set of parameters (Table 4.4); we will refer to this model as *X3DAccPar*.

We tested the model *X3DAccPar* on a randomly selected window of data taken from the validation dataset `datax3d_3`; the results are plotted in Figure 4.3, along with the control inputs given to the model (Figure 4.4).

The very good quality of fit for the angular velocities p and q is immediately clear, confirming that the combination of the PID equations and the first principle equations is indeed appropriate. The good fit for p and q translates directly into good predictive abilities for the angular variables θ and ϕ . However those variables show the effects of integration, with the small error in angular velocities building up with time as we would expect, but still maintaining good prediction. The yaw velocity prediction appears qualitatively correct but is delayed, possibly indicating that the value estimated for the blade drag coefficient C_d is too low; as a result the heading is underestimated although qualitatively correct. The lateral linear velocity v shows generally good qualitative prediction, but its value is lower than the measured value in the second part of the data window where the roll angle ϕ also exhibits its largest error. Similarly, the forward velocity u appears to be influenced by the error in θ in the second half of the data window. The vertical velocity w is the most accurate of the three linear velocities, and as a result the altitude error is small. The x and y position predictions suffer from the heading error which produces an erroneous re-projection of the body velocities u and v , and their predictions quickly drift away from the real positions. The error obtained by our model in the particular window plotted in Figure 4.3 can be compared with the error evaluated across multiple windows (RMSEm) using the CR value as explained in Section 4.1.1. In Table 4.5, we see that for most of the variables the error is in line with the average error. The three exceptions are the angle θ and the linear velocity u , which are predicted better than in the average situation, and the yaw angular velocity which has a larger error than the average in the dataset.

Since the author is well trained in flying the real X3D helicopter, there was an opportunity to fly the model manually to obtain a qualitative intuitive evaluation. To do this we created a rudimentary simulator by interfacing our dynamic model to a 3D visualization

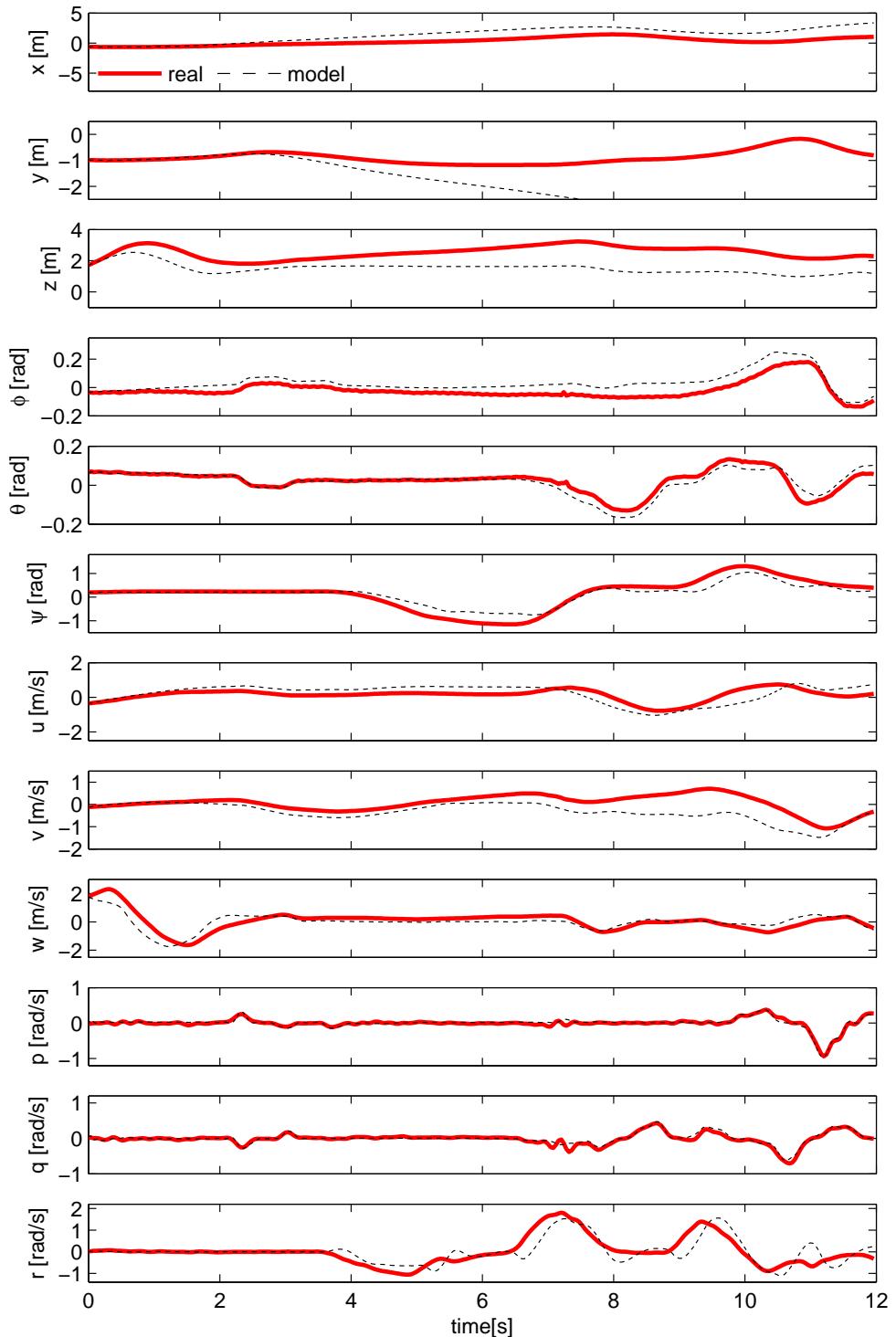


Figure 4.3: Proof of match: the state evolution predicted by the *X3DAccFP*, versus the measured real state.

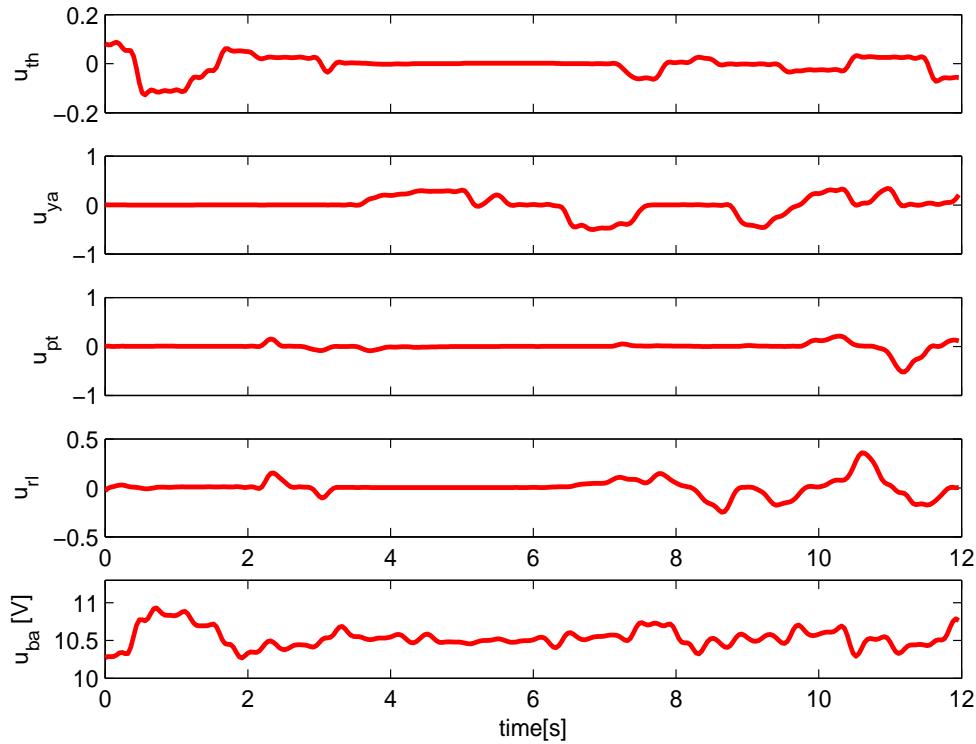


Figure 4.4: Control inputs given to the *X3DAccFP* model during the time window in Figure 4.3.

	RMSE	CR
u	0.37 m/s	0.43
v	0.52 m/s	0.69
w	0.40 m/s	1.08
ϕ	0.057 rad	0.64
θ	0.025 rad	0.25
ψ	0.24 rad	1.27
p	0.039 rad/s	0.67
q	0.049 rad/s	0.87
r	0.36 rad/s	2.43
<i>total</i>	0.87	

Table 4.5: Prediction error of the *X3DAccFP* model computed over the window of data plotted in Figure 4.3.

of a quadrotor. The same RC controller and serial interface used to fly the quadrotor for data collection were used to provide the inputs to our simulator. To the extent that a 3D visualization can allow comparison²³, the model was felt to replicate the behaviour of the real quadrotor very well. Both the type of motion and its speed were judged very realistic. Of course, since the model is completely deterministic and therefore not noisy, while air disturbances affect the flight of the real X3D, the model was much smoother, and this was a noticeable difference.

We then turned our attention to a more comprehensive evaluation of the predictive performance of our model by testing it across all the windows in the validation dataset. Applying the same validation metrics used for the ATTAS aircraft (RMSEm, RMSEsd and RAEm as defined in Section 4.1.1) gave us the results in Table 4.6. The results for

	RMSEm	RMSEsd		RAEm
u	0.9	0.41	m/s	1.17
v	0.7	0.27	m/s	1.17
w	0.4	0.15	m/s	0.52
ϕ	0.1	0.04	rad	1.02
θ	0.1	0.067	rad	0.88
ψ	0.2	0.17	rad	0.72
p	0.06	0.027	rad/s	0.21
q	0.06	0.025	rad/s	0.18
r	0.1	0.12	rad/s	0.47
<i>total</i>	1.3	0.38		

Table 4.6: Prediction error of the *X3DAccFP* model computed on the validation dataset.

the RMSEm metrics show that the yaw velocity has the largest error (among the angular velocities); this is approximately twice as great as the error in p and q . This error in r translates into an error for the heading ψ larger than that in ϕ and θ . Among the linear velocities, u and v show the largest errors and the prediction of the vertical velocity w the smallest. The RAEm metric compares the *X3DAccFP* model with a *constant model* that would simply predict no changes in the state (see 2.2.5 for details), and its values confirm the trends seen in the plots of Figure 4.3, with low relative errors for the rotational velocities p and q and the vertical velocity w . The worst performances are obtained for the angle ϕ and for the linear velocities u and v , the predictions of which are on average no better than a *constant model*. In examining those results, we need to keep in mind

²³The perception of depth and distances that a 3D visualization provides is of course very different from the reality and limits our ability to make comparisons.

that variations of performances have to be expected between different windows, since these metrics are averages. It should also be noted that the RMSEsd is in general of the same order of magnitude as the RMSEm, indicating that the model exhibits a wide range of performances in different parts of the datasets.

The *X3DAccFP* model has very good predictive abilities, but as discussed in the beginning of this it would be better to identify all model parameters simultaneously based on the error over the whole state vector. Unfortunately, in practice all our attempts at identifying the model parameters simultaneously led to failure. More precisely we obtained a model that barely predicted the means of the state variables and did not even qualitatively represent their dynamic variation but that scored a likelihood almost as high as the one obtained by optimizing the model parameters in stages. Is likely that such a solution represents a local maximum of the likelihood function in which the LMA algorithm gets trapped when all the model parameters are optimized simultaneously. This underpins our observation that although the *X3DAccFP* model is qualitatively good, its quantitative performance is in some cases (e.g. u and v velocities) close to that of a *constant model*.

This result points us towards reexamining the basic assumption that all OE methods make about noise in the system, namely that the process noise is negligible. Because of the way we represent the dynamics in our system, any error in prediction is effectively magnified by process of integration, and even a good model can lead to a level of error that, leads to implausible results, when used to jointly optimize the model parameters.

It could of course be argued that this is not due to the complexity of the modelling task, but is instead due to our suboptimal choice of model (i.e. we oversimplified the system dynamics). We will comment on this in Sections 4.2.2 and 5.7 when investigating the automatic design of quadrotor models.

4.2 Black-Box Models

While we have seen that techniques based on first principle models can be very effective, there are obviously situations in which an in-depth knowledge of the systems concerned is simply not available, and of course those are the ones we are most interested in. In Section 2.3 we described how black-box models have been widely used in the domain of vehicle modelling with good results, and here we take two of the most common techniques,

namely the nearest neighbour classifier (*NN*) and the feed-forward neural network (*MLP*) and apply them to both our toy car and our quadrotor platforms.

In Section 2.3 we discussed how modelling a rigid body, by predicting linear and angular accelerations is a very sound strategy, since it allows us to make the most of the physical laws of motion without requiring any further knowledge about the specific robotic platform in question. To test the strength of this observation for both the *MLP* and the *NN* models, we will investigate two different approaches: modelling in acceleration space, and modelling in velocity space (we introduced the two concepts in Section 2.2.1).

The models will be trained using standard supervised learning methods; this assumes that input and output data for the model are available during the training. In other words, we will not be integrating the model forward in time to predict the evolution of the state, but instead we will use the measured state and controls as the inputs for the training, and the pre-computed accelerations (or velocities in the case of a model in velocity space) as the output targets. For both function approximators described in this section, the training method is an instance of the EE type of methodology discussed in Section 2.2.5, and therefore we will again operate under the assumption that the inputs (\mathbf{x} and \mathbf{u}) to our models are error-free. By using pre-computed values for the target outputs, the error metric will not reflect the effects of error integration, and so comparing the models obtained in this section with other models which do not make this assumption enables us to test the assumption in practice.

In the case of the models predicting velocity, the training metric will be based solely on the velocity, and not on the potentially inaccurate acceleration, and so the comparison between the velocity based and acceleration based approaches will also allow us to understand how the noise present in the pre-computed accelerations (discussed in Section 3.2) affects the outcome of the training.

Some versions of the experiments presented in this section were published by the author and his co-workers in [234], but we have since added several new examples (namely the *carVelNN*, the *carAccMLP* the *X3DVelMLP* and the *X3DAccMLP* models). The experiments already published have been repeated here using exactly the same dataset as for all the other models in the thesis, thus allowing for a more meaningful and exact comparison of the different methods.

4.2.1 Nearest Neighbour (NN)

The nearest neighbour algorithm (*NN*) is among the simplest machine learning algorithms that can be applied to a regression problem. As we will see, its abilities are somewhat limited in our application domain, but it will provide us with a meaningful baseline against which to compare other approaches.

Given a training dataset consisting of inputs expressed in feature space, along with their associated outputs, the nearest neighbour algorithm works by turning the regression problem into a simple classification problem. Any new input is associated with the closest training point according to some appropriate metric of distance in feature space. The output associated with the selected input is extracted from the training dataset and returned as the prediction. The underlying assumption with this method is obviously that points close together in the feature space are associated with points that are close together in the output space; in other words, the function to be learned is assumed to be in some sense smooth. However, since the technique returns only the single closest point as output, it is unable to produce a smooth interpolation between data points.

No assumption whatsoever is made about the nature of the data, therefore the methodology can deal equally well with linear or non-linear relationships. Noise in the data is clearly a potential problem, since any noise in the training data will transfer directly to the predicted output. Since this method is nothing more than a principled way of “replaying” the training data, the number of examples present in the training set, the features chosen as inputs, and the distance metric used, are all of paramount importance. In practice this method will produce good results when large datasets with limited amounts of noise are available, and when samples provide good coverage of the parts of the feature space for which predictions are required.

A more detailed analysis of the properties of the *NN* algorithm can be found in [159], but the key fact we want to emphasise is that numerous extensions to the *NN* concept have been explored in the supervised learning literature in order to address the drawbacks just mentioned. A straightforward extension is the k -nearest neighbours algorithm, in which instead of selecting the closest point to the input, the k nearest neighbours are taken and their associated output averaged to produce the output result. This concept opens the possibility of using a weighted contribution of the neighbours according to some specific kernel function, which leads to the whole family of locally weighted regression (*LWR*)

methods.

Since the data used to compute the predictions are local, the data can be organized into data structures that make retrieving the neighbours a very efficient operation (e.g. KD trees, ball trees etc.). Incorporating new data is very straightforward, since it is just a matter of extending the training dataset, and for this reason these methods can be used for on-line learning. Both of these qualities make these techniques very useful in domains like robotics where the data have large dimensionality and on-line learning is a welcome addition ([201]). The type of kernel function used, their parameters, optimal ways of determining the parameters, and efficient ways of making predictions have been and still are active topics of research; see [10] for a comprehensive review.

The need to retain the complete training dataset is often mentioned as a drawback in terms of memory requirements for the nearest neighbour algorithm; fortunately with modern computers, the size of the datasets used here is by no means problematic. In our case, since we use a form of batch processing, even a straightforward implementation where a simple search through the dataset is used to find the nearest neighbour proved to be sufficiently fast.

Nearest Neighbour Velocity Modelling: Toy Car

In this first example we make use of the *NN* classifier to learn a velocity model for our toy car. In other words we learn the function of equation 3.12:

$$[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (4.25)$$

in which the velocities $[\mathbf{v}, \boldsymbol{\omega}]$ at time $t+1$ are predicted given the current state and inputs.

As a first step we need to define both the input feature space for the *NN*, and an appropriate metric over it. A simple choice for the feature would be the vector formed by the concatenation of \mathbf{x} and \mathbf{u} , with the Euclidean distance as the closeness metric. However, such a naive implementation does not take into account the fact that the throttle and steering control inputs can take only discrete values (see Section 3.3.3):

$$u_{th}, u_{st} \in \{1, 0, -1\}. \quad (4.26)$$

As a consequence of this, the multidimensional feature space is discrete in two of its dimen-

sions, and so, the meaning of the degree of closeness between points will be significantly affected. For instance it is easy to see how a change in just one of the control signals could produce a change in distance much larger than that produced by a set of smaller changes in all the other state variables. This effect obviously depends on the metric used in the feature space, and we could of course design a metric such that these effects are minimized. Instead we take a different but in many ways more straightforward approach. As we are happy to use knowledge about the input and output spaces of our systems, we can exploit the fact that at any time the couple (u_{th}, u_{st}) can take only one of nine different configurations:

$$(u_{th}, u_{st}) \in \{(0, 0), (0, 1), (1, 0), (1, 1), (-1, 0), (0, -1), (-1, -1), (1, -1), (-1, 1)\}. \quad (4.27)$$

This allows us to split the feature space into nine different subspaces, and therefore to use nine different *NN* classifiers, one for each input combination in 4.27. The feature vector is thus defined by the concatenation of \mathbf{x} and the continuous input u_{ba} . For each of the classifiers every feature is therefore continuous, and so the Euclidean distance constitutes a sensible metric. However, since each component of the feature vector may have a different characteristic magnitude, we first apply a normalization, scaling each component by its standard deviation measured across the whole dataset. It could be argued that a more sophisticated scaling than a simple normalization could be beneficial (see Section 2.2.4), but since defining such scaling factors would ultimately require knowledge about the specific platform, we prefer to use a normalization that relies only on the data.

In Figure 4.5 we can see an example of state prediction made using the *carVelNN* model that was trained²⁴ on the `datasetcar_6` and tested on a randomly chosen 12 seconds windows taken from `datasetcar_7`.

To produce such plot and also to compute validation metrics for the models, we propagated the model forward in time using a procedure similar to that used for the model based on first principles (Section 4.1.1). This was done as follows: at the beginning of the data window (t_{0w}) the state was initialized with a value from the recorded dataset, and then at each subsequent time step, the previously predicted state was fed into the model and used to predict the current state of the system. The length of the data windows was

²⁴For consistency with the other types of models we use the word training also for the *NN* classifier, although the training amount to nothing more than scaling the collected data.

set to $T = 300$ time steps which is again a compromise between computation, accounting for long range effects and providing a sufficient number of data windows²⁵. Choosing the same window length as was used for the X3D model based on first principles is also in line with our idea of making choices that are platform independent. The additional variables x, y, ψ which are not part of the state are updated as per equations 3.17 and 3.18; these are plotted in Figure 4.5 along with the state variables to give an understanding of the effects of error integration.

Not surprisingly, given the simplicity of the NN algorithm, the prediction results are quite poor, but we can still make some interesting observations by looking more closely at the plots. In the forward direction, the model clearly under-predicts the linear velocity u , and perhaps more importantly does not always produce a negative velocity in response to a negative throttle input. The lateral velocity v is also characterized by poor predictions, but this is less surprising as its relationship to the state of the system is undoubtedly very complex, since this velocity is determined by the interplay between the car's lateral acceleration and the frictional force developed by the car tyres on the carpet surface. Due to anisotropy and non-linearity, frictional forces are notoriously difficult to model; we will see this in many of the car models that we will analyse. The rotational velocity shares the fate of the forward velocity, being underestimated when the car is going forwards and being badly predicted when the car is going backwards.

Since the model directly predicts the state of the system, the state prediction error depends only on the state and input variables, and not on the time index t . This is clearly not true for the derived quantities x, y, ψ that are computed by integration, as their error tends to grow as the time increases. Having a state prediction error that does not grow with time is a clear benefit of any method based on velocity prediction. At the same time, since no integration is taking place the model does not have any explicit “memory” of its state, making inertial effects difficult to model; as a result, a single bad prediction can drastically change the state of the system. This can force the whole classifier into the wrong area of the feature space, and therefore can have an effect on all the subsequent predictions. The forward velocity u also shows another drawback of the NN algorithm, namely its lack of interpolation abilities; this sometimes leads it to produce a velocity prediction varying in magnitude even if the remainder of the state vector and the inputs are not changing very

²⁵In Section 5.9.2 we will see in detail the effects of using training windows of different lengths.

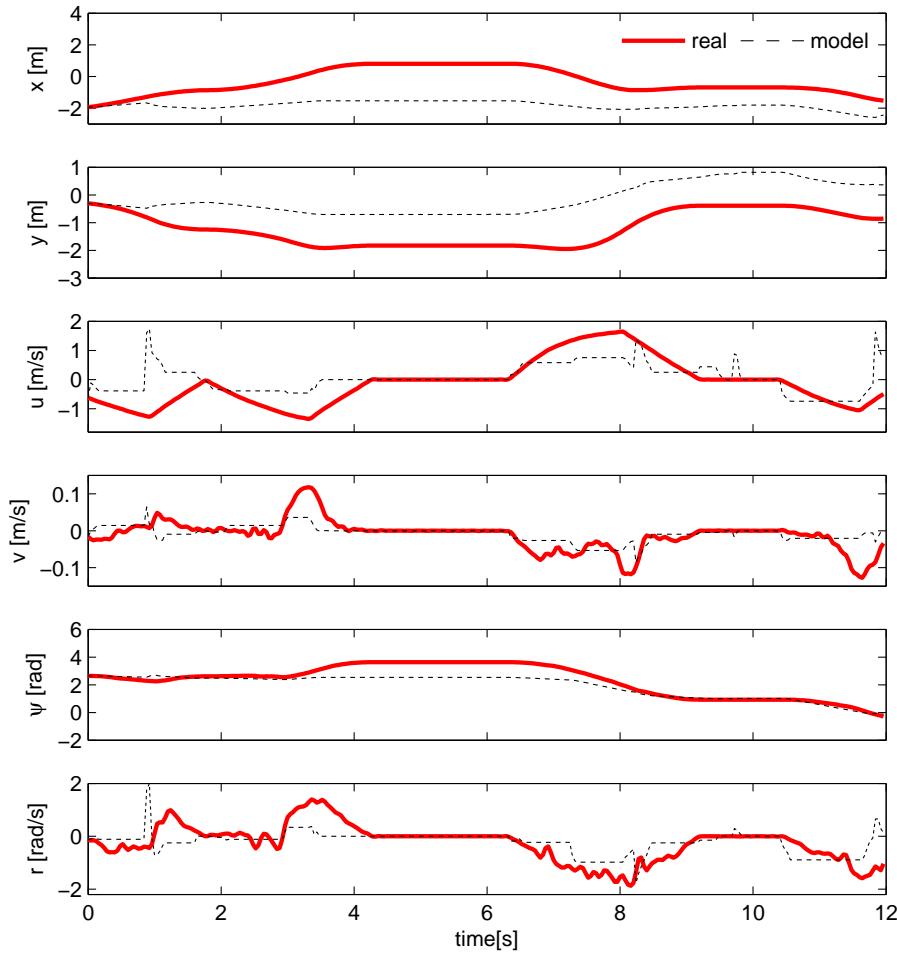


Figure 4.5: Proof of match: the state evolution predicted by the *carVelNN* model versus the measured real state. The result of the simple *carVelNN* model is poor, in particular in response to a negative throttle input.

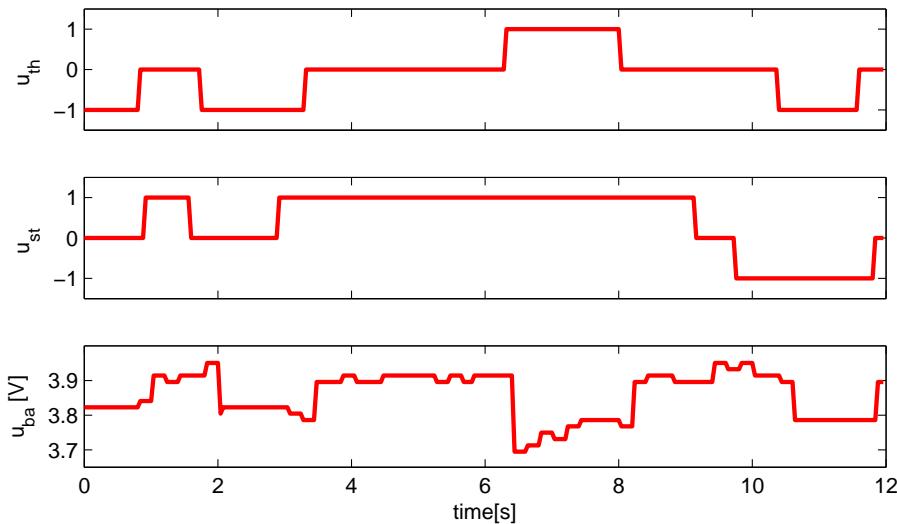


Figure 4.6: Control inputs given to the *carVelNN* model during the time window in Figure 4.5.

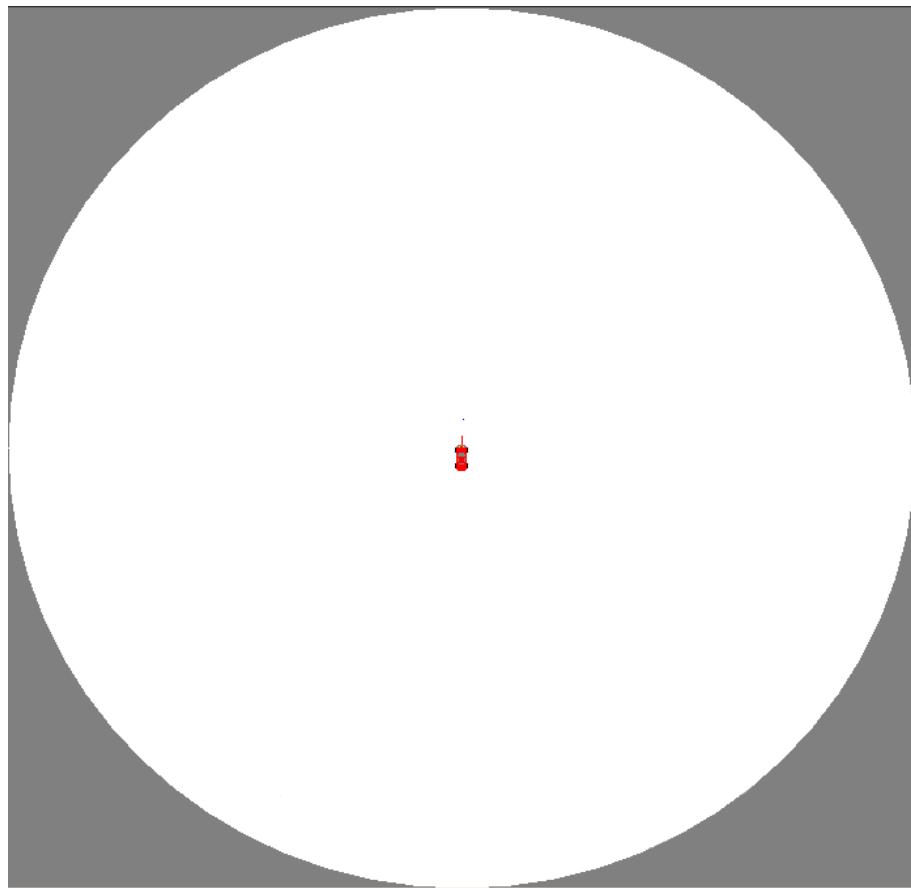


Figure 4.7: Simple 2D visualization used to test the car model. The white area represents our test arena, the car is drawn to scale.

much (e.g. see $t \simeq 10s$).

To confirm those observations, we tested the obtained model by manually driving it via a computer keyboard after it was interfaced with a 2D visualization (see Figure 4.7) to produce a simple car simulator. The driver, very familiar with the behaviour of the real car used during data collection, immediately noticed all of the above problems. The car was almost impossible to drive backwards, and was not able to turn when going backwards. A quirk was also noticed; when a left steering input was applied, a drastic and unrealistic increase in speed (forward or backward) was predicted. Another important weakness was also recognized, in that the car often appeared to move at a fixed speed even though no throttle command was applied, instead of slowing down as a result of friction. Even more unrealistic was the fact that by giving a right steering command in the presence of no throttle the car would move backwards, although very slowly.

The performance of the model computed over the window of data depicted in Figure 4.5 is shown in Table 4.7. Looking at the CR coefficients²⁶ we see that on the randomly chosen

	RMSE	CR
u	0.62 <i>m/s</i>	0.97
v	0.03 <i>m/s</i>	0.91
r	0.55 <i>rad/s</i>	1.13
<i>total</i>	0.83	

Table 4.7: Prediction error of the *carVelNN* model computed over the window of data plotted in Figure 4.5.

window, the model performances are in line with their averages on the dataset, indicating that we have picked a window representative of the model’s behaviour.

Figure 4.5 certainly gives us some insight into the ability of the *carVelNN* model, but to have a better understanding of how the model performs we need to evaluate it across the dataset since is reasonable to believe that in some portions of it the model would work better than in others. For each of the state dimensions, we report the *RMSEm* the *RMSEsd* and the *RAEm* (the validation metrics were computed as explained in Section 4.1.1).

	RMSEm	RMSEsd		RAEm
u	0.64	0.15	<i>m/s</i>	0.63
v	0.034	0.015	<i>m/s</i>	0.72
r	0.49	0.19	<i>rad/s</i>	0.57
<i>total</i>	0.82	0.22		

Table 4.8: Prediction error of the *carVelNN* model computed on the validation dataset. The large *RMSEsd* shows how the predictive ability varies widely in different parts of the dataset.

For all four state variables, the value of the *RAEm* indicates that the model is only moderately better than a *constant model*, confirming the impression that we had from the POM plot in Figure 4.5. We should not get confused by the fact that for u and v the *RAEm* is similar while the *RMSEm* is not. This is correct, and the *RSRSEm* for the variable u is low due to the fact that the car never reaches high lateral velocities; the *RAEm* in contrast is a relative metric so it automatically rescales when the magnitude of the signal changes.

The *RMSEsd* reported in Table 4.8 shows how, the predictive ability varies for all the dimensions depending on which window of data the model was tested on. For v and r the standard deviation is equal to half of the *RMSEm*, indicating non-negligible differences in behaviour across different data windows.

²⁶The comparative ratio coefficient CR is defined in Section 4.1.1

Nearest Neighbour Acceleration Modelling: Toy Car

We turn our attention now towards the idea of using our function approximator to model accelerations instead of velocities. In this section we use the *NN* classifier to learn a model of the form of equation 3.13:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (4.28)$$

With this method we predict accelerations in body coordinates, and then exploit the very general and well known laws of rigid body physics (equations 3.14 and 3.15) to compute the state of our system at time $t+1$.

The considerations made in the previous section about the discrete nature of the inputs u_{th} and u_{st} still apply in this setting, so we again use the idea of splitting the dataset into 9 subsets each associated with an input pair as in Section 4.2.1. As a result the feature space used as the input to the classifiers is exactly the same as in the previous example, but now a vector of accelerations is associated with each of the input samples. The inputs have again been normalized by scaling them by their standard deviation, and the Euclidean distance is again used as the closeness metric.

The POM of the predictions made by the new *NN* model (we will call it *carAccNN*), is plotted in Figure 4.8, again using a randomly chosen window. The training is again done using pre-computed data, but for the POM and the performance evaluation instead the predicted accelerations are integrated at each time step using equations 3.14 and 3.15. The remaining variables $[x, y, \psi]$ are computed using equations 3.17 and 3.18. For direct comparison with the *carVelNN* model we used the same training and validation datasets, the same size for the data windows and also the same randomly chosen data window for the POM plots.

The first impression is undoubtedly of a mixed result. The prediction of the forward velocity u is generally good, but for v and r , prediction errors tend to build up as a result of the integration, and the velocities drift away from their recorded values. In some cases this pushes the velocities to values much larger than those in the training dataset. At this level of error the predicted state trajectory in some parts of the data is uncorrelated with its actual counterpart, as can be seen for variable v in the second part of the time window. The fact that the linear velocity u appears better behaved than v and r suggests that the integration is not a problem in itself, but becomes problematic when the predicted

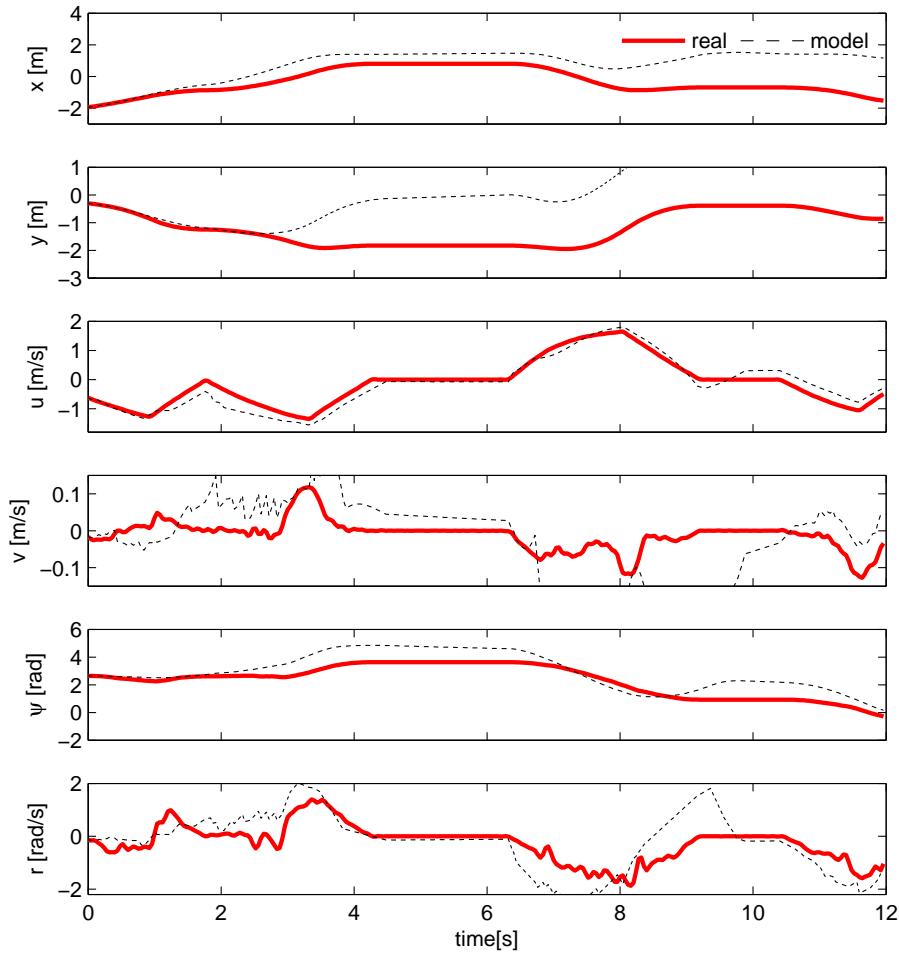


Figure 4.8: Proof of match: the state evolution predicted by the *carAccNN* model versus the measured real state. It is evident how poor acceleration prediction as a result of error accumulation through integration can produce large state errors.

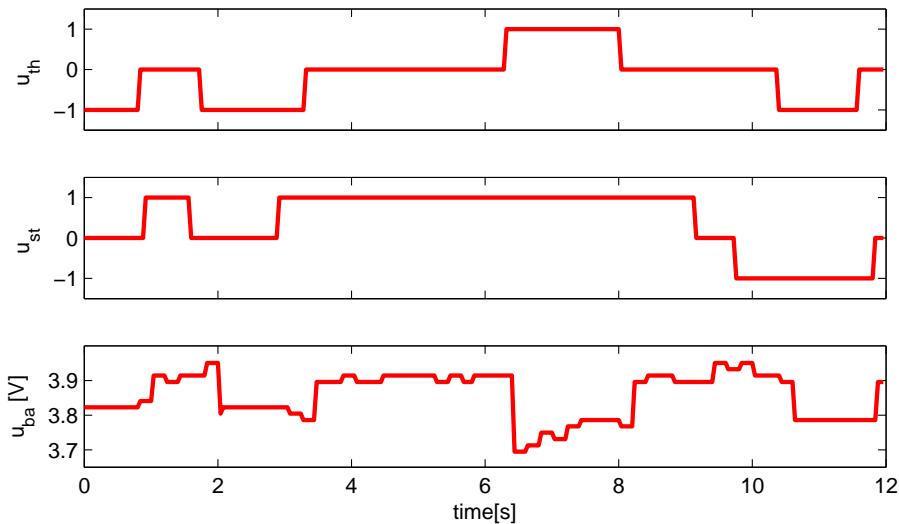


Figure 4.9: Control inputs given to the *carAccNN* model during the time window in Figure 4.8.

accelerations are not sufficiently accurate. Another effect of integration, but this time a positive one, is also visible, in that the velocities are much smoother than those predicted by the *carVelNN* model, as would be expected for the type of system that we are modelling. The effects of integration error are understandably even more pronounced for the derived variables x , y , ψ , in all of which it is clear how the error builds up as the integration time progresses; the y coordinate shows this most strongly.

In Table 4.9 we show the *RMSE* metric and the *CR* value computed over the window of

	RMSE	CR
u	0.25 <i>m/s</i>	0.39
v	0.22 <i>m/s</i>	0.47
r	0.70 <i>rad/s</i>	0.38
<i>total</i>	0.77	

Table 4.9: Prediction error of the *carAccNN* model computed over the window of data plotted in Figure 4.8.

data used to plot Figure 4.8. For all four variables, the *CR* value is low, and indicates that over the chosen window the model performs better than it does on average. It is interesting also to note that u and v have similar *RMSE*, although we would expect v to have a much smaller *RMSE* since the magnitude of v is generally much smaller than that of u (as we explained in the previous section).

Testing the trained model manually with the help of our rudimentary simulator revealed that the *carAccNN* model in general feels much more “natural” than the *carVelNN* model. By natural we mean that its ability to build up speed gradually and to slow down in absence of control inputs gives the driver a distinct feeling of its similarity to the real car. This feeling also arises from the fact that the model goes backwards and steers appropriately, and does not show sudden speed changes. Also, in the case of this model, steering commands in the absence of any throttle do not produce any motion. The ability to reproduce the inertial properties is the main reason why we advocate the use of an acceleration based model, since ultimately such a model is closer to the underlying physical system.

But the news is not all good, because after a few seconds of driving it was also clear that in some specific situations the model would behave incorrectly, achieving unreasonably high speeds, especially in the lateral direction, leading to a severe skidding behaviour not present in the real car. This appears to be due to one of the weaknesses of the *NN* classifier,

in that, since is not using any interpolation scheme, it can produce an association with a wrong (e.g. very noisy) state acceleration output that can push the state out of its envelope towards a region in which the performance of the classifier is poor.

For a more rigorous evaluation of the performance of the *carAccNN* model we computed the familiar validation metrics *RMSEm*, *RMSEsd* and *RAEm*. The values obtained are shown in Table 4.10.

	RMSEm	RMSEsd		RAEm
<i>u</i>	0.63	0.56	<i>m/s</i>	0.59
<i>v</i>	0.46	0.59	<i>m/s</i>	8.79
<i>r</i>	1.85	1.53	<i>rad/s</i>	2.37
<i>total</i>	2.09	1.63		

Table 4.10: Prediction error of the *carAccNN* model computed on the validation dataset. While the *u* error is on the same order of magnitude as that of the *carVelNN* model, the *v* and *r* dimensions are noticeably worse.

Looking at the *RAEm* metric, we immediately see that the performance of the model varies widely over the various dimensions; for the forward velocity *u* the predictions are considerably better than a *constant model*, while the prediction of the rotational speed *r* and in particular the lateral velocity *v* are much worse than would be achieved by simply predicting no change. This is in agreement with the poor performance of *v* shown on the window of data plotted in Figures 4.8 and 4.9.

Comparing the total *RMSEm* of this model with that of the *carVelNN* model shows a general decrease in performance. However, in the forward direction (*u*), the acceleration based model is actually better. This is noticeable also when comparing Figures 4.5 and 4.8. On the remaining two dimensions, *v* and *r*, the acceleration based model is worse than the *carVelNN* model with *RMSEm* of *v* being as much as an order of magnitude larger, and *RMSEm* of *r* more than three times as large.

The large standard deviation (*RMSEsd*) in all four dimensions indicates a very variable level of performance on different parts of the dataset, considerably larger than we saw for the *carVelNN* model.

Observations

The limited performance of the *NN* confirms that modelling our toy car is certainly not trivial. As expected, the most difficult relationship to learn is the lateral velocity *u*. Quali-

tatively we have seen some positive signs in favour of the concept of modelling accelerations, namely the smoothness of the predicted state and the “natural” feel of the model. However the quantitative results produced by the *carAccNN* show that, unless the acceleration predictions are good, the acceleration error tends to build up dangerously, affecting the state prediction accuracy.

4.2.2 Multi Layer Perceptrons Neural Networks (MLP)

In this section we will look again at the two possible modelling strategies (i.e. velocity space and acceleration space), using a more sophisticated function approximator, namely a neural network. We will test the two strategies on the toy car problem, in order to have a direct comparison with the previous section, and also on the more challenging X3D quadrotor problem.

A neural network is a very well known and well studied type of function approximator, and a plethora of applications to regression and modelling problems can be found in the literature. Their ability to learn complex non-linear input-output relationships regardless of the specific type of problem makes them a favourite choice when domain knowledge is not available. Many years of research in the field have delivered very effective and well understood training methodologies which have also contributed to the popularity of this regression method. The term “neural network” is very general, and from the computational point of view simply refers to a set of interconnected functional units; very different computational abilities can be achieved depending on the type of units used, their number and their connection topology. Discussing the properties and topologies of artificial neural networks is clearly outside the scope of this work; there are excellent accounts in [23, 24, 159].

In this section, we will make use of feed-forward multi layer perceptron networks (*MLP*) quite possibly the most widely used topology in artificial neural networks. For the network we have chosen a very standard fully connected topology with two hidden layers (see Figure 4.10 and Figure 4.13). A hyperbolic tangent activation function (*tanh*) is used for the hidden layers, while the output neurons are simply linear. It has been proven (Hornik *et al.* [102]) that this type of network is capable of approximating with arbitrarily small error any bounded continuous function²⁷ and therefore is a very suitable candidate to model

²⁷The number of hidden units required depends on the function to be approximated.

systems without requiring of knowledge about specific platforms.

In the following sections we train our models using the popular backpropagation algorithm ([197]) since such method has been successfully applied to a wide range of modelling problems including aircraft ([20, 192]). The backpropagation algorithm adapts the connection weight on the basis of the square error between the predicted and the target output; when the quadratic error function and no regularization terms are used ([142]) it is equivalent to maximum likelihood training.

Many other methods could be used for determining the network weights, from those following a gradient descent approach (e.g. quickprop, delta-bar-delta) to those based on both the slope and the curvature of the error surface (second order methods e.g. Levenberg-Marquardt and Gauss-Newton)²⁸. The main advantage of these methods is that some of them (especially the second order ones) require lower training epochs than backpropagation. Since we carry out the training off-line, training time is not of paramount importance in our settings and backpropagation was found to be an effective choice.

In the case of a *MLP*, discrete inputs do not represent an additional challenge to the learning process, and for this reason we do not need to explicitly divide the training dataset into subspaces as we did in the case of the *NN*. Each of our models is therefore formed by one *MLP* for each of the velocities (or accelerations) that we wish to predict.

To produce a model that can account for input delays we used the simplest of the methods described in Section 2.2.1; in addition to the current state, we also fed the network with the inputs $\mathbf{u}_{t-D:t}$. The maximum input delay was chosen equal to $D = 5$ (equivalent to 200ms) since a larger delay would increase the number of network inputs excessively²⁹.

Both the input data and the target outputs were normalized by scaling them by their standard deviation over the whole dataset; this is often done for neural networks ([219]) since it makes the weight search behave better.

In all the training runs we initialized the connection weights randomly by drawing them from a Gaussian distribution with zero mean and standard deviation 0.1. Each data point in the training dataset represents an example used to train the network, and one training epoch corresponds to updating the network weights once for each of the examples in the dataset. To avoid over-fitting the early stopping method was used. With this technique,

²⁸See [198] for a good presentation of a variety of learning methods.

²⁹200ms is a reasonable maximum value for small electromechanical vehicles and controllers. The choice could be said to involve domain knowledge, but is does not involve knowledge of any particular vehicle.

the training is terminated if the *MSE* error computed over the validation dataset starts increasing while the *MSE* over the training dataset continues to decrease³⁰. A maximum of 1000 epochs was also set to terminate the algorithm when the error reached a plateau without showing signs of over-fitting.

When training a network using backpropagation, it is common procedure to pick the training examples from the dataset in a random order with the aim of avoiding biasing the learning towards a specific part of the function to be learned. Unexpectedly, we found experimentally that training the network using the training examples in the chronological order in which they were collected would actually produce much better results both in terms of the prediction error and of its standard deviation. In the first portion of the dataset the car is at rest and is commanded to move forward and speed up, eventually reaching its top speed. By starting the training from the beginning of the dataset we are effectively biasing the learning towards this specific part of the system envelope. Since it ultimately leads to better models, we decided to use the training data in chronological order, but we will comment on the possible significance of such a choice in Section 4.3.

Neural Network Velocity Modelling: Toy Car

As a first example of using a *MLP* we focus on modelling our toy car in velocity space, and so we tackle the learning of the relationship expressed by equation 3.12:

$$[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (4.29)$$

To predict each of the variables in the state vector $[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T$ we will use three identical networks each of which is trained independently (i.e. for each network, only the square error in the predicted variable is used to optimize the network weights). The input to each of the networks is made up of the current state \mathbf{x}_t , the current and delayed control inputs $\mathbf{u}_{t-D:t}$ and a bias term³¹ for a total of 22 inputs; for the hidden layers seven neurons were used (see Figure 4.10)³². The targets are different for each network, namely u_{t+1} , v_{t+1} and

³⁰When validation data is available this is a simple and effective way of ensuring that the weights of the network are not over-tuned to the specific dataset used for training ([198]).

³¹To simplify the implementation of the network, instead of having a specific bias connection for each of the neurons in the network we simply set one of the inputs to the constant value 1. Since the network is fully connected is not difficult to see that the two approaches are equivalent.

³²We experimented with network topologies with a larger number of hidden neurons, but this produced no clear benefit.

r_{t+1} ; but in all cases these come from the training dataset since we do not make use of the predicted state during the training phase.

Figure 4.11 shows the prediction produced by this model (*carVelMLP*) when trained on **datasetcar_6**, and tested on a randomly selected data window of 12 seconds (300 time steps) taken from the unseen dataset **datasetcar_7**. In the usual manner, both for the

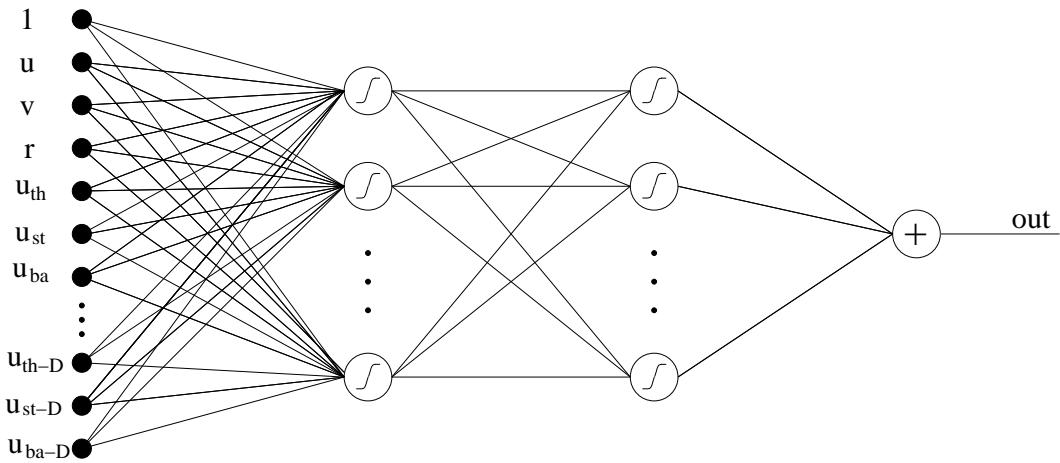


Figure 4.10: Structure of each of the three networks used for the *carVelMLP* and *carAccMLP* models. The only difference between the networks is the variable *out* being predicted ($out \in \{u_{t+1}, v_{t+1}, r_{t+1}\}$ for velocity space models and $out \in \{a_{x_t}, a_{y_t}, \alpha_{z_t}\}$ for acceleration space models).

POM and for measuring the performance of the model, the predicted state at time t is used as the input to the model equations in the following time step ($t+1$) to propagate the model forward in time. The additional variables x, y and ψ are integrated using equations 3.17 and 3.18. To ease the visual comparison, we have chosen for the POM plots the same data window previously used for the *NN* models. The results in Figure 4.11 show that across all the state variables the *carVelMLP* model is far superior to the *NN* models. The model produces a very good fit for the forward velocity u , and the fact that CR_u is equal to one (Table 4.11) tells us that this performance is on average just as good in other parts of the dataset. The lateral velocity is again confirmed as a very complex and noisy relationship to learn, but the model successfully learns the general trend of v while predicting smoother changes in speed than were measured. The rotational velocity r is qualitatively well approximated but for $t = 4s$ we can notice how, after reaching the maximum rotational speed, the model takes longer than the real car to stop its rotational motion.

The values of CR in Table 4.11 are all close to unity indicating that the behaviour

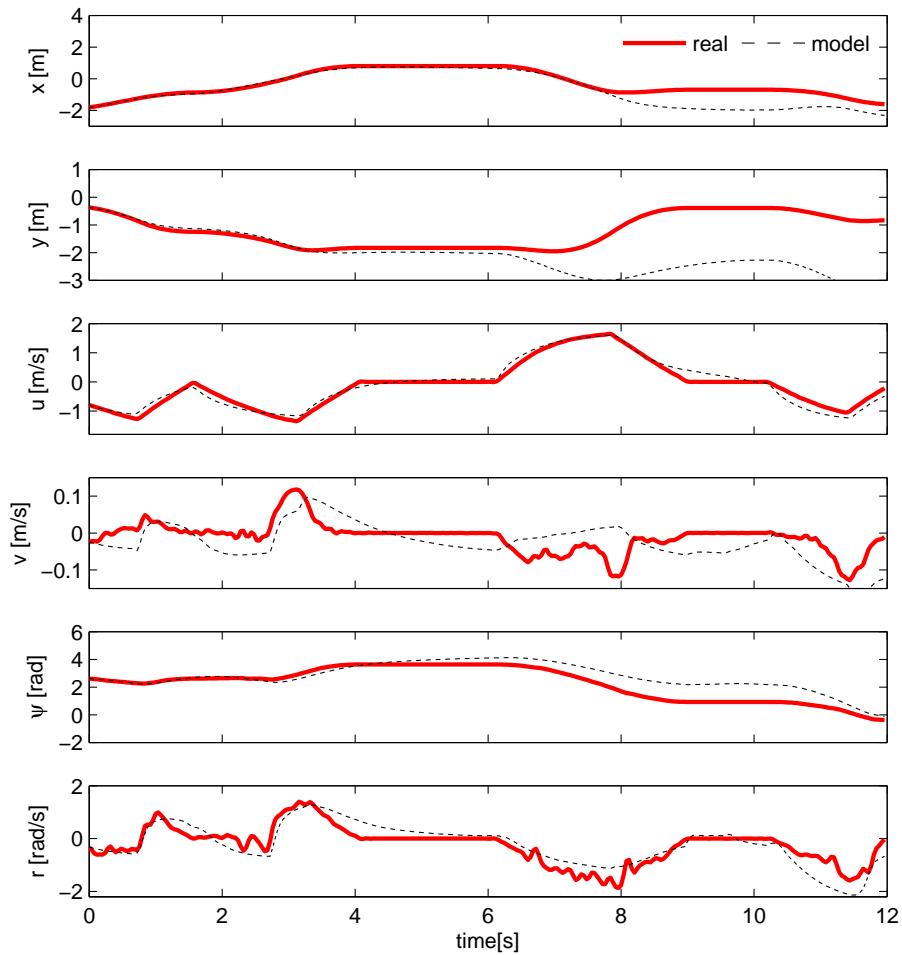


Figure 4.11: Proof of match: the state evolution predicted by the *carVelMLP* model, versus the measured real state. The model is clearly better than any of the *NN* models.

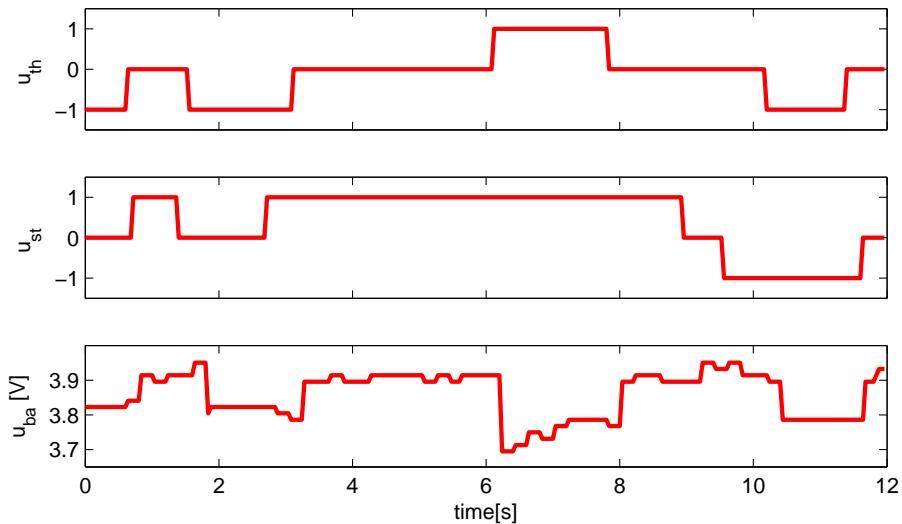


Figure 4.12: Control inputs given to the *carVelMLP* model during the time window in Figure 4.11.

analysed for the chosen random window is indicative of the average behaviour of the model.

	RMSE	CR
u	0.16 m/s	1.00
v	0.05 m/s	0.86
r	0.39 rad/s	1.13
<i>total</i>	0.42	

Table 4.11: Prediction error of the *carVelMLP* model computed over the window of data plotted in Figure 4.11.

As previously, we manually tested the obtained model before examining more comprehensive error metrics. The model immediately appeared very realistic, and showed the expected ability to speed up and reach a top speed, and also to slow down as friction comes into play when no throttle input is applied. Backward motion was similarly well behaved and consistent with the behaviour of the real car. When we described the real car in Section 3.3.3 we mentioned how, the car has asymmetrical turning radii due to its poor quality; the *carVelMLP* model was able to represent that, with a smaller left turning radius and a larger right one. However, one unrealistic effect was noticeable: even without any throttle input, the car tended to move forward slowly instead of standing still. Overall the model appeared stable, and no set of inputs produced unrealistic speeds.

In Table 4.12 we show the results of testing the model over the validation dataset. As was previously done for the *NN* models, we split the dataset into consecutive windows of length $T = 300$ steps and computed the usual performance metrics (see Section 4.1.1 for more details). Comparing the model with a *constant model* using the *RAEm* indicates clearly that the *carVelMLP* model has better prediction abilities for the yaw dynamics (r) and especially for the forward velocity. For the lateral velocity the *carVelMLP* model appears no better than what would be obtained by predicting no change. Comparing the *RMSEm* with the results obtained for the *carVelNN* model reflects the improvement that the plots of Figure 4.11 suggested. In the u dimensions the *RMSEm* is only 0.25 times the $RMSEm_u$ obtained by the *carAccNN* model (the best of the *NN* models in the u dimension); similarly the $RMSEm_r$ error is 0.39 times that of the *carVelNN* model. The standard deviations $RMSEsd_u$ and $RMSEsd_v$ confirm the same trend toward reduction indicating that the *carVelMLP* model is more consistent across the validation dataset than the *NN* models. In the case of the lateral velocity v , the performance is worse than

	RMSEm	RMSEsd		RAEm
u	0.16	0.046	m/s	0.17
v	0.06	0.017	m/s	1.40
r	0.34	0.09	rad/s	0.47
<i>total</i>	0.38	0.09		

Table 4.12: Prediction error for the *carVelMLP* model. This model shows a net improvement when compared with the *NN* models, both in terms of *RMSEm* and of *RMSEsd*.

that of the *carVelNN* model, confirming that the lateral velocity is not easy to learn; the $RMSEsd_r$ of the two models is however very similar.

Neural Network Velocity Modelling: X3D

It is perhaps not surprising that in the case of the toy car the *MLP* model has far superior abilities to the *NN* model, but our understanding of the physics governing the toy car tells us that being constrained to move in only three dimensions simplifies the general problem. For this reason it is of interest to compare our modelling techniques on a more complex problem like the one represented by the X3D quadrotor.

In the case of the X3D quadrotor we are dealing with six degrees of freedom, and so when predicting velocities we are after the relationship of equation 3.2:

$$[\mathbf{v}, \boldsymbol{\omega}]_{t+1}^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (4.30)$$

In what follows we will refer to this model as *X3DVelMLP*.

As with the toy car, a single *MLP* is used for each of the output variables. The six identical networks are trained independently; for each network, the weights are optimized in order to minimize the square error of the single variable being predicted. The input to each network is formed by the concatenation of the current state \mathbf{x}_t , the current and past controls $\mathbf{u}_{t-D:t}$ and a bias term, for a total of 40 inputs. The target outputs for training are the velocities \mathbf{x}_{t+1} from the collected dataset. For both of the hidden layers we used 15 neurons³³ obtaining the topology depicted in Figure 4.13. The *X3DVelMLP* model was trained with backpropagation using data from the training dataset `datax3d_2` (for details of the training see Section 4.2.2) .

³³As for the toy car, we also experimentally tested networks with a different number of neurons in the hidden layers, but noted no clear benefit in having larger or smaller sizes.

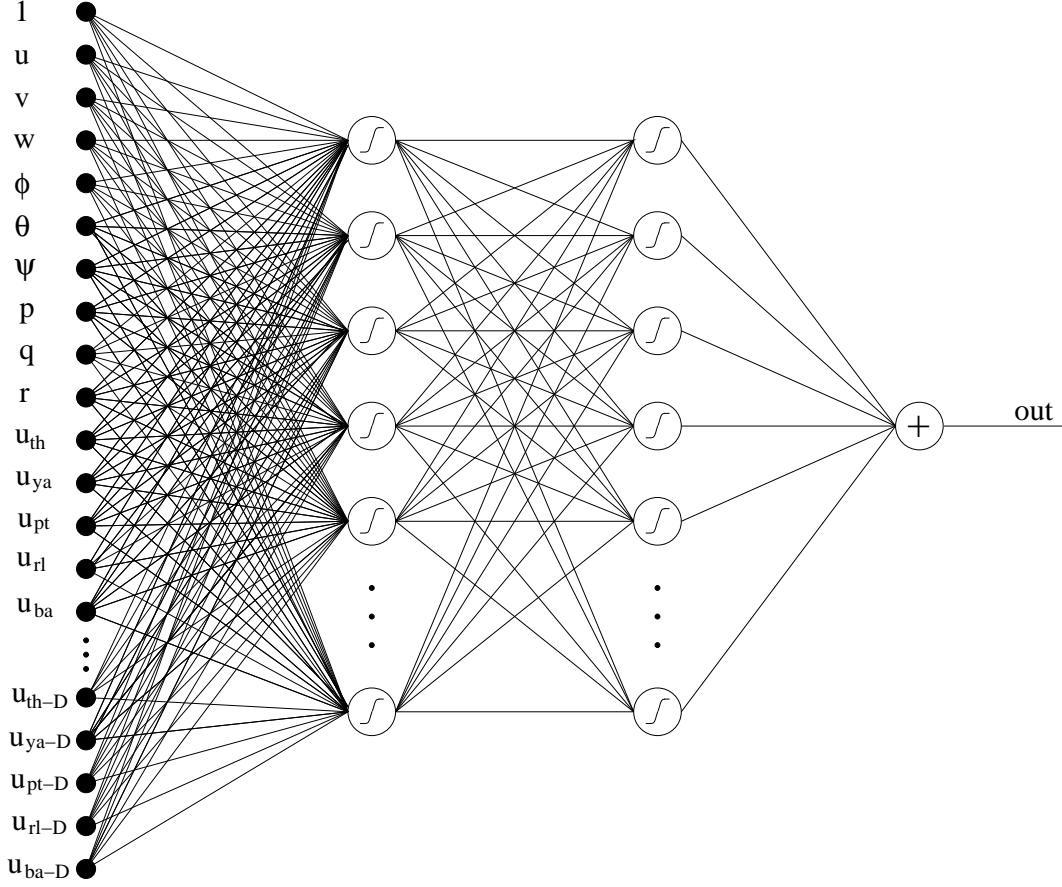


Figure 4.13: Structure of each of the six networks used for the *X3DVelMLP* and *X3DAccMLP* models. The only difference between the networks is represented by the variable *out* being predicted ($out \in \{u_{t+1}, v_{t+1}, w_{t+1}, p_{t+1}, q_{t+1}, r_{t+1}\}$ for models in velocity space and $out \in \{a_x, a_y, a_z, \alpha_x, \alpha_y, \alpha_z\}$ for models in acceleration space).

When testing the model against real data we take the usual approach of propagating the state forward in time. The velocities ($[\mathbf{v}, \boldsymbol{\omega}]$) and integrated angles ($\boldsymbol{\Phi}$) resulting from the prediction at time t are given as input to the model at time $t+1$ along with the controls ($\mathbf{u}_{t-D:t}$). Repeating this procedure for all the steps in the chosen data window (using equation 3.3 for the angle update) delivers the predicted state trajectories that we then compare against the validation data. The variables x, y, z are computed by integration using equation 3.9. In Figures 4.14 and 4.15 we can see the qualitative performance of the *X3DVelMLP* model on the same 12s window that was used for the POM plots of the *X3DAccFP* model. Looking at the predicted data, we can see that the performance of the model is good. The rotational velocities show good predictions, and this is particularly true for p and q ; however the yaw velocity r is sometimes underestimated ($t = 7s$). The good quality of the rotational velocities translates into reasonably good predictions of the

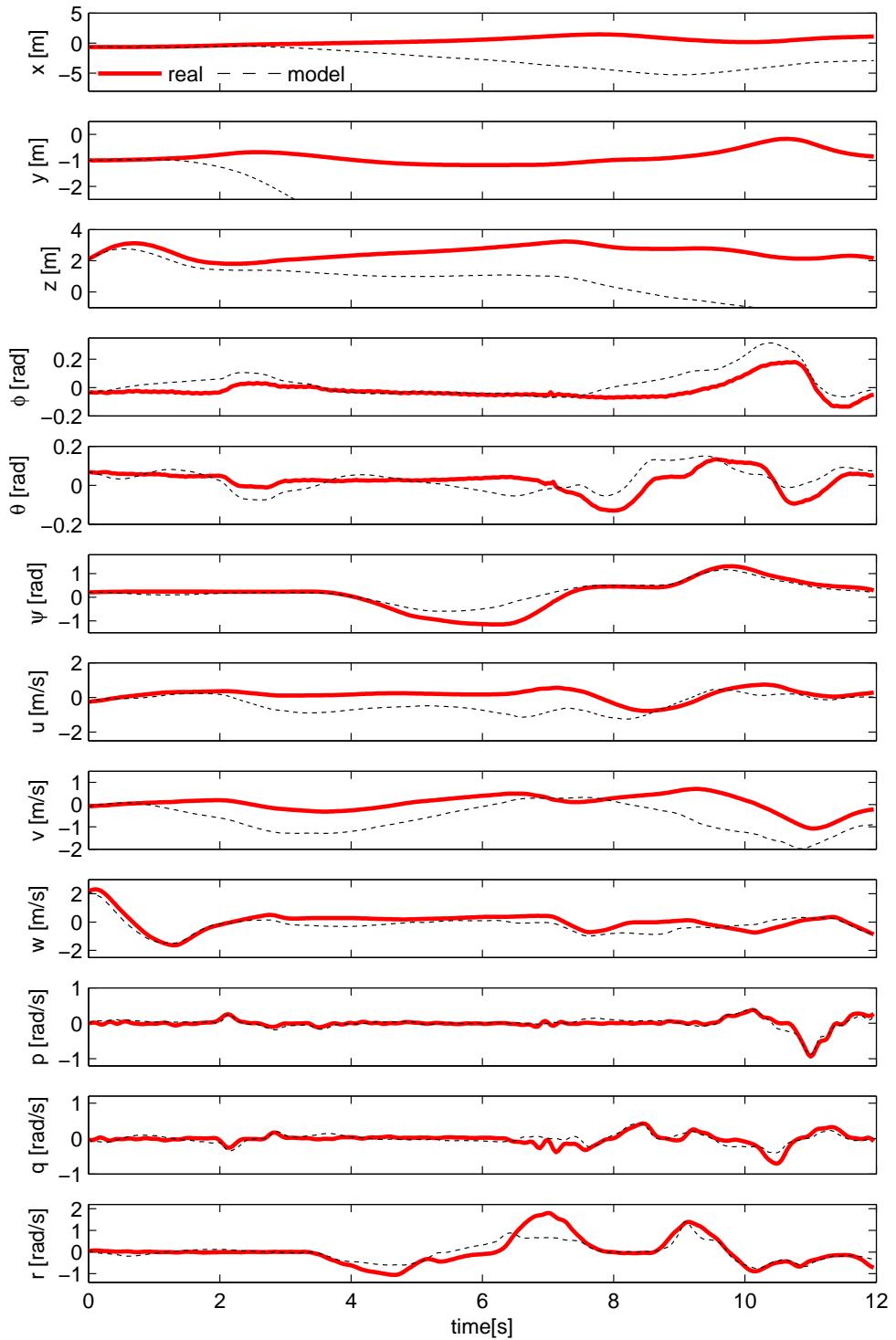


Figure 4.14: Proof of match: the state evolution predicted by the *X3DVelMLP* model versus the measured real state. The *MLP* achieves mixed results.

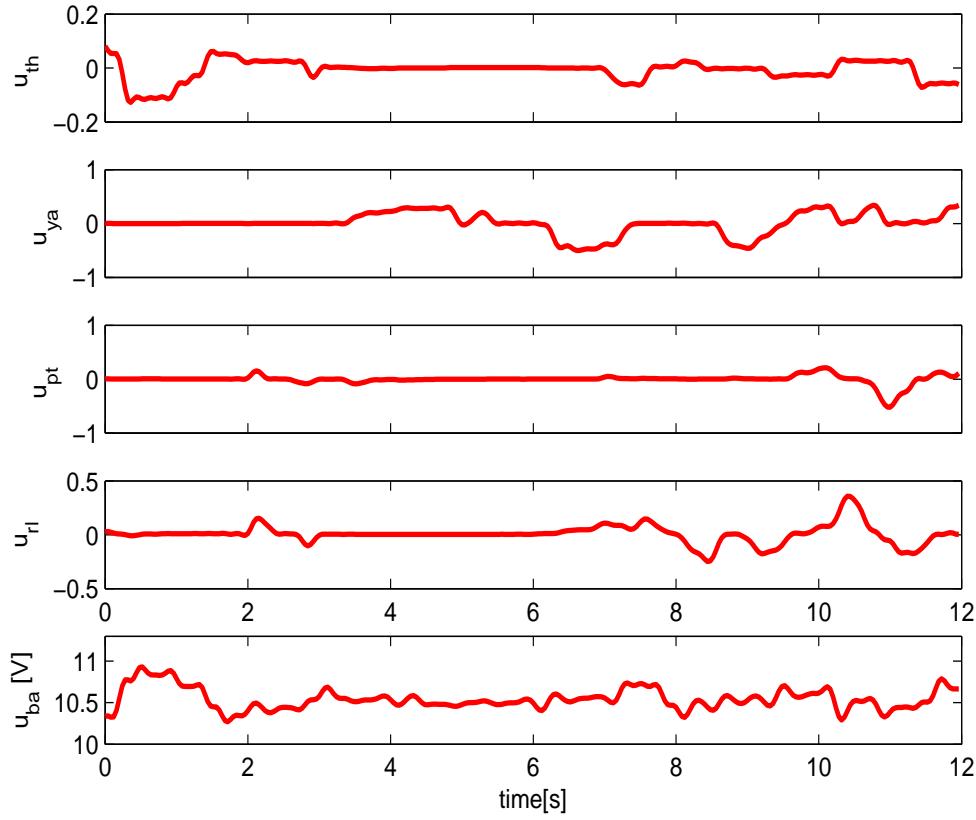


Figure 4.15: Control inputs given to the *X3DVelMLP* model during the time window in Figure 4.14.

attitude, but it is clear how as result of integration and coupling, the error in r propagates into all three angular variables (ϕ , θ and ψ).

The model predicts the vertical velocity w well, and the forward and lateral velocities (u and v) are qualitatively correct. As expected (since the accelerations a_x and a_y are directly related to the attitude), u and v show larger errors in the areas in which the predictions of θ and ϕ are poor. As a consequence of the errors in the linear velocities, the derived variables x,y,z have a tendency to drift away from the recorded trajectories, but overall their error is contained.

For most of the dimensions the CR value is not far from one (see Table 4.13) suggesting that on average the performance of the model is similar to what shown in Figure 4.14. The exception is the rotational velocity r , which is worse than the average across the validation dataset.

The model was then tested manually by interfacing it to a remote controller and a simple 3D visualization (as explained in Section 4.1.1). In general, the model seemed to

	RMSE	CR
u	0.70 <i>m/s</i>	0.85
v	0.82 <i>m/s</i>	0.83
w	0.40 <i>m/s</i>	0.78
ϕ	0.071 <i>rad</i>	0.88
θ	0.055 <i>rad</i>	0.73
ψ	0.28 <i>rad</i>	0.75
p	0.065 <i>rad/s</i>	0.78
q	0.11 <i>rad/s</i>	0.85
r	0.31 <i>rad/s</i>	1.8
<i>total</i>	1.23	

Table 4.13: Prediction error of the *X3DVelMLP* model computed over the window of data plotted in Figure 4.15.

	RMSEm	RMSEsd		RAEm
u	0.82	0.20	<i>m/s</i>	1.05
v	0.98	0.18	<i>m/s</i>	1.57
w	0.51	0.13	<i>m/s</i>	0.75
ϕ	0.081	0.030	<i>rad</i>	0.91
θ	0.074	0.024	<i>rad</i>	0.60
ψ	0.37	0.21	<i>rad</i>	1.49
p	0.082	0.037	<i>rad/s</i>	0.35
q	0.13	0.049	<i>rad/s</i>	0.48
r	0.17	0.15	<i>rad/s</i>	0.59
<i>total</i>	1.49	0.19		

Table 4.14: Prediction error of the *X3DVelMLP* model computed on the validation dataset.

have a slower response than the real flying machine, and this was particularly noticeable for the yaw motion. It was also noticed that in certain cases, when the pitch or roll angles were large, the model would not respond correctly to the control inputs, and recovering horizontal flight would become difficult, a behaviour that clearly does not correspond to the dynamics of the real helicopter.

The qualitative tests carried out so far are promising overall but as usual we need a more balanced quantitative measure of the model performance (we refer to Section 4.1.1 for details on the validation procedure). To make the comparison more meaningful we chose the same length for the validation windows as was used for the *X3DAccFP* model, namely $T = 300$ steps. The *RMSEm*, *RMSEsd* and *RAEm* metrics obtained for each of the dimensions are shown in Table 4.14. We start by looking at the *RAEm* which compares our model to the baseline *constant model*. The angular velocities p , q and r , the angle θ

and ϕ and the vertical velocity w are predicted by the *X3DVelMLP* model better than by a *constant model*. It is interesting to note that θ is predicted better ψ , although we know that the system is symmetrical. This agrees with our analysis of the POM plots. Although we saw in Figure 4.14 that the yaw angle ψ and the linear velocities u and v are qualitatively correct, they are no better in terms of error than the predictions of a constant model, and in fact ψ and v are considerably worse. The trend indicated by the *RAEm* is confirmed by the *RMSEm*, with p and q being more accurate than r , and ψ and θ being predicted much better than ψ .

When examining the *RMSEsd* to seek to understand how the prediction ability changes in different windows, for most of the variables we see standard deviations of magnitude comparable to half of the *RMSEm*. Performance therefore varies noticeably within the dataset.

Comparing the *RMSE_{total}* of the *X3DVelMLP* model with that of the *X3DAccFP* model indicates that the model based on first principles has better prediction accuracy. However, for some of the variables the *X3DVelMLP* is actually more accurate (i.e. for the variables u , θ , ψ) while for the remainder, the *RMSEm* of the *X3DVelMLP* model is only marginally higher.

Neural Network Acceleration Modelling: Toy Car

To complete our investigation of black-box models for the toy car we look now at training a *MLP* model to predict the instantaneous accelerations in body coordinates as per equation 3.13:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (4.31)$$

The acceleration model (*carAccMLP*) is again made up of three distinct networks all having the same topology and inputs as for the *carVelMLP* model (see Figure 4.10). The three networks predict a_x , a_y and α_z , respectively the forward, lateral and rotational accelerations. Each of the networks is trained independently using both input and output data from the recorded datasets, and its weights are optimized using backpropagation to minimize the square error between the predicted acceleration and its experimentally measured counterpart. During the test phase we again need to integrate the acceleration predictions in order to have a clear picture of the effects of error propagation. At every

time step we compute the updated state from the predictions of the three networks using equations 3.14–3.15; this forms the model inputs (along with the controls $\mathbf{u}_{t-D:t}$) at time step $t+1$.

Figure 4.16 shows the *carAccMLP* model tested on the same randomly selected data window used for the POM plots of the previous toy car models. Looking at the velocity prediction and visually comparing it with the result obtained with the *carVelMLP* model we can see a small improvement in prediction abilities. The improvement is visible in the velocities r and v which show a more accurate fit to the real data. In contrast to the *carVelMLP* model, this model does not appear to have an excessively smooth lateral response, but it is still unable to fit completely the fast dynamics of this variable. This is evident in the central part of the data window where the model predicts a positive lateral velocity instead of a negative one. In comparison to the *carVelMLP* model, this model has a much more accurate rotational speed response, both qualitatively and quantitatively.

We again manually tested the *carAccMLP* model to get a qualitative feeling for the model’s ability. As first we noticed that without any throttle input the car stands still indefinitely, correctly replicating the behaviour of the real car. When driving the car, the inertial effects in speeding up and slowing down are also replicated correctly both in the forward and in the backward directions. The model qualitatively reproduces the fact that the top speed of the car is limited, and that the right and left turning radii differ. The model also appears stable, and no set of inputs produces unrealistic speeds or positions. The *RMSE* of the model computed over the window of data used in Figure 4.16 is shown in Table 4.15. The *CR* values indicate that the lateral velocity v in the window of data

	RMSE	CR
u	0.13 m/s	0.72
v	0.05 m/s	1.41
r	0.36 rad/s	1.10
<i>total</i>	0.39	

Table 4.15: Prediction error of the *carAccMLP* model computed over the window of data plotted in Figure 4.16.

plotted in Figure 4.16 is not predicted as well as in the average case, whereas r and u are representative of the capabilities of the model across the whole validation dataset.

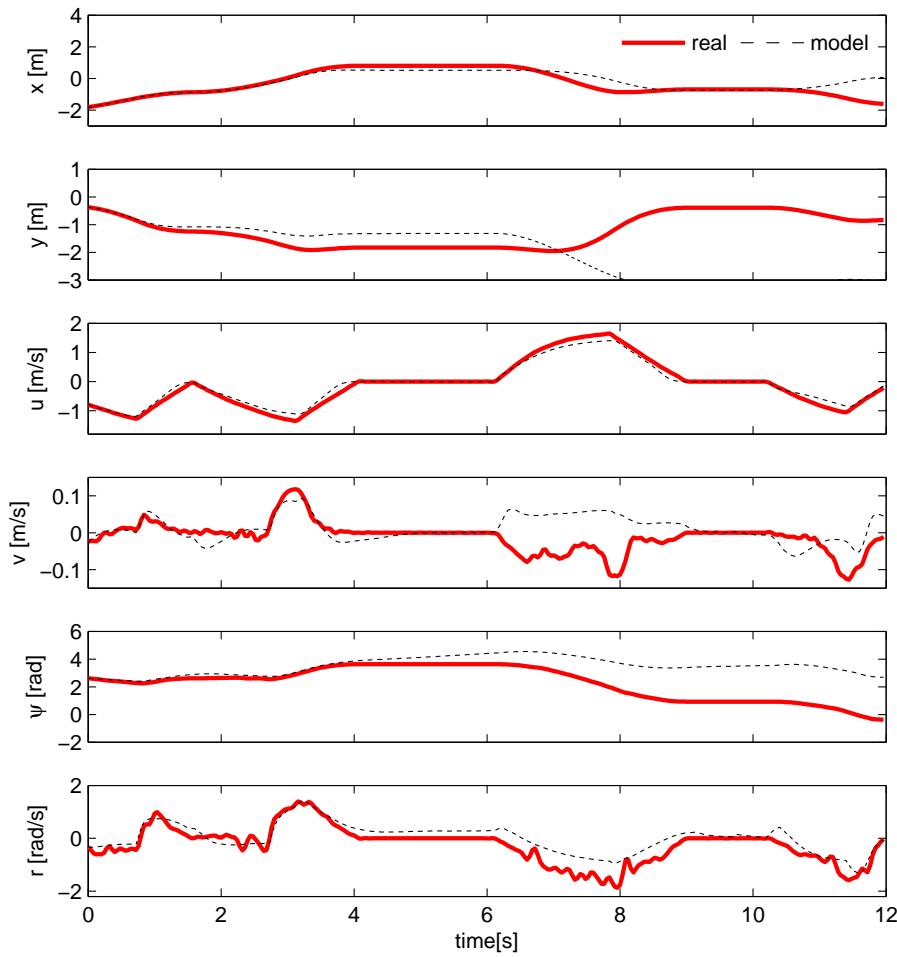


Figure 4.16: Proof of match: the state evolution predicted by the *carAccMLP* model versus the measured real state. The model shows very good prediction abilities even when compared with the *carVelMLP* model.

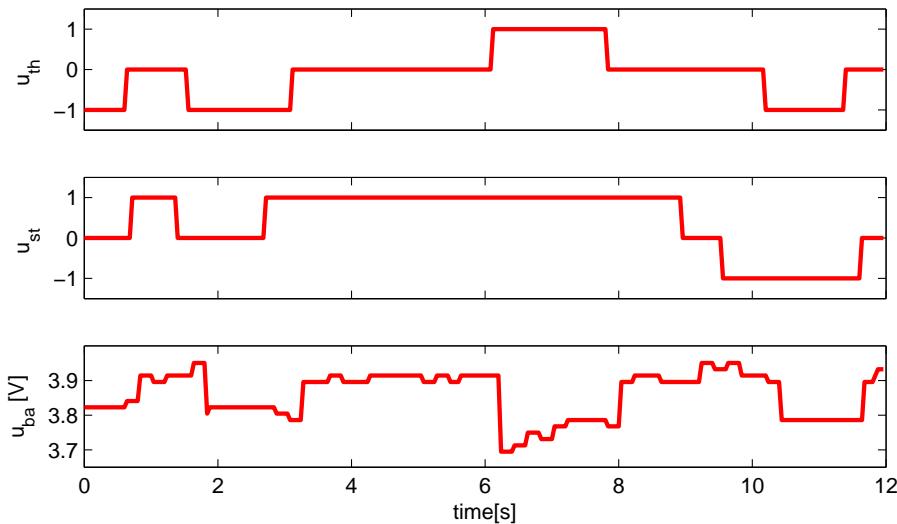


Figure 4.17: Control inputs given to the *carAccMLP* model during the time window in Figure 4.16.

Finally, we extensively tested the *carAccMLP* model by employing the usual performance metrics (see 4.1.1). To facilitate the comparison between models, we used the same window length used in validating the previous car models (namely $T = 300$ steps); the results are shown in Table 4.16. We already knew that we had a good model, so it was very interesting to compare its performance against the best model obtained so far, namely the *carVelMLP* model. The total *RMSEm* for both models is identical. A comparison of the error for each of the variables shows very similar performances between the two models for the forward and rotational velocities, but the prediction of the lateral velocity v is more accurate for the *carAccMLP* model. This is confirmed by the *RAEm* metric, which in the case of the model based on acceleration prediction also indicates that the lateral velocity v is more accurate than what would be obtained by predicting no change; this was not the case for the *carVelMLP* model.

Overall the *carAccMLP* model appears to be marginally better than the *carVelMLP* model since for equal total *RMSEm* it has smaller values of *RAEm*. However, given the *RMSEsd* of both models in all dimensions, we cannot exclude the possibility that on some specific parts of the dataset the *carVelMLP* is in effect a better model than its acceleration counterpart.

Neural Network Acceleration Modelling: X3D

Finally we want to test the idea of using acceleration based prediction on the complex and coupled problem of the X3D quadrotor helicopter. We therefore aim to obtain the relationship that delivers the platform accelerations given the current state and inputs, as per equation 3.4:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t), \quad (4.32)$$

we will call this model *X3DAccMLP*.

	RMSEm	RMSEsd		RAEm
u	0.18	0.06	m/s	0.19
v	0.038	0.02	m/s	0.76
r	0.33	0.14	rad/s	0.44
<i>total</i>	0.38	0.14		

Table 4.16: Prediction error of the *carAccMLP* model computed on the validation dataset. The predictive ability is marginally better overall than that of the *carVelMLP* model.

We aim to learn six input-output relationships (one for each component of the acceleration vector) and therefore our model is again formed by six neural networks of identical topology. We again used the network structure and inputs that we used for the *X3DvelMLP* model (see Figure 4.13). The model was trained using dataset `datax3d_2` and the validation was carried out using backpropagation on the dataset `datax3d_3`.

As in the previous section, the training does not involve integrations, but the performance of the model is tested by integrating the state forward in time using equations 3.5 and 3.7. For the validation we again used data windows (see Section 4.1.1) with a window length $T = 300$ time steps. Although they were not included in the performance metrics, the variables x, y, z , are also computed using equation 3.9 and plotted along with the state variables. In Figure 4.18 we plot the state trajectories predicted by the *X3DAccMLP* model along with the real ones recorded during data collection. Looking at the proof of match plots it is immediately clear that the *X3DAccMLP* is a good model, and that overall its behaviour appears rather similar to that of the *X3DVelMLP* model.

The pitch and roll rotational velocities are predicted well, but the fit of the yaw velocity r is not very accurate; the control u_{ya} is always correctly translated into a rotational velocity (e.g. $t \simeq 9s$). The error in r has obvious effects on the yaw angle with an error that is particularly evident in the last part of the time window. In terms of r the prediction of the *X3DVelMLP* model appeared more accurate, and the yaw error was also smaller. As a consequence of integration, the small p and q errors, and the error in r , propagate to θ and ψ and affect these predictions. At first glance the *X3DAccMLP* model appears more accurate in the prediction of θ and ψ , but both MLP model falls short when compared to the model based on first principles. The prediction of the vertical velocity w is accurate and generally as good as those produced by the *X3DAccMLP* and the *X3DAccFP* models; for all three models the accuracy of w yields a contained altitude error (z). As expected the attitude errors translate into errors in the forward and lateral velocities, both of which are less accurate than the predictions of the *X3DAccFP* and *X3DAccMLP* models.

The results of computing the *RMSE* for the model over the data window of Figure 4.18 are shown in Table 4.17. The value of the *CR* coefficient in almost all dimensions indicates that the chosen window is representative of the average performance of the model; the two exceptions are the yaw velocity (r) and the yaw angle (ψ), both of which behave worse on average than on the chosen window. We then tested the *X3DVelMLP* model

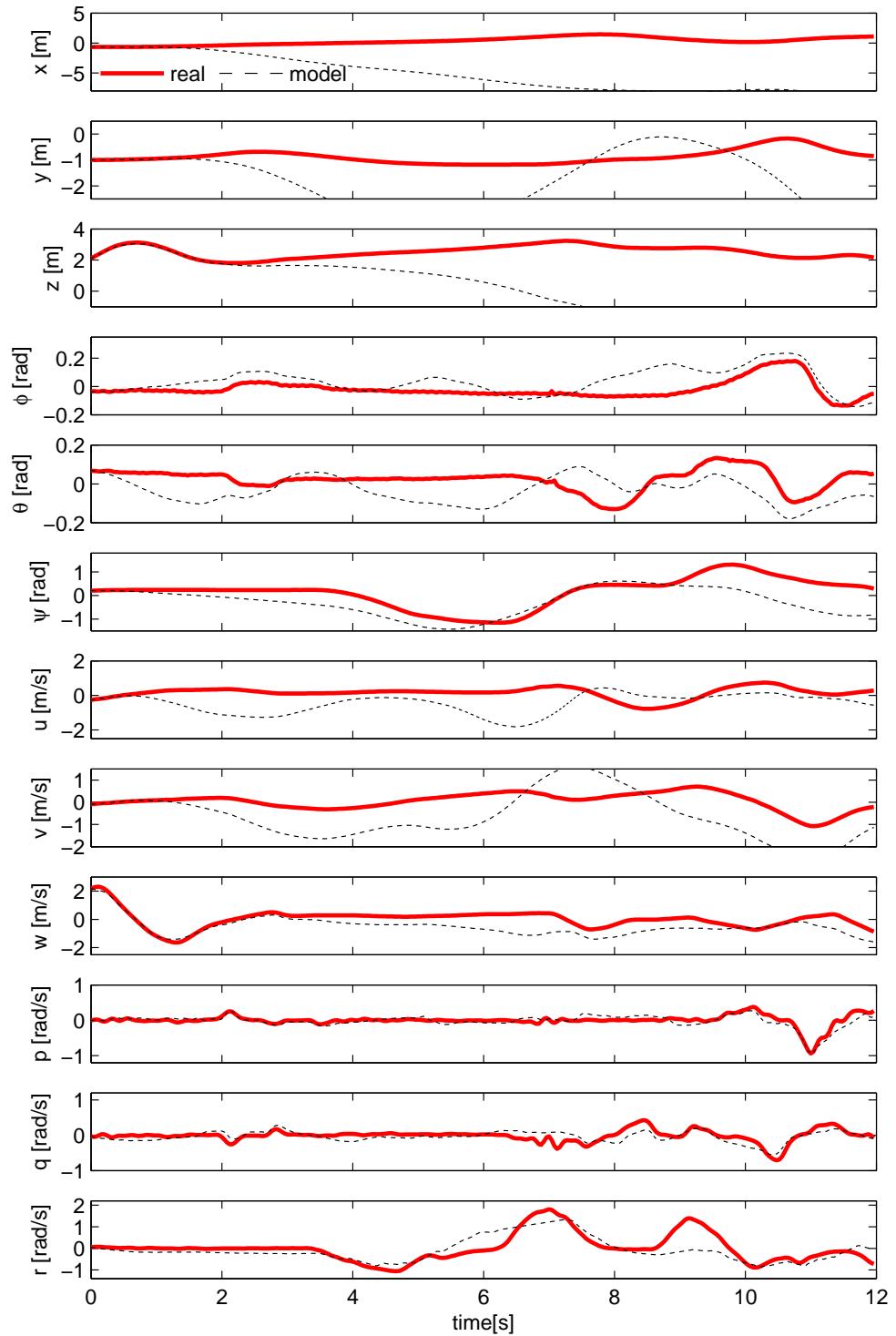


Figure 4.18: Proof of match: the state evolution predicted by the *X3DAccMLP* model versus the measured real state. The prediction results are worse than that produced by the *X3DVelMLP* model.

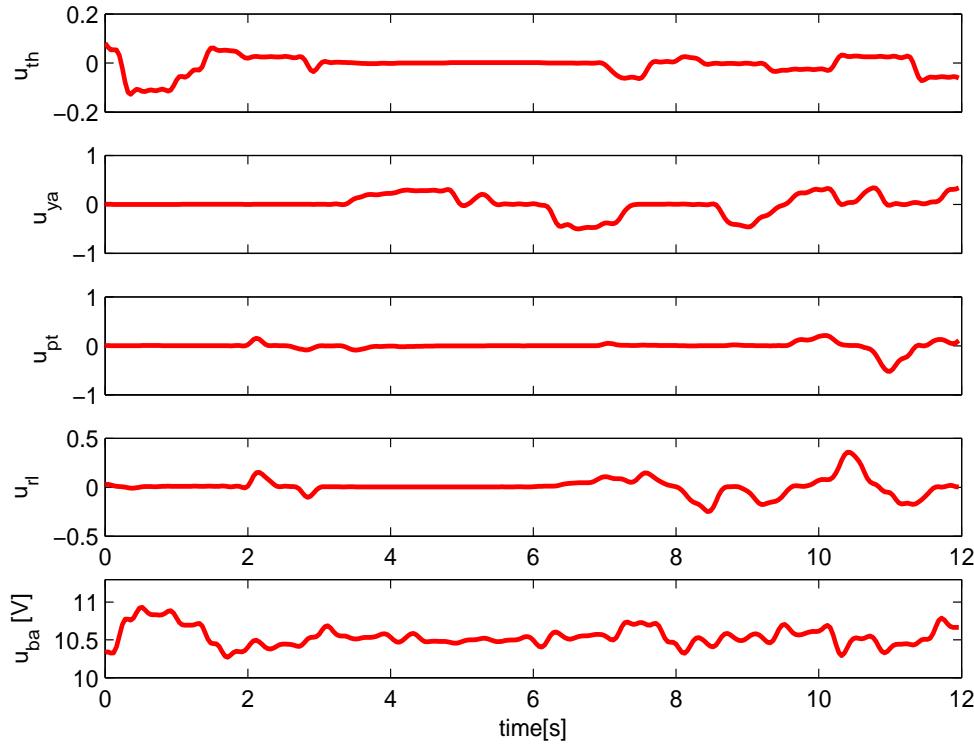


Figure 4.19: Control inputs given to the *X3DVelMLP* model during the time window in Figure 4.18.

manually by interfacing it to the remote used to control the real helicopter and to a basic 3D visualization. The model appeared to respond correctly to all four control commands, and in general the motion appeared smooth. Occasionally we noticed the tendency of the model to slightly pitch or roll, although it was fairly easy to compensate for this. Overall we did not find that the model displayed any behaviour that did not correspond qualitatively to the real platform. However, as for the previous models, the lack of any disturbance was a very noticeable difference between the simulated model and the real helicopter.

Finally in Table 4.18 we show the results of computing the usual validation metrics *RMSEm*, *RMSEsd* and *RAEm*. The *RAEm* metric simply confirms our expectation, with p and q being the most accurate predictions followed by r and the angles θ and ϕ . All the remaining variables have worse accuracy than would be obtained using a *constant model*.

In terms of total *RMSEm* the *X3DAccMLP* model is the worst performing X3D model so far, but not by much. Interestingly, the pattern of *RMSEm* errors across the different variables closely matches what we have already seen for the *X3DVelMLP* model, with the main difference being the vertical velocity w which is predicted noticeably better by

	RMSE	CR
u	0.93 m/s	0.80
v	1.12 m/s	1.05
w	0.68 m/s	0.79
ϕ	0.08 rad	1.03
θ	0.10 rad	0.90
ψ	0.61 rad	1.56
p	0.094 rad/s	0.98
q	0.14 rad/s	0.82
r	0.43 rad/s	1.9
<i>total</i>	1.8	

Table 4.17: Prediction error of the $X3DAccMLP$ model computed over the window of data plotted in Figure 4.19.

	RMSEm	RMSEsd		RAEm
u	1.17	0.33	m/s	1.48
v	1.06	0.34	m/s	1.67
w	0.86	0.75	m/s	1.19
ϕ	0.079	0.027	rad	0.91
θ	0.11	0.037	rad	0.90
ψ	0.39	0.18	rad	1.50
p	0.096	0.036	rad/s	0.42
q	0.17	0.071	rad/s	0.66
r	0.21	0.13	rad/s	0.79
<i>total</i>	1.96	0.70		

Table 4.18: Prediction error of the $X3DAccMLP$ model computed on the validation dataset.

the velocity based model. A comparison with the $X3DAccFP$ shows that the two models have very similar $RMSEm$ errors, but with the $X3DAccMLP$ model always being less accurate; in this case the difference in accuracy for the w velocity is even higher.

Across the dataset the variability in performance for the $X3DAccMLP$ model is in general similar to that obtained for the $X3DAccMLP$ model, but for some specific variables with a higher error (i.e. u , v and w) the $RMSEsd$ is larger.

Toy Car: Prediction Error as Function of Time

We have already reviewed and compared the prediction error for the four different black-box models that we developed for the toy car, but metrics like the $RMSEm$ do not show how the model error changes as time progresses within the data window. Since all of our models need to be integrated to be useful, having an understanding of the relationship

between time and model error is very relevant.

To investigate this, we used the same predicted trajectories that were used in the previous section to compute the $RMSEm$ but instead of averaging together the errors regardless of their time index, we computed a separate average (e^i) for each of the time instants $t = [0,..,T]$. More explicitly:

$$e_t^i = \sqrt{\frac{1}{W} \sum_{w=1}^W (z_t^i - y_t^i)_w^2}, \quad i \in \{x, y, \psi, u, v, r\}, \quad t \in [t_{0w}, .., t_{0w} + T] \quad (4.33)$$

where, following the notation introduced in Section 2.2.5, W is the number of windows in the validation dataset, and z^i and y^i are the measured and predicted values for the variable i . In this way, while averaging out any window-specific dependency, any general trend of error vs. time should emerge. Figure 4.20 shows the results of computing such a metric for each of the dimensions of all the car models we have obtained so far.

As expected, the *carAccNN* model is the worst of all the models; after an initial rapid increase its velocity errors stabilize to a roughly constant value, and as a consequence the error in the integrated variables x , y and ψ increases linearly and very rapidly with time. The remaining three models have a qualitatively similar behaviour although showing a much lower level of error. During the first second the velocity error increases rapidly to reach a plateau value that is approximately constant for the remaining of the test window. In the integrated variables x , y and ψ the error increases linearly as the simulation time progresses but at a much lower rate that is shown by the *carAccNN* model.

For the lateral and rotational velocities the *carVelNN* model has errors very similar to those of the *MLP* models while the errors are higher for the forward velocity and the position y . In contrast, the two *MLP* models are qualitatively and quantitatively very similar.

4.3 Discussion

In this chapter we have taken a very hands-on approach for analysing some standard modelling methodologies commonly used in the fields of system identification, robotics and machine learning.

In the first part of the chapter we confirmed that using specific domain knowledge

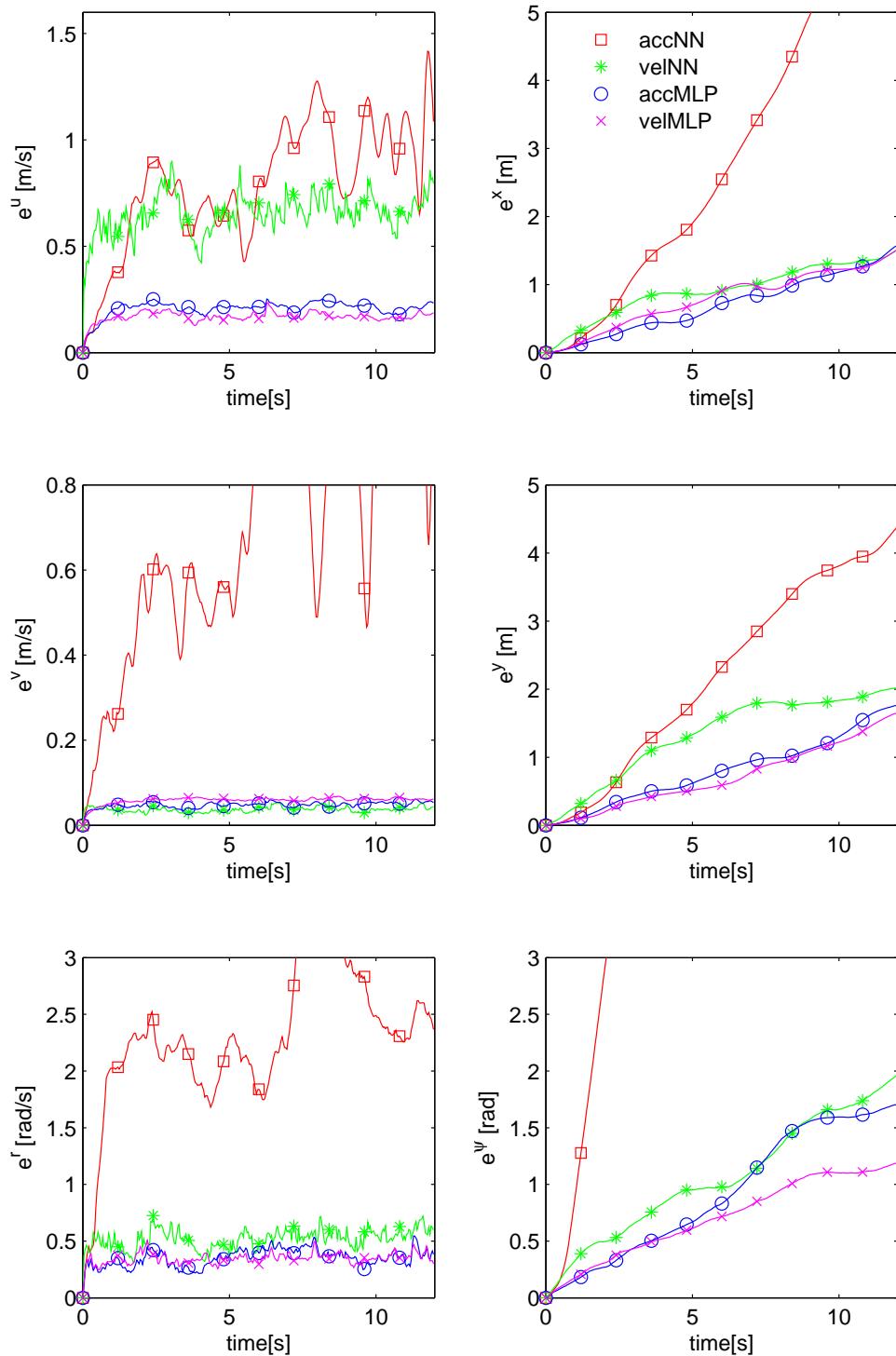


Figure 4.20: Average prediction error versus time. The *carAccMLP* model confirms its very good performance.

about the system under study can get us a long way. In the case of models based on first principles, domain knowledge is involved critically in three different stages of the modelling process:

- deciding which inputs, system states and physical phenomena are relevant to the evolution of each state variable;
- devising the functional relationships that allow the prediction of the system state;
- defining starting values for the parameters to be fitted, and rearranging the model to enable efficient parameter estimation.

For both the ATTAS and the X3D systems, the physics led us to decide which inputs were relevant to the system variable to be modelled, and the available literature gave us grounds for deciding which phenomena could be neglected. The choice of decoupling the dynamics of the ATTAS and neglecting blade flapping and ground effect for the X3D are examples of those informed choices.

Again, physics was essential to devise the dynamic equations of our system, while static modelling and static measurements were needed to initialize (when possible) the system parameters. In the absence of such knowledge, many pragmatic ways had to be used. Often an iterative process of modelling and testing was needed to identify effective modifications to the model. This was the case for example in the rearrangement of the PID equations (see Section A.5), which required us to find a form suitable for our numerical optimization technique. This example shows not only that domain knowledge is essential to devise the model, but also that the specific formulation of the model will have an impact on how easy it is to identify. During our tests we had the opportunity to see at first hand how modelling based on first principles depends heavily on experience³⁴ and requires a considerable amount of time and effort.

When using the *NN* and *MLP* techniques, a learning process based solely on the collected data has to substitute for the platform specific domain knowledge. In our examples, all the control inputs and the full state vector were used as inputs to the function approximator of choice; as a result, it was necessary to learn which inputs are relevant and which have to be neglected. The function approximators we chose are in principle all able

³⁴In fact we found ourselves getting better and better at rearranging and complementing the mathematical expressions for our models.

to learn linear and non-linear relationships of arbitrary complexity, therefore we are not indirectly imposing any particular type of structure on our dynamic models. Finally, since the procedure adopted for training depends on the specific function approximator, any problems of parameter identification that might appear during training are not dependent on the type of platform being modelled.

In practice however, we have seen that the *NN* is simply not powerful enough to accurately learn non-trivial models like that of our toy car, while the *MLP*, although promising, is not as accurate as the technique based on first principles. As well as the issues concerning the power of the function approximators, there are also issues of process noise and noise in the pre-computed accelerations; all these need to be taken into account when evaluating the results obtained in the second part of this chapter.

In the case of the toy car, both the velocity and acceleration based *MLP* models perform reasonably well, confirming that the power of the networks used is sufficient to approximate the non-linear dynamics of the model. In the light of the results obtained, we can certainly say that for the toy car problem the relationship that governs the forward velocity u and rotational velocity r are amenable to standard black-box techniques such as a *MLP*. A different conclusion must be drawn for the lateral velocity v , which is confirmed to be a non-trivial function. Two factors come into play; the first is that, due to the effects of friction, the functional relationship in question could be highly non-linear; the second is that, since the amplitude of the velocity is small, its value might be comparable with the level of noise in the system, as was discussed in Section 3.2. Distinguishing between these two hypotheses is very difficult at present, and we will continue to address this question in the following chapters; however, the fact that four different modelling techniques all failed to perform well on this dimension probably points towards something rather deeper than the limited ability of our function approximators.

The good performance of the *carAccMLP* model supports the case made in Section 2.2.1 that modelling accelerations is well suited to reproducing the inertial properties of our vehicles and therefore can lead to better models. Confirmation that acceleration based models are indeed effective also comes from the *X3DAccFP* and *ATTASAccFP* models, both of which express the state update in terms of accelerations and deliver excellent results. The qualitative “natural” feeling experienced by the pilot when driving the car simulator based on acceleration prediction is also very reassuring, even though it is

formally weak.

For our helicopter problem, we see that the decision not to use platform specific knowledge brings in two well-known problems in machine learning. The first is the problem of dimensionality: since we do not actively decide which inputs or which state variables are relevant to the dynamic relationship that we are trying to learn, our algorithm must be able to deal with the large learning space that this decision produces. The second problem, colorfully named the problem of “poisonous inputs” by Togelius *et al.* [236], refers to the fact that using inputs that are partially correlated with the function to be learned, but which are not essential, can hinder rather than help the learning process.

In Section 3.2 we discussed the problem of the noise introduced by numerical differentiation. Obviously, this could in part explain the quality of our results, and the fact that we have seen that the *X3DAccMLP* model is not better overall than the *X3DVelMLP* supports this conclusion. In the EE settings integration is not used during training, and any effects of noise propagation are simply ignored, making those techniques very vulnerable to such noise. The fact that the *X3DVelMLP* is not as good as the *X3DAccFP* model suggests that avoiding accelerations and using velocity alone is not the answer to our problems. However, since the *X3DAccFP* model, although it predicts accelerations, is trained on velocity data and delivers good performance, we can definitely exclude the possibility that there are serious noise problems with the velocity data. Nonetheless we have to recall that our attempts to jointly identify all the parameters of the *X3DAccFP* model led us nowhere, since the prediction error, although small, was sufficient to make the prediction uncorrelated with the training data, making the error effectively useless. At this point any further conclusions seem premature, but we can clearly see the need for the integration over time of the model predictions in order to evaluate the extent of error propagation.

In the case of the *MLP* networks another open question is highlighted by the counter-intuitive experimental evidence that training our networks with randomized rather than ordered data from our dataset reduces the performance of our models. During the data collection we attempted to make sure that we sampled the flight envelope of our machine in a representative way; however, this is something very difficult to guarantee in practice. The evidence suggests that not all the data in the training set are equally informative, and that although an effort was made during data collection to homogeneously sample

the system envelope, some parts of the envelope might be over-represented. Training our models on the whole dataset leads therefore to a bias towards the over-represented parts of the dynamic envelope. This obviously opens the issue of how to select only the most informative part of the training data in order to improve the modelling; this is one of the active topics of Chapter 5.

4.4 Summary

In the first part of the chapter we have shown how knowledge about the platform can be used to provide a solution to the problem of modelling but we have also made clear in how many ways such knowledge informs the modelling process; making very relevant the research questions that we are tackling in this thesis.

In the second part of the chapter we tested some well known black-box techniques, which showed their limitations when facing challenging platforms like the X3D, for which number of dimensions, coupling and the noise potentially present in the data have a considerable impact on the learning problem.

The work in the forthcoming chapters will leverage on this results and tackle this open issues.

Chapter 5

Coevolutionary Modelling

After exploring and assessing the performance of conventional modelling methods, in this chapter we start answering one of the key questions behind this thesis: how far can we get in automatically building a model without using any platform specific knowledge?

Our answer will come in the form of a novel and efficient algorithm based on coevolution (Section 5.3). In keeping with the experimental emphasis of the thesis, we will put our algorithm to the test on four very challenging vehicles (Sections 5.5-5.7) and assess the degree to which the algorithm is able to produce accurate models.

5.1 Fundamental Considerations

The experiments carried out in the previous chapter have given us a considerable insight into what is actually effective on our real world platforms.

The MLP models included some desirable traits: the model structure used was generic, and the training was solely based on the collected data. However, the problem of input selection was deliberately avoided by presenting the models with all the available inputs, and the choice of network size (i.e. number of hidden neurons) was informed by our understanding of the complexity associated with each platform. For instance, we used a larger number of hidden neurons for the X3D models than for the car models, since we knew the former to be a more complex vehicle with a larger number of inputs. Since ideally even decisions at this level should not be dealt by the designer, we seek a methodology that can deal with all these decisions simultaneously.

The analysis in the previous chapter showed that there is good reason to believe that our

training methodologies were not making the most of the experimental data. In particular, treating all points in the dataset as being equally important appeared not to be the best idea since the learning might end up being biased toward those parts of the dynamic envelope that are more heavily represented in the dataset. With our platforms, collecting more data by actively generating test inputs is simply impractical, as already discussed in Section 2.2.3. We therefore need to look at a method that makes an active selection from the available (pre-collected) data gathered when a pilot was safely controlling the vehicle.

The training of MLP networks based on the EE setup produced adequate results for the toy car, but revealed its shortcomings when a more challenging problem like the X3D was tackled. In contrast, the OE methodology proved very effective in terms of training, without requiring any use of platform specific knowledge. The relationships used to integrate the state forward in time as required by the OE method (e.g. equations 3.5-3.7) are not platform dependent since they assume nothing more than 6DoF rigid body physics. The specific domain knowledge used in Sections 4.1.1 and 4.1.2 was because first principles models were used; it was not used for the OE training itself.

How can we carry forward our understanding into the idea of modelling without using any platform specific knowledge?

5.2 The Idea

In line with the considerations of the previous section, the constraints imposed by the fact that we use non-recoverable vehicles, and the spirit of this thesis we can identify three crucial points that our modelling method needs to address:

- *platform agnostic modelling*: the modelling methodology should be able to automatically select the structure of the model, its inputs, and the input delays using only the information contained in the training data;
- *physical consistency*: the quantities predicted should be meaningful accelerations expressed in body coordinates¹, and ideally it should be possible to inspect the models to identify any meaningful terms;

¹This requirement ensures that the model produces accelerations and velocities that are expressed in a sound and standard form. A model of this kind is easier to inspect and potentially more useful (e.g. it could be used in conjunction with conventional design and control techniques).

- *intelligent data selection*: the available data should be selected in order to make the most of the available information and avoid the inevitable “biases”² that might be present in a specific dataset, while also limiting the computational requirements.

To achieve our aim we will leverage and complement some of the approaches and concepts present in the literature. Since we are aiming at a proof of concept for our framework and we expect future improvement to be possible both in terms of techniques and implementation, we will emphasize the rationale behind our design and implementation choices, instead of focussing on their possible optimality.

The requirement of physical consistency leads directly to the choice of producing a model in acceleration space and propagating the state forward in time by means of integration as we did with all the models in acceleration space considered in Chapter 2.

The first design decision concerns the type of training methodology to use. In the light of the considerations made in the previous section and of the simplifying assumption of negligible process noise that we will continue making in this chapter, the OE methodology appears to be the most appropriate choice for our settings.

Choosing a representation for the model is our logical next step. In practice a representation often goes hand in hand with a training algorithm or a family of training algorithms, therefore in making our choice we need to take into account the training requirements imposed by the OE methodology. In Section 2.2.4 we saw that the OE methodology does not assume the output of the model to be measurable, and, as a consequence, none of the training methodologies (e.g. backpropagation) that needs the output of the model (i.e. accelerations) to perform the training can be used. This is a crucial point since the strength of the OE methodology depends on the fact that the error is evaluated after the model predictions have been integrated (i.e. the error in the state).

Many of the training algorithms used in supervised learning assume that the target outputs are directly available, and therefore are not suitable in our settings however, there are other techniques, such as training based on Maximum Likelihood or Bayesian methods, that could be adapted to the situation. But as we saw in Chapter 2, the integration equations relating the predicted accelerations to the system’s state are complex and non-linear, and as a result, deriving training algorithms based on those two methods is very

²In a dataset certain parts of the dynamic envelope might be over-represented; the dataset is therefore “biased” towards that part of the system’s envelope.

challenging.

A training approach that does not suffer from those restrictions is the use of evolutionary algorithms. Since the optimization approach used is completely numerical, these algorithms can be applied regardless of the problem and its formulation, as long as a meaningful fitness function able to measure progress can be devised. As a result of their flexibility, evolutionary algorithms have found widespread use, especially on problems not amenable to more standard optimization methods. For our problem we already have a well defined fitness function provided by the error between the predicted and recorded state values, and so using an evolutionary approach is both appropriate and straightforward.

Other approaches based on global search techniques such as particle swarm optimization, differential evolution, simulated annealing, ant colony optimization, and hybrids created by mixing these techniques with evolutionary methods, have been applied to the problem of modelling in several cases (e.g. [9, 254] and [257]) and could in principle be adopted in our settings. The available literature suggests that such techniques have in general comparable levels of performance, and which one is the best is very problem dependent.

In this work we restrict our treatment to evolutionary approaches since they are among the least specific, and also because, they have been very successfully extended to the idea of coevolution, as we will see later in this section. Our choice remains a pragmatic one, and the framework that we will present in the sections that follow could possibly be adapted to other types of global search techniques (e.g. differential evolution or particle swarm optimization). It would certainly be interesting to see if any of those approaches would yield similar results to ours.

Evolutionary algorithms can provide an effective way of optimizing a model, but they do not prescribe the form that the model should have (e.g. neural network, analytical equation etc.). In fact, much of their strength lies in this indifference to form. Given a measure of progress, they can be applied to a variety of representations as long as the genetic operators for the chosen representation are defined³. Genetic operators are usually defined as heuristics, especially for the most common representations, but some very interesting research has been carried out on principled ways of designing operators (e.g. [163]). From the many possibilities we need to select a representation that does not require platform

³Genetic operations like for example mutation and crossover act directly on the model representation and therefore are representation specific.

knowledge for the structure nor for the input selection, and that is as interpretable as possible.

Genetic algorithms have been used extensively and successfully to identify the parameters of first principle models (e.g. [234, 256]). While those examples indicate the abilities of genetic algorithms as optimizers, this type of approach is unsuitable for our needs since it is based on the availability of platform knowledge.

Evolutionary neural networks⁴, the practice of combining neural networks and evolutionary algorithms, is a well established branch of the computational intelligence field. A great deal of research has been conducted investigating the evolution of connection weights, architectures, learning rules, and input features. Presenting this topic in depth is outside the scope of this thesis, but a good introduction to the field, if a little dated, can be found in [255]. A significant drawback that makes this representation unsuitable to our needs is the fact that the models obtained are not transparent, and attributing any meaning to the network weights is very hard.

Evolutionary algorithms have also been applied to fuzzy logic modelling, both for extracting fuzzy rules, and for optimizing membership functions (see [51] for a list of references). But this type of representation base on logic rules does not appear very suitable for representing the dynamic equations of a 6 DoF rigid body.

An important area for evolutionary computation is symbolic regression, the process of formulating an analytical equation from a given set of data points. Symbolic expressions can accommodate a large variety of different functions, and in principle allow the representation of arbitrarily complex linear or nonlinear relationships. Since the model is produced in the form of an analytical expression, the unique benefit of symbolic regression is that the models are potentially interpretable. This allows some level of insight into the phenomena underlying the system being modelled. Inspecting the resulting models may also allow the identification of variables or variable combinations that play a key role. Although it cannot be guaranteed that the model produced will be fully understandable, it is often the case, as we will show in our examples (Section 5.5-5.7), that the expert eye is able to make sense of the main terms in the model. Interpretability is one of the main requirements that we set for our methodology; given this unique benefit ,evolutionary symbolic regression is an ideal candidate for our modelling approach.

⁴Sometimes also termed Neuroevolution.

The best known way of representing symbolic expressions are expression trees ([125]), but graphs ([157]) and lists of instructions (linear genetic programming [36]) have also been proposed⁵. Linear genetic programming offers a way of directly evolving the binary bit patterns actually used by the computer; in this way the use of a compiler can be avoided, saving on computation. In our settings the evaluation of the fitness of an individual requires far more computation than the genetic operations carried out during evolution, making any computational saving involving them of minor importance. Therefore using linear genetic programming does not provide us with any direct advantage. Tree and graph encodings are very similar representations, with the latter being more flexible since multiple edges are allowed between nodes. Comparison of the two representations when regressing functions of different complexity ([203]) showed that the graph encoding solutions are more efficient in terms of computation and of graph size, but tree encoding offers a higher convergence rate. Ultimately, the latter is our main concern, and makes the tree encoding our encoding of choice.

Some authors ([177] and [249]) have proposed the use of a grammar in addition to the representation to handle the problem of closure⁶ or to explicitly bias the search. The model equations of interest to us require only real valued numbers, and do not pose the problem of closure, and biasing the search (i.e. on the basis of platform knowledge) conflicts with the spirit of this work. The use of a grammar will therefore not be considered any further.

From the evidence provided by the available literature, a direct tree representation seems a suitable choice for encoding our symbolic expressions, but we do not exclude the possibility that more effective representations (in terms of search results or of computational cost) may exist.

The combination of evolutionary methods with a tree based representation of symbolic expressions brings us to what is perhaps the best known application of evolutionary computation to symbolic regression; symbolic regression via genetic programming ([124])⁷. In

⁵Graph and trees can be evolved directly, or encoded indirectly as binary ([177]) or real valued ([16]) strings. The genetic operators depend on the encoding used.

⁶Closure, as defined by Koza [125] indicates that the functions available for evolution are well defined for any combination of arguments. Such a property is implicitly guaranteed if only one data type is used (e.g. all functions accept and return real valued numbers); however, for multiple data types, closure needs to be guaranteed explicitly.

⁷Technically our approach does not correspond to conventional genetic programming ([124]), since the evolving individuals are not part of a multi-individual population, and we do not make use of a recombination operator (the reasons for this choices are made clear in the sections that follow). The latter, in particular, makes our approach also similar to evolutionary programming ([72]). Recent years have seen cross fertilization between different variants of evolutionary algorithms, to the point where distinctions between classes have become fuzzy at best. In the light of this we take the liberty to use the name “genetic

this technique an initial set of random expressions is generated and tested to determine their ability to model the training data (i.e. their fitness). Generation by generation, those programs that perform well are chosen to breed by crossover and/or mutation⁸ and produce new and hopefully better expression trees. The procedure is stopped when the desired accuracy or the target amount of computation is reached.

Last but certainly not least of the requirements for our algorithm is that of making an intelligent selection from the available data. In Section 2.2.3 we discussed at length many of the possible approaches, and identified their requirements and limitations. The coevolutionary approach proposed by Schmidt and Lipson ([204]) appears to be the most suitable for our settings since it does not require direct interaction with the vehicle (i.e. the precollected data can be used), it can accommodate any model representation (e.g. symbolic expressions) and it does not require platform specific knowledge. In order to exploit all the merits of using segments of data for the computation of the model’s accuracy, discussed in Section 2.2.4, the use of simple data windows from the training dataset as test individuals appears both appropriate and efficient.

The idea behind the use of competing populations in coevolution is to encourage an evolutionary “arms race”, so that progress in one population forces the other populations to improve, and vice-versa. During evolution the models are rewarded for how well they can predict the data present in the tests, while the fitness of the tests depends on how challenging they are for the models, that is, the error rates they elicit from the models. This active selection of the training data has the aim of inducing a coevolutionary dynamic in the hope that it will deliver faster and better convergence to good models. In [204] and [205] Schmidt and his co-workers investigated the use of coevolution in the symbolic regression setting, and showed how a data selection process carried out by the coevolution produced solutions that performed better than those delivered by standard symbolic regression, solving problems not amenable to conventional genetic programming.

In standard symbolic regression the fitness of the expression being evolved is generally computed over the whole training dataset ([125, 11]). This becomes a significant computational drawback in applications like ours, where because of the complexity of the relationships to be learned, and the presence of noise in the data, large training sets must

programming”.

⁸Crossover indicates the creation of a new expression by combining randomly chosen parts from two selected parent expressions. Mutation indicates the creation of a new expression by randomly altering a randomly chosen part of the parent expression.

be used. With coevolution, instead of using the whole dataset, a limited number of data windows tailored to the current models can be used to evaluate the models' fitness. A further advantage of coevolution is therefore the potential for obtaining a computational saving.

Thanks to its ability to improve search performance and reduce computation, coevolution is well suited to symbolic regression, and features in the literature [60, 180, 204].

Competitive Coevolution

In the type of competitive coevolution introduced in the previous section, the fitness of an individual is made dependent on other individuals in a different population, with the aim of producing global progress. In this work we use the definition of progress proposed by Miconi in [153]; given two individuals A and B he defines:

- *local progress*: an individual A is better than B, when comparing performances against the current opponents;
- *historical progress*: A is better than B, when comparing performances against all previously encountered opponents - that is, current opponents and their ancestors;
- *global progress*: A is better than B when comparing their performances against all possible opponents - that is, the entire search space.

From earlier work in competitive coevolution [46, 70, 246], it became apparent that the interplay between competing populations has a very complex dynamics; as a result, coevolution does not always produce global progress, the desired objective of the optimization. Three problems are well known to affect competitive coevolution: the Red Queen effect, disengagement, and cycling. It is important to discuss them since we will need to take them into account when implementing the algorithm (Section 5.3).

The Red Queen Effect [46] identifies the situation in which the coevolution is effectively producing global improvement, but due to reciprocal adaptation (overspecialization) of the competing populations, this improvement is not reflected in the metric of local progress. For example, in our situation, the choice of tests may indicate that the fitness metric of a model is decreasing, whereas the model may actually be getting better in terms of global progress.

Disengagement takes place when one or more members of one population consistently “beat” all the individuals of the competing population. This leads to the loss of a selective gradient which prevents the improvement of the individuals in the worse performing population. In the coevolutionary modelling example, the evolutionary algorithm that is producing the tests is constantly searching for new challenging tests. It is therefore possible that tests too difficult for the current population of models might be produced, leading to a disengagement between the two populations. Bongard and Lipson in [29] suggest a method for alleviating disengagement by reserving the tests that are too challenging for future iterations.

The problem of cycling arises due to the fact that the fitness of one population is relative to the population of opponents, and so there is no guarantee of transitive dominance between individuals in different generations. For the specific case of coevolving models and tests, the fact that a model performs well against the current set of tests (local progress) does not guarantee ([153]) that it will perform well against all the tests coevolved up to the current generation (historical progress). A situation of dynamic equilibrium can therefore arise, during which the algorithm repeatedly cycles through the same tests and models without making any real progress towards the objective of the optimization.

In [196], Rosing and Blew proposed the “hall of fame” method to avoid the problem of cycling and ensure historical progress. In their method, the best individuals of each generation are retained and used to test the opposing population; with this method, new individuals cannot overspecialize, and therefore the phenomenon of cycling is avoided. The “hall of fame” idea has been widely employed in coevolution, and coevolutionary modelling is no exception (see [27, 31]).

In [153] Miconi also points out that often in the coevolutionary literature, the idea of historical progress is implicitly assumed to imply global progress, but in reality this is not guaranteed. In analyzing the convergence of our algorithm we will therefore need to measure its performance against an unseen validation dataset (to measure global progress) to verify that historical progress does indeed lead to global progress.

5.3 The Algorithm

In this section we present in detail the coevolutionary algorithm used in all the coevolutionary modelling experiments presented in this work; we first define the terms epoch, run and generation that will be used extensively throughout the remainder of this chapter:

- *generation*: a set of the three operations of mutation, fitness evaluation, and selection that an EA carries out cyclically when evolving a population of individuals;
- *epoch*: a series of generations of model evolution followed by a series of generations of test evolution;
- *run*: a consecutive series of epochs from the initialization to the termination of the algorithm.

5.3.1 Design

Before looking in detail at the mechanics of the coevolutionary algorithm, we need to design the building blocks of the coevolutionary system. These are the two evolutionary algorithms for evolving models and tests, the techniques that avoid the problems commonly encountered in competitive coevolution (Section 5.2) and a practical method for applying the idea of coevolution to the types of multidimensional model we are interested in.

In an evolutionary algorithm the size of the population of individuals is an important control parameter that has an impact on the algorithm's performance. In [186] Poli *et al.* wrote on the problem of how to choose the size of the population of individuals:

“As a rule one prefers to have the largest population size that your system can handle gracefully.”

In our setting, the most computationally expensive step in the evolution of models and tests is the evaluation of the fitness. After some preliminary investigation of fitness evaluation times, we estimated that for a single core implementation we could comfortably handle populations of at most a few hundred individuals. This can be considered small in respect to the sizes commonly used in the literature, where the number of individuals is often of the order of hundreds of thousands ([186]). This limitation will have a strong impact on the design of our two evolutionary algorithms.

Fundamental to any population-based evolutionary algorithm is the need to maintain diversity between individuals, so that the operations of crossover and mutation can maintain their effectiveness. When a small population is used, the probability that a small number of individuals will take over the whole population in response to the selection pressure is higher than it would be for a large population. Loss of population variety is a well known problem in evolutionary algorithms ([186]); the usual strategies used to counteract this problem are based on modifying the selection mechanism (e.g. tournament selection [64]), modifying the genetic operators (e.g. size fair crossover [130]), or splitting the population into semi-isolated subpopulations (e.g. niching [146]).

Given our limitations in terms of computation, the approach proposed in [31] of using a set of (1+1)EAs to operate the search, and embedding some of the standard strategies to preserve diversity, is particularly suitable⁹. In the evolutionary computation field the denomination $(\mu+\lambda)$ EA identifies an algorithm in which λ offspring are produced and the best μ of them are selected from the union of parents and offspring to form the new population. More specifically, in the case of a set of (1+1)EAs used to evolve the models, an offspring is generated for every model at each generation by copying the parent model and mutating it. The offspring is tested, and if its fitness is higher than that of its parent it is retained, otherwise it is discarded. In the method proposed in [31] no form of recombination is performed between the members of the population, and so the evolution of the different models proceeds in parallel. Using this simple mechanism, each element of the population is in effect “protected”, and thus has the time to evolve and adapt to the tests. Specific traits in a model are therefore prevented from spreading to the whole population and producing a loss of diversity. One potential drawback of this method, is that the spread of beneficial traits within epochs is also prevented; later in this section we will see how this problem is addressed. In some ways, this can be seen as an extreme example of niching ([146]), where the niche consists of only one individual. During evolution, we reinitialize the worst individuals with a small probability (0.1) to promote the exploration of different areas of the search space.

The absence of recombination will impose strong requirements on the design of the mutation operators (see Section 5.3.3) since mutation will need to promote both exploration and exploitation of the search space.

⁹A (1+1)EA is sometimes called a stochastic hill climber.

Limitations in terms of computation also apply to the evolution of the tests, and so there we can again employ the same (1+1)EA setup. For evolving models we took note of the folk wisdom of the genetic programming community, and made our population size as large as possible (without being impractical), at 100 individuals. In the case of the EAs evolving the tests we still need to limit the computation but in this case we also know the search space to be much smaller. A much smaller population of 6 individuals appeared to be a suitable value.

During model evolution, as the number of generations increases, the mutation if left unchecked will lead the average size of the trees in the population to increase. This is a well known phenomenon in genetic programming known as bloat. In the context of evolving symbolic expressions, bloat also implies more complex expressions that are probably more difficult to interpret. Controlling bloat is therefore intimately related to the objective that we set for our algorithm of obtaining interpretable models. In the literature three main classes of techniques have been proposed to simplify genetic programming expressions: parsimony pressure, operator modification and code modification. The first entails defining a metric for parsimony and using the metric along with the normal fitness to guide the selection process. However the true effectiveness of parsimony pressure is still a highly debated topic ([18]). Operator modification focuses on reducing the destructiveness of the genetic operators. For example, in [214] Saule *et al.* suggested the use of a crossover that accepts an offspring only if it is better performing than the parents. The idea is that, if code growth occurs to protect expression trees against the potentially destructive effects of crossover ([150]), a non destructive crossover should lead to smaller expressions. This is also the idea used in our parallel (1+1)EA, and so we are implicitly already applying such a technique. The third type, code modification, involves modifying the expression structure during or after evolution using various methodologies. In the case of symbolic regression, both algebraic simplification and numerical simplification can be applied. Since in general there are an exponentially large number of equivalent expressions, automatic simplification is in practice not a trivial matter ([186]), and for this reason we prefer a numerical one. The method proposed in ([31]) works in two distinct phases: during fitness evaluation, it keeps track of the maximum and minimum values (respectively c_m and c_M) to which the subtree originating at each of the nodes of the expression evaluates, and then, when it is time to apply the simplification, each node in the tree is selected with

probability $0.5/N_{nodes}$ (where N_{nodes} is the total number of nodes in the tree) and the entire subtree of which it is the root is replaced with a constant, the value of the constant being drawn from the uniform distribution $\mathcal{U}[c_m, c_M]$. The idea behind this simplification is clear: if a subtree makes little contribution to the expression output, replacing it with a constant will not affect the predictive ability of the model very much, and therefore on average the population will end up containing smaller and simpler trees. In general, simplification can cause drastic changes in the expression tree and can therefore produce strong effects on the evolutionary progress. Empirical investigations ([252]) have shown that applying simplification at every generation tends to have disruptive effects, and so in our implementation we apply simplification only at the end of each coevolutionary epoch.

Since the constants used for simplification are chosen randomly, is likely that even in cases in which the simplification is meaningful (i.e. in which the pruned subtree was superfluous) the simplified subtree scores poorly in terms of fitness. Now, in our models, even a small constant bias can have a large impact on the final error because of the integration. Therefore, in order to increase the effectiveness of the simplification operator, we apply a further procedure designed to re-converge the constant factors in the expression tree. A local search technique could be used for this (e.g. [213]) but to simplify the implementation¹⁰ we use a (1+1)EA to evolve only the constant nodes present in the tree (see 5.3.3 for details). The parameters of the model are evolved for 100 generations. If the resulting tree scores better than its parent, it will replace it, otherwise it will be discarded.

In addition to the simplification of the tree expressions, we also limit the maximum depth of the expressions; we chose a maximum depth of seven to allow for expressions more complex than those we would expect for our systems.

We now need to structure the interplay between the two (1+1)EAs carrying out the model and test evolution respectively. The chief requirements for this step of the design are avoiding the shortcomings typical of coevolutionary methods (Section 5.2) and as usual, limiting the computational complexity.

Avoiding cycling is a fundamental issue, and the “hall of fame” method proposed in [196] appears appropriate because it is simple and because, it allows us to limit the computation by regulating the size of the hall of fame. The idea of the hall of fame simply requires the best test evolved in each generation to be stored so that the complete set (the test history)

¹⁰It is often the case that some of the evolving expressions have a tendency to diverge, a problem that complicates the use of standard optimization methods.

can be used for testing the models. This technique was shown to be effective in [31] but obviously has the drawback that the size of the set of tests keeps increasing. In our case this is problematic because it means that evaluating the fitness of models will become more and more expensive as the run proceeds.

To obviate such a problem, we propose an approach working at different time resolutions. At each generation only the m most recently evolved tests (the test bank) are used to compute the models' fitness, while the whole test history is used at each epoch to select the best performing models from those evolved and the current set of best models. In this way the history of tests still grows, but the increase in computational requirements is less dramatic since the history of tests is evaluated only at every epoch.

The selection procedure applied after each epoch has two functions, the first being that of discarding any model that scores poorly against old tests, and the second that operating a selection of the model at the population level. The latter allows good solutions to spread in the population, but at a much lower rate than would be the case for the within-epoch spreading that is disallowed by our choice of parallel (1+1)EAs.

The selection at the level of epochs is achieved by using a classic elitist scheme. The whole population (i.e. current models plus models from the last epoch of model evolution) is ranked by fitness and the b best performing individuals are taken as the new current models. Those are used to seed the next epoch of model evolution as well as to compute the fitness of the tests. The value of b determines how many of the population are initialized with current models, and how many with their mutated copies. After some preliminary testing we settled for a value for b of 80.

This strategy of separating the individual-based and population-based selection is a compromise, but it does allow us to control the deleterious effects of loss of diversity at the cost of slowing down some beneficial effects.

Since only the best test from every epoch of optimization is added to the test bank, m defines the maximum rate at which a complete replacement of the population of tests can take place. We chose $m = 6$ as a compromise between computation demands and a long (and more informative) history. We will comment on the effectiveness of this dual approach when analyzing its convergence ability for our different platforms (Sections 5.5.1, 5.6.1, 5.7.1 and 5.8.1).

An additional problem that could be induced by coevolution is overspecialization, which

reduces the disagreement between models. In our scheme this problem is counteracted in two ways. First, the tests are evolved for producing maximum disagreement among models (see Section 5.3.4), and second when a new test is added to the test bank we do not allow it to overlap with any of the old tests. If the tests were allowed to overlap, they could all converge on a very limited part of the dataset, effectively overspecializing. In our implementation, if a test overlaps a test that is already in the bank, it will replace that test, the idea being that is better to prefer a test that has been optimized on the current models. If the test does not overlap any of the pre-existing tests, it will simply replace the oldest test in the bank, again following the same rationale of preferring newer tests.

During our initial test runs, we did not see signs of disengagement between the two evolving population (the last of the pathologies presented in Section 5.2), and so no measures to counteract it were introduced. We comment further on this in Section 5.4 when analyzing a typical coevolution run.

The key component that bridges model generation and the evaluation of model performance is the integration forward in time of the analytical expression in order to produce the full state of the vehicle, which is needed to evaluate the model fitness in a OE fashion. The integration exploits the knowledge that we are modelling a rigid body, focussing attention on the crucial problem of error propagation, and forcing the model expressions generated to be physically meaningful. A model of one of our vehicles requires several analytical equations that will predict the current acceleration given the current state and possibly delayed control. One equation is needed for each DoF of the model up to a maximum of six in the case of our quadrotors.

Although it would be preferable to attempt the identification of all the model equations concurrently, this is not a sound idea. For a fixed expression depth, the search space increases exponentially in the number of variables ([31]) making for a very difficult search problem. Partitioning the problem and performing the optimization of the equations one at the time gives a more manageable search space that scales polynomially with the number of variables ([31]). A second and more pragmatic reason for regressing each equation in the model independently is the fact that, if the total error is used models that are partially correct might be neglected during the search. As we saw in several experiments in the previous chapter, any error in one of the dimensions will inevitably be propagated into the other dimensions due to dynamic coupling, leading to a high total error. During evolution

such a model is unlikely to be selected, and so some correct equations may be lost.

It is therefore preferable to identify the dynamics of each of the model's dimensions separately. For each equation we will have a specific population of models and of tests that will be coevolved¹¹. In order to do so, and at the same time to account for the correct propagation of the model error, we need to integrate the model forward in time as usual. However a problem then arises: when evolving a single equation, we have no equations for the remaining dimensions of the model. A pragmatic solution already used in Section 4.1.2 is to use the precomputed values available in the dataset for the remaining accelerations. Those estimated accelerations in fact represent what the “correct” model would predict, plus a noise component (see Section 3.2) which will be inevitably present since the accelerations are calculated from real data.

With this modification, we can estimate a dynamic equation for each dimension of the system using our coevolutionary approach, and the equations obtained can then simply be combined to deliver the full dynamic model.

5.3.2 Coevolutionary Loop

We now explain how the parts designed in the previous section compose the main loop of our algorithm. We will focus only on the coevolutionary mechanism; the specific details of the two algorithms used to evolve the models and tests are left to Sections 5.3.3 and 5.3.4 respectively.

The architecture of the algorithm is shown in Figure 5.1 in terms of functional blocks and the flow between them. In order to aid understanding we describe the algorithm in the order in which it is executed, such order is marked in each functional block with a number that we will report in parenthesis during our description.

The coevolutionary run starts in the first epoch with a set of randomly generated tests and models (1); the random tests also seed the tests bank (2) and the history of tests (19).

For each of the randomly created models, a (1+1)EA is started, and each model is independently evolved (2,3,4,5); the fitness of each model is computed using the same set of tests from the test bank (2).

After each model has evolved for the maximum number of generations (600 in our experiments), the model undergoes the process of simplification with the aim of reducing

¹¹This is different from what was proposed in [31] where the model's equations are evolved independently but the whole model is used to coevolve the tests.

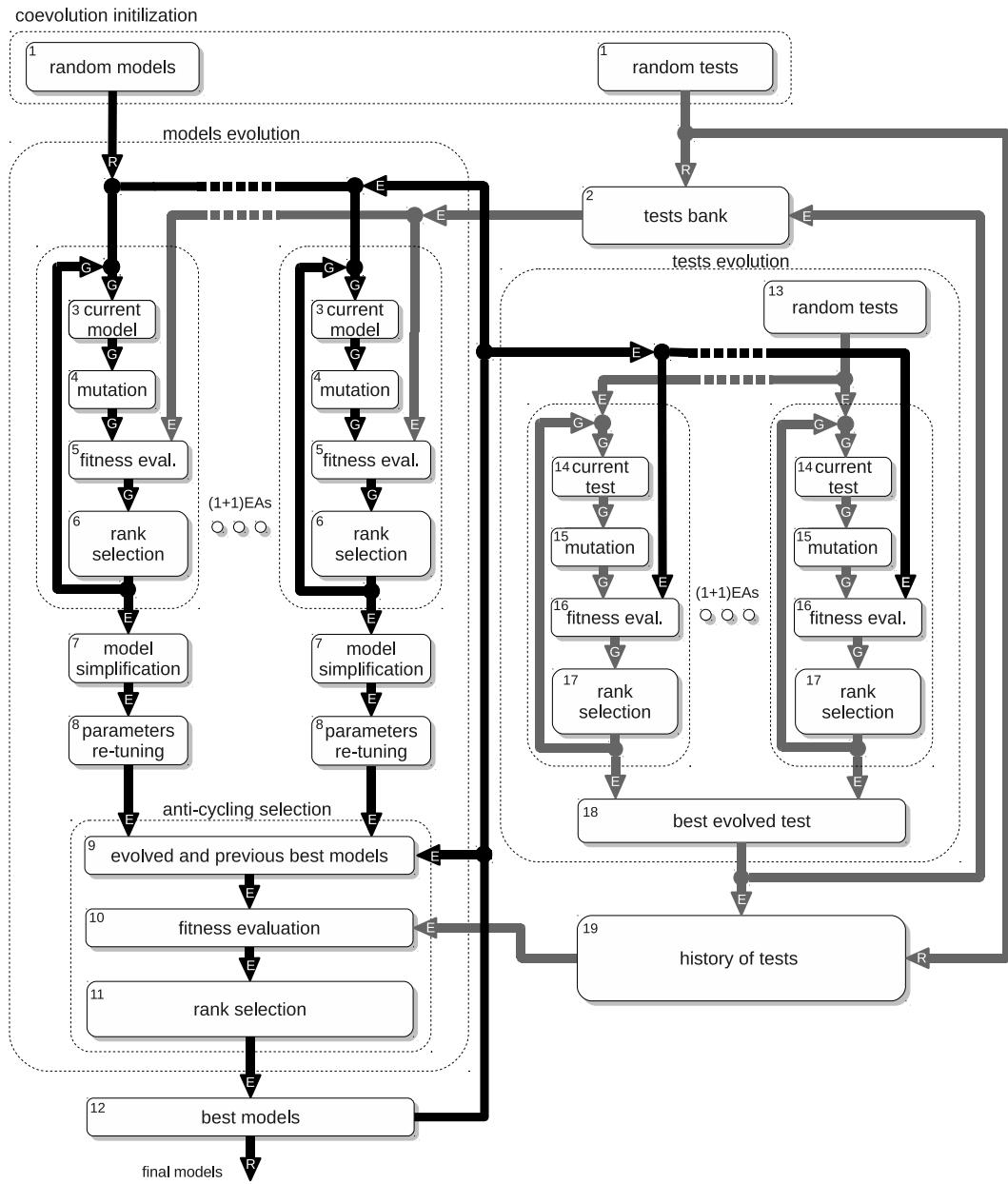


Figure 5.1: Architecture of the coevolutionary algorithm. The letter “R”, “E” and “G” indicate operations executed at every run, epoch and generation respectively. Black arrows indicate the flow of models, while grey arrows are used for tests. The number within each block denotes the order in which the operations are executed.

the model complexity (6). This is followed by a short evolution run (200 generations), with the aim of “re-converging” the parameters of the model possibly disrupted by the simplification (7). After this operation, the better performing of the simplified and the unsimplified models becomes one of the candidate models for the next stage. The numbers of generations used for model evolution and for re-converging the models were chosen based on the results of several exploratory runs of the algorithm.

These models are added to the best models produced in the previous epoch (9), and the union of the two is then evaluated using the tests (i.e. data windows) belonging to the history (10). The models are ranked by fitness and the best 80 are selected (11) as the best models produced by the current epoch (12). Since no previously generated best models are present during the first epoch, and the history of tests contains only the random tests used for the first evolution, this phase corresponds to selecting the best 80 from the 100 models produced by the parallel (1+1)EAs.

The best models obtained are then passed to the second evolutionary algorithm that is responsible for evolving the tests; this will make use of the models to compute the tests’ fitness¹².

In contrast to what happens when evolving the models, the tests are randomly generated (13) at the beginning of every epoch, not just in the first; this ensures diversity among the tests, and makes overspecialization less likely.

Each of the tests thus generated is passed to one of the parallel (1+1)EAs and is evolved (14,15,16,17) for 60 generations. As we will describe in detail later, the fitness of each model is computed by evaluating the variance of the models’ predictions. From the tests, the one that induces the most variance is selected (18) and added to the test bank. This test is also added to the test history.

This concludes the description of the first coevolution epoch; in all subsequent epochs (up to 200 in our experiments), the same steps are repeated, the only difference being that the population of models is seeded by the best models of the previous epoch rather than being randomly generated¹³.

The best model in the set of best models after 200 epochs is the equation that we deem the solution for the dimension that is being evolved; we then repeat the process for each

¹²The evolution of tests happens therefore sequentially to the evolution of models.

¹³Since the number of best models is equal to 6 and the number of parallel (1+1)EAs is 14, the EAs are seeded with each of the best models plus another 8 models chosen at random with uniform probability from the best models.

of the remaining DoFs, and the resulting set of equations defines the full model. We will see later that in all our experiments the algorithm will converge within much less than 200 epochs; the use of such a long evolution horizon enables us to perform a better analysis of the algorithm's properties.

In the next two sections we report the details of our implementation that were omitted here for reasons of clarity.

5.3.3 Evolution of the Models

Representation

Each dynamic equation in our models is represented as an expression tree. The inner nodes in the tree belong to a basic algebraic function set \mathcal{F}

$$\mathcal{F} = \{+, -, *, \%\},$$

where $\%$ indicates protected division¹⁴, while the terminal nodes (leaves) can be any of the state variables, any of the control inputs, or constant terms. In the case of one of our quadrotors for example, the set \mathcal{T} of terminal nodes takes the form:

$$\begin{aligned} \mathcal{T} = & \{u, v, w, \phi, \theta, \psi, p, q, r, u_{th-\delta_{th}}, u_{ya-\delta_{ya}}, u_{rl-\delta_{rl}}, u_{pt-\delta_{pt}}, c\} \\ & \delta_{th}, \delta_{ya}, \delta_{pt}, \delta_{rl} \in [0, \Delta_{max}], \end{aligned} \quad (5.1)$$

where c represent a constant, and $\delta_{th}, \delta_{ya}, \delta_{pt}, \delta_{rl}$ are the input delays. To avoid the possibility of having two instances of the same input with different delays¹⁵, the delay δ is defined at tree level. In this way, if the same input appears more than once in the expression, both instances will have the same delay since they belong to the same tree (see Figure 5.3).

To evaluate the tree in order to compute the predicted acceleration, the tree is simply traversed *inorder* and each state variable and input is replaced by its numerical value at that time step (see Figure 5.2).

¹⁴As is standard in genetic programming (see [186]) protected division is defined to return 0 when a division by 0 is attempted (even in the case of 0 divided by 0) and to return the normal quotient otherwise.

¹⁵This would be in conflict with our Markovian assumption.

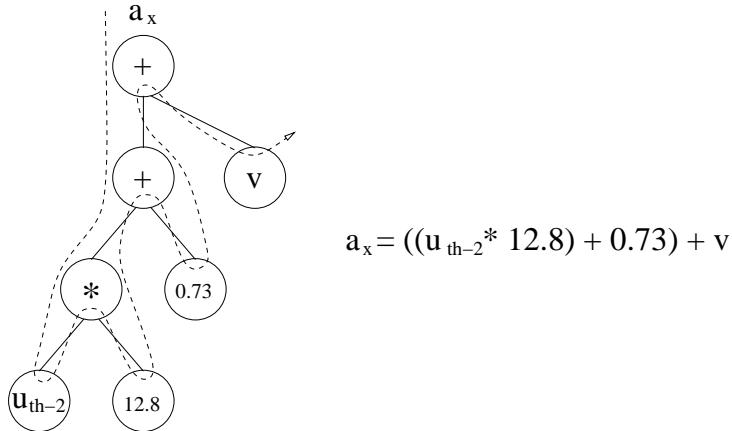


Figure 5.2: An expression tree traversed *inorder*, and the associated analytical expression.

Initialization

The principle of avoiding platform specific knowledge determines the way in which we initialize our models. To avoid any specific bias in the size or shape of the expression trees, we initialize the starting population of models using the “ramped half-and-half” method ([125]). This is based on creating half of the population as fully grown trees and the remainder as partially grown trees to give a uniform sampling of the search space.

For the first fully grown trees, nodes are drawn with uniform probability from the function set \mathcal{F} until the maximum depth¹⁶ is reached; at that level the nodes are drawn with uniform probability from the terminals \mathcal{T} . For the partially grown half, nodes are drawn with uniform probability from the set $\mathcal{A} = \mathcal{F} \cup \mathcal{T}$, and from \mathcal{T} when the maximum depth is reached. In this way trees of irregular shape and size are created.

When a node representing a constant is selected from the set \mathcal{A} , its value c at initialization is drawn from a uniform distribution; given the magnitude of the input and output variables in our datasets, a suitable interval is:

$$c \in \mathcal{U}[-30, 30].$$

For each tree the delay for each of the inputs is selected at random with equal probability from the set $\mathcal{D} = \{0, ..\Delta_{max}\}$, to provide an unbiased initialization. In our experiments we limited Δ_{max} to 5 (equivalent to 200ms) to limit the search space as was done for the models based on MLP in Section 4.2.2.

¹⁶The depth of a tree is defined as the length of the longest non backtracking path from the root to a terminal node.

Mutation Operators

At each generation of the evolutionary algorithm, every model is mutated to produce a single offspring. Since we are not using recombination, we require a type of mutation that can perform small adjustments in the expression structure and parameters, as well as larger changes to promote the exploration of the search space. These are typical requirements when only mutation is used to evolve expression trees, and an appropriate way of addressing them is to use multiple types of mutation simultaneously ([127, 44]).

Following the reasoning in [44] we designed a type of mutation for each operation that appeared to be useful for the type of expression trees that we are dealing with; we then used a stochastic scheme to combine them sequentially. We defined four mutations that work on the tree's structure (*macro*, *micro*, *insert*, *remove*), one that works on the tree's parameters, and one that deals with the input delays.

A macro mutation (Figure 5.3) has the aim of producing large changes in the tree

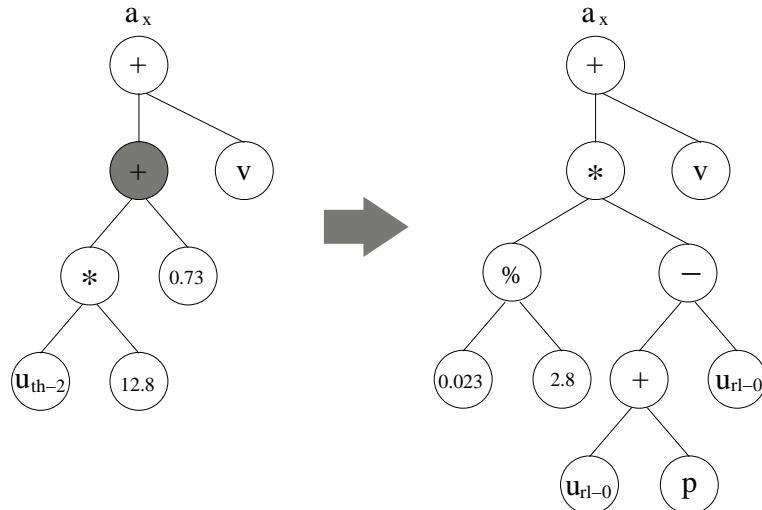


Figure 5.3: Example of macro mutation: the node '+' and its children are deleted and replaced by a newly grown subtree. It is worth noting that in the new tree u_{rl} happens to appear twice, but both instances are constrained to have the same delay $\delta_{rl} = 0$.

structure to promote the exploration of the search space. In this mutation one node in the tree is selected at random and deleted along with its subtrees; it is replaced by a newly generated tree. The nodes of the new subtree are selected at random from the sets \mathcal{T} and \mathcal{F} , but the choice is biased towards terminal nodes (using a probability equal to 0.7)¹⁷ to prefer smaller trees.

¹⁷This probability goes to one when the maximum depth is reached.

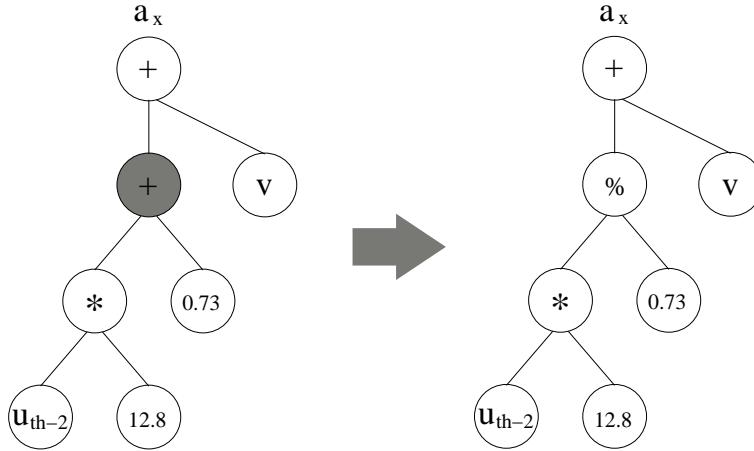


Figure 5.4: Example of a micro mutation: the node '+' is mutated into '%' and its children are preserved.

A micro mutation aims at tuning the tree structure by making small changes. During a micro mutation (Figure 5.4), one node in the tree is selected at random (all the nodes having the same probability of being selected), and is then replaced with a node selected with equal probability from the set \mathcal{A} . If the replacing node has the same arity as the replaced node, the children (if any) are retained. If the new node has a lower arity, superfluous child subtrees are randomly deleted with equal probability. If there are a higher number of children in the replacing node, the pre-existing children are retained and the necessary number of new child subtrees is added. The new child subtrees are generated randomly by drawing nodes from the set \mathcal{T} , in order not to grow the tree any further.

The micro and macro mutations are very general ways of modifying expression trees, however in the case of symbolic expressions it would be beneficial to be able to also directly insert and remove factors from an expression. For example to remove a superfluous input (see Figure 5.6) or the adding a scaling factor (see Figure 5.5). We therefore designed two mutation operations that can do these modifications directly.

The insert mutation (Figure 5.5) is the direct counterpart of the remove mutation and selects a node in the tree (all nodes having the same probability of being selected) and inserts a node above it. The inserted node is chosen with equal probability from the set \mathcal{F} . If its arity is bigger than one, the additional children are created from the set \mathcal{T} (as opposed to growing a new subtree), following the rationale that we aim at a relatively small insertion into the tree.

The remove mutation (Figure 5.6) selects a non-terminal node in the tree (all non-

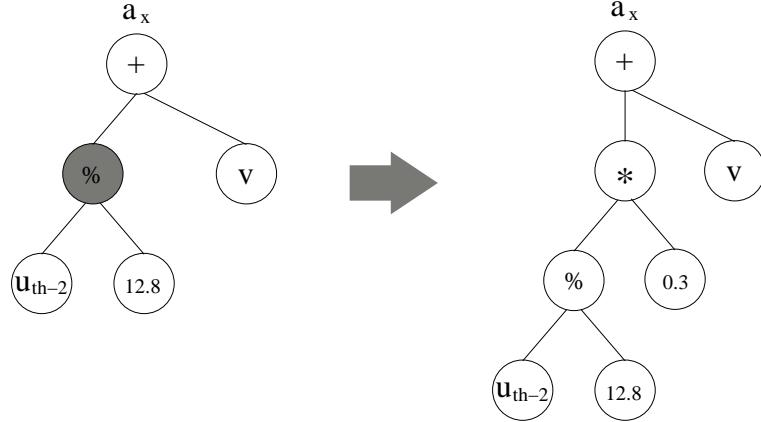


Figure 5.5: Example of an insert mutation: a new node '*' is inserted above the node '%'.

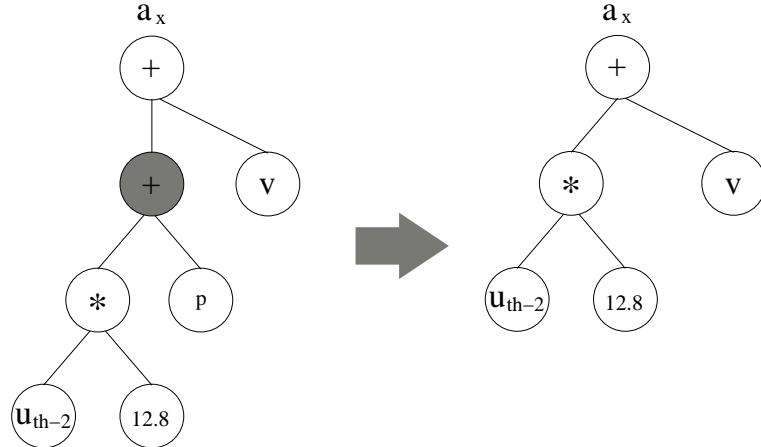


Figure 5.6: Example of a remove mutation: the node '+' is removed and only one of its children is preserved.

terminal nodes having the same probability of being selected) and removes the father of the node. If the father has other children in addition to the one selected, they are discarded.

The constant mutation aims at tuning the parameters within the model's expressions, and is based on the idea of adding a random quantity generated from a Gaussian distribution as suggested in [206]. But we also allow for reinitializing the value of the constant (with a small probability) for situations in which a more drastic effect on the expression is needed (a similar scheme is used in [31]). One constant node is selected at random from the expression tree (all constants in the tree having an equal probability of being selected), and with a high probability (0.7) we add to its value a sample from a Gaussian distribution

with mean zero and variance 0.02,

$$c = c + \epsilon \quad \epsilon \in \mathcal{N}(0, 0.02).$$

In the remaining cases, the constant is redrawn from the distribution used to initialize the constants (namely $\mathcal{U}[-30, 30]$). Exploratory runs showed parameters from the literature ([206, 31]) to be appropriate, for the variance of the Gaussian distribution from which we draw samples and for the probability of selecting the type of mutation applied to the constant. This type of mutation is also used to re-converge the tree expressions after simplification, as explained in Section 5.3.1.

The final mutation deals with the input delays associated with each of the inputs so that all the terminal nodes representing the same input have the same delay (see Section 5.3.3). In dynamic systems like those we are working with, there will be an optimal delay, but delays very close to that value will very often be suboptimal but viable solutions. We have embedded this understanding into our mutation operator by defining it as a Gaussian perturbation of the original delay value. With equal probability, one of the input delays is selected and its value (δ_i) is changed:

$$\delta_i = \min(\Delta_{max}, \max(0, \delta_i + \lfloor \mathcal{N}(0, 1) \rfloor)) \quad i \in \{th, rl, ya, pt\},$$

where $\min()$ and $\max()$ simply return respectively the minimum and the maximum of their arguments while $\lfloor \cdot \rfloor$ indicates the round off operation. By choosing the variance to be equal to 1 we ensure with a high probability that the change in delay is of only one time step, while still allowing less probable larger changes.

The mutations are combined as suggested in [44]. More specifically, a sample n is generated from a Poisson distribution $\mathcal{P}(4)$ and for n times one of the six mutations is selected with equal probability (with replacement) and applied to the current individual. While the exact number will change, on average four types of mutation will be applied, delivering mutations with different levels of impact on the original tree.

Fitness Function

A vital part of our evolutionary algorithm is the definition of a fitness function that can reflect the progress of the modelling process. In machine learning terminology, we need to

define a loss function for our model fitting.

In the domain of symbolic regression many ways to measure error have been proposed, including squared error, absolute error, correlation, and methods based on combinations of statistical features of the data to be fitted ([107]). In fact, loss functions are directly connected to the idea of robust estimation ([104]), since different functions will weight the error residuals differently. For example, a squared error loss will weigh a large error more heavily than will absolute error. For symbolic regression the general understanding is that, although the choice is often not critical to the optimal solution, different metrics can work better on different problems ([205]).

For the experiments in this paper, we will use the mean absolute error, applying the rationale that avoiding over-penalizing models with large errors should lead to a better behaved search¹⁸. In addition the absolute error has also been shown to be suitable for a similar dynamic modelling methodology ([29]). More formally the fitness function f_M for the model evolution will be defined as:

$$f_M = -\frac{1}{TW} \sum_{w=0}^W \sum_{t=t_{0w}}^{t_{0w}+T} |z_t^i - y_t^i| \quad i \in \mathbf{x}, \quad (5.2)$$

where, using the nomenclature introduced in Chapter 2, z^i are the measured values of the variable i while y^i are its predicted values and t_{0w} indicates the starting time of the window w ¹⁹.

Let us assume for instance that we are training an expression tree to predict the linear acceleration a_x . To calculate f_M for each of the data windows (i.e. tests in the bank), we first initialize the state at time t_{0w} with its value $\mathbf{z}_{t_{0w}}$ taken from the recorded dataset. We then evaluate the expression tree in order to compute the predicted a_x . In this example a_x will be the first component of the acceleration vector, while the remaining accelerations are precomputed from the dataset. The resultant vector is integrated forward in time to produce the state at time $\mathbf{y}_{t_{0w}+1}$ that is used to compute the absolute error. This process is repeated T times to cover the whole time window, each time starting from the previously predicted state. The fitness f_M computed in this way embeds the ideas of both windows and error integration that we have already discussed thoroughly.

¹⁸A higher probability of producing accurate models.

¹⁹The minus sign has the role of simply transforming the optimization problem from a minimization to a maximization.

We have purposely avoided being specific about the state variable i that is used in the fitness function; this will in fact depend on which of the specific accelerations we are evolving a model for. In our algorithm we use the state variable directly connected to the acceleration in question; for instance, if we are predicting the local linear acceleration in the x axis (a_x), we will use the local linear velocity in the x axis (u). The rationale behind this choice is based on the fact that in general, due to the nonlinear transformations in the rigid body dynamics of our systems, complex nonlinear dependencies exists between the model error (in acceleration) and state variables. However, if we limit the fitness to consider only the velocity error in the dimension of the acceleration for which we are building a model this dependency is much simpler²⁰ and this should help the search. Our choice of using only one variable avoids the problem of using scaling factors to adjust the contributions of the different variables, which as we commented earlier (Section 4.1.1) is not straightforward. A complete list of the state variables associated with each of the accelerations is shown in Table 5.1.

It is worth noting that more than one variable could be used for the computation of the absolute error; however, this would introduce the problem mentioned previously of weighting the separate contributions (see 2.2.5), a problem that we know does not have a platform agnostic solution.

Predicted acceleration	Variable used to compute fitness
a_x	u
a_y	v
a_z	w
α_x	p
α_y	q
α_z	r

Table 5.1: State variables used to compute the model fitness f_M .

²⁰More precisely, it is linear for the angular velocities (equation 3.6), and approximately linear for the linear velocities in the case of moderate changes of orientation between two consecutive time steps(i.e. when the small angle assumption would hold for equation 3.5).

5.3.4 Evolution of the Tests

Representation

Choosing an adequate representation for the windows of data used as test individuals is straightforward. Since in our datasets the entries are time stamped, and each window (i.e. test) has a predefined fixed size, the starting index t_{0w} of the window is a simple and appropriate representation.

Initialization

At the beginning of each epoch the population of tests is initialized at random. The starting index for each of the test data windows is chosen with uniform probability from all the possible indexes in the dataset:

$$t_{0w} = \mathcal{U}[\Delta_{max}, N - T], \quad (5.3)$$

where the minimum allowed index is Δ_{max} so that delayed inputs are available even in the first window; the maximum index is equal to the size of the dataset N minus the size of the data window T to ensure that all the windows have the same length.

There is a distinct possibility of randomly producing windows that are partially or completely overlapping. Since we are searching for the data chunk that is most effective in discriminating between the models' performance, using the same data multiple times is not a very effective or efficient choice. To avoid this problem, every time a new window is generated, it is tested against the existing windows; if it overlaps, it is discarded and regenerated according to equation 5.3.

Mutation Operators

As in the case of the models, no recombination operation is used during evolution. We therefore need the mutation operator to be able to produce both large changes in the genotype to produce exploration, and also smaller changes to be able to “tune” the test in question.

In the experimentally collected data there are segments that are more discriminatory than others, either due to a manoeuvre that was executed by the pilot, or to a specific set of system states. Since our sampling rate is high compared to the dynamics of the

control inputs and of the systems, discriminatory data points will almost invariably be in the form of contiguous chunks. Building on this idea, we designed the first of our two mutation operators: the *slide mutation*. This operator simply slides the data window (i.e. its starting index) forwards or backwards in time by adding a random displacement, with the aim of producing an improvement in the discrimination ability of the test. More formally, for the window w :

$$t_{0w} = t_{0w} + \lfloor \mathcal{N}(0, T/2) \rfloor,$$

where $\mathcal{N}(0, T/2)$ is a zero mean Gaussian distribution with variance $T/2$ and $\lfloor \cdot \rfloor$ denotes the *round()* operation. By choosing the variance to be equal to $T/2$ we ensure with a high probability (i.e. 0.97) that the increment will not be larger than the window size and therefore that the window will tend not to slide off its current position by a large amount.

To explore different parts of the dataset it is certainly useful at least occasionally to “jump” further than the *slide* mutation would allow. With this aim, we therefore introduce a second mutation operator, *jump* mutation, which generates a new starting index for the window according to a uniform distribution over the whole dataset. This is identical to the way the window indexes are initialized according to equation 5.3

At the end of every generation, each of the tests in our (1+1)EA undergoes mutation, with equal probability of being a slide mutation or a jump mutation.

Fitness Function

A crucial part of the evolution of the tests is finding an effective way of defining the quality of a test. In the spirit of competitive coevolution, one could define the quality of a test in a very naïve way, as its ability to reveal error in the model. The worse the performance of the model, the better the ability of the test at highlighting its deficiencies. However, after the remarks made in Section 5.1, we can easily see how such a fitness function would be a recipe for disaster, since it would almost certainly produce tests that were too difficult. In the case of our application, the data coming from our experimental runs might contain sections in which the platform behaviour was affected by external disturbances²¹. Since those inputs are not measurable, their effects on the dynamics are not modellable, and therefore trying to learn them would simply lead nowhere. But our naïve fitness function

²¹Air turbulence in the case of the quadrotors, or anisotropic friction in the contact with the floor for the toy car.

would select those spots in the data, imposing on the models a task that is impossible to learn. The result of this would obviously be disengagement between the two evolving populations.

In [208] Seung *et al.* proposed a selection method for test queries based on the idea of maximizing the disagreement between the learners. Such a criterion was adopted in the specific context of coevolutionary learning by Bongard and co-workers in [28, 29, 31]. Seung *et al.* provided some theoretical justification of the proposed method, but the method also has an intuitive interpretation. By choosing the test that maximizes the disagreement between the models, we choose a test that is highly discriminative and is able to reveal the differences among models. In the case of data selected from the training dataset, selecting data on which all the models perform well, or all of them perform badly, does not provide information about which models are better. In contrast, a portion of data on which some models perform well and some others do not is highly informative in exactly the way we require.

We have implemented this concept by using a fitness metric that, given a test, computes the predicted variable over the whole test for each of the best models. For this set of trajectories, it is then possible to compute the mean prediction error and the error variance for each time step (see Figure 5.7). The sum of this variance over all time steps and all

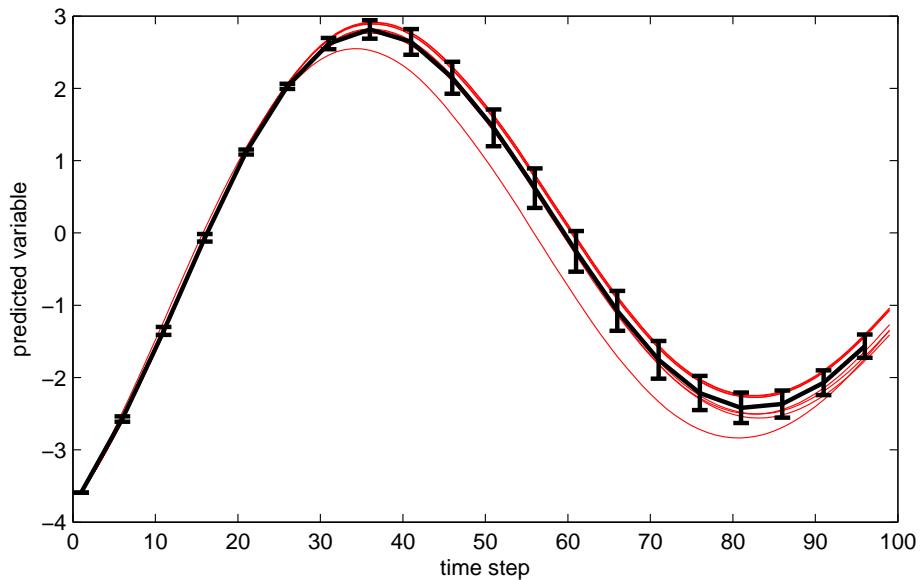


Figure 5.7: Prediction output of 6 different models (thin lines). We also show the mean prediction (thick line), and the variance (error bars) which we use as index of model disagreement.

best models can be used as indication of the disagreement between tests. More formally the fitness of a test w is computed as:

$$f_{Tw} = \sum_{m \in \mathcal{M}} \sum_{t=t_{0w}}^{t_{0w}+T} (z_{m,t}^i - \bar{y}_t^i)^2 \quad i \in \mathbf{x}, \quad (5.4)$$

where $z_{m,t}^i$ is the prediction of model m for the variable i at time t , \bar{y}_t^i is the mean prediction from all the models:

$$\bar{y}_t^i = \frac{\sum_{m \in \mathcal{M}} (y_m^i)_t}{|\mathcal{M}|}$$

and \mathcal{M} is the set of best models.

5.4 Analysis of a Typical Coevolution Run

Before examining the task of evolving models for each of the platforms, it will be interesting to look at a typical coevolution run to examine the interplay between models and tests during coevolution. For this, we added additional logging capabilities to our software implementation, and carried out a single coevolution run using the toy car data (see Section 5.6 for further details on the setup). For the sake of clarity we preferred to use the toy car data in this specific context since it is only three dimensional, and the analysis results are simpler while still being very informative.

We first look at the progress of the models' fitness during the 12000 generations (600 generations for each of the 20 epochs) that constitute a typical coevolution run. In Figure 5.8, for each of the three equations being evolved we plot the local fitness of the best model (i.e. the model with the highest fitness in the set of best models). In looking at the plot we have to remember that the quantity shown is the local fitness, or in other words the errors of the models on the current tests. The absolute value of the local fitness is of limited significance, since the tests change at every epoch. Instead, what is meaningful is the progress within each of the epochs, as this shows the performance of the (1+1)EAs in optimizing the models. For all three equations and for all epochs, it is clear that within each epoch the models are improving their fit, to varying degrees. This tendency is obviously more marked in the initial phase of coevolution, but it is also retained through the remaining epochs. At the beginning of a new epoch, a sharp reduction in fitness is sometimes seen, obviously because the tests have changed, but the lost ground is quickly

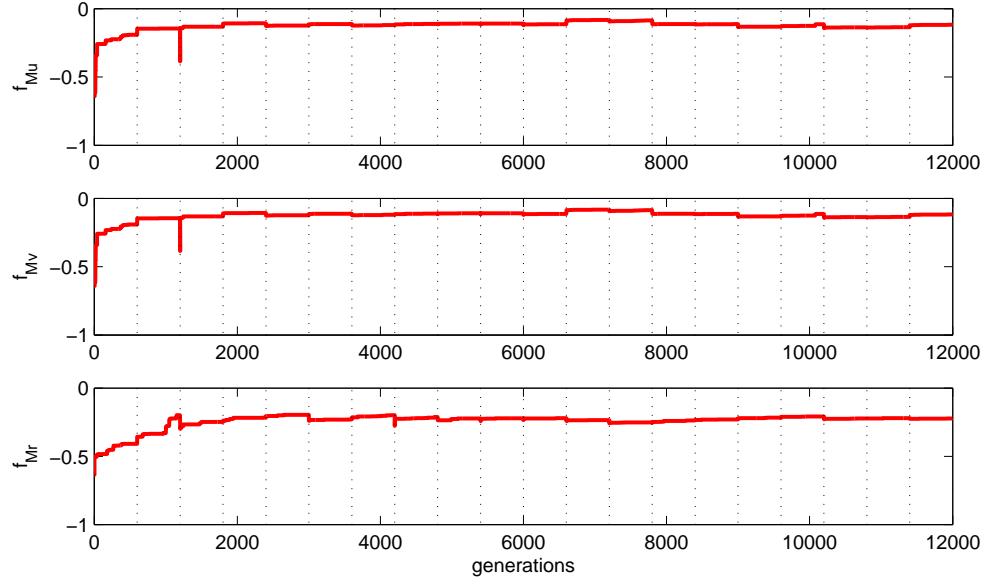


Figure 5.8: Local fitness of the best model during the 20 coevolutionary epochs. Dotted lines delimit the epochs.

recovered within the first few generations of the epoch.

Reassuringly, none of the epochs show visible fluctuations in fitness or unexplained drops in fitness which would suggest possible disengagement between the two competing populations.

At the start of each run, trees are constructed sampling uniformly (using the ramped half-and-half method) from the space of model tree sizes; during a run the sizes of trees will change, ideally converging towards the size that the problem requires. Figure 5.9 shows how the mean size of trees progresses during the run. Initially the mean size of trees is around 20 nodes, but in the first few generations, the trees reduce in size and quickly settle to a smaller mean size of around 12 or 13 nodes. Comparing Figure 5.9 with Figure 5.8 we see that, during those first generations distinguished by a rapid reduction in the size, fitness increases rapidly. This suggests that for the toy car problem, small expressions are sufficient to express the principal contributions of the model dynamics. In the remaining epochs, two phases can be distinguished. The first is the first half of the epochs in which the size of trees grows more or less steadily and is marked by an increase in fitness, suggesting that as the tree expressions become more complex, they can better explain the training data. In the second phase, the rate of change of the size of trees reduces. For the lateral acceleration a_y , the mean size reaches a plateau, but for a_x and α_z only a change in rate is visible and it is not clear that the trees are settling to a particular

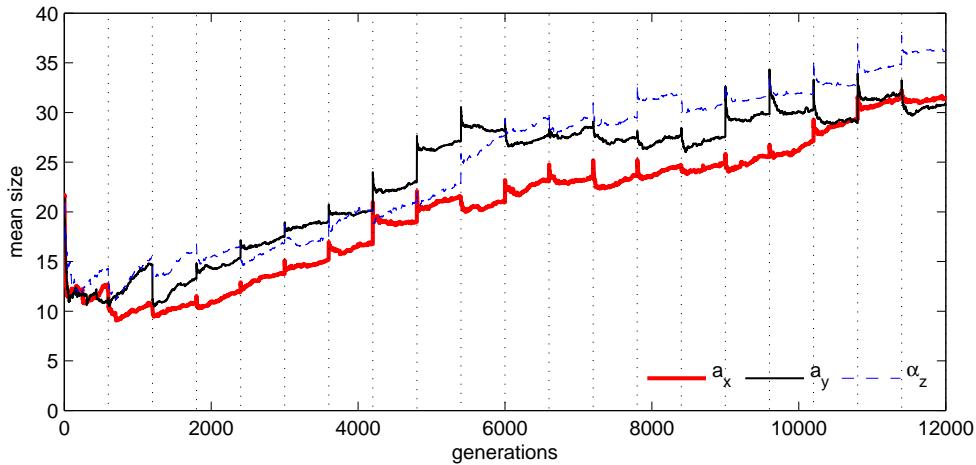


Figure 5.9: Mean size of the genetic programming trees during the coevolution run.

size. Comparing this plot with Figure 5.8, we can see that this phase of the coevolution is not so clearly associated with any substantial change in fitness. This suggests the presence in the expression tree of components that do not have a direct effect on the model fitness. In the genetic programming literature such components of the expression trees are called introns ([186]) and some authors ([17]) have suggested that their presence can be beneficial by providing genetic material that an appropriate mutation could convert in a useful part of the model.

Some of the choices made during the design of our coevolutionary algorithm were aimed at maintaining a substantial amount of genotypic diversity in the population of models during the coevolution run. Given the variety of equations that can be represented by the expression trees, defining diversity is not at all trivial. While we have to avoid any definition of diversity that requires knowledge about the specific platform for which we are evolving models, we need a definition of diversity that is well suited to the fact that our expression trees represent dynamic equations. More specifically we would like a measure that expresses the fact that, in an equation, the structure of a tree is important. For instance a function at the root of the tree is applied to the whole tree and therefore has a larger effect than for example the same function at a terminal node. The definition of diversity should also take into account the type of node, since for example a multiplication is clearly different from an addition operation.

In the literature various methods have been proposed to define the diversity of expression trees; an in-depth discussion is outside the scope of this work (see e.g. [69]) but for

our purposes it will suffice to look at the principal ways of defining diversity.

Koza in [125] defined diversity as the percentage of individuals for which a duplicate in the population does not exist, while Langdon proposed a similar measure based on the number of unique individuals in a population [129]. Although those are sound ways of defining diversity, they do not embed any understanding of the fact that we are dealing with evolving mathematical expressions, and therefore are not particularly suited to our use.

Other authors have proposed measuring diversity by introducing the concept of distance between trees. Lu [139] (and similarly Keller and Banzhaf [118]) defined the concept of *edit distance* which measures the number of changes needed to transform one tree into a second one. In [65] Ekárt and Németh introduced an edit distance specifically designed for genetic programming that reflects the structural differences between trees. In their edit distance, types of nodes are taken into account, and differences near the root have more weight. Moraglio and Poli [164] defined a similar structural distance (*structural Hamming distance* (SHD)), that has the added benefit of being normalized between 0 and 1. The SHD is defined as:

$$dist(T1, T2) = \begin{cases} \delta(p \neq q) & \text{if } arity(p) = arity(q) = 0, \\ 1 & \text{if } arity(p) \neq arity(q), \\ \frac{1}{m+1} \left(hd(p, q) + \sum_{i=1,..,m} dist(s_i, t_i) \right) & \text{if } arity(p) = arity(q) = m, \end{cases} \quad (5.5)$$

where p and q are nodes of the trees $T1$ and $T2$ respectively, $arity(p)$ indicates the number of children of the node p and s_i and t_i are children of the nodes p and q respectively. The Hamming distance $hd(p, q)$ is defined as:

$$hd(p, q) = \begin{cases} 1 & \text{if } p = q \\ 0 & \text{if } p \neq q, \end{cases} \quad (5.6)$$

Looking at equation 5.5 we can see that what the SHD metric is doing is simply taking two trees, visiting them *inorder*²² and comparing their nodes. If the nodes are the same

²²To traverse a binary tree in *inorder*, the following operations are carried out recursively at each node:

1. traverse the left subtree;
2. visit the root
3. traverse the right subtree.

their distance is 0, otherwise 1. Since the metric is recursive and the distance between two subtrees is scaled by the factor $\frac{1}{m+1}$, differences between nodes close to the root are weighted more heavily than differences at the terminal nodes. In the extreme situation of two trees that are completely different, their distance is one, while any tree will have distance 0 from itself.

The SHD distance fulfils the qualitative requirements that we set out in order to capture the fundamental elements that define the similarity between equation expressions; this makes it well suited to our analysis and for this reason it is the metric chosen in this thesis to define tree diversity. In Figure 5.10 we display the progress of the mean SHD computed over the population of models during the coevolution run. Since the distance is defined between two trees, the mean is computed over all the possible couples that can be formed using the models in the population.

For all the three functions, a general trend can be seen during the whole coevolution run (i.e. across epochs). The mean SHD is initially very close to one since the initial population consists of randomly generated models. During the first generations a rapid decrease in mean distance can be seen as the models start converging. In the epochs immediately following the first, the diversity keeps reducing up to about 5-6000 generations. Due to the effect of coevolution, changes in the selection of tests can produce large changes in mean SHD as it is the case for example for the a_x expressions. Although the diversity decreases as the evolution proceeds, importantly it does not collapse to very low levels ensuring a continuing exploration during the coevolution run.

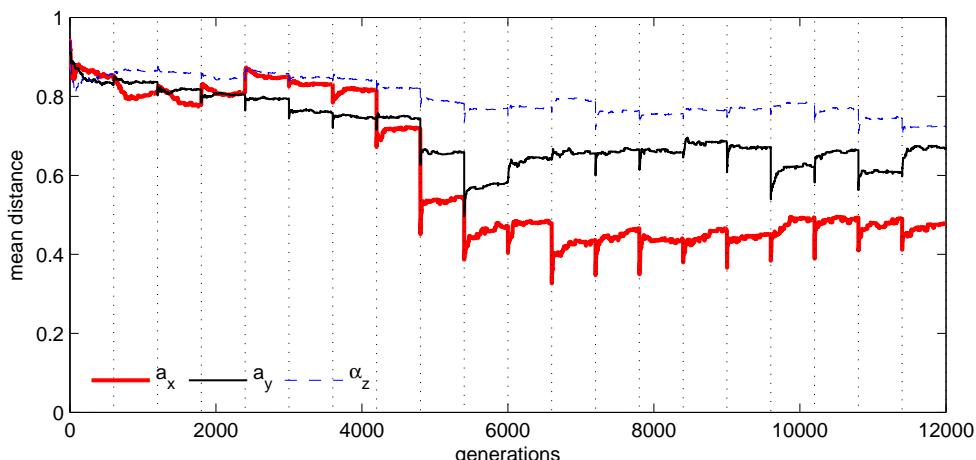


Figure 5.10: Mean structural Hamming distance of the genetic programming trees during the coevolution run.

Within a single epoch, an expected decrease in mean SHD is noticeable at the beginning of each epoch. This is due to the selection scheme applied which preserves 80% of the individuals and fills the remaining of the population with mutated copies of the best models (as explained in Section 5.3.2). As an effect of mutation, diversity is quickly re-established within a small number of generations.

We now turn our attention towards the evolution of tests. At the end of each epoch of test evolution, we logged the population of tests. For each of the data points in the training dataset, it is then possible to count the number of epochs during which each such data point was actually used to compute the models' fitness.

Figure 5.11 plots the resulting histogram for each of the three equations being evolved. For each equation we can see that certain areas of the dataset have been selected more than others, since they are able to produce larger disagreements among models. In general, different chunks of data might be expected to be better suited to the evolution of each of the three model equations. However in the case of the toy car, we see that certain areas have been heavily used for all three equations. In contrast, other parts of the dataset have not been used for any equation.

It will be interesting to look at the state and input data during these time intervals,

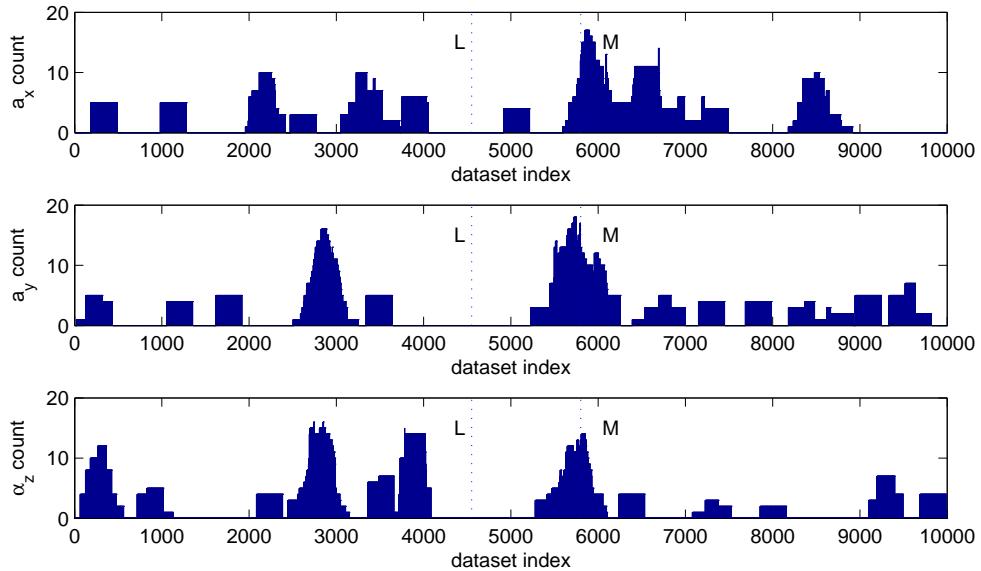


Figure 5.11: Histogram of usage of the data samples from the training dataset for each of the three equations of the model (a_x, a_y and α_z). The count indicates the number of times (epochs) that the data point was used for training. The dotted lines mark the data points chosen as representative of most often and least often used data (M and L respectively).

and to do so we examined two window of 300 data samples centred respectively at one of the most (Figure 5.12) and one of the least (Figure 5.13) used points in the dataset (those are indicated by M and L respectively in Figure 5.11).

Comparing the two windows, it is clear that the most used data window (Figure 5.13) is characterized by more activity. More precisely, if we look at the steering input u_{st} it is clear that for the window of least used data, the duration of the control inputs applied was very short, and as a consequence, due to the low bandwidth (slow response) of the toy car, the resulting rotational and lateral velocity changes are very limited. The opposite is true for Figure 5.12, which shows much slower changes in the steering commands, strongly correlated with large changes in both rotational and lateral velocity.

These two examples of selected and avoided data support the intuitive understanding that we used during the design of our algorithm; with the coevolutionary algorithm selecting the parts of the dataset that are information rich, which ultimately should improve identification.

The analysis just presented is limited to a single run on only one of our platforms. Ideally, of course, a similar analysis should be extended to all our runs and all platforms. Unfortunately, given the number of platforms analyzed, and the sheer number of coevolutionary runs involved, such a detailed analysis must be left for future work. However in Section 5.9 we continue the investigation of our coevolutionary algorithm by looking specifically at the effect of various choices of parameters.

In the following sections we apply our novel coevolutionary algorithm to each of our platforms. With the exceptions of the size of the data test windows, and obviously of the number of equations (i.e. dimensions) being evolved, all the remaining parameters of the coevolutionary algorithm were kept the same to avoid any tailoring of the algorithm to the specific platform. Such an operation would be obviously in conflict with one of the main aims of this work.

5.5 Automatic Modelling of the ATTAS Aircraft

With all the ingredients of our coevolutionary setup in place we can now start testing it on our first platform; the ATTAS aircraft.

As with the parametric models presented in Section 4.1.1, our aim will be to produce

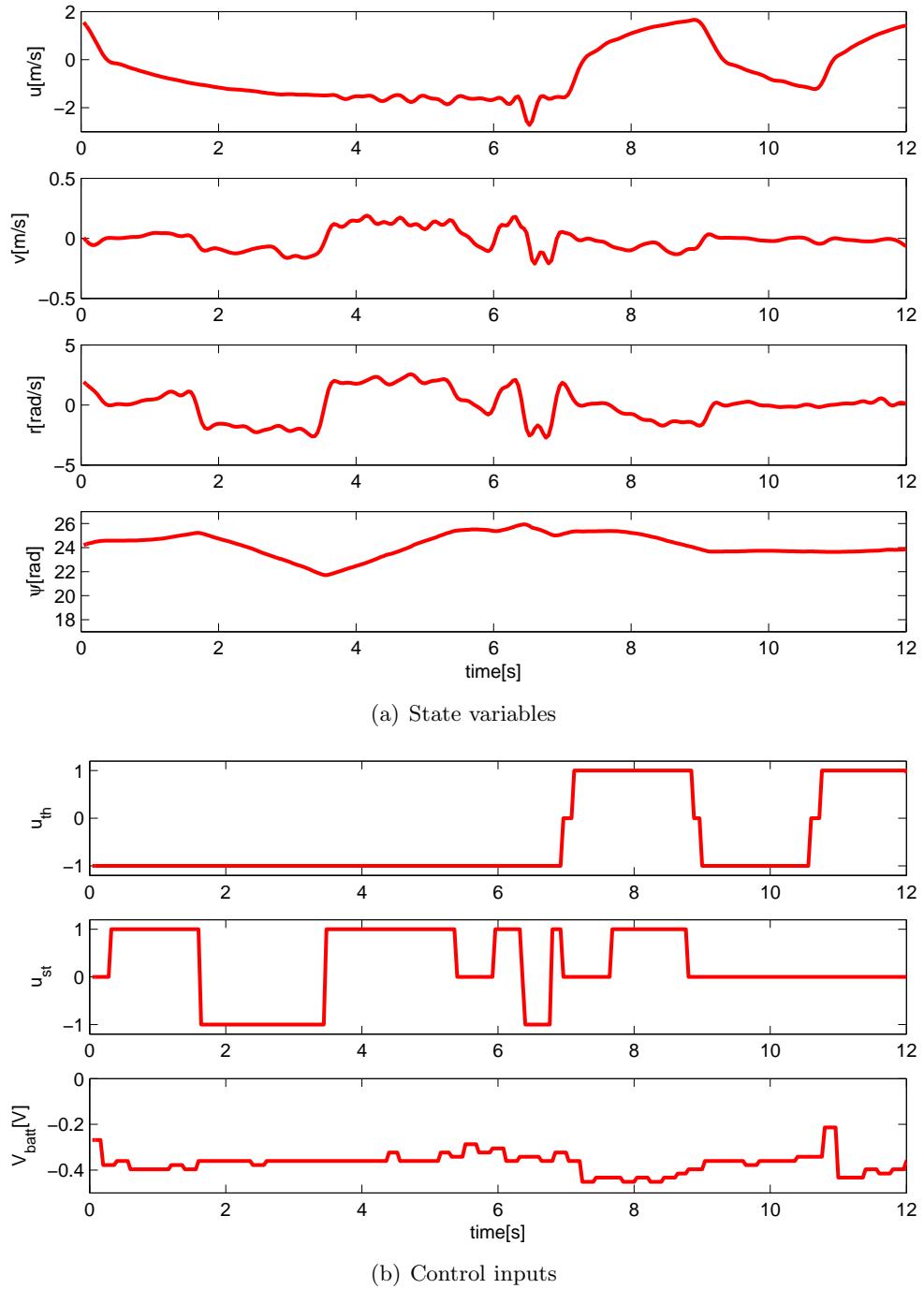


Figure 5.12: Plot of the state (a) and control (b) variables in a window of 300 data points centred on one of the most used data points (dotted line M in Figure 5.11).

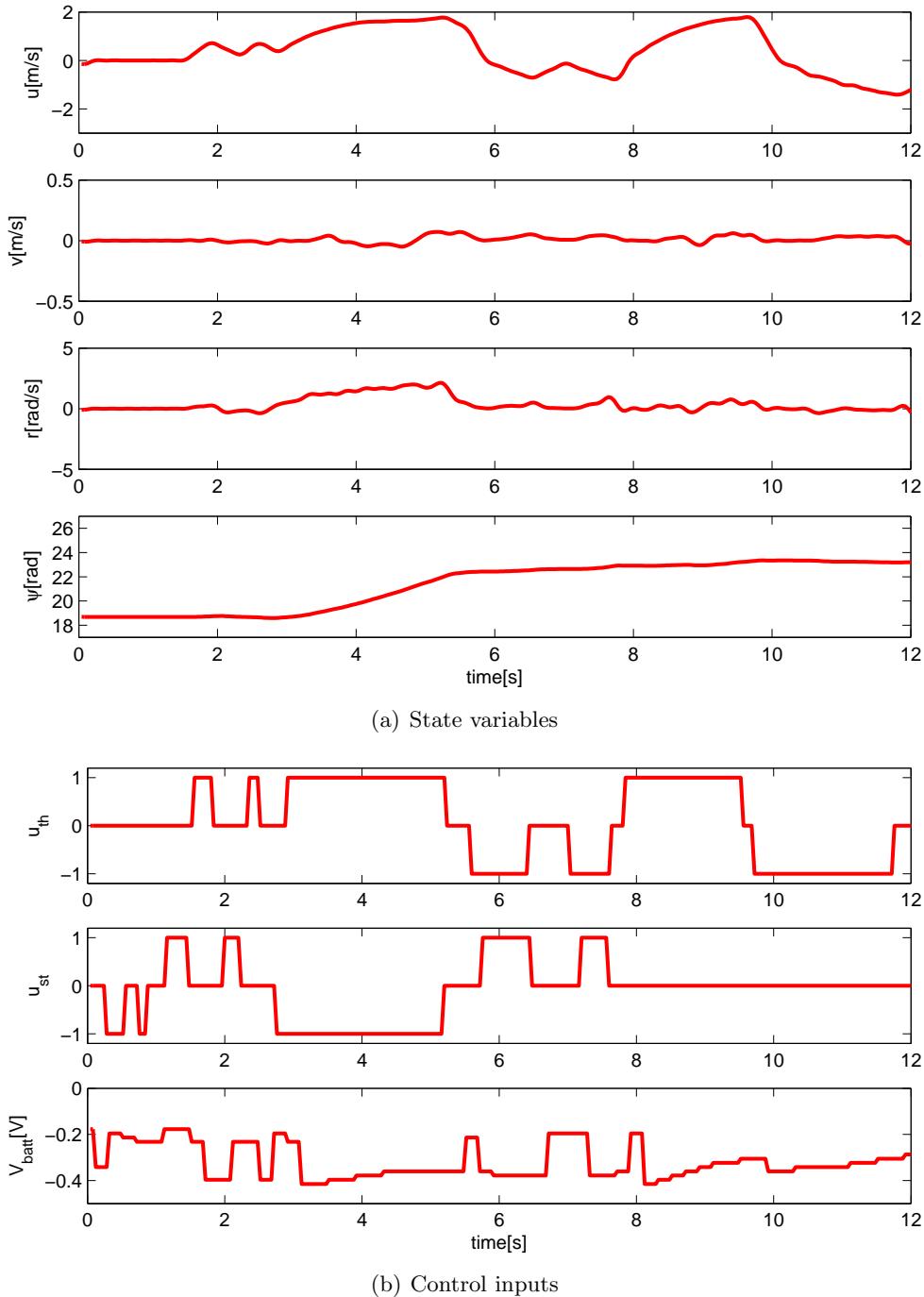


Figure 5.13: Plot of the state (a) and control (b) variables in a window of 300 data points centred on one of the least used data points (dotted line L in Figure 5.11). This data window when compared to the one of Figure 5.12 shows a lower level of activity in both the state and control variables.

a model in acceleration space. Defining the state as in Section 3.3.4, we will be learning a model in the form expressed by equation 3.20 that we copy here for convenience:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (5.7)$$

Three different expressions will be learned, one for the prediction of a_y and other two for the angular accelerations α_x and α_z respectively. We train the three models using data from dataset `fATTASAilRud2` and using the same size of windows ($T = 100$) as used in Section 4.1.1 for the models based on first principles. This will help the comparison of the two approaches. The inputs to our model are the state $\mathbf{x} = [u, \phi, \theta, p, r]^T$ and the controls $\mathbf{u} = [u_{ai-\delta ai}, u_{ru-\delta ru}, \beta]^T$. The maximum allowed delay for the control inputs was set to 5 time steps²³.

5.5.1 Convergence

To investigate the repeatability of the results of the coevolutionary algorithm, 30 independent runs were carried out. For each of the epochs during coevolution, we saved the best model (the one with the highest fitness among the best models) for each of the dimensions being evolved. The local fitness computed at runtime during evolution will depend on the current set of historical tests, which in turn is changing during the run. The saved models will therefore not have been tested against the same data. Moreover, we have already discussed (Section 2.2.5) how evaluating the performance of a model on the training dataset is not truly indicative of model quality. For both of these reasons, we compute the *RMSE_m* of each saved model against the validation dataset `fATTASAilRud1` which was not used during evolution. In this way we obtain an unbiased measure of model error which ultimately is what we require.

As a consequence of the choice of using a validation dataset, it is possible for a model deemed to be good by the coevolutionary algorithm to register an error score which may not be low. This can happen because the validation dataset might contain a section of data that the model is not able to predict. This problem can be limited by choosing, as we did in our work, two datasets for training and validation that both cover adequately the

²³In order to maintain the same setting across all the different platforms the maximum allowed delay was set to 5 time steps. However, in the case of the ATTAS aircraft we know from [113] that good models can be obtained without any delay in the control inputs.

dynamic envelope of the platform.

As explained in Section 5.3, the mathematical expression of each of the models' dimensions is evolved independently, and so it is interesting to look not only at the error of the whole model but also at the error of each single equation; this allows us to understand the relevance of the coupling between the different equations. To do so we compute the $RMSE_m$ of each of the specific variables used for evolution (according to Table 5.1) throughout the progress of the coevolutionary runs. The $RMSE_m$ is computed as in equation 4.8²⁴.

In Figure 5.14 we display the performance ($RMSE_m$) of the best model for each of the 30 runs throughout the 20 coevolutionary epochs. All three dimensions show marked convergence for a large proportion of the models within the first few epochs. In the case of the equations of the angular velocities p and r , all the models converge by the 20 epochs mark, but this is not the case for the linear velocity v models, a small number of which have higher error.

Looking at the plots, is also noticeable how some models exhibiting a low error at a given epoch might then score a worse fitness at a later generation; this is particularly obvious for the best model for the dimension v , the fitness of which seems to be actually diminishing from the 3rd to the 4th epochs. Such effects are not unexpected, and are due to the fact that we are measuring the performance of the models on a validation dataset as previously explained. If the two datasets are essentially covering the same data envelope, any effects of this type should be of minor importance, as they appear to be in our case.

Ultimately, what defines the quality of our models is their combined performance as a full 3DoF model of the aircraft. For each run, the best models for each of the dimensions, (those that we have already discussed) have been combined into full three dimensional dynamic models, and the $RMSE_{m_{total}}$ has been calculated and plotted in Figure 5.15 as a function of the training epochs.

The overall convergence behaviour closely resembles the individual convergence plots previously analyzed, confirming the intuition that convergence in each of the dimensions is also associated with convergence of the complete model. The fact that some of the single dimension models have higher errors than others is reflected in the complete model; some of the models clearly do take more epochs to converge. Nevertheless, all the models still

²⁴As explained in Section 5.3, values from the dataset are used for accelerations which are not being predicted.

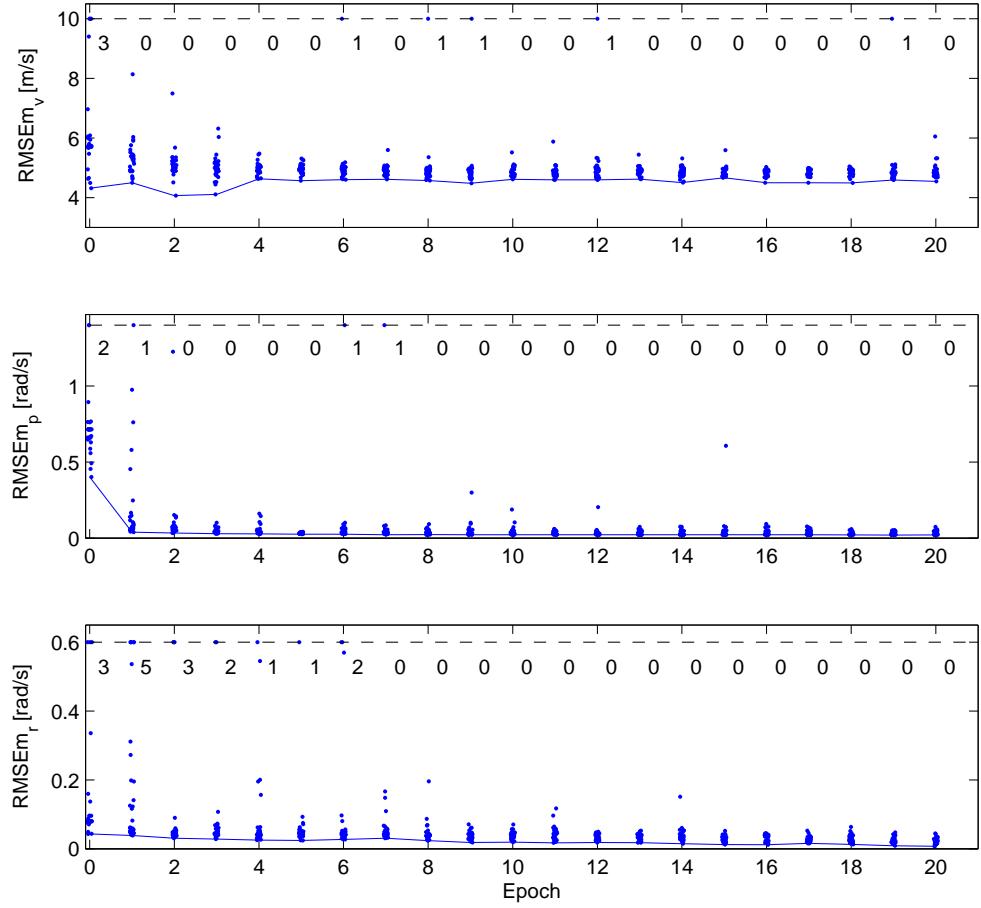


Figure 5.14: The error on single dimensions of the best model during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. The vertical scale of the plot is chosen proportionally to the standard deviation of the signal being predicted to allow for direct visual comparison between the three plots. The value of the points that exceed the maximum of the scale has been limited to the level indicated by the dashed line, and the figures at the top of each plot indicate how many points have been limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

converge before reaching the 20th epoch.

We can now look at how the size of the expression trees changed during coevolution. The average size of the best model from each of the 30 coevolutionary runs is shown in Figure 5.16. For all three equations, the average expression size of the best models grows very quickly during the first epoch, after which it shows a much slower rate of increase. Unfortunately 20 epochs are not sufficient to reveal whether the increase eventually stops.

Comparing Figure 5.16 with Figure 5.15 shows that the initial fast growth in size is associated with a large reduction in $RMSEm_{total}$ suggesting that the expressions become progressively more complex in order to account for the training data. It is interesting to see

how the angular acceleration α_z is associated with smaller expressions, perhaps suggesting that the yaw dynamics is intrinsically less complex than the lateral and roll dynamics. The fact that the average size of tree dimensions reaches values much smaller than the maximum tree size²⁵ suggests that the simplification mechanism explained in Section 5.3 is indeed effective in preventing excessive tree growth.

5.5.2 Performance Analysis

So far we have been analyzing the ability of our coevolutionary algorithm to consistently converge to a solution, assuming by implication that the algorithm was indeed converging to good solutions. It is now time to have a close look at the quality of the models being evolved.

We start as usual by plotting the time predictions of our models against the experimentally collected real data. Simply plotting the performance of all 30 models obtained will lead to difficulties of interpretation, so we limit ourselves to analyzing the qualitative behaviour of the best (*ATTASAccGPbest*) and the median (*ATTASAccGPmedian*)

²⁵For trees with a maximum depth of 7, the maximum number of nodes is 128.

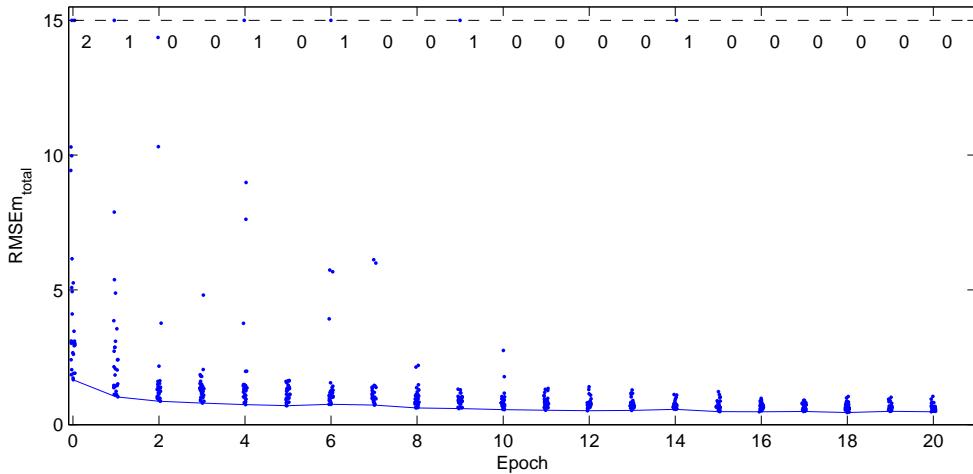


Figure 5.15: Total model error ($RMSEm_{total}$) during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. To focus on the most interesting part of the plot, the value of some of the points has been limited to the level indicated by the dashed line. The numbers at the top of each graph indicate how many points have been limited. The position of the dashed line was chosen as a compromise between providing a good picture of the distribution of models performances and containing the number of points limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

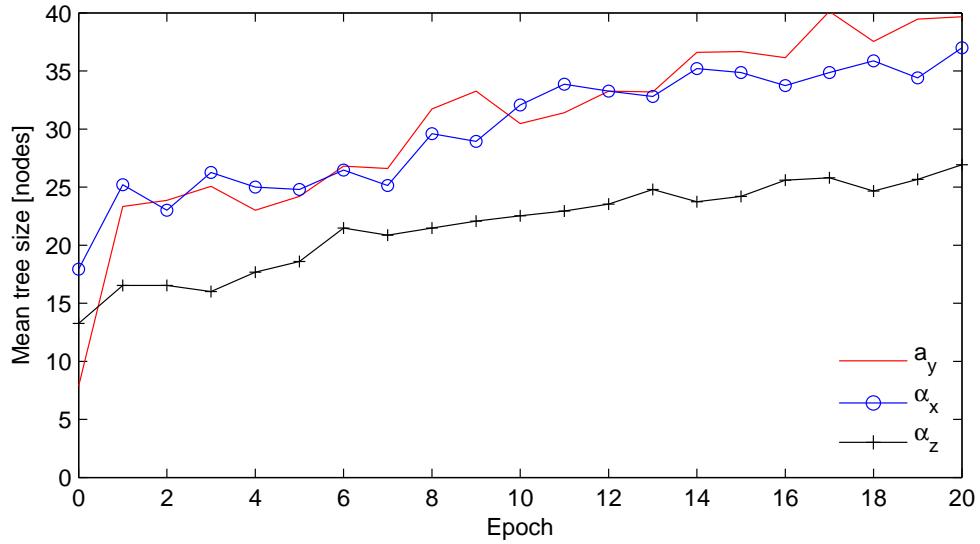


Figure 5.16: Size of the expression trees throughout the 20 coevolution epochs. The mean size computed on the best models of each coevolution run is reported. On average the expression for the angular acceleration α_z are smaller suggesting a simpler dynamic model.

models; we will measure the performance of all the other models in our quantitative analysis²⁶. To allow a straightforward comparison with the first principles model produced in Section 4.1.1, we use the same window used in Section 4.1.1. The predicted state and the controls are shown in Figure 5.17 and Figure 5.18 respectively.

In this specific window, the predictions of the *ATTASAccGPbest* and *ATTASAccGP-median* models are both reasonably good for most of the state variables. The angular quantities ϕ and ψ are subject to the effect of error accumulation, and tend to have worse performance than their velocity counterparts. The best model exhibits better predictive ability in all three velocities (p, r and v), and as a result also accumulates smaller errors in the ϕ and ψ state variables but especially with ϕ .

We know that this is a randomly selected window of data, so it is interesting to see how the performance on this window (see Table 5.2) is representative of the *RMSE* of the model. Unfortunately the *CR* shows that the performance of both of the evolved models in the plotted window are optimistic for almost all the variables. The only exceptions are the yaw velocity and angle, the prediction errors of which have a *CR* close to one and are therefore more representative of the performance of the real model. This is not a positive sign since it indicates that across the dataset (on average) we should expect worse

²⁶Best and median are determined in terms of the metric $RMSE_{total}$ computed as explained in Section 4.1.1.

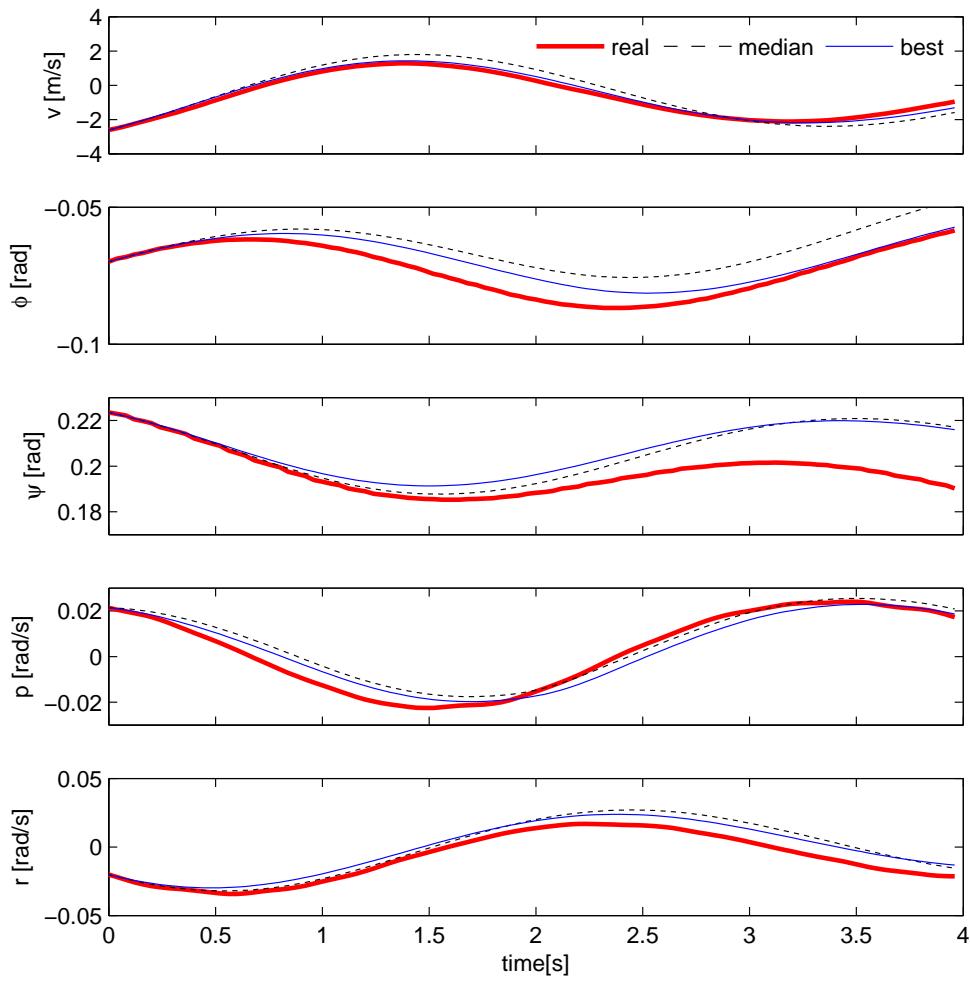


Figure 5.17: Proof of match: the state progress predicted by the *ATTASAccGPbest* and median *ATTASAccGPmedian* models versus the measured real state.

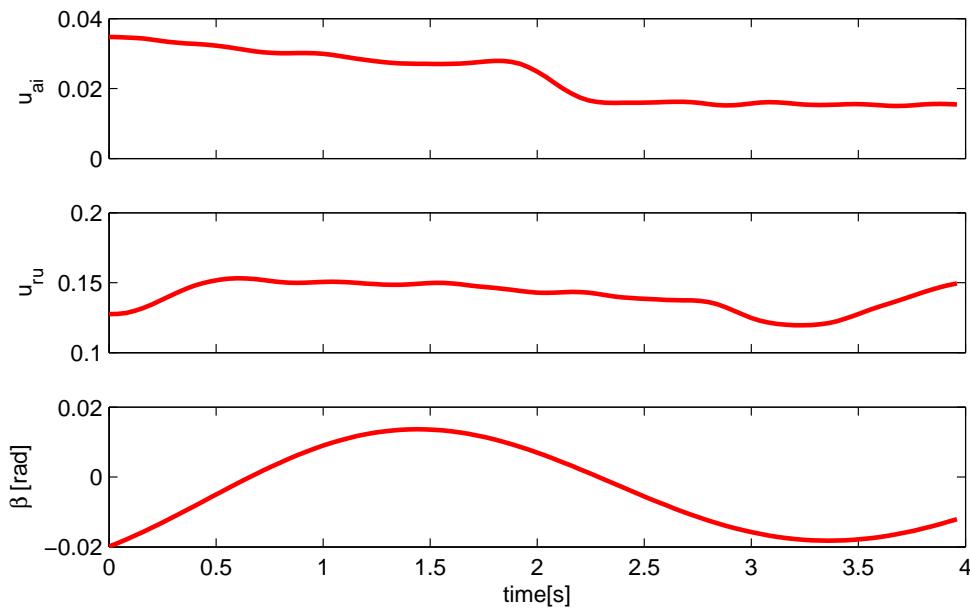


Figure 5.18: Control inputs given to the *ATTASAccGPbest* and *ATTASAccGPmedian* models during the time window in Figure 5.17.

	Best Model		Median Model	
	RMSE	CR	RMSE	CR
v	0.16 m/s	0.33	0.21 m/s	0.34
ϕ	0.005 rad	0.37	0.001 rad	0.07
ψ	0.012 rad	0.98	0.013 rad	0.97
p	0.003 rad/s	0.22	0.001 rad/s	0.14
r	0.007 rad/s	0.75	0.007 rad/s	0.64
<i>total</i>	0.16		0.21	

Table 5.2: Prediction error and *CR* of the *ATTASAccGPbest* and the *ATTASAccGPmedian* models computed on the window of data plotted in Figure 5.17.

	Best Model			
	RMSEm	RMSEsd		RAEm
v	0.48	0.50	m/s	0.17
ϕ	0.014	0.02	rad	0.26
ψ	0.012	0.01	rad	0.42
p	0.013	0.020	rad/s	0.10
r	0.009	0.008	rad/s	0.31
<i>total</i>	0.48	0.50		

Table 5.3: Prediction error of the *ATTASGPbest* model computed on the validation dataset.

performance than those exhibited in this test window.

In the same way as we measured the performance of the ATTAS model based on first principles in Chapter 4, we compute the *RMSEm* and *RAE* metrics over the 14 consecutive windows into which the validation dataset is split²⁷. These metrics provide us with an unbiased measurement of the models' abilities and are reported in Tables 5.3 and 5.4.

Both models have low *RAEm* in most of the dimensions, confirming that each of the models is considerably better than a *constant model*, with the *ATTASGPbest* model being obviously the better of the two. For both models the *RAEm* shows the yaw velocity and angle to be the most difficult quantities to predict, while, angular velocity p is the easiest. Comparing the two models, we see that what makes the *ATTASGPbest* a better model is essentially its ability to better predict the lateral velocity v ; in the remaining dimensions the two models are very similar.

The *ATTASGPbest* model is not simply better at predicting the lateral velocity but it is also more consistent across the various windows of the dataset, as shown by its

²⁷Details of the computation of the *RMSEm* and *RAEm* metrics are reported in Section 4.1.1

	Median Model			
	RMSEm	RMSEsd		RAEm
v	0.61	0.71	m/s	0.22
ϕ	0.01	0.01	rad	0.17
ψ	0.01	0.01	rad	0.43
p	0.008	0.007	rad/s	0.073
r	0.011	0.011	rad/s	0.36
<i>total</i>	0.61	0.71		

Table 5.4: Prediction error of the *ATTASGPmedian* model computed on the validation dataset.

lower $RMSEsd_v$. For the remaining dimensions the two models exhibit similar levels of $RMSEsd$, suggesting that for the model produced by our algorithm, performance levels and performance consistency are related.

We now look at the $RMSEm$ and $RAEm$ across the 30 independently produced models. In Table 5.5 we see that, all 30 runs produced fairly similar models (in terms of prediction error), with a standard deviation of 0.14 equal to 22% of the mean $RMSEm_{total}$.

Regarding the $RAEm$ (Table 5.6), we see instead that the low values of $RAEm_p$ and $RAEm_\phi$ show that in general all the evolved models have good prediction abilities for the roll dynamics. The lateral dynamics v has a similar quality, but across all the models, the

	RMSEm _{total}
best	0.48
median	0.61
mean	0.63
s.d.	0.14 (22% of mean)
diverging	0

Table 5.5: Prediction error ($RMSEm_{total}$) of the set of 30 models obtained from 30 independent runs of the coevolutionary algorithm.

	RAEm mean	RAEm sd
v	0.23	0.05
ϕ	0.21	0.05
ψ	0.43	0.05
p	0.09	0.01
r	0.36	0.05
diverging	0	

Table 5.6: $RAEm$ of the set of models obtained by 30 independent runs of the coevolutionary algorithm. The yaw dynamics (r and ψ) appears as the most difficult to predict.

yaw dynamics (ψ and r) is the most difficult to predict. The limited standard deviation in all the dimensions shows that these characteristics are shared by all the models produced.

Lastly, let us compare the performance of the models obtained with our automatic method and limited domain knowledge against the one produced in Section 4.1.1 using the understanding of aircraft dynamics. The overall measure of error $RMSE_{total}$ achieved by the automatically obtained models, is somewhat higher than that of the *ATTASFP* model (0.48 versus 0.40). Even looking at the error dimension by dimension, the model based on domain knowledge consistently scores a lower error, confirming that the model based on first principles is in all respect of higher quality. Looking at the $RAEm$ we can see that the largest differences in $RAEm$ between the *ATTASFP* and *ATTASGPbest* models occur for the yaw velocity r and the heading angle ψ . This confirms our expectations since we have seen that the yaw dynamic is not easy to model. However when compared with the *ATTASFP* model the, *ATTASGPbest* model also shows poor performance in capturing the evolution of the roll angle ϕ .

5.5.3 Analysis of the Equations

When designing our coevolutionary algorithm in Section 5.3, we highlighted that one of the advantages of symbolic regression is the possibility of examining the model expression obtained. This is not necessarily a straightforward task, since the expressions produced can be quite complex, but it is worth trying to analyze at least some of the models. In the case of the ATTAS aircraft it is helpful that we have the expression of the *ATTASFP* model against which to compare the expressions produced automatically.

In Table 5.7 we see the expression obtained for the median and best models. The expressions have been automatically simplified²⁸ and the numerical constants have been truncated to make the expressions easier to read. Is immediately clear that for both of the models the equations of α_x are very similar in structure and parameters to the first principle model. The *ATTASGPbest* includes some superfluous terms ($-0.56u_{ru}\beta - 0.04\psi$) while the *ATTASGPmedian* model lacks the bias term and includes a term $3.9\psi r$ instead of $0.88r$. After investigating the data present in the training set, we noticed that the angle ψ has only relatively small variations around the mean value of 0.22. We then tried substituting

²⁸The automatic simplification is simply an algebraic rearrangement of the expression and was carried out using a combination of the functions `ExpandAll[]` and `FullSimplify[]` provided by the commercial package Mathematica®.

a_y
best: $8.25\phi - \beta\phi + r - (-0.24 + \beta + 2u_{ru-1}(-0.63 + \phi)(-0.08 + \psi))(-0.78 + 3.32\phi - v)$
median: $(-4.81(-1.78u_{th} + u))/(3.16 - r - u_{th} - 2u_{ba})$
first principles: $\phi(9.53 + \beta - \psi) - (-\beta + 0.83 - u_{ru} - u_{ru}^2 + v)/(4.38 + p^2 - (\psi)/(0.20 + \psi) + 1.18r)$
α_x
best: $-2.32p + 1.00r - 1.32u_{ai} + 0.091u_{ru} - 3.55\beta + 0.023 - 0.56u_{ru}\beta - 0.04\psi$
median: $\alpha_x = -2.31p + 3.9\psi r - 1.31u_{ai} + 0.11u_{ru} - 3.90\beta$
first principles: $\alpha_x = -2.21p + 0.88r - 1.33u_{ai} + 0.094u_{ru} - 3.84\beta + 0.0008$
α_z
best: $0.032 - 0.02u_{ai} - 0.1p - 0.12u_{ru} - 0.02u_{ru}^2 - 0.02pu_{ru} + 4.23\phi + r + r/\psi - v$
median: $(\psi(-0.40 + 1.31u_{ru} + 0.31p + 1.97r + \beta(-1.17 - \beta + u_{ru} + p - v) - 0.31v))/(-3.53 + 3.00\psi + r)$
first principles: $-0.17p - 0.16r - 0.04u_{ai} - 0.12u_{ru} + 2.95\beta + 0.003$

Table 5.7: Expressions obtained automatically for the *ATTASGPbest* and *ATTASGPmedian* models. The expressions of the parametric model *ATTASFP* are also reported to help the comparison.

the value 0.22 in place of ψ in the expression $3.9\psi r$. This returns the expression $0.86r$ which is remarkably close to the term $0.88r$ of the *ATTASFP* model. In effect, since the angle ψ does not change too much in the dataset, the algorithm is using it as a proxy for a constant. This leads us to think that the amount of information present in the dataset is not actually sufficient to differentiate the variable ψ from a constant.

Unfortunately, for the expressions of a_y and α_z any similarities to the first principle expressions are not so clear. Since all the state variables were made available in the terminal set for the genetic programming, is not surprising to see that some of the expressions use the angles ϕ and ψ and the linear velocity v in addition to the inputs and variables used in the *ATTASFP* model. In conjunction with the presence of fractional terms, this leads to complex expressions in which it is difficult to recognize the form of the equation of the models based on first principles.

Is worth noting that the yaw dynamic, for which the obtained models have a structure

similar to that of the first principle models, is also the dimension that was associated with the smallest $RAEm$ in Table 5.6. There is therefore evidence of a link between well structured and meaningful models, and model accuracy.

5.5.4 Conclusion

The application of our novel coevolutionary algorithm to the modelling of the ATTAS aircraft has shown that the algorithm is capable of reliably producing a model of the system. Both the $RMSE_{total}$ and the $RAEm$ metrics reveal similar performance across the 30 independent runs of the algorithm. However, in pure performance terms, the automatically produced models are not as good as the model based on first principles.

In trying to understand what could lie behind the inability of the coevolutionary process to produce very good models, we need to remind ourselves of one of the fundamental requirements of the modelling process identified in Section 2.2.3, namely the fact that during data collection all the parts of the system dynamic envelope that are of interest need to be sampled. If this is not the case the model will at best only represent a fraction of the system dynamic envelope. Establishing whether the lower performance of the evolved models is due to a lack of informative data or to poor search abilities in our coevolutionary algorithm, is not at all straightforward. Certainly, as shown by the very good performance on the equation of the roll dynamics (α_x) the algorithm is able to reliably produce linear models of the type required for all the dimensions of the ATTAS dynamics. This leads us to think that, while the information present in the training dataset is sufficient to identify the full system once its structure is defined, which is the case when training the parametric models, the information necessary to evolve the system structure is actually not available. In particular, the data may not present enough information to specify the structure of the equations for the roll and lateral dynamics. The fact that in one of the models the genetic programming uses a state variable (ψ) as a proxy for a constant, is clear evidence that the changes in that variable within the dataset are not informative enough.

Unfortunately the amount of experimental data available for the ATTAS platform is very limited, and we are not in the position of being able to acquire more data to confirm our understanding experimentally. In the following sections we will have the opportunity to investigate our coevolutionary algorithm thoroughly on more complex platforms, and to test its abilities. The ATTAS study can be regarded as a paramount example of how

the experimental data plays a key role in modelling, especially when platform specific knowledge is not employed, as in our work.

5.6 Automatic Modelling of the Toy Car

As in the case of the ATTAS aircraft, the toy car also requires the modelling of three dimensions; as described in Section 3.3.3, they are the longitudinal, lateral and yaw dynamics. The coevolutionary algorithm will be applied to generate a model of the form of equation 3.13, restated here for convenience:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (5.8)$$

We therefore need to evolve three expressions that predict the accelerations a_x , a_y and α_z respectively.

As with the *MLP* models of the car, the dataset `datasetcar_6` is used for the training, and the same state and control vectors ($\mathbf{x} = [u, v, r]^T$ and $\mathbf{u} = [u_{th-\delta_{th}}, u_{st-\delta_{th}}, u_{ba-\delta_{ba}}]^T$ respectively)²⁹ are used as the inputs to the model expression tree. The maximum allowed delay for the control inputs is again set to 5 time steps, while the size of the test windows is set to 300 samples³⁰.

The coevolutionary algorithm was run independently 30 times in order to allow judgment of its repeatability; each run was stopped at the 20 epochs mark.

5.6.1 Convergence

As usual, to produce an unbiased performance measure we compute the performance of the model on a validation dataset. We are interested in the performance of each individual equation in the model, as well as the performance of the whole model.

As for the ATTAS aircraft, we start by taking each of the equations in the evolved models, and computing its *RMSEm* using all the windows of the validation dataset as explained in Section 2.2.5. Since we are interested in evaluating each equation independently,

²⁹As explained in Section 5.3.3 we directly evolve the delay parameter for each of the control inputs. Therefore the input vector \mathbf{u} only contains the version of each control signal associated to the delay chosen. This is different from what we have used in Sections 4.2.2 and 4.2.2 for the *MLP* models where all the possible delayed version of the control inputs were contained in \mathbf{u} .

³⁰The reader will recall that the reasons behind this choice of parameters were considered in Sections 4.2.2 and 4.2.1

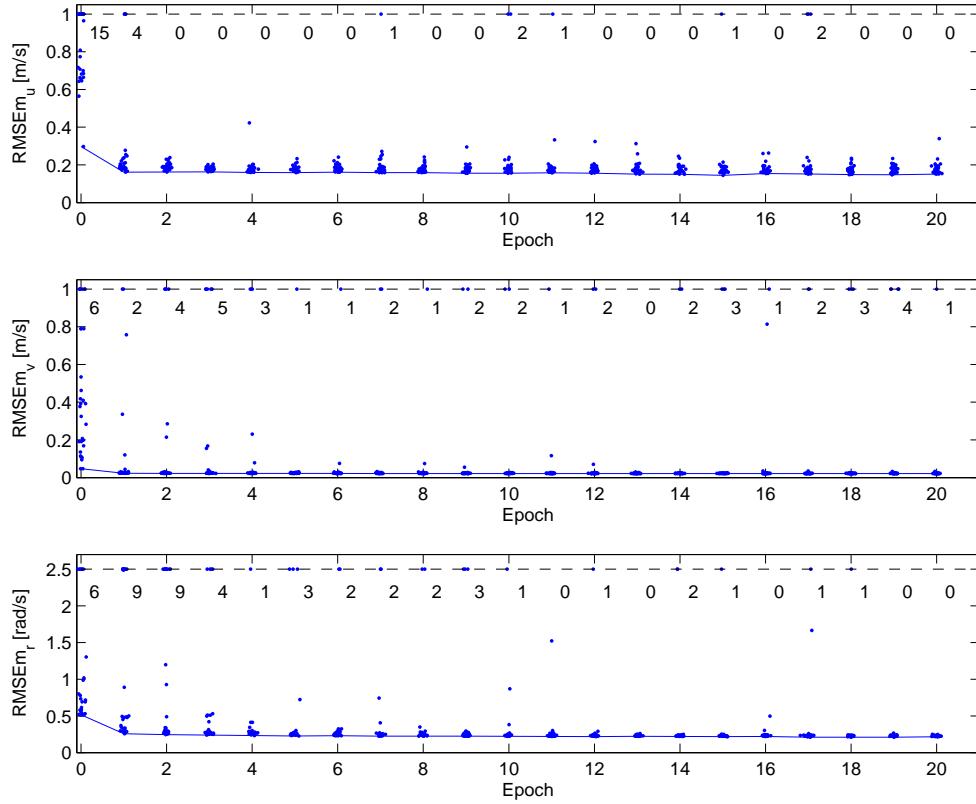


Figure 5.19: The error on single dimensions of the best model during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. The vertical scale of the plot is chosen proportionally to the standard deviation of the signal being predicted to allow for direct visual comparison between the three plots. The value of the points that exceed the maximum of the scale has been limited to the level indicated by the dashed line, and the figures at the top of each plot indicate how many points have been limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

only the equation in question is used for predicting the acceleration associated with it, while values taken from the dataset are used for the remaining accelerations. In Figure 5.19 we have plotted the progress of the model error during the 20 epochs of model evolution for each of the three equations in the model. As with previous models, the performances were evaluated in terms of mean $RMSEm$ (i.e. $RMSEm_u$, $RMSEm_v$, $RMSEm_r$; see 4.1.1 for details on the computation of the $RMSEm$ metric.). As expected the initially randomly generated models have very different performance levels.

For all three dimensions, the majority of the models converge in the initial epochs, but improvements in performance, although sometimes small, are also taking place at later times in the run³¹. In the case of the longitudinal and lateral dynamics (u and v

³¹We define a model as not converging when its error level is much larger than the one of the other

respectively), even in the 20th epoch there are still a small number of models that fail to converge. We also see instances for some models in which the performance worsens during the progress of the run instead of improving as might be expected. As discussed in Section 5.5, this is result of the fact that, while at the end of each epoch the models are selected according to their performance on the history of models, here they are being evaluated over the whole validation dataset. As a consequence models can encounter portions of data on the unseen dataset that produce large errors.

To evaluate the progress of the complete model during the evolution runs, for each run and each epoch we aggregate the best equation for each of the dimensions into a 3DoF model and we test this model on a set of independent windows that covers the whole validation dataset. The performance is measured in terms of the familiar $RMSE_{m_{total}}$ that we defined in equation 4.10.

The total error of the models plotted in Figure 5.20 shows the same pattern of convergence throughout the coevolution as is exhibited by the single dimension models. The first epoch is crucial for the majority of the models, but the remaining models converge

evolved model. This is easy to recognize from the error plots since the error of such a model does not belong to the cluster formed by all other models.

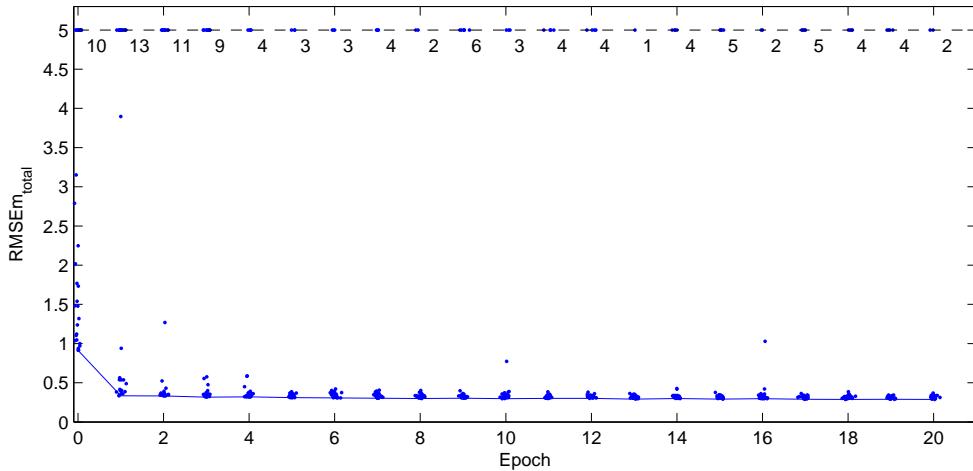


Figure 5.20: Total model error ($RMSE_{m_{total}}$) during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. To focus on the most interesting part of the plot, the value of some of the points has been limited to the level indicated by the dashed line. The the numbers at the top of each graph indicate how many points have been limited. The position of the dashed line was chosen as a compromise between providing a good picture of the distribution of models performances and containing the number of points limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

to similar performance during the following epochs. Generally speaking the number of full models not converging is larger than the number of single dimension model not converging. This is to be expected, in view of the dynamic coupling that is present between dimensions. The errors that are inevitably present in each of the single equations will propagate to the remaining equations, and may eventually prevent the full model from converging.

Is worth noting that this issue is intimately related to our design choice of modelling each of the single dimensions independently (see Section 5.3.1). Perhaps evolving all the equations of a model simultaneously could be a way of highlighting the situations in which error coupling leads to poor models; however, as already discussed, this is unfeasible due to the exponential increase in the size of the search space.

In the case of the toy car, evolving each dimension independently represents a good compromise between performance and computational complexity, since only two out of the 30 independent runs do not converge. More precisely, if we take into account the fact that one of the runs for the v equation did not converge even when tested independently (see Figure 5.19) only one of the models fails to converge due to error coupling.

We now look at the mean size of the expression trees during the evolution runs (Figure 5.21). Small models are the best performers initially; as the coevolution proceeds, the models' expressions increase in complexity. Comparing the progress of tree sizes with

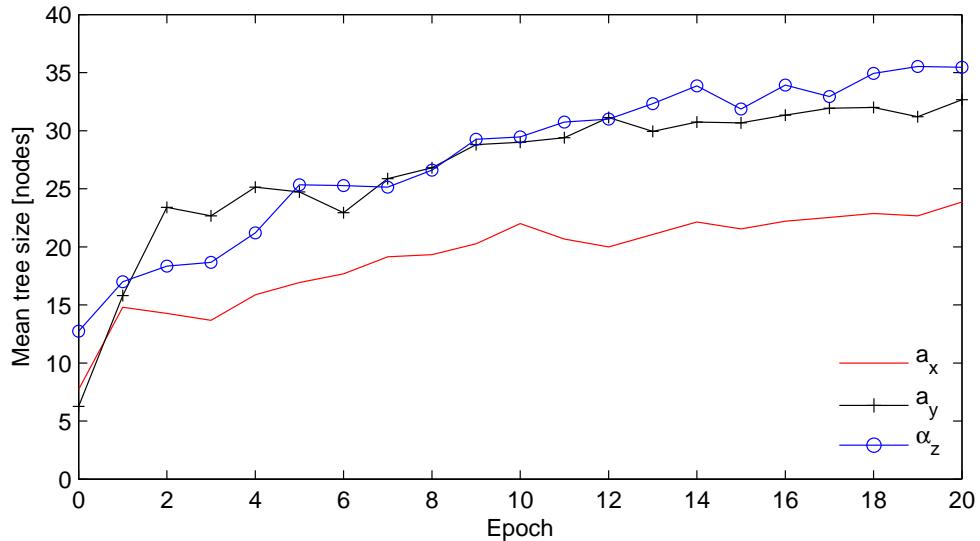


Figure 5.21: Size of the expression trees throughout the 20 coevolution epochs. The mean size of the best models of each coevolution run is shown. On average the expressions for the lateral acceleration a_y are larger, suggesting a more complex dynamic relationship.

the progress of the model error reported in Figure 5.19, we see that the initial growth is correlated with a large improvement in fitness, while the subsequent addition of genetic material has less of an impact on overall performance. Interestingly, the expressions of the forward acceleration (a_x) are in general smaller than the equations of a_y and α_z suggesting that there might be a simpler dynamic relationship for this equation.

5.6.2 Performance Analysis

Plotting the evolved model predictions against the experimentally recorded data is the first step in assessing the model performance, at least qualitatively. We have chosen the best and the median models (we will call them *carAccGPbest* and *carAccGPmedian* respectively) as representative of the 30 evolved models³². In Figure 5.22 we plot their predictions on the same randomly chosen window of data from the validation dataset that was used for the *NN* and *MLP* models (see Sections 4.2.1 and 4.2.2). The predicted state during the window in question is computed as usual by initializing the state to its measured value at time $t = t_{w0}$ and then integrating it forward using the accelerations produced by the three model equations.

At first glance, is clear that both the *carAccGPbest* and *carAccGPmedian* models are capable of good predictions. The forward velocity u exhibits the best prediction, with the models' outputs very close to the experimental data. The u prediction of the *carAccGPbest* and *carAccGPmedian* models are almost indistinguishable. Overall the models approximate the acceleration phase of the car very well, while the deceleration corresponding to the pilot releasing the throttle are marginally slower than the true data.

Again, there are small differences between the *carAccGPmedian* and the *carAccGPbest* models in predicting the rotational velocity r . In this dimension, both models show very good predictions, and as with the forward velocity, they decelerate more slowly than the real car. When analyzing the model equations, we shall see how the prediction of α_z depends directly on u which explains the similar time response behaviour in the two dimensions.

The last of the state variables v is difficult to predict as previously discussed, and this holds even for the genetic programming models. The time interval $t = [6, 8]s$ time interval clearly shows a situation in which the change in lateral velocity does not seem to be correlated with the control inputs nor with the other state variables. However, in that

³²Best and median are determined in terms of the metric $RMSE_{m_{total}}$ computed as explained in Section 4.1.1

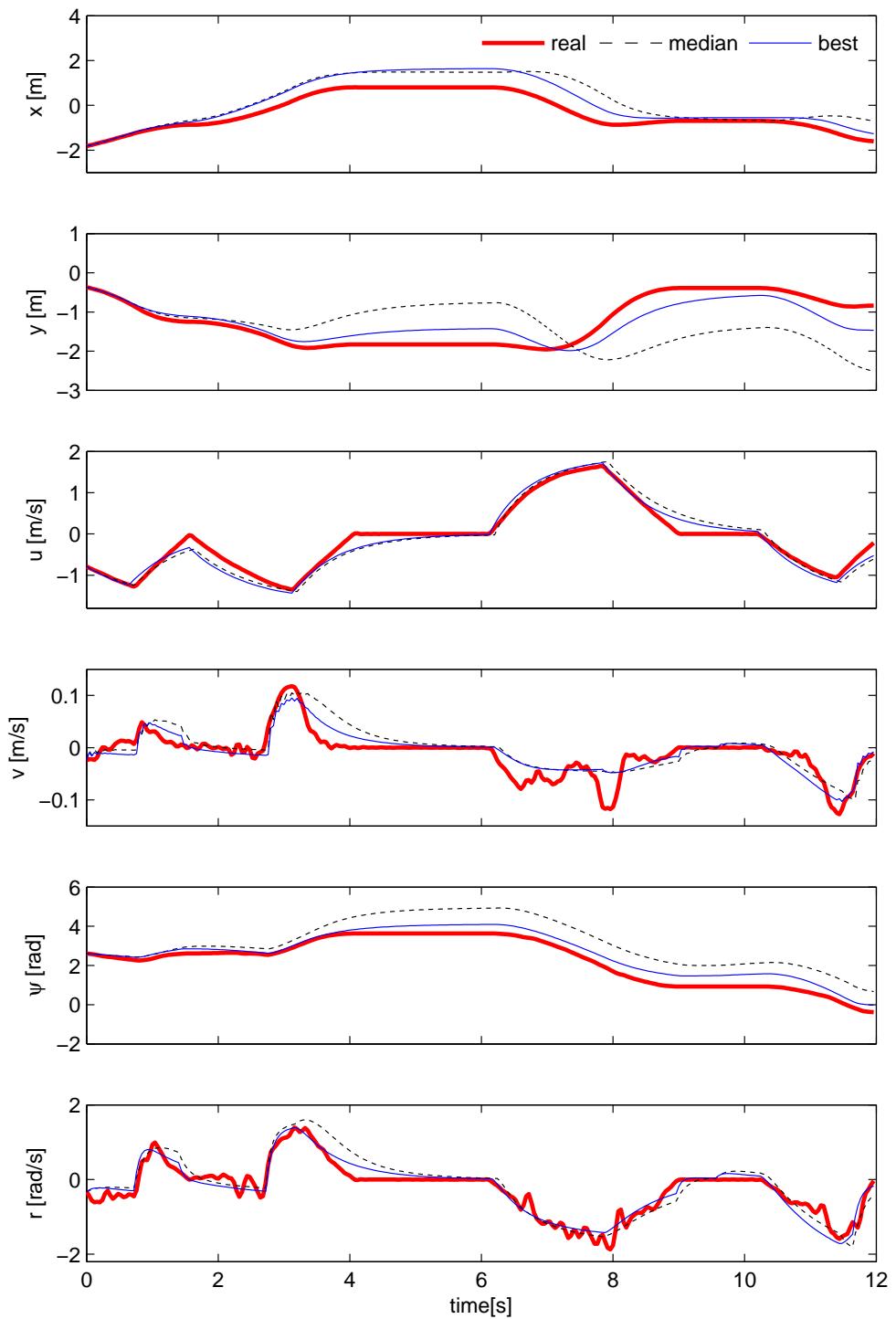


Figure 5.22: Proof of match: the state progress predicted by the *carAccGPbest* and *carAccGPmedian* models versus the measured real state.

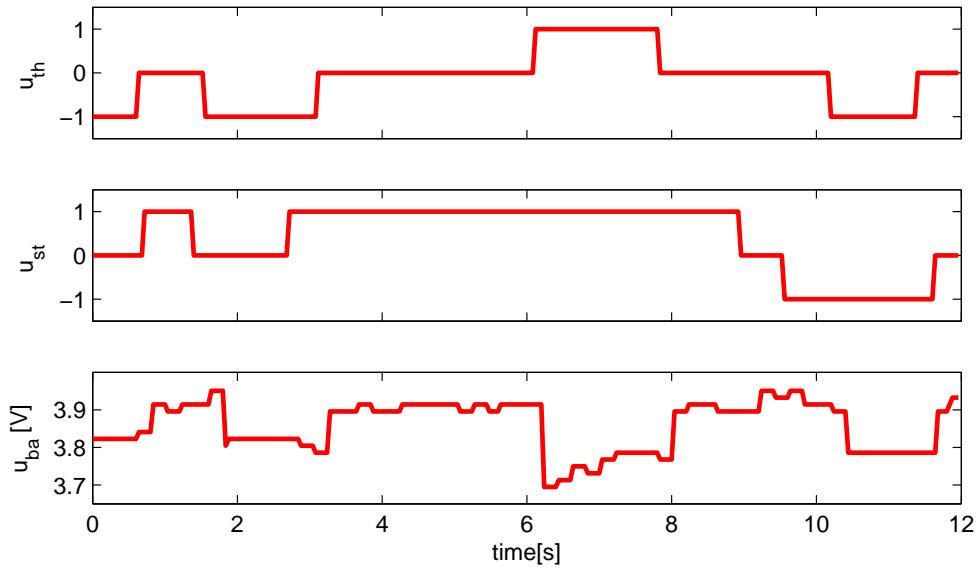


Figure 5.23: Control inputs given to the *carAccGPbest* and *carAccGPmedian* models during the time window in Figure 5.22.

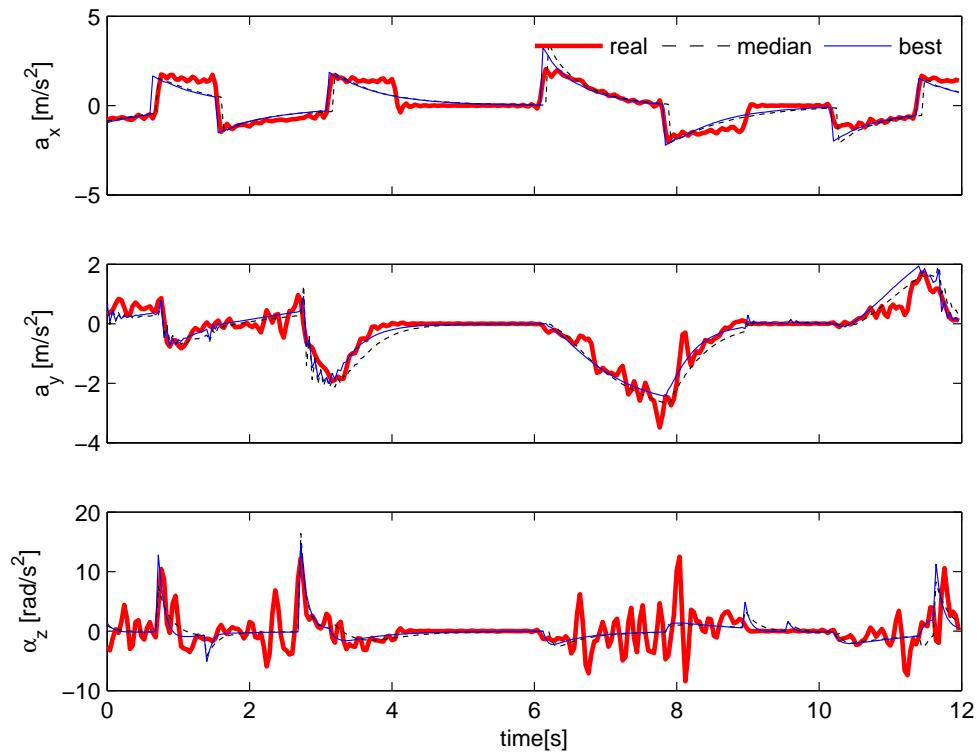


Figure 5.24: Proof of match: the acceleration progress predicted by the *carAccGPbest* and *carAccGPmedian* models versus the acceleration computed from the real data.

interval the car was accelerating up to its maximum speed while performing a turn. It is at least possible that, during this driving phase, the behaviour of the toy car could be highly nonlinear perhaps as result of wheel slippage which can sometimes happen in high speed turns. Without dedicated sensors we cannot expect to be able to predict this type of phenomenon with any precision.

Comparing the prediction of the genetic programming models with the prediction of the MLP based models (seen in Figures 4.11 and 4.16), we see a marked superiority of the models produced with genetic programming. This is particularly true for the lateral velocity v which in the case of the *carVelMLP* model only roughly approximates the true velocity.

Finally we look at the derived variables x , y and ψ obtained by integration of the state. Inevitably the error in such variables between the predicted and the collected data tends to increase as time goes on because of error integration. Even so, it is interesting to note that after 12 seconds of rapidly changing motion over $7.3m^{33}$, we can predict the position of the car with an x and y error of less than $1m$ solely from the control inputs.

The objective of our models is to predict accurately the progress in time of the vehicle's state. However, one of the constraints in designing our algorithm (Section 5.3.1), was to have a model that was also able to produce physically meaningful accelerations. It is therefore of interest to plot the acceleration predicted by the models against that computed by differentiation for at least one of our platforms.

Figure 5.24 shows that the accelerations predicted by both the *carAccGPmedian* and the *carAccGPbest* models are very consistent qualitatively and quantitatively with those obtained from the logged data. This was expected given the good predictions that we have seen for the state. The predictions of the two models are very similar, with the best model having, as expected, a higher accuracy. In the case of a_x for example, the prediction of the *carAccGPbest* model is better aligned in time with the real acceleration, while in the case of a_y the magnitude predicted by *carAccGPbest* is more accurate. The amount of noise present in the real data is very noticeable, especially in the case of the angular acceleration α_z for which is not entirely clear which features are the result of the model dynamics and which are the result of the noise amplified by the numerical differentiation. As a consequence, although at first sight both models appear to neglect part of the dynamics,

³³Computed by integrating the absolute value of the forward velocity u .

it is difficult to know how much of the signal content is in fact predictable.

As usual, we want to have an understanding of how the performance in the chosen window compares with the performance of the model over the whole dataset. We therefore compute the *RMSE*, and also the *CR* metric, for on the predictions made in the selected window (Table 5.8). Both models have a *CR* close to one for the forward velocity u ,

	Best Model		Median Model	
	RMSE	CR	RMSE	CR
u	0.16 m/s	1.03	0.17 m/s	0.95
v	0.02 m/s	0.87	0.03 m/s	1.07
r	0.21 rad/s	0.87	0.35 rad/s	1.08
<i>total</i>	0.26		0.33	

Table 5.8: Prediction error of the *carAccGPbest* and the *carAccGPmedian* models computed on the window of data plotted in Figure 5.22.

leading us to expect good velocity prediction (as seen in the sample window) over the whole validation dataset. The *CR* is higher for the rotational and lateral speeds, highlighting that the randomly chosen window is actually more difficult to predict than the average window in the dataset. This is clearly comforting since it means that on average the evolved models will perform even better than what we have seen for the chosen data window.

Before performing a quantitative evaluation of the models' performance, we manually tested the models. As in the previous chapter, a simple simulator was built by interfacing the models produced by the coevolutionary algorithm with a 2D visualization and a control keyboard. The manual test involved driving the simulated car in the simulated arena in order to subjectively assess its ability to behave qualitatively like the real car. In particular, various combinations of control inputs were tested to see if any of them elicited any behaviour markedly incompatible with the one of the real car.

Both of the genetic programming models appeared to be well behaved, with an ability to speed up and slow down similar to the real car. As with the real car, the top forward speed achievable was higher than the reverse. The turning radius was markedly asymmetrical with the left turning radius smaller than the right one, again in agreement with the behaviour of the real car. Any steering input applied with the car at rest produced no effect, again just like the real car. During the driving test the behaviour of the *carAccGPbest* and the *carAccGPmedian* models was extremely similar and we were unable to notice any difference.

Qualitatively, both evolved models appear to reproduce the measured data quite well, but obviously we need to test them quantitatively. Again we use a validation dataset split into consecutive windows of 300 samples.

The model's state is initialized to its recorded value at the beginning of the data window and the acceleration predictions from the model's equation are then integrated forward in time for the remaining window data points. The $RMSE_m$ and RAE predictions are then computed over all the windows in the validation data. Table 5.9 and 5.10 show the values obtained for the *carAccGPbest* and the *carAccGPmedian* models respectively.

	RMSE_m	RMSE_{sd}		RAE_m
<i>u</i>	0.16	0.03	<i>m/s</i>	0.17
<i>v</i>	0.023	0.009	<i>m/s</i>	0.46
<i>r</i>	0.24	0.07	<i>rad/s</i>	0.30
<i>total</i>	0.29	0.06		

Table 5.9: Prediction error of the *carAccGPbest* model.

	RMSE_m	RMSE_{sd}		RAE_m
<i>u</i>	0.20	0.035	<i>m/s</i>	0.20
<i>v</i>	0.023	0.008	<i>m/s</i>	0.47
<i>r</i>	0.25	0.074	<i>rad/s</i>	0.33
<i>total</i>	0.32	0.06		

Table 5.10: Prediction error of the *carAccGPmedian* model.

The RAE_m metric quantitatively confirms the qualitative analysis that we carried out based on the proof of match plots. The forward velocity u is the dimension that shows the best performance (lowest RAE_m) followed by the rotational and lateral velocities. Again the performances of the two models are very similar with the best model only marginally better.

While analyzing the $RMSE_m$ simply reinforces our understanding of the similarity between the *carAccGPmedian* and the *carAccGPbest* models, the $RMSE_{sd}$ is much more informative. For both models, the standard deviation of the $RMSE_m$ between the various windows in the dataset (see Table 5.11) is low in comparison to the mean error (8.8%), meaning that the models deliver good performance consistently over all windows in the dataset.

The fact that the median and best models have very similar performance is also a strong indicator that our coevolutionary algorithm is reliably able to produce good models.

	RMSEm_{total}
best	0.29
median	0.31
mean	0.32
std	0.02 (6.2% of mean)
diverging	2

Table 5.11: Prediction error ($RMSEm_{total}$) of the set of 30 models obtained from independent runs of the coevolutionary algorithm (diverging models are not included in the computation of the standard deviation).

However, it is worth quantifying this ability by computing the $RMSEm$ and $RAEm$ across the 30 independently produced models. In Table 5.11 we see that, as seen in Figure 5.20, all 30 runs produce fairly similar models (in terms of prediction error), with a standard deviation of 0.048.

The mean $RAEm$ computed across the set of models (see Table 5.12) confirms the qualitative analysis based on Figures 5.22 and 5.23. In particular, u is confirmed as the variable for which the models produce the most accurate predictions, while v is the weakest point. The low standard deviation shows that this pattern is shared by the majority of the evolved models.

	RAEm mean	RAEm sd
u	0.19	0.025
v	0.48	0.010
r	0.33	0.016
diverging	2	

Table 5.12: $RAEm$ of the set of models obtained by 30 independent runs of the coevolutionary algorithm (diverging models are not included in the computation of the standard deviation). The lateral velocity v has the largest $RAEm$ confirming the lateral dynamics as the most difficult to predict.

Lastly we compare the performance of the model produced with genetic programming with the best model obtained for the toy car in Chapter 4, namely the *carVelMLP* model (Table 4.12). While the two models have very similar forward velocity performance, the error in rotational velocity is lower for both the *carAccGPbest* and *carAccGPmedian* models. The prediction ability of the evolved models shows an even more marked improvement in the prediction of the lateral velocity.

Is interesting to notice that, while in the case of the *carVelMLP* model $RAEm_v$ is higher than one, the $RAEm_u$ for the evolved models is 0.46. This means that, while the

lateral velocity predictions of the *carVelMLP* model are worse than a *constant* model, this is clearly not the case for the models produced with our coevolutionary method.

5.6.3 Analysis of the Equations

In this section we look at the equations of the models produced with genetic programming to see if we can get any insight into the models themselves. With the ATTAS aircraft we had a set of first principle equations to check our model against, but with the toy car (as well as with our quadrotors), this is not the case.

Analyzing the automatically produced equations involves inspecting all the equations obtained from the 30 coevolution runs, and looking for any structure that appears in several models. The structure of the obtained models is not always easy to interpret, especially when parts of an equation are divided by other parts. To aid our analysis the expressions produced were first automatically simplified²⁸. While most of the simplified expressions have a polynomial form which makes for easier comparison, some expressions do not fit this form (e.g. the median model for a_x in Table 5.13). When appropriate we substituted some terms within the equations with their respective series expansion in order to obtain a polynomial expression approximately equivalent to that produced by genetic programming³⁴.

These straightforward manipulations were sufficient to show that a large majority of the models obtained from the 30 independent runs of the algorithm indeed share the same general structure; we call such a structure the *general form*. More precisely, we define a term as part of the *general form* if it is present at least in 70% of the models. In this way we avoid including into the *general form* terms that appear in only a small number of models, and are therefore probably not essential for modelling the dynamics. The few models that disagree with this *general form* come either from equations in which some of the terms of the *general form* are missing, or from equations that we were simply unable to reduce to the *general form*.

The best and median models along with the *general form* obtained for each of the

³⁴In particular we often substituted fractional terms appearing in the model by the Taylor's series expansion of $\frac{1}{1-x}$:

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n \quad |x| < 1$$

truncated to the 3rd order.

dimensions of the car model are shown in Tables 5.13-5.15³⁵.

For all three dimensions, good levels of agreement are found across the evolved models, with at least 70% of the models sharing a common equation structure. Looking at the *general form* of the equations of a_x we see how the forward acceleration is dependent on the throttle input and on the forward velocity u . While the dependence on throttle is expected (we would certainly agree that a throttle command makes the car accelerate forward), the model shows that this dependency is second order. The negative factor $C_u^{a_x}$ multiplied by u has the role of a linear friction coefficient, which is also expected.

The reason for the nonlinear combination of $uu_{st-\delta_{st}}$ is less obvious. Since the term is multiplied by the negative coefficient $C_{uu_{st-\delta_{st}}}^{a_x}$, its effect is to reduce the acceleration when the car is travelling at a speed u and throttle is applied, and to increase the acceleration when the throttle is released. Its function is therefore to change the value of the linear friction depending on whether or not a throttle command is applied. This leads us to think that this term may model the mechanical friction taking place in the drive and motor system which obviously depends on whether the motor is powered or not.

Interestingly, both the *carAccGPmedian* and *carAccGPbest* model also show a dependency of the forward acceleration on the battery level, a dependency that is not difficult to understand but that is not shared by many other models.

In Tables 5.13-5.15 we also report the mean and standard deviation obtained from the 22 models for which we found agreement on the *general form* of the a_x equation. For some of the parameters (i.e. $C_{th}^{a_x}$ and $C_{thu}^{a_x}$) the standard deviation is quite large, showing that in some of the models the additional terms present in addition to those of the standard form are having a non-negligible effect.

The rotational acceleration α_z is directly dependent on the steering input u_{ya} multiplied by the forward speed u , since obviously the car would not turn unless it is moving. This motion is damped proportionally to the rotational velocity r (see term $C_r^{\alpha_z}r$). The tendency of the car not to move in a straight line even when u_{st} is zero is captured by the term $C_u^{\alpha_z}u$, while the term $C_{ru}^{\alpha_z}ru$ produces the asymmetry in turning radius since its effect is positive or negative depending on the rotational speed r . Interpreting the remaining term $C_{uv}^{\alpha_z}$ is more difficult, but we can speculate that this could be a way of accounting for the fast rotation associated with large lateral velocities, perhaps as a result of skidding.

³⁵To aid readability the expressions of the best and median models reported have been automatically simplified and the constant terms have been truncated, but no substitutions of terms has been carried out.

a_x
best:
$-0.59(-u_{th}th + 0.57u)(4.31 + 1.11u_{th} - 0.11v + 2.08uu_{ba}/u_{th})$
median:
$(-4.81(-1.78u_{th} + u))/(3.16 - r - u_{th} - 2u_{ba})$
general form:
$C_{th}^{ax}u_{th-\delta_{th}} + C_{th^2}^{ax}u_{th-\delta_{th}}^2 + C_u^{ax}u + C_{thu}^{ax}u_{th-\delta_{th}}u$
$C_{th}^{ax} = 2.6 \pm 0.24 (9.2\%)^a$
$C_{th^2}^{ax} = 0.7 \pm 0.52 (74.2\%)$
$C_u^{ax} = -1.5 \pm 0.19 (12.6\%)$
$C_{thu}^{ax} = -0.4 \pm 0.12 (30\%)$
delays:
$\delta_{th} = 0 \ 40.9\% (9/22)^b \quad \delta_{th} = 1 \ 50\% (11/22) \quad \delta_{th} = 2 \ 9.1\% (2/22)$
agreement:
$73.3\% \ (22/30)^c$

Table 5.13: Structure of the evolved toy car models: a_x equations.

^aMean and standard deviation of the parameters of the *general form* computed across all the agreeing models. The relative standard deviation ([59]) is shown in parentheses.

^bPercentage of models among the ones that agree with the *general form* having the same delay. The actual number is shown in parentheses.

^cPercentage of models that agree with the *general form* (we define a term as part of the *general form* if it is present at least in 70% of the models). The actual number (out of 30) is shown in parentheses.

a_y
best:
$2r - 39.18v + 4.62uv + r(-0.43 + 1.57u - 0.27u_{ya})/(2.67 + r(2.45 + 1.94r))$
median:
$r(1.80 + 0.42u) - 0.21u(1 + u) - 42.48v$
general form:
$C_r^{ay}r + C_u^{ay}ru + C_v^{ay}v$
$C_r^{ay} = 1.7 \pm 0.31 (18.2\%)$
$C_u^{ay} = 0.4 \pm 0.11 (27.5\%)$
$C_v^{ay} = -40 \pm 5.8 (14.5\%)$
agreement:
$87\% \ (26/30)$

Table 5.14: Structure of the evolved toy car models: a_y equations.

α_z	
best:	$r + 2u - ru + 4.85uv + (-11.17 - u + v - u_{st})(r + u_{st})$
median:	$u + (-3.05r - 0.20u(-17.11 + 3.22u_{th})(v - u_{ya}))(4 + u - 0.52r/u_{st})$
general form:	$C_r^{\alpha_z}r + C_u^{\alpha_z}u + C_{ru}^{\alpha_z}ru + C_{uv}^{\alpha_z}uv + C_{u\bar{u}_{st-\delta_{st}}}^{\alpha_z}u\bar{u}_{ya-\delta_{ya}}$
	$C_r^{\alpha_z} = -10 \pm 5.00$ (50%)
	$C_u^{\alpha_z} = 1.5 \pm 0.72$ (0.48%)
	$C_{ru}^{\alpha_z} = -1.8 \pm 1.2$ (66.7%)
	$C_{uv}^{\alpha_z} = 7 \pm 4.02$ (57.2%)
	$C_{u\bar{u}_{st}}^{\alpha_z} = -13 \pm 1.33$ (10.2%)
delays:	$\delta_{ya} = 0$ 95.2% (20/21) $\delta_{st} = 1$ 4.8% (1/21)
agreement:	70% (21/30)

Table 5.15: Structure of the evolved toy car models: α_z equations.

Only the primary term $C_{u\bar{u}_{st}}^{\alpha_z}$ has a value clearly shared by most of the models; the remaining parameters are much more variable.

The *general form* of the lateral acceleration contains only three terms. We interpret the rotational velocity r as the driving input as the lateral acceleration is obviously connected to the rotational velocity since both are manifestations of centrifugal force. The term $C_v^{a_y}v$ provides a damping effect, while for $C_{ru}^{a_y}ru$ we can apply the same interpretation given in the case of the rotational acceleration, that of producing asymmetry in the car's behaviour.

Reasonable agreement on parameter values exists among the 26 models presenting the *general form* for a_y .

5.6.4 Conclusions

The application of our coevolutionary method to the car platform confirmed our understanding that the most critical part of the dynamics in this platform is the lateral dynamics. The nonlinear phenomena of friction and skidding are possibly and probably at the root of the problem, and our automatic method proved able to improve on the results obtained by more standard techniques. In this respect we have to emphasise that in the literature the lateral dynamics is generally neglected ([7, 222]), both because it is difficult to model, and because it is less relevant than the forward and rotational dynamics.

Using genetic programming we reliably produced good models of the toy car and which

have higher accuracy than the ones obtained with more conventional techniques. In addition (thanks to the relative transparency of the equations produced) we were able to gain insight into the factors affecting the dynamics of the car. In general a good proportion of the models showed agreement on a *general form* of the model equation, but due to the stochastic nature of the techniques that we are using, some models were difficult to interpret, and even some models that shared the *general form* had residual terms which are difficult to understand.

5.7 Automatic Modelling of the X3D Quadrotor

In this section we apply the same coevolutionary algorithm already tested with the ATTAS and toy car platforms to a more challenging vehicle, both for the number of dimensions and for the dynamic coupling between them; the X3D quadrotor.

All the degrees of freedom are relevant in the case of a quadrotor, so our modelling method will produce equations to predict each of the linear and angular accelerations (i.e. $a_x, a_y, a_z, \alpha_x, \alpha_y$ and α_z).

The input to the model consists of the full 6DoF state \mathbf{x} plus the input \mathbf{u} ; we report both here for completeness:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T$$

$$\mathbf{u} = [u_{th-\delta_{th}}, u_{ya-\delta_{th}}, u_{rl-\delta_{rl}}, u_{pt-\delta_{pt}}, u_{ba-\delta_{ba}}]^T.$$

Even in the case of evolving X3D models, we followed the reasoning presented in Section 4.2.2 and used a maximum allowed input delay of 5 time steps.

To allow meaningful comparison of the results with the models based on *MLP* networks (Section 4.2.2) the same length $T = 300$ was used for both the test and training windows.

5.7.1 Convergence

Again, we look at the prediction error of the best models during the 20 epochs that constitute each of the 30 coevolution runs. The six equations of the model are independently evolved, so it is interesting first to look at the progress during evolution of the best model for each of the single equations.

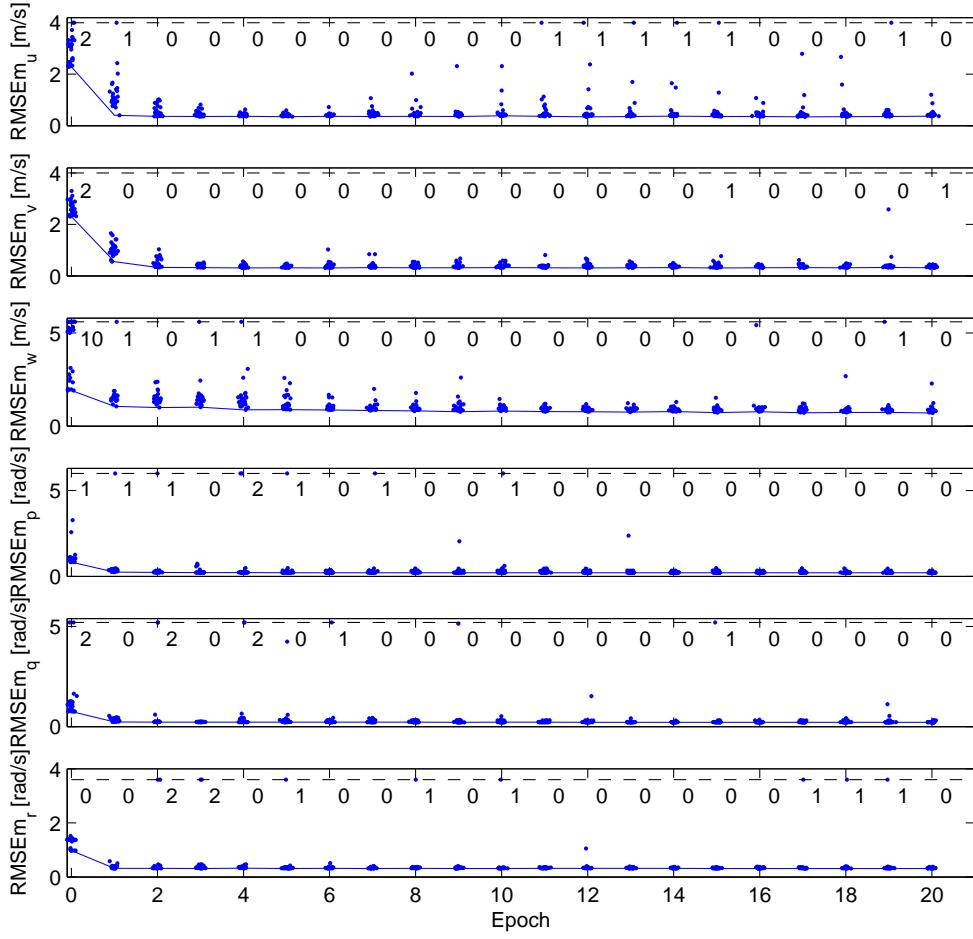


Figure 5.25: The error on single dimensions of the best model during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. The vertical scale of the plot is chosen proportionally to the standard deviation of the signal being predicted to allow for direct visual comparison between the three plots. The value of the points that exceed the maximum of the scale has been limited to the level indicated by the dashed line, and the figures at the top of each plot indicate how many points have been limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

For each run and for each epoch of coevolution, the $RMSEm$ computed from the prediction made over the windows of the validation dataset is used as indication of the equation's performance. The acceleration predicted by the equation being tested is integrated forward in time along with the remaining accelerations (the remaining five dimensions) taken from the training dataset.

For each dimension the progress of the $RMSEm$ for the best model of each of the 30 coevolution run is shown in Figure 5.25.

For all dimensions, the first few epochs are critical for the convergence of the models; in

fact the majority of them converge to a low error in the first two epochs, but in some runs a few more epochs are necessary. The initial population of models is generated randomly so differences between runs have to be expected. There are noticeable differences between the three angular and linear dimensions. The former are distinguished by a larger number of good models in the initial population, but also by a faster convergence. Additionally, the angular accelerations show a smaller variation in performance both during evolution and in the final population.

For each run and each epoch we can combine the single dimensional equations to form a complete 6DoF model. In this case, the measure of performance of the models becomes the $RMSE_{m_{total}}$ metric (defined in equation 4.10), which is computed over the predicted state obtained by integration of the six body frame accelerations. In Figure 5.26 we plot the progress of the $RMSE_{m_{total}}$ during the 30 coevolutionary runs.

Convergence of the full model is much slower than that of each individual equation. Interestingly, by comparing Figure 5.26 with Figure 5.25 we can see that improvements in single dimension performance which are not so obvious after the initial epochs have considerable impact on the overall quality of the model. This clearly emphasises that a

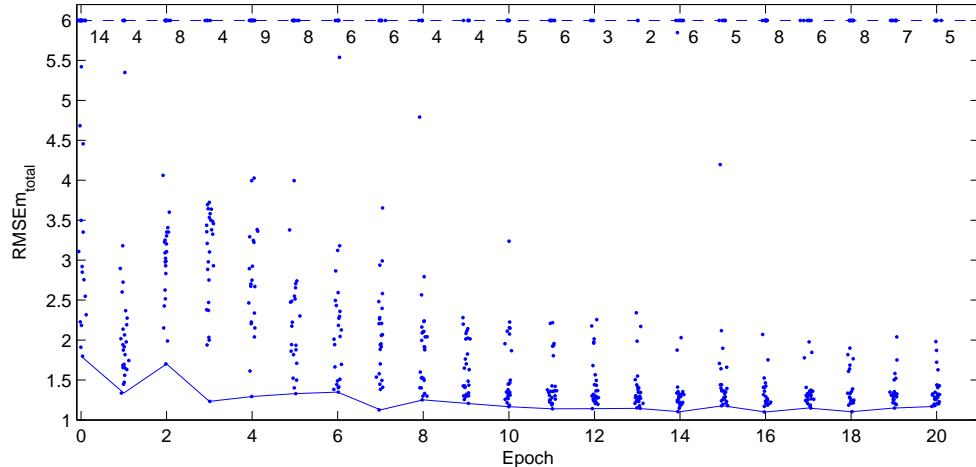


Figure 5.26: Total model error ($RMSE_{m_{total}}$) during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. To focus on the most interesting part of the plot, the value of some of the points has been limited to the level indicated by the dashed line. The numbers at the top of each graph indicate how many points have been limited. The position of the dashed line was chosen as a compromise between providing a good picture of the distribution of models performances and containing the number of points limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

smaller number of epochs would not be sufficient in general to reliably produce convergence, but more importantly it shows the inherent dependency between dimensions present in the dynamics of this platform.

Although only one of the single dimension equations (predicting the lateral acceleration a_y) was shown not to be converging at the end of the coevolution run, four more models show problems of convergence when the equations are combined into a 6DoF model. Among the remaining 83.3% of models that converge properly, differences in performance are also noticeable, with a small number of models having a distinctly lower performance level than the majority.

Figure 5.27 shows the progress of the mean tree size of each equation during the 20 epochs of coevolution³⁶.

The pattern shown for all dimensions is rather similar to that shown by the two previous platforms. The first epochs are marked by a sharp increase in size, and as the evolution progresses the rate of increase slows down. It is difficult to determine clearly whether the size of trees reaches a plateau but certainly the plot suggests a stabilization of the mean size to a value close to 30 nodes. The size of the expression for the vertical acceleration a_z always seems to be larger than that of the others and by the end of the coevolutionary epoch it reaches 35 nodes.

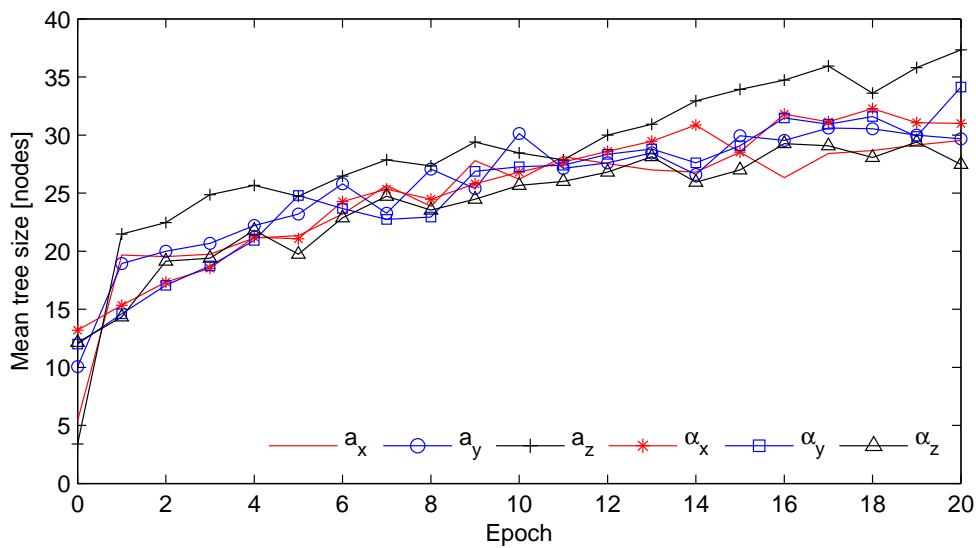


Figure 5.27: Mean size of the expression trees throughout the 20 coevolution epochs. All the expressions are of similar size.

³⁶As previously, the mean size is computed by averaging the equation size of each best model across the 30 coevolutionary runs.

5.7.2 Performance Analysis

Two representative models are selected for the performance analysis; the best ($X3DAccGPbest$) and the median ($X3DAccGPmedian$)³⁷.

Figure 5.28 shows the time series of the state variables obtained by integrating forward in time the predictions of the two selected models. The predictions are computed over a randomly chosen data window from the validation dataset `datax3d_2`; this is the same window used in Chapter 4 to display the predictions of the models based on first principles, and also those of the artificial neural networks (see Figures 4.3, 4.14 and 4.18). In the selected data window the predictions of both the $X3DAccGPbest$ and $X3DAccGPmean$ appear good. For the rotational velocities p , q and r the predictions of the two models are indistinguishable, and in a large part of the window they are also indistinguishable from the experimental data. For the variables p and q the data present a small oscillation around $t = 7\text{s}$ not reproduced by the models. Unfortunately it is impossible to determine with certainty if this oscillation is part of the model dynamics or if instead it is just a manifestation of unmeasurable external inputs (e.g. turbulence), but the short duration of the oscillation makes the second explanation more likely. The model slightly overestimates the angular velocity r at times $t = 5\text{s}$ and $t = [10, 11]\text{s}$.

The differences between the $X3DAccGPbest$ and $X3DAccGPmean$ models become visible on the angles. Although both are very good (we need to take into account in our judgement that these are after all integrated variables), the best model shows the better performance. Overall the error accumulated in attitude after 12 seconds of integration is very limited.

Larger errors appear for the linear velocities, in particular for u and v . As we learned from deriving the quadrotor model based on first principles in Section 4.1.2, the lateral and longitudinal linear accelerations of a quadrotor are directly dependent on the angles θ and ϕ through the projection of the gravity force. Therefore it does not come as a surprise that the errors in lateral and longitudinal velocity appear in regions where the angular errors are located.

The vertical velocity w does not share the fate of u and v as it is not directly dependent on the angle predictions, and as a result the predictions made by both models are very

³⁷As previously we determine the best and median models based on the $RMSE_{total}$ metric computed over the validation dataset; see later in this section for further details.

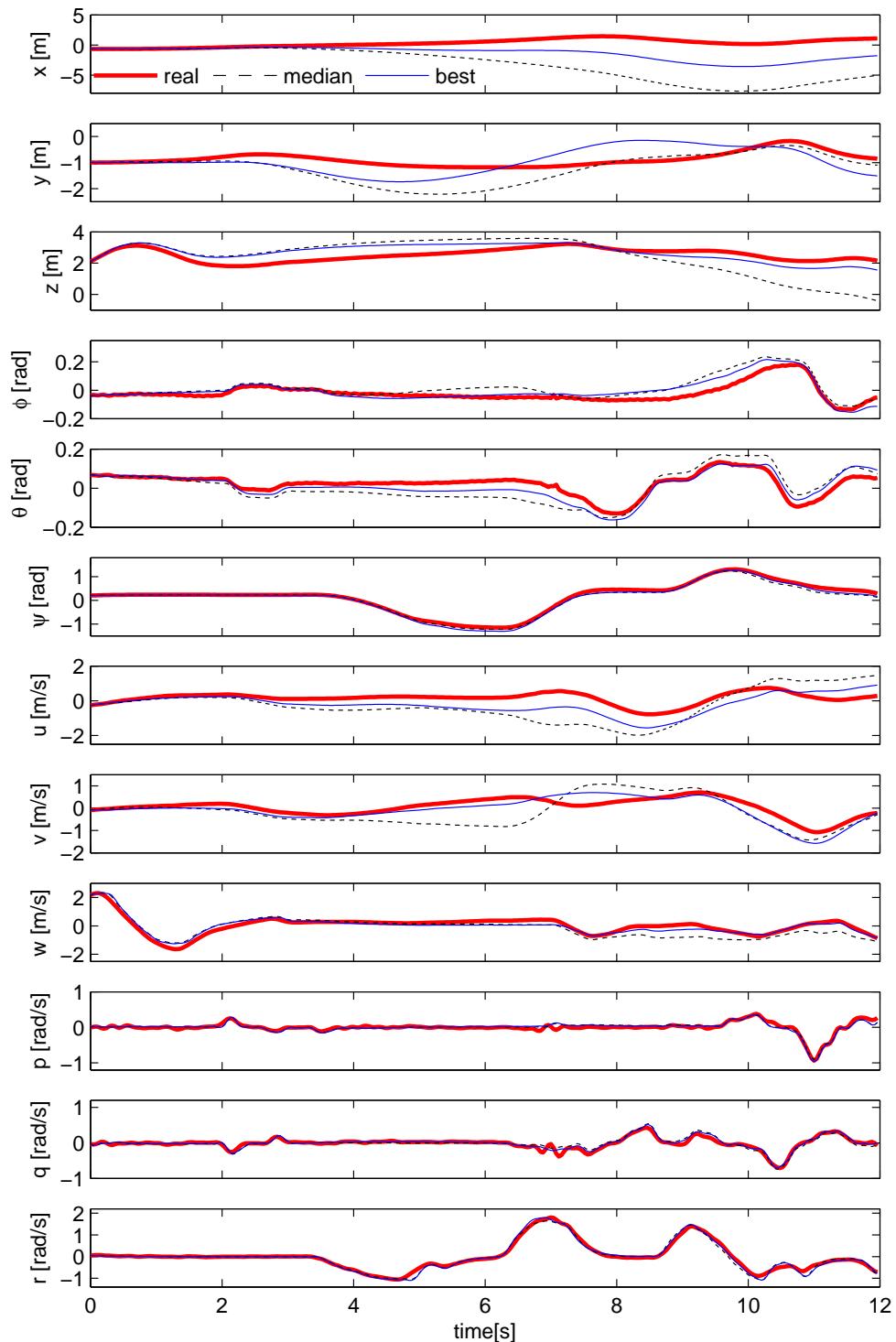


Figure 5.28: Proof of match: the state progress predicted by the *X3DAccGPbest* and median *X3DAccGPmedian* models versus the measured real state.

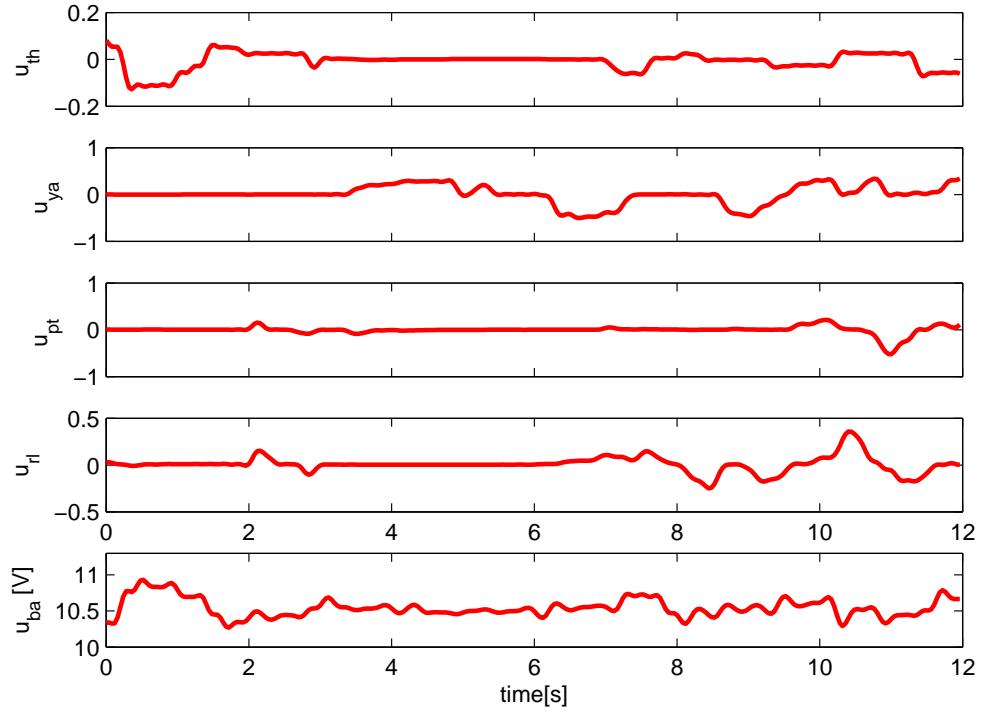


Figure 5.29: Control inputs given to the *X3DAccGPbest* and *X3DAccGPmedian* models during the time window in Figure 5.28.

good. In practice w is almost indistinguishable from the experimental data.

The integrated variables x , y and z show larger errors obviously due to the errors in the u and v velocities, and it is clear that the *X3DAccGPmedian* model has more of a tendency to move away from the trajectory of the recorded data than the *X3DAccGPbest* model. The altitude remains accurate throughout the whole 12 seconds of predictions with an error of less than 1m.

Let us now have a look at the quantitative error of the two models over this window of data that we just commented on. The *RMSE* and the *CR* for both the best and median models are reported in Table 5.16. With the exception of ψ which has a CR of only 0.46, the *X3DAccGPmedian* model, presents CR coefficients in the range 0.82 to 1.42, sufficiently close to one to indicate that the performance shown in Figure 5.28 for this model is largely representative of the behaviour of the model over the whole validation dataset.

In the case of the *X3DAccGPbest* model the CR values are generally lower than one, informing us that in this randomly selected window the chosen model appears to be better than its average performance on the whole dataset.

Before making a thorough quantitative evaluation of performance, we wanted to gain a qualitative feeling of the models' quality by interfacing them with our 3D visualization and

	Best Model		Median Model	
	RMSE	CR	RMSE	CR
u	0.53 m/s	0.60	0.88 m/s	0.85
v	0.29 m/s	0.49	0.56 m/s	0.86
w	0.21 m/s	0.89	0.40 m/s	1.42
ϕ	0.036 rad	0.87	0.049 rad	1.15
θ	0.032 rad	0.61	0.053 rad	0.88
ψ	0.085 rad	0.43	0.090 rad	0.46
p	0.048 rad/s	0.84	0.046 rad/s	0.82
q	0.045 rad/s	0.81	0.063 rad/s	1.07
r	0.090 rad/s	1.24	0.096 rad/s	1.32
<i>total</i>	0.66		1.14	

Table 5.16: Prediction error of the $X3DAccGPbest$ and the $X3DAccGPmedian$ models computed on the window of data plotted in Figure 5.28.

the control remote, and flying them manually. The visual experience of the simulator is certainly not what a pilot would have with a real quadrotor, but since the model can be run in real time the speed and magnitude of the response of the system are very realistic. In the hands of an experienced pilot this can give a good feel for the dynamic characteristics of the quadrotor. During the test we explored the envelope of the model by flying it similarly to the way in which the real helicopter was flown during data collection. We also tried a variety of control combinations to see if any would lead the model to behave differently.

Both the $X3DAccGPbest$ and $X3DAccGPmedian$ model performed well qualitatively, being able to reproduce responses similar in motion and speed to those of the real platform. The ability of the model to rotate and change position is a close match to the real flying machine, at least to the extent that can be appreciated in simulation.

As with the $X3DAccFP$ model (Section 4.1.2), the most noticeable difference between the real helicopter and the model lies in the absence of turbulence that characterizes the latter. In the real flying machine small drifts and vibration are continuously having small but noticeable affects on the helicopter’s position and orientation; the models, being fully deterministic, do not show this behaviour. For both the median and best models, our attempts at forcing the model into divergence did not produce any indication of instability, nor of behaviour incompatible with the motion of the real platform.

While these tests are not meant to provide any quantitative measure of the model’s abilities, we found them important in increasing our confidence in the models. In Chapter 7 we will use these models to build controllers to fly the real quadrotor autonomously.

Without in the first place being able to trust our models, we would certainly not feel comfortable in testing the evolved controllers on the real helicopter.

To test the models we proceed in the usual fashion, using the validation dataset `datax3d_2` split into consecutive windows, and computing the metrics *RMSEm* and *RAEm*. We report the performance of both the *X3DAccGPbest* and *X3DAccGPmedian* models in Tables 5.17 and 5.18 respectively.

We start by looking at the *RAEm*. Both models show values of $RAEm_p$, $RAEm_q$, $RAEm_r$ and RAE_w much lower than one, so the two models are therefore much better than a *constant model* at predicting the four state variables. As we know and will confirm in the next section when looking at the models' equations, these four equations do not depend directly on other state variables, and the performance of the models is then very good since it is not affected by error accumulation.

	RMSEm	RMSEsd		RAEm
u	0.88	0.48	<i>m/s</i>	1.14
v	0.58	0.30	<i>m/s</i>	0.92
w	0.24	0.13	<i>m/s</i>	0.33
ϕ	0.041	0.019	<i>rad</i>	0.47
θ	0.052	0.024	<i>rad</i>	0.44
ψ	0.20	0.23	<i>rad</i>	0.76
p	0.057	0.023	<i>rad/s</i>	0.24
q	0.056	0.022	<i>rad/s</i>	0.20
r	0.072	0.053	<i>rad/s</i>	0.25
<i>total</i>	1.17	0.51		

Table 5.17: Prediction error of the *X3DAccGPbest* model over the set of consecutive windows covering the validation dataset.

	RMSEm	RMSEsd		RAEm
u	1.04	0.48	<i>m/s</i>	1.31
v	0.66	0.30	<i>m/s</i>	1.04
w	0.28	0.11	<i>m/s</i>	0.39
ϕ	0.042	0.018	<i>rad</i>	0.49
θ	0.06	0.024	<i>rad</i>	0.51
ψ	0.19	0.23	<i>rad</i>	0.74
p	0.056	0.026	<i>rad/s</i>	0.23
q	0.058	0.021	<i>rad/s</i>	0.21
r	0.073	0.055	<i>rad/s</i>	0.24
<i>total</i>	1.32	0.53		

Table 5.18: Prediction error of the *X3DAccGPmedian* model over the set of consecutive windows covering the validation dataset.

Slightly worse predictions are seen for the angular quantities ϕ , θ and ψ , which is to be expected since they depend on the integration of the angular velocity and are therefore affected by any prediction error in the angular velocity dimensions.

The remaining variables u and v have $RAEm$ close to one showing that the prediction errors made by our models are in line with the errors that a *constant model* would produce. As we know from our analysis of the model based on first principles (see Section 4.1.2), the lateral accelerations, and therefore the lateral velocities, depend primarily on the re-projection of the resultant between the thrust and gravity forces and therefore on a correct prediction of the pitch and roll angles. In the light of this, it does not come as a surprise that the u and v variables are the least accurate predictions made by the model.

In absolute terms, the performances of the median and best models are remarkably similar, with the *X3DAccGPmodel* being slightly better. The most noticeable differences are in the pitch angle θ and in the longitudinal velocity u where the better predictions of the *X3DAccGPbest* model are clear.

Repeating the computation of the $RMSEm$ on all of the best models obtained in the 30 independent coevolutionary runs, we can compute the statistics of their performances (see Table 5.19). Excluding the diverging models, the standard deviation amounts to 22.6% of

	$RMSEm_{total}$
best	1.17
median	1.32
mean	1.37
s.d.	0.21 (15.3% of mean)
diverging	5

Table 5.19: Best and median and average prediction error in the 30 runs of the *X3DAccGPbest* model.

the mean $RMSEm$ value; although not very large, variations in performance (15.3% of the mean $RMSEm_{total}$) are present, an inevitable result of the stochastic nature of genetic programming.

We now know that our coevolutionary methodology can produce models capable of good predictions, and also that it can produce such models reliably. While producing models better than those obtained using more conventional techniques is not among the aims of this research it is still interesting to compare these models with those based on first principles obtained in Section 4.1.2.

Since the same training and validation datasets were used, we can compare the *X3DAccFP* model (namely the best X3D model obtained in Chapter 4) with the *X3DAccGPbest* model directly (comparing Table 4.6 with Table 5.17).

The overall *RMSE_{total}* performance identifies *X3DAccGPbest* as the better performing model. If we look at the errors in the various state variables, we see that the predictions of the angular velocities p and q and of the heading angle ψ have very similar errors in both models. However, the yaw velocity error of the *X3DAccFP* model is much larger (approximately double) than that of the best model obtained with genetic programming. For reasons previously mentioned, this error has a direct impact on the prediction of the angles ϕ and θ which for the *X3DAccFP* model have noticeably higher *RMSE_m* than the *X3DAccGPbest* model. In terms of linear velocities, the pattern of errors in the two models is very similar, with a low error for the vertical velocity w and a higher error in the lateral and longitudinal velocities. The *X3DAccGPbest* model performs better than the first principles model for the variables v and w , but is marginally less capable in predicting the longitudinal velocity u .

Finally we would like to gain some insight into which of the state variables are easier to learn. To do so, we look at the *RAEm* (see Table 5.20) for each of the state variables across the 30 models from our coevolutionary runs³⁸. The pattern of *RAEm* for the 25 models that do not diverge is the same for both the best and median models, confirming that throughout the independent runs the performance of our coevolutionary runs are

	RAEm_{mean}	RAEm_{sd}
u	1.26056	0.22943
v	1.15886	0.15931
w	0.37519	0.06355
ϕ	0.53868	0.06196
θ	0.49751	0.09024
ψ	0.76893	0.04790
p	0.26613	0.06390
q	0.21605	0.02794
r	0.25618	0.01645
diverging	5	

Table 5.20: *RAEm* of the set of models obtained by 30 independent runs of the coevolutionary algorithm (the diverging model are not included in the computation of the standard deviation).

³⁸We define as easier to learn the variables that have lower *RAEm* on average across the obtained models.

quantitatively repeatable. As seen for the median and best models, the easier variables to learn are the velocities p , q , r and w followed by the angular quantities ϕ , θ and ψ .

5.7.3 Analysis of the Equations

It is now time to inspect the equations that the coevolutionary algorithm produced in the 30 independent runs. We proceed by first automatically simplifying the models' equations and then looking for models that share the same structure (see Section 5.6.3 for details of how this was done), we again call the model structure common to many of the models the *general form*, and Obviously, since the coevolutionary algorithm is required to make accurate predictions of the system's accelerations without any additional constraints on the models' structure, we should expect that the models produced are likely to have a lumped nature. The contributions of each rotor and of the drag and lift forces affecting it will not appear in the model equations 4.14-4.19; instead, we are likely to see terms that lump together and approximate the forces and moments from many different sources.

The X3D quadrotor is symmetric about its x and y axes³⁹, and as a results its dynamics should be approximately the same for the longitudinal and lateral directions of motion. Happily, our algorithm, although not primed with any of this information, produces for the accelerations a_x and α_x equations with the same structure that appear in the equations of a_y and α_y respectively.

In the case of the linear acceleration a_x we have a general form

$$a_x = C^{ax} + C_\theta^{ax}\theta + C_u^{ax}u, \quad (5.9)$$

in which we can recognize a “driving” term C_θ^{ax} that expresses the fact that the platform accelerates proportionally to the tilt it has in the longitudinal direction. The motion is damped proportionally to the linear velocity u by what appears as a friction term with coefficient C_u^{ax} . Comparing the *general form* with equation 4.14 that we report here to facilitate the comparison

$$a_x = -(H_{0x} + H_{1x} + H_{2x} + H_{3x}) - \frac{1}{2}C_{xy}A_c\rho u|u| - \sin(\theta)g, \quad (5.10)$$

we recognize the term $C_\theta^{ax}\theta$ as being the small angle approximation of the term $\sin(\theta)g$.

³⁹Axes are defined in Figure 3.7.

a_x
best: $-0.41(1.65 + u_{pt-1} + u) + \theta(9.97 - 0.28u - q)$
median: $-0.23 + 10.1\theta - 0.41u - 0.24u_{ba} + 0.24r$
general form: $C^{ax} + C_\theta^{ax}\theta + C_u^{ax}u$ $C^{ax} = -0.26 \pm 0.025$ (9.6%) $C_\theta^{ax} = 10 \pm 0.26$ (2.6%) $C_u^{ax} = -0.42 \pm 0.030$ (6.9%)
agreement: 100% (30/30) ^a

Table 5.21: Structure of the evolved X3D models: a_x equations.

^aPercentage of models that agree with the *general form* (we define a term as part of the *general form* if it is present at least in 70% of the models). The actual number (out of 30) is shown in parentheses.

^bWithin the small angles approximation (see text) this models also agree with the first principles one.

a_y
best: $-0.44v + (0.84 + \psi(0.41 + u_{th-2}))/(-4.87 - \psi - u_{th-2} - u + w + r) + \phi(-9.89 + 2u_{ya-1})$
median: $((1.05 + \theta)(1.26 - 3.2u_{pt} + 3.19v + r))/(-8.65 - 4.71\theta + r - 1.1u_{ya} - \theta u_{ya}) + \phi - \phi(11.06 - \psi + u_{rl} - \theta)$
general form: $C^{ay} + C_\phi^{ay}\phi + C_v^{ay}v$ $C^{ay} = -0.17 \pm 0.014$ (13%) $C_\phi^{ay} = -9.83 \pm 0.21$ (2.1%) $C_v^{ay} = -0.41 \pm 0.032$ (7.8%)
agreement: 100% (30/30) ^a

Table 5.22: Structure of the evolved X3D models: a_y equations.

^aWithin the small angles approximation (see text) this models also agree with the first principles one.

a_z
best:
$-0.67 - 7.03(-6 + r)u_{th-1} + \theta + (-\phi + \theta)(4.86\theta + v) + 1.38u_{ba} - w$
median:
$(11.81u_{th}(11.44 + \theta))/(3.08 + \phi + q + 11.79\phi\theta) + 3.27pu_{rl}(q + \theta + u_{ba}) + (-1.25 + 2.51u_{ba})/(2.14 + qu_{ba}) - w$
general form:
$C^{az} + C_{u_{th}}^{az}u_{th} + C_{u_{ba}}^{az}u_{ba} + C_w^{az}w$
$C^{az} = -0.5 \pm 0.13$ (26%)
$C_{u_{th}}^{az} = 42 \pm 3.57$ (8.5%)
$C_{u_{ba}}^{az} = 1.2 \pm 0.29$ (24.1%)
$C_w^{az} = -0.9 \pm 0.13$ (14.4%)
delays:
$\delta_{th} = 0$ 56.7% (17/30) $\delta_{th} = 1$ 40% (12/30) $\delta_{th} = 2$ 3.3% (1/30)
$\delta_{ba} = 0$ 76.7% (23/30) $\delta_{ba} = 1$ 23.3% (7/30)
agreement: 96.7% (29/30) ^a

Table 5.23: Structure of the evolved X3D models: a_z equations.^aSee text for a comparison with the first principles model.

α_x
best:
$0.5 - \phi - q((-1.47 + u_{pt})q + u_{rl}) + (-0.59p + u_{rl})(79.5 - 0.31v) + 2v$
median :
$-45.6p - \phi + 77.4u_{rl} + (2.2 + u)v + u_{ba}$
general form:
$C_p^{\alpha_x}p + C_{\phi}^{\alpha_x}\phi + C_v^{\alpha_x}v + C_{u_{rl}}^{\alpha_x}u_{rl}$
$C_p^{\alpha_x} = -45 \pm 1.64$ (3.6%)
$C_{\phi}^{\alpha_x} = -2 \pm 1.36$ (68%)
$C_v^{\alpha_x} = 1.3 \pm 0.43$ (33%)
$C_{u_{rl}}^{\alpha_x} = 77 \pm 3.25$ (4.2%)
delays:
$\delta_{rl} = 0$ 96.3% (24/26) $\delta_{rl} = 1$ 7.7% (2/26)
agreement: 86.7% (26/30) ^a

Table 5.24: Structure of the evolved X3D models: α_x equations.^aSee text for a comparison with the first principles model.

α_y
best:
$0.26 - 4.3\phi - u_{pt} - r - u_{rl} - 4.9\theta - u + (-6.9u_{pt} - 3.8q)(11.6 - p - 0.45u_{pt} + u_{ba})$
median :
$4.37u_{th} + (q + u_{pt-1}(1.7 + u_{th}))(-48.1 + \theta) - 4.9\theta - u - q^2(3.1 + \theta)u$
general form:
$C_q^{\alpha_y}q + C_\theta^{\alpha_y}\theta + C_u^{\alpha_y}u + C_{u_{pt}}^{\alpha_y}u_{pt}$
$C_q^{\alpha_y} = -45 \pm 1.84$ (4%)
$C_\theta^{\alpha_y} = -4 \pm 1.06$ (26.5%)
$C_u^{\alpha_y} = -1 \pm 0.14$ (14%)
$C_{u_{pt}}^{\alpha_y} = -83 \pm 3.73$ (4.5%)
delays: $\delta_{pt} = 0$ 88% (22/25) $\delta_{pt} = 1$ 12% (3/25)
agreement: 83.3% (25/30) ^a

Table 5.25: Structure of the evolved X3D models: α_y equations.^aSee text for a comparison with the first principles model.

α_z
best:
$(3.14 + 0.90\phi)(4.6 + u_{pt} - 2.25/(1.14 - u_{th}) + r(r + w))(-0.44r - 1.7u_{ya})$
median :
$r(-4.2 - v) - (14.8 + \psi + u_{rl} - w)u_{ya}$
general form:
$C_r^{\alpha_z}r + C_{u_{ya}}^{\alpha_z}u_{ya}$
$C_r^{\alpha_z} = -3.9 \pm 0.33$ (8.5%)
$C_{u_{ya}}^{\alpha_z} = -15 \pm 1.1$ (7.7%)
delays: $\delta_{ya} = 0$ 100% (30/30)
agreement: 100% (30/30) ^a

Table 5.26: Structure of the evolved X3D models: α_z equations.^aSee text for a comparison with the first principles model.

The constant C_θ^{ax} has a value close to g , and from trigonometry we know that for small angles:

$$\sin(\theta) \simeq \theta \quad \theta \text{ is small.}$$

In the dataset in question the pitch and roll angles are small (less than 0.2rad in most part of the dataset) and therefore the approximation “discovered” by our coevolutionary algorithm is reasonable.

The term C_u^{ax} is an approximation of the drag force $\frac{1}{2}C_{xy}A_c\rho u|u|$; here, probably because the velocities reached by the helicopter are never very high, the genetic programming is approximating the second order term $u|u|$ simply with u . Due to the presence of the rotors and the complex shape of the helicopter fuselage, we cannot analyze in any detail why a linear model is preferred to the more standard viscous relationship. It is possible that a linear relationship allows the capture of aerodynamic effects that were neglected in the model based on first principles when the fuselage was represented as a planar surface. Dedicated wind tunnel measurements outside the scope of this work would be needed to shed more light on the possible reasons behind this approximation.

The term $(H_{0x} + H_{1x} + H_{2x} + H_{3x})$ in equation 5.10 does not appear in the *general form* of the equations a_x , but we cannot exclude the possibility that some of the models actually present terms with similar effects among the terms not forming part of the *general form*.

Considerations analogous to those just made for the equations of a_x can be made for the lateral acceleration a_y by comparing its *general form* with equation a_y and equation 4.15; however in this case we also have to bring in the small angle approximation for the cos function:

$$\cos(\theta) \simeq 1 \quad \theta \text{ is small.} \quad (5.11)$$

For both the longitudinal and lateral linear accelerations we see in Tables 5.21-5.22 that a unanimous agreement exists among all 30 models sharing the same *general form*. The agreement also extends to parameter values, which have low standard deviations.

The equation of the angular acceleration α_x takes the *general form*

$$\alpha_x = C_p^{\alpha_x} p + C_\phi^{\alpha_x} \phi + C_v^{\alpha_x} v + C_{u_{rl}}^{\alpha_x} u_{rl}, \quad (5.12)$$

where the control input u_{rl} is in this case the “driving” term of the equation, lumping together the dynamics of the left and right rotors. A damping effect on the rotational dynamics is introduced by the term $C_p^{\alpha_x} p$ which given its negative value for $C_p^{\alpha_x}$ has a stabilizing effect on the rotational motion. It is important to remember here that we are reproducing the dynamic behaviour of both the helicopter and its stability augmentation system, a system that is implemented in the embedded electronics in order to make the helicopter stable and therefore manually flyable (we describe the system in Section A.4).

In the light of this, we can rewrite the term $C_p^{\alpha_x} p + C_{u_{rl}}^{\alpha_x} u_{rl}$ as :

$$C_p^{\alpha_x} p + C_{u_{rl}}^{\alpha_x} u_{rl} = C_p^{\alpha_x} \left(p + \frac{C_{u_{rl}}^{\alpha_x}}{C_p^{\alpha_x}} u_{rl} \right), \quad (5.13)$$

in which, remembering the fact that $C_p^{\alpha_x}$ is negative, we can recognize the error computation of the PID stabilization loop that ensures an angular speed proportional to the u_{rl} input. The genetic programming appears to be lumping the group formed by the helicopter’s rotational dynamics and the stability augmentation system into the much simpler stable system produced by the interaction of the two. The term $C_\phi^{\alpha_x} \dot{\phi}$ produces a restoring torque that always tend to bring the helicopter back to level flight; this expresses the fact that the centre of force and the centre of mass of the helicopter do not coincide. Finally the term $C_v^{\alpha_x} v$ brings in all the aerodynamic effects produced by the fact that the helicopter is moving through the air. This term represents the fact that the advancing and retreating sides of the rotor experience different airspeeds, as well as capturing the moments produced by the drag forces acting on the helicopter fuselage.

These considerations concerning the angular velocity α_x apply unchanged to the equations produced for the pitch angular acceleration α_y since their *general form* is identical to that of α_x .

The level of agreement among models for α_x and α_y , although not as high as for the linear accelerations, is still very good with 86.7% and 83.3% of the models having the same *general form*. As regards parameter values, the $C^{\alpha_x p} p$ ($C^{\alpha_x p} p$) and $C_{u_{rl}}^{\alpha_x} u_{rl}$ ($C_{u_{pt}}^{\alpha_y} u_{pt}$) coefficients have low standard deviations. For $C_\phi^{\alpha_x} \dot{\phi}$ ($C_\theta^{\alpha_y} \dot{\theta}$) and $C_v^{\alpha_x} v$ ($C_u^{\alpha_y} u$) the agreement is less clear, and there are larger standard deviations. This suggests the former two terms as being the main players in the dynamic behaviour.

For both the pitch and roll accelerations, the vast majority of the models agree on an

input delay equal to zero, denying the presence of any substantial effect of delay in the system.

For the vertical acceleration a_z , we identified a *general form* possessed by 29 out of 30 models:

$$a_z = C^{a_z} + C_{u_{th}}^{a_z} u_{th} + C_{u_{ba}}^{a_z} u_{ba} + C_w^{a_z} w. \quad (5.14)$$

The coefficients $C_{u_{th}}^{a_z} u_{th}$ and $C_w^{a_z} w$ express an expected component of the vertical dynamic behaviour. The first of the two simply says that the acceleration is directly proportional to the throttle. The reader might note that in Section A.3 when we determined experimentally the relationship between the throttle input and the thrust generated, we observed that for values of u_{th} larger than 0.3, the relationship could be approximated as linear.

The coefficient $C_w^{a_z}$, introduces a velocity dependent damping that ensures the stability of the system. In line with what was found for the lateral and longitudinal accelerations, this damping has an intuitive explanation in the drag forces experienced by the helicopter when moving through the air. Again the genetic programming produces a relationship that is linear in the vertical speed. In this case it is difficult to determine whether the choice made by our algorithm is rooted in the nature of the aerodynamic phenomena taking place on the flying machine, or if it is simply an approximation made by the algorithm based on the fact that the velocity of the helicopter is never very high.

While the term C^{a_z} in the equation is simply a bias that is needed to ensure correct hovering behaviour in the absence of any control input, the most interesting term in the equation is surely $C_{u_{ba}}^{a_z} u_{ba}$, which explicitly links the vertical acceleration to the battery level. Our practical experience confirms this dependency (qualitatively at least) with the X3D helicopter being more responsive when the battery is fully charged, and becoming more and more sluggish as the battery depletes.

In terms of parameters, the agreement between the 29 models is satisfactory but there is certainly room for improvement. In terms of delays, the vast majority of the models (76.7%) do not indicate the need for any delay for the battery input u_{ba} , and the models are almost equally split between a delay of zero and one time step for the throttle input. This is not a clear result, but it is certainly possible that the algorithm is trying to model the inertia of the propulsion group (see A.3) as a delay with various degrees of success.

Finally we examine the equations of the yaw acceleration α_z which has the *general*

form:

$$\alpha_z = C_r^{\alpha_z} r + C_{r^3}^{\alpha_z} r^3 + C_{r^2 u_{ya}}^{\alpha_z} r^2 u_{ya} + C_{u_{ya}}^{\alpha_z} u_{ya}. \quad (5.15)$$

In this case also we can recognize a *general form* that reproduces the dynamics of the controlled system formed by the quadrotor and its stability augmentation system. The acceleration α_z is proportional to the difference between the control input and the current rotational velocity (terms $C_r^{\alpha_z} r$ and $C_{u_{ya}}^{\alpha_z} u_{ya}$). The model also shows higher order dependencies between α_z and both the control input u_{ya} and the rotational velocity r which do not lend themselves to a direct interpretation.

5.7.4 Conclusions

This exercise in modelling the X3D helicopter, with the greater complexity in its inputs and dimensions, proved to be a very interesting challenge for our algorithm.

The overall results are very satisfactory, with the algorithm being able to produce good models and also to produce models that are as good, or better than, models produced manually. It is interesting to note that the equations that we identified automatically are actually (and not surprisingly) similar to the models that have been used in recent publications [8] as part of the estimation and control loop for an indoor autonomous helicopter.

Due to the stochastic nature of the algorithm that we employ, we cannot guarantee that every single model produced will deliver the best possible performance, but in practice the methodology has been shown to be able to produce good quality models with high reliability.

The analysis of the structure and terms of the model equations showed that although no platform specific information is used, the algorithm is indeed able to “discover” equations that embed and express the characteristics of the platform (e.g. its symmetry).

Overall, it is important to notice that for both the automatically produced models and the model obtained from first principles, the errors in the state variables were heavily affected by the dependencies between the dimensions and the dynamic relationships that characterize the specific platform. Since no algorithm exists that will produce a perfect model, these dependencies are the ones that determine how hard a system is to model, and what level of results can be expected.

In comparing the equations obtained automatically against the model based on first

principles, we have seen a clear example of how our coevolutionary methodology reliably discovers approximate solutions that are valid within the envelope of data present in the training dataset. This again emphasises the paramount importance of the data collection stage in this technique; this is also true of any other technique that relies solely on the evidence provided by the data.

5.8 Automatic Modelling of the Owl Quadrotor

Finally we approach the last of the platforms considered in this work, the Owl quadrotor helicopter. Although conceptually similar to the X3D, the Owl is a larger and heavier platform, and has a completely different suite of sensors as well as a different control system; is it therefore interesting to see how the coevolutionary algorithm will cope with this platform. Naturally we will investigate the two automatically obtained analytical models and look for any similarity in structure between them.

Like the X3D the Owl dynamic model is also characterized by six dynamic equations, one for each of the DoF (i.e. it must predict $a_x, a_y, a_z, \alpha_x, \alpha_y$ and α_z).

The set of inputs \mathbf{x} and \mathbf{u} will also be the same:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r]^T$$

$$\mathbf{u} = [u_{th-\delta_{th}}, u_{ya-\delta_{th}}, u_{ro-\delta_{ro}}, u_{pi-\delta_{pi}}, u_{ba-\delta_{ba}}]^T.$$

In the case of the Owl, we keep the same settings for the coevolutionary algorithm that were used for all the other platforms, again avoiding any platform specific setup for the algorithm. The maximum allowed input delay was set to the standard value of 5 time steps, and a length of 300 samples was chosen for the test data window.

The coevolutionary algorithm was run for 30 times using the training dataset `dataowl_1`; each of the runs was terminated after 20 epochs.

5.8.1 Convergence

We start our analysis by looking at the convergence ability of the coevolutionary algorithm; and again using the same procedure explained in Section 5.7.1 we first look separately at each of the model's equations.

The result of computing the $RMSEm$ metric for each of the dimensions and for each of the best models in the 30 coevolutionary runs is shown in Figure 5.30. We recognize the

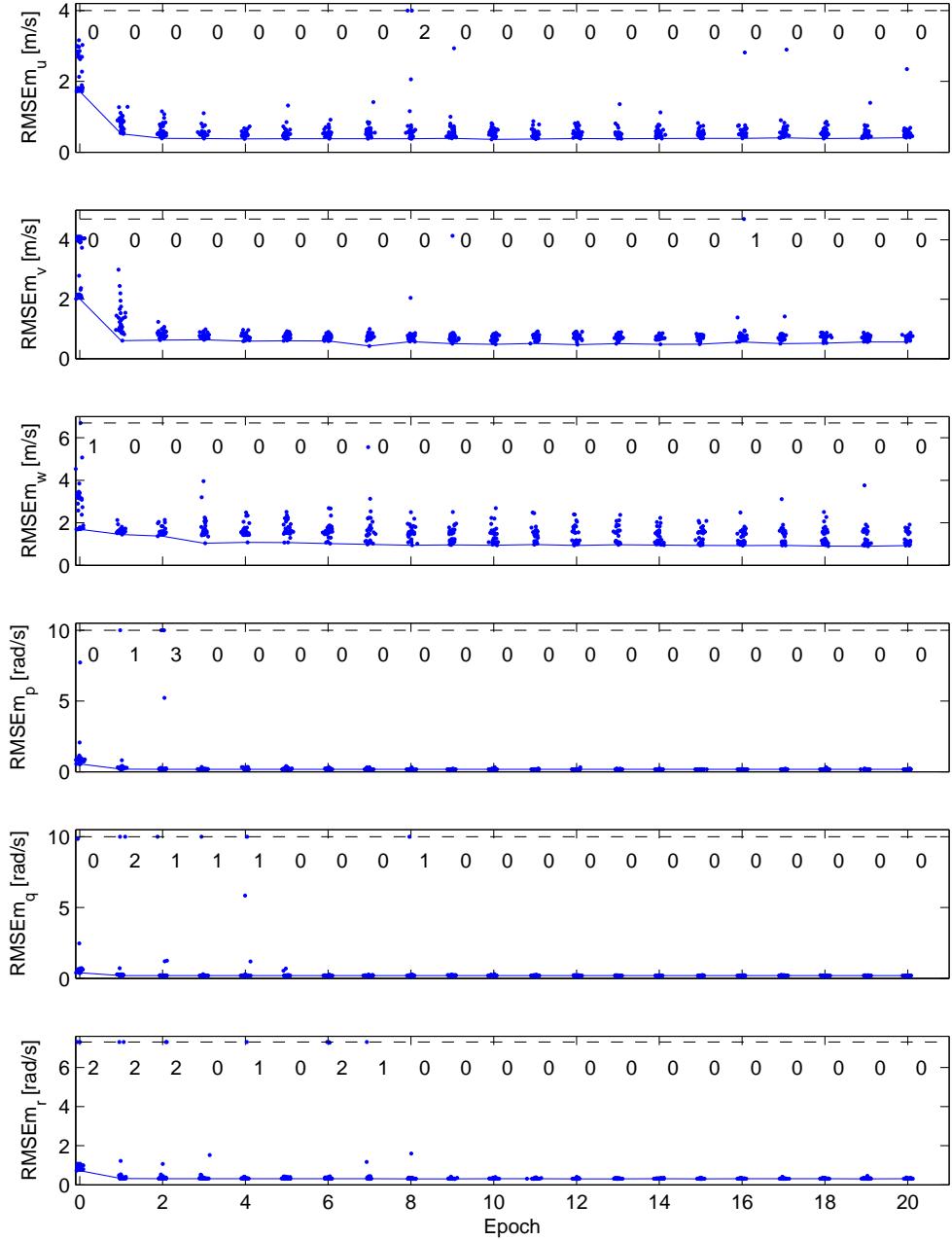


Figure 5.30: The error on single dimensions of the best model during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. The vertical scale of the plot is chosen proportionally to the standard deviation of the signal being predicted to allow for direct visual comparison between the three plots. The value of the points that exceed the maximum of the scale has been limited to the level indicated by the dashed line, and the figures at the top of each plot indicate how many points have been limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

familiar pattern of convergence seen for the platforms already examined, in that a large majority of the models converge during the initial epochs of coevolution. The additional epochs give a chance to the remaining models to converge but are also marked by the refinement of the models' fitness. As seen in Figure 5.25 for the X3D quadrotor, we see that quicker and more reliable convergence is exhibited for the angular velocities. At the 20 epoch mark, one of the equations for the a_x dynamics delivers quite a large error, but again this is understandable since the model is being tested on an unseen validation dataset.

Concentrating now on the full 6DoF model, we use all six equations of each best model simultaneously to make predictions. The performances of the best models during coevolution computed as already described for the X3D quadrotor (Section 5.7.1), are shown in Figure 5.31. Although we have seen that each single equation converges rapidly in the first few epochs, we know from the experiments on the X3D (Section 5.7.1), that the subsequent phase of equation improvement is vital for the convergence of the full model. Small improvements in fitness in Figure 5.30 deliver substantial improvements to the full models throughout the 20 epochs of coevolution. As the coevolution progresses, both the number of diverging models and the $RMSE_{total}$ of the models that are already converging

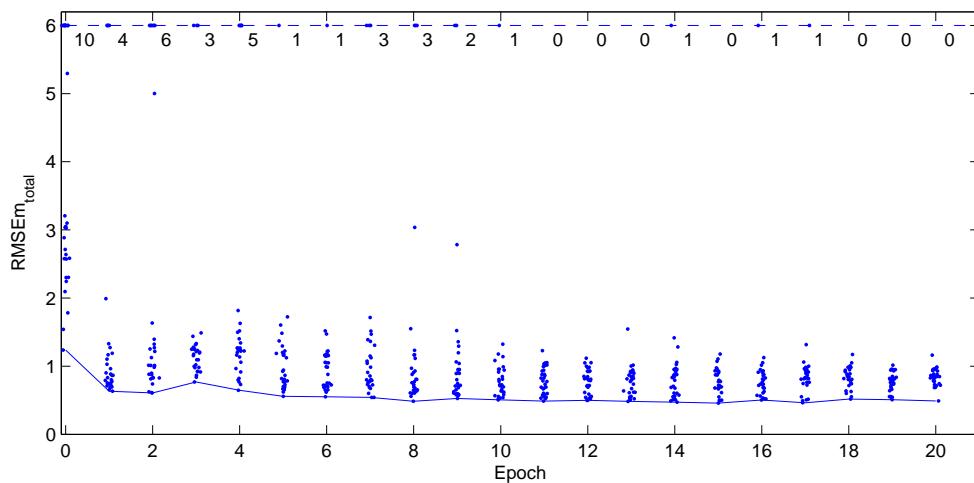


Figure 5.31: Total model error ($RMSE_{total}$) during the 20 coevolution epochs computed over the validation dataset. The best model for each of the 30 independent runs is plotted. The continuous line indicates the best among all the models. To focus on the most interesting part of the plot, the value of some of the points has been limited to the level indicated by the dashed line. The numbers at the top of each graph indicate how many points have been limited. The position of the dashed line was chosen as a compromise between providing a good picture of the distribution of models performances and containing the number of points limited. To avoid overlapping points a random jitter drawn from the distribution $\mathcal{U}[-0.1, 0.1]$ has been added to the epoch number of each point.

decrease. At the 20 epochs mark none of the models fails to converge.

During coevolution we also logged the size of the best models in the population; averaging this information across the 30 independent runs delivers for each of the model equations a history of tree sizes as shown in Figure 5.32. In the case of the Owl quadrotor we again recognize the familiar progression in the mean size of the models. In the initial population, small models have better performance, but their size increase quickly in the first two epochs. This coincides with a sharp decrease in error (see Figure 5.30). The epochs following the second are then marked by a much slower size increase. By the end of the coevolution phase, the size of the equations appears to stabilize at between 25 and 30 nodes. It appears that the measure we designed into the algorithm to control the tree size through the mutation operators and the explicit simplification procedure are effective in limiting bloat.

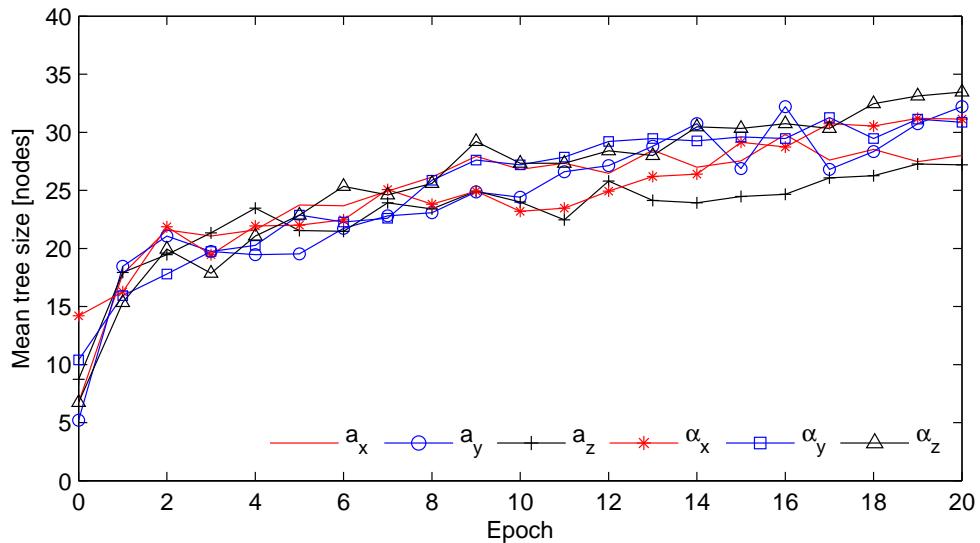


Figure 5.32: Mean size of the expression trees throughout the 20 coevolution epochs.

5.8.2 Performance Analysis

In this section we put to the test the Owl models obtained with our coevolutionary method, first qualitatively and then quantitatively. A good way to start to get an idea of the model quality is by plotting its predictions against the real data, the familiar proof of match. As for the previous platforms we plot only the best and the median models (we will call them *owlAccGPbest* and *owlAccGPmedian* respectively).

We plot the predictions over a randomly chosen sample window from the validation

dataset `dataowl_1` in Figures 5.33 and 5.34. The first thing that strikes us is that the angular velocities p and q are very noisy; as we commented in Section 3.3.2, this is due to the vibration of the Owl platform. Using a narrower low pass filtering at the data pre-processing stage could limit the effect of this type of vibration, but getting closer to the range of frequencies that are characteristic of the system would also increase the risk that the filtering might distort the information about the system dynamics present in the data. In this instance we considered it unacceptable to reduce the bandwidth of the smoothing stage because it would represent the use of platform dependent knowledge. Instead, we treat this as an opportunity to test our coevolutionary algorithm in the presence of data heavily affected by noise.

In reproducing the angular velocities, both the *owlAccGPmedian* and *owlAccGPbest* models appear very good at capturing the low frequency behaviour of the model, effectively ignoring the stochastic part of the dynamics produced by vibration.

The best and median models also appear indistinguishable in the prediction of the yaw velocity r . While providing a good fit, neither looks as smooth as the measured yaw velocity variable.

Although the fitting of the angular velocities looks satisfactory, our understanding of rigid body dynamics tells us that small differences between the predicted and the measured angular velocities, mostly due to vibrations, will certainly degrade the estimates of the platform angles. Our expectation are confirmed by the plots of ϕ , θ and ψ all of which show how the error, initially small, is integrated and grows with time. The *owlAccGPbest* model shows better prediction abilities for the pitch angle θ , while its ψ and ϕ predictions are very similar to that of the median model.

We expect any error in estimating orientation to have a direct effect on the prediction of the accelerations a_x and a_y and therefore in the velocities u and v (see Section 4.1.2 for more details). Looking at the plot of u and w we see exactly this; although the models clearly replicate correctly the changes affecting the two linear velocities, the predictions tend to drift away from the real data as time progresses. Due to better θ predictions, the u prediction produced by the *owlAccGPbest* model has considerably less drift than that produced by the *owlAccGPmedian* model.

For the variables x and y this problem of drift is even more evident since they are primarily dependent on the integration of u and v .

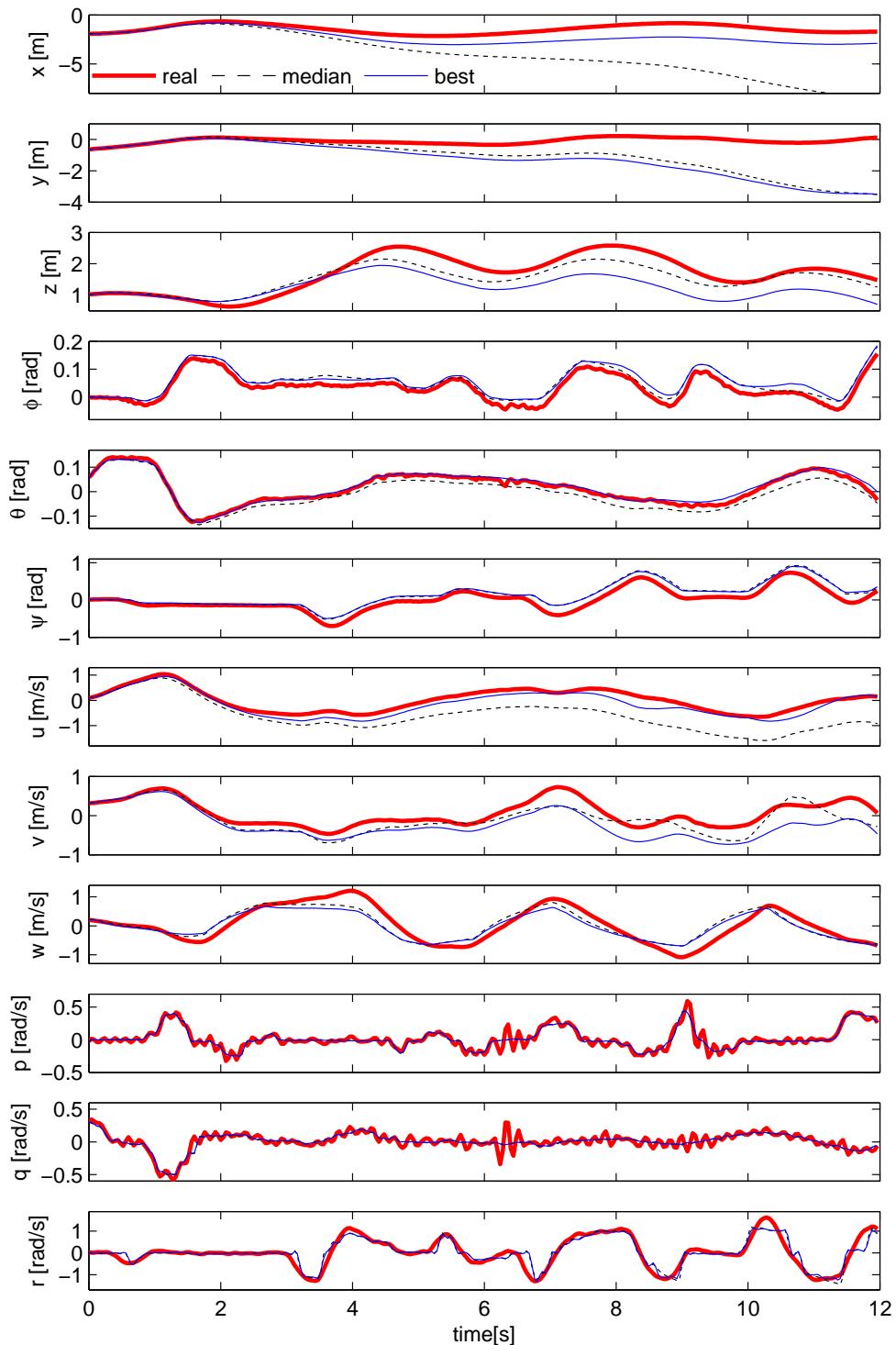


Figure 5.33: Proof of match: the state progress predicted by the *owlAccGPbest* and median *owlAccGPmedian* models versus the measured real state.

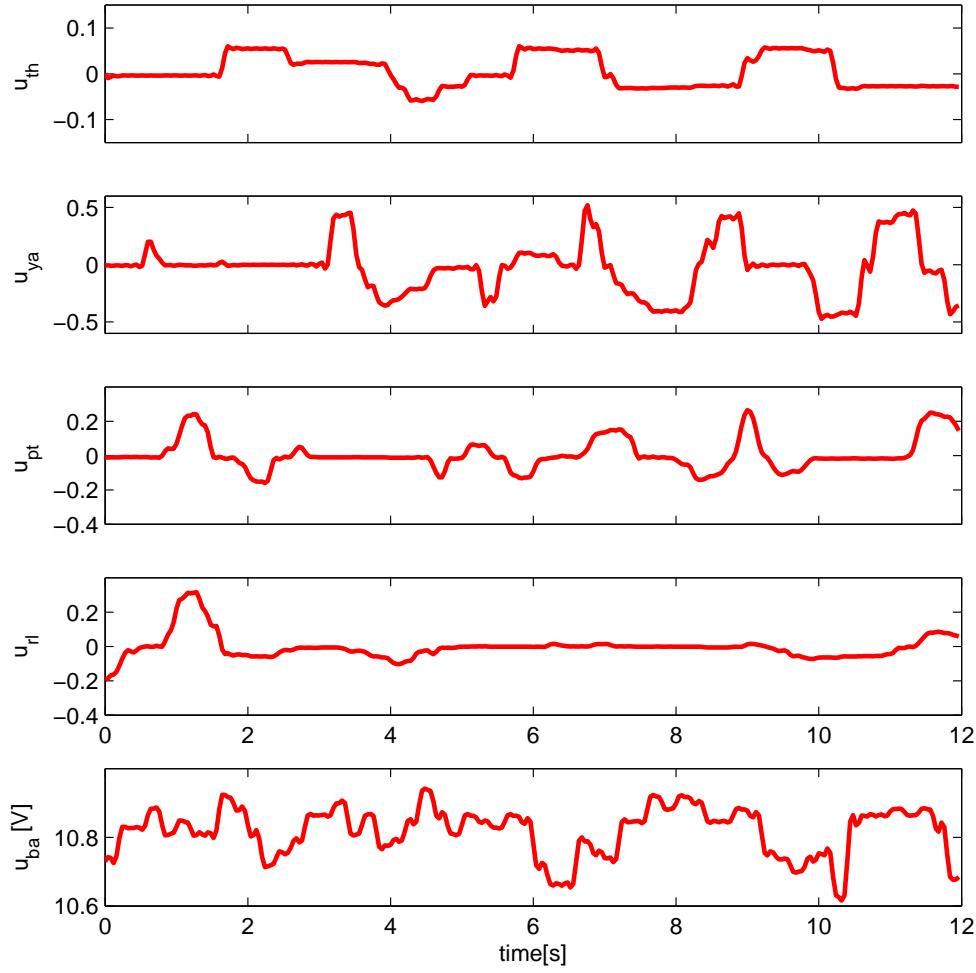


Figure 5.34: Control inputs given to the *owlAccGPbest* and *owlAccGPmedian* models during the time window in Figure 5.33.

We know that the vertical dynamics is less dependent on the platform angles, and as a consequence the prediction of w is almost unaffected by drift. For this state variable the predictions of the two models are very similar; both have a tendency to underestimate this velocity, but this is less pronounced for the *owlAccGPmedian* model. The altitude z is derived primarily from the integration of w , and shows how the better prediction of vertical velocity made by the median model also delivers better altitude predictions.

Before proceeding with other qualitative tests on the obtained models, we want to have an understanding of how the performance in this window is representative of the overall performance in the complete dataset. To do so we compute the *RMSE* for the window of data shown in Figure 5.33 and we compare it to the *RMSE* computed over the whole validation dataset by means of the coefficient *CR*; the results are shown in Table 5.27. From the values of the *CR* coefficients we can see that in general the randomly chosen window is fairly representative of the models' performance, and for the majority of the

	Best Model		Median Model	
	RMSE	CR	RMSE	CR
u	0.20 m/s	0.76	0.69 m/s	0.99
v	0.33 m/s	1.26	0.23 m/s	0.67
w	0.28 m/s	1.26	0.26 m/s	1.17
ϕ	0.023 rad	1.28	0.021 rad	0.89
θ	0.010 rad	0.77	0.024 rad	0.81
ψ	0.15 rad	1.72	0.16 rad	1.32
p	0.052 rad/s	1.22	0.052 rad/s	1.22
q	0.062 rad/s	1.24	0.062 rad/s	1.24
r	0.23 rad/s	2.97	0.24 rad/s	2.91
<i>total</i>	0.56		0.83	

Table 5.27: Prediction error of the *owlAccGPbest* and the *owlAccGPmedian* models computed on the window of data plotted in Figure 5.33.

variables the *CR* coefficients are not far away from one. An exception to this is the yaw velocity r for which the large *CR* value indicates that the performance on the selected window is much worse than the average for the yaw velocity. We can also see that for the *owlAccGPmedian* model, the prediction of v is particularly good in the selected window, explaining how the median model appears to be better than the best model in predicting v .

Before measuring the performance of our models quantitatively we wanted to have a firsthand impression of the quality of our model. We again interfaced the best and median model to our custom made software to obtain a simple 3D quadrotor simulator.

After testing all the basic flight manoeuvres (i.e. pitch, roll and yaw) and ensuring the behaviour of the simulator did indeed match that of the real flying machine, we tried to use the simulator to fly manoeuvres similar to those carried out on the real helicopter during data collection. The overall motion of the simulator appeared very realistic and during the test neither of the two models demonstrated instabilities nor unexpected dynamic behaviour.

In terms of flight behaviour we did not notice any apparent difference between the *owlAccGPbest* and the *owlAccGPmedian* model. However, we noticed a clear difference between the X3D and the Owl models in that the former are faster and more responsive in changing their altitude than the latter; this was expected because the X3D is a much lighter flying machine, and has a higher power to weight ratio. Unfortunately this was the only clear difference that we were able to identify in simulation between the Owl and X3D models. Although we use the same controller as with the real helicopter, the 3D

visualization simply does not provide the same visual experience as when flying the real machine.

For the Owl, the clear difference between the real platform and its simulation was again the complete absence of any disturbances. Another important point to make is the fact that when using the simulator we did not notice any effect of the attitude error we commented on when analyzing Figure 5.33. It is likely that when flying the model using the 3D visualization we unconsciously compensate for any small but systematic differences from the real flying machine just as a closed loop controller trying to govern the helicopter attitude would do.

A quantitative comparison between the *owlAccGPbest* and the *owlAccGPmedian* models is straightforward to obtain by calculating the *RMSEm* metric over the 36 consecutive windows that constitute the validation dataset `dataowl_1`. To obtain a direct perception of the dimensions in which the two models differ we compute the *RMSEm* for each dimension as well as the *RMSEm_{total}*; the results are shown in Tables 5.28 and 5.29.

Both models show low *RAEm* in almost all dimensions, confirming that the obtained models are much better than a *constant model*. The only exception is the longitudinal velocity v of the *owlAccGPmedian* model which has quite a large error and scores a *RAEm* of one. Both models show a similar pattern of performance, with the *owlAccGPmedian* model slightly worse than the best model in each dimension except for the *RMSEm_u* of the median model which is more than twice that of the best model. The error in the u dimension is thus the principal source of the difference in performance between the two models.

Both models have a standard deviation of the *RMSEm* across different data windows

	RMSEm	RMSEsd		RAEm
u	0.26	0.17	m/s	0.36
v	0.26	0.13	m/s	0.28
w	0.22	0.06	m/s	0.34
ϕ	0.018	0.008	rad	0.22
θ	0.013	0.007	rad	0.23
ψ	0.089	0.06	rad	0.46
p	0.04	0.012	rad/s	0.20
q	0.052	0.015	rad/s	0.28
r	0.078	0.07	rad/s	0.23
<i>total</i>	0.49	0.15		

Table 5.28: Prediction error of the *owlAccGPbest* model.

	RMSEm	RMSEsd		RAEm
u	0.70	0.23	m/s	1.00
v	0.34	0.16	m/s	0.35
w	0.22	0.07	m/s	0.34
ϕ	0.023	0.009	rad	0.28
θ	0.03	0.009	rad	0.57
ψ	0.12	0.08	rad	0.64
p	0.043	0.012	rad/s	0.20
q	0.050	0.014	rad/s	0.29
r	0.083	0.073	rad/s	0.24
<i>total</i>	0.85	0.24		

Table 5.29: Prediction error of the *owlAccGPmedian* model.

	RMSEm_{total}
best	0.49
median	0.85
mean	0.84
s.d.	0.12 (14.1% of mean)
diverging	0

Table 5.30: Best and median prediction error among the 30 runs of the *owlAccGPbest* model.

(reported as *RMSEsd* in Tables 5.28 and 5.31) of the same order of magnitude as the corresponding *RMSEm*; this indicates that the performance of the models varies considerably over different part of the dataset.

Computing the *RMSEm_{total}* across all 30 models allows us to get a quantitative measurement of the performance of the whole set of models as well as of the variation in performance within models. From the results in Table 5.30, we see that the median and the mean of the distribution have similar values, suggesting a symmetrical distribution of performance around the mean. The 14.1% standard deviation is in line with that of the other platforms analyzed so far and confirms the good convergence ability of the coevolutionary algorithm.

Finally we look at the *RAE* across the 30 models produced by 30 different coevolution runs. Computing the mean and the standard deviation of the *RAE* allows us to understand which of the variables are on average more difficult to learn⁴⁰. Since the *RAE* is a relative measure, we can compare it across dimensions. The rotational velocities p , q , r are, in general, dimensions that are easier to learn. This result matches that obtained for the

⁴⁰Defined as having lower *RAEm* on average across the obtained models.

	RAEmmean	RAEmsd
u	0.9	0.19
v	0.33	0.038
w	0.44	0.081
ϕ	0.23	0.038
θ	0.6	0.11
ψ	0.53	0.084
p	0.20	0.011
q	0.2861	0.00095
r	0.2337	0.0092
diverging	0	

Table 5.31: $RAEm$ of the set of models obtained by 30 independent runs of the coevolutionary algorithm.

X3D models, and can be explained similarly by the fact that those quantities depend primarily on themselves and the control inputs, and are not directly coupled to other state variables. Low values of $RAEm_p$ make for low values of $RAEm_\psi$, and also for low values of $RAEm_v$ due to the dynamics of the helicopter. For similar reasons, but at the other end of the performance spectrum, we see $RAEm_q$ which affects the performance of both θ and u . The vertical velocity w has a moderately low $RAEm_w$ since it is not directly dependent on the angle predictions. Importantly, we see that, with the Owl as with the X3D, the ability of the models to make good predictions is strongly influenced by the dynamic couplings intrinsic to the platform. The angular velocities are confirmed once again as being the key elements for producing good models.

5.8.3 Analysis of the Equations

In this section we look at the equations produced by our automatic modelling method to see if the model can give us direct insight into the dynamics of our platform.

The first step of the analysis involves the automatic simplification of all the equations of the 30 dynamic models as well as the appropriate series expansion of some of the terms present in the equations. Both operations are carried out in exactly the same way as for the X3D quadrotor and the toy car (see Section 5.6.3).

The terms of each equation shared by many of the models constitute what we call the *general form* of the specific equation being analyzed. In Tables 5.32-5.37 we report the *general form* for each of the six model equations along with the equations of the *owlAccGPbest* and *owlAccGPmedian* models³⁵. As with the previous quadrotor, the

a_x
best:
$(-2.45(9.9 + r)\theta + u + (-2.1 + \phi + v)/(-4.9 + u_{ya}))/(-2.4 + u_{th})$
median:
$-0.19 + (10.3 + p + q + u_{th-1} - \theta)\theta - u/(2.7 + r)$
general form:
$C^{ax} + C_\theta^{ax}\theta + C_u^{ax}u$
$C^{ax} = -0.18 \pm 0.023$ (12.4%)
$C_\theta^{ax} = 9.97 \pm 0.31$ (3.1%)
$C_u^{ax} = -0.38 \pm 0.029$ (7.5%)
agreement: 96.7% (29/30) ^a

Table 5.32: Structure of the evolved Owl models: a_x equations.

^aPercentage of models that agree with the *general form* (we define a term as part of the *general form* if it is present at least in 70% of the models). The actual number (out of 30) is shown in parentheses.

a_y
best:
$0.40 + (-4.1 + \phi)\phi - \phi(6 + \theta) + (u_{ba}(-v + (p - u_{ya}))/-2.48 + r))/(-2.58u_{th-1} + 2.41u_{ba})$
median:
$-1.62(-p + (-0.04 + \phi)(6.16 + \theta) + v/(3.26 + \psi + u_{rl-1} + u_{ba}) - p(-1.15 + r + u_{ya}))$
general form:
$C^{ay} + C_\phi^{ay}\phi + C_v^{ay}v$
$C^{ay} = 0.40 \pm 0.012$ (2.9%)
$C_\phi^{ay} = -9.9 \pm 0.15$ (1.5%)
$C_v^{ay} = -0.4 \pm 0.12$ (30.3%)
agreement: 100% (30/30)

Table 5.33: Structure of the evolved Owl models: a_y equations.

a_z
best:
$-31.7(-0.86 + u_{th-1})u_{th-1} + w + w(-2.4 - 0.45(u_{rl} + u_{ya}))$
median:
$(2.45 - u_{pt})/(-2.29 + u_{th}) + 30.1u_{th} + 0.87p(u_{th} + v) + u_{ba-1} - w$
general form:
$C_{u_{th}}^{az}u_{th} + C_w^{az}w$
$C_{u_{th}}^{az} = 30 \pm 5.0 \text{ (17\%)}$
$C_w^{az} = -1.1 \pm 0.21 \text{ (18.9\%)}$
delays:
$\delta_{th} = 0 \text{ } 51.7\% \text{ (15/29)} \quad \delta_{th} = 1 \text{ } 24.1\% \text{ (7/29)}$
$\delta_{th} = 2 \text{ } 13.8\% \text{ (4/29)} \quad \delta_{th} = 3 \text{ } 10.3\% \text{ (3/30)}$
agreement:
96.7% (29/30)

Table 5.34: Structure of the evolved Owl models: a_z equations.

α_x
best:
$0.40 - 29.9p + 48u_{rl} + \phi(-3.7 + \phi + u_{pt-1} + v)$
median:
$0.42 - 29.3p + 47u_{rl} - (4.5\phi)/(1.4 + \phi v)$
general form:
$C^{\alpha_x} + C_p^{\alpha_x}p + C_\phi^{\alpha_x}\phi + C_{u_{rl}}^{\alpha_x}u_{rl}$
$C^{\alpha_x} = 0.41 \pm 0.055 \text{ (13.2\%)}$
$C_p^{\alpha_x} = -30 \pm 1.01 \text{ (3.3\%)}$
$C_\phi^{\alpha_x} = -3.5 \pm 0.30 \text{ (8.6\%)}$
$C_{u_{rl}}^{\alpha_x} = 49 \pm 1.64 \text{ (3.3\%)}$
delays:
$\delta_{rl} = 0 \text{ } 96.4\% \text{ (27/28)} \quad \delta_{rl} = 1 \text{ } 3.6\% \text{ (1/28)}$
agreement:
93.3% (28/30)

Table 5.35: Structure of the evolved Owl models: α_x equations.

α_y
best:
$-0.24 - 0.35p + 1.64r - 6.9\theta + (4.5u_{pt} + 2.9q)(-13 + u_{rl} + \theta w)$
median:
$-40q + u_{pt}(-63.2 + r) + 1.6r - 4.9\theta - 2.1/(5.9 + \phi + w)$
general form:
$C^{\alpha_y} + C_q^{\alpha_y}q + C_\theta^{\alpha_y}\theta + C_r^{\alpha_y}r + C_{u_{pt}}^{\alpha_y}u_{pt}$
$C^{\alpha_y} = -0.31 \pm 0.052 \text{ (17\%)}$
$C_q^{\alpha_y} = -39 \pm 1.79 \text{ (4.6\%)}$
$C_\theta^{\alpha_y} = -5.4 \pm 0.70 \text{ (13\%)}$
$C_r^{\alpha_y} = 1.2 \pm 0.23 \text{ (19.8\%)}$
$C_{u_{pt}}^{\alpha_y} = -62 \pm 2.97 \text{ (4.8\%)}$
delays:
$\delta_{pt} = 0 \text{ 100\% (27/27)}$
agreement:
90\% (27/30)

Table 5.36: Structure of the evolved Owl models: α_y equations.

α_z
best :
$-0.61 + u_{pt} - 58.6u_{ya} + r(-23.2 + 2.33\psi u_{ya})$
median :
$-0.82 + 0.39p - 0.77\phi - 0.77\psi + 0.38u_{pt} + 3.58(-7.39 + 0.83q + r - w)(r + 2.54u_{ya})$
general form:
$C^{\alpha_z} + C_\psi^{\alpha_z}\psi + C_r^{\alpha_z}r + C_{u_{ya}}^{\alpha_z}u_{ya}$
$C^{\alpha_z} = -0.7 \pm 0.1 \text{ (13.6\%)}$
$C_\psi^{\alpha_z} = -1.14 \pm 0.49 \text{ (42.9\%)}$
$C_r^{\alpha_z} = -23 \pm 2.6 \text{ (11.4\%)}$
$C_{u_{ya}}^{\alpha_z} = -58 \pm 6.6 \text{ (11.3\%)}$
delays:
$\delta_{ya} = 0 \text{ 100\% (28/28)}$
$\delta_{pt} = 0 \text{ 100\% (28/28)}$
agreement:
93.3\% (28/30)

Table 5.37: Structure of the evolved Owl models: α_z equations.

symmetry of the model dynamics in the longitudinal and lateral direction is apparent in the *general form* equations which for both a_x and a_y have the same form:

$$a_x = C^{a_x} + C_\theta^{a_x} \theta + C_u^{a_x} u. \quad (5.16)$$

More interestingly, such *general form* equations are identical to those obtained for the X3D quadrotor in Section 5.7.3. Trying to get some insight into the obtained equations leads to the same observation made for the X3D; the term $C_\theta^{a_x} \theta$ is again a small angle approximation of the projection of the gravity force, $C_u^{a_x} u$ is the effect of drag forces, and the term C^{a_x} is simply a way of correcting for biases. Entirely similar consideration apply to the expression of a_y .

The general form equations are shared by 29 out of 30 models in the case of the longitudinal acceleration a_x and by all the models in the case of a_y , showing excellent agreement. Looking at the parameters we see generally low standard deviations for the coefficients, with the exception of $C_v^{a_y}$, the standard deviation of which reaches 30.3% of the mean value. Good agreement is present between the values of corresponding parameters in the longitudinal and lateral acceleration equations, with $C_\theta^{a_x} \simeq C_\phi^{a_y}$ and $C_u^{a_x} \simeq C_v^{a_y}$, confirming once more the symmetry between the two dimensions. The bias coefficients are different, probably due to manufacturing tolerances and/or small misalignments between the helicopter structure and the reference frame.

The vertical acceleration a_z has a simple general form

$$a_z = C_{u_{th}}^{a_z} u_{th} + C_w^{a_z} w, \quad (5.17)$$

with a linear dependency on the throttle input and with the aerodynamic drag proportional to the vertical velocity w . Those two terms were also present in the *general form* of the equation a_z obtained from the X3D quadrotor (see Section 5.7.3). In that instance however the *general form* also presented a dependency on the battery level. While some of the Owl models do show the presence of the battery input (see the median model equation of a_z in Table 5.34), this term is not shared by most models. The X3D and the Owl helicopter use the same types of motor and motor controller, but the Owl uses a battery with larger capacity. We are inclined to believe that the difference in the battery size, and the fact that the Owl uses more rigid and efficient blades which can produce larger change of thrust

with smaller speed changes, might make the effect of battery voltage less important. At this point this is little more than a speculation; specific tests going beyond the scope of this work would be needed to characterize the discharge curves of various batteries along with parameters such as internal resistance.

We look now at the rotational accelerations α_x and α_y , and while we would expect fully symmetrical models as for the linear accelerations, we find only a partial agreement, in that the *general form* of the equation of α_y presents a term $C_r^{\alpha_y} r$ which does not have a corresponding term in the equation of α_x .

In the part of the *general form* that the two dimensions have in common:

$$\alpha_x = C_p^{\alpha_x} + C_p^{\alpha_x} p + C_\phi^{\alpha_x} \phi + C_{u_{rl}}^{\alpha_x} u_{rl}, \quad (5.18)$$

we can recognize the $C_p^{\alpha_x} p$ and $C_{u_{rl}}^{\alpha_x} u_{rl}$ coefficients that as we have seen already for the X3D helicopter, simply reproduce the dynamics imposed by the stability augmentation controller. $C_\phi^{\alpha_x} \phi$ is also recognizable as the term that reproduces the restoring moment due to the fact that the centre of mass and centre of forces do not coincide. Interestingly, in the roll and pitch acceleration equations of the Owl model, we do not recognize the terms in equation 5.12 that depend directly on the forward (lateral) velocity and that we understand to be a combination of various drag effects.

Since the size and design of blades are the main aerodynamic difference between the Owl and the X3D model, we believe this to be the main reason why the drag term does not appear in the rotational dynamics. However, we have also to take into account the fact that the two flight machines use completely different stability augmentation systems. It is possible for the PID in the Owl helicopter to have a significantly higher gain and therefore a better ability to reject any aerodynamic effects coming from the rotor blades. Unfortunately comparing the PID controllers in the two systems is not possible since the manufacturer of the X3D quadrotor does not disclose such parameters; specific tests would be needed to provide better support for what are at the moment just intuitions.

The term $C_r^{\alpha_y} r$ that appears only in the *general form* of the α_x equation for the Owl platform shows a coupling between the yaw motion and the roll motion. This type of cross axis couplings is difficult to explain, but might be understood by considering the way in which the yaw motion is controlled on a quadrotor platform. As noted in Section 3.3.1, the

yaw motion is controlled by slowing down two of the rotors (those that control the pitch) and speeding up the remaining two. It is therefore clear that there is bound to be some interference between the yaw control, and the pitch and roll control. In the ideal situation the pitch and roll PIDs with their high bandwidth should be able to reject any effect of the yaw control but, while tuning the stability augmentation system of the Owl, we found empirically that this is not always the case. In particular, if the gain of the yaw controller is very high, we have noticed that a yaw manoeuvre is associated with a perturbation in the pitch and roll dimensions. While during tuning we selected PID values that did not appear to show this problem, we cannot be sure that part of the problem is not still present on the platform. Our automatic modelling indicates such a dependency, even though it is not visible to the naked eye, and not apparent to the pilot.

Finally we analyze the yaw dynamics that takes the *general form*:

$$\alpha_z = C^{\alpha_z} + C_{\psi}^{\alpha_z} \psi + C_r^{\alpha_z} r + C_{u_{ya}}^{\alpha_z} u_{ya}. \quad (5.19)$$

We recognize again the terms $C_r^{\alpha_z} r$ and $C_{u_{ya}}^{\alpha_z} u_{ya}$ related to the PID stabilization system. We also see the bias coefficient C^{α_z} which has the role of offsetting the mean value of the u_{ya} control. At present we find it difficult to give any interpretation to the term $C_{\psi}^{\alpha_z} \psi$, but the fact that a large standard deviation (42.9%) is associated with the value of its coefficients indicates that it is likely to have only a marginal role in the model dynamics. The remaining coefficients show a much higher level of agreement with much lower standard deviations.

5.8.4 Conclusions

Testing the coevolutionary algorithm of Section 5.3 on the Owl data has undoubtedly delivered good results.

The ability to repeatedly converge to good solutions, and to yield good overall performance have been convincingly demonstrated.

The level of noise present in the data, and specifically in the roll and pitch angular velocities, certainly has an effect on the overall performance. While this noise directly challenges the assumption of negligible noise we made in Section 2.2.1, the coevolutionary algorithm is still able to produce good models.

As in the case of the X3D, the analysis of the evolved equations revealed meaningful models that capture platform specific characteristic of the dynamics (i.e. symmetry). Importantly these characteristics are exclusively due to the data, since the algorithm is identical to that used for the other platforms.

In addition, the equations of the models are very similar to those we obtained for the X3D showing how the algorithm is capturing the fundamental dynamics that the two platforms have in common, although being different in size, weight and electronics.

Of course the nature of the algorithm often produces terms in addition to those of the *general form* that are difficult both to interpret and to relate to the platform physics.

5.9 Further Algorithm Investigation

When designing our algorithm in Section 5.3.1 we made some principled choices about several parameters in our coevolutionary algorithm. In doing so, we made it clear that obtaining an optimal algorithm was not our aim, and that instead we wanted to build a “proof of concept” to demonstrate our coevolutionary approach. Our choices of parameters and settings were therefore mostly informed by similar research in evolutionary computation, our understanding of the problem at hand (excluding platform-specific knowledge), and a small number of exploratory runs.

While the results in Sections 5.5-5.8 are very positive, we still feel the need to do some additional investigation on two of the most important control parameters in our coevolutionary setup: the size of the population of models, and the length of the window used as an individual test.

5.9.1 Effect of Models’ Population Size

In Section 5.3 we took note of the folk wisdom of the genetic programming community, and made our population size as large as possible. However, some authors have shown good results by using much smaller populations especially when using mutation as the main search operator; relevant examples in the literature are [31] and [33] which also share many of the concepts used in our implementation.

We therefore decided to experiment with a much smaller population size and see what influence such a change would have on the overall performance of the algorithm. We

selected a size of 14 for the population (as used in [31]) which is clearly at the other end of the spectrum when compared to the 100 individuals that we have been using so far. To ensure a meaningful comparison we allowed roughly the same number of fitness evaluations for the two versions of the algorithm; since the size of the test population and of the test window is the same for both versions this boils down to allowing for a larger number of coevolution epochs. Using a maximum number of coevolution epochs equal to 150 fulfils our requirements⁴¹. The only other parameter that needs to be changed along with the size of the population is the number of best models preserved between epochs. Given a population of 14 models, maintaining the same ratio of 0.8 between the number of models preserved and population size that we used with a population of 100 individuals gives us a number of models preserved equal to 11. All other parameters of the algorithm were left unchanged.

In this work we are lucky enough to have more than one platform to test, and although it was very time consuming, we chose to re-run our coevolution on both the quadrotor helicopters and also on the toy car. We did not use the ATTAS aircraft since as we saw in Section 5.5.4 we believe that the limited size of its dataset has a direct influence on the results of the algorithm, and we obviously want to avoid reaching biased conclusions.

As in Sections 5.5-5.8 we executed 30 independent coevolutionary runs for each of the platforms. We again computed the performance of each of the models in terms of *RMSEm*.

The results for the toy car are shown in Table 5.38; to aid the comparison, we also include the results for a population of 100 from Table 5.11. The two versions of the

	large population (100) RMSEm	small population (14) RMSEm
best	0.47	0.32
median	0.54	0.39
mean	0.54	0.38
std	0.048	0.08
diverging	2	10

Table 5.38: Toy car: mean best and median prediction error computed over 30 coevolutionary runs using two different sizes for the population of models.

algorithm produce models with very similar performance; the median performance between

⁴¹Setting the number of epochs to 150 guarantees the same number of fitness evaluations for both instances of our algorithm, but in terms of computational resources the two versions are not exactly the same. Since each epoch is associated with the evaluation of the models against the history of tests, the instance of the algorithm with a small population is marginally more computationally demanding.

	Large Population (100)		Small Population (14)	
	R AEmmean	R AEmsd	R AEmmean	R AEmsd
u	0.19	0.025	0.21	0.030
v	0.48	0.010	0.54	0.077
r	0.33	0.016	0.37	0.070
diverging	2		10	

Table 5.39: Toy car: *RAEm* error computed over 30 coevolutionary runs using two different sizes for the population of models.

the two setups does not differ significantly (Mann-Whitney U test, $n_1 = 28$, $n_2 = 20$, $P < 0.01$ two-tailed, where n_1 and n_2 are the number of non-diverging models obtained by the two versions of the algorithm). Nonetheless is important to note that the number of diverging models is much smaller in the case of the algorithm with a larger population, making this version of the algorithm preferable.

A comparison of the mean *RAEm* of the two populations shows very similar performances across the four state variables (see Table 5.39), but the lower *RAEm* standard deviations show more consistent performance for the algorithm which uses a larger population.

We now apply the same analysis to the models obtained for the X3D platform (see Tables 5.40 and 5.41). In this situation, the best model is actually produced by the version of the algorithm based on a population of only 14 models. However a comparison of the performance of the non diverging models of the two populations (Mann-Whitney U test, $n_1 = 25$, $n_2 = 19$, $P < 0.01$ two-tailed) shows no statistical difference in performance between the two populations.

As for the toy car, the algorithm based on a smaller population of models produces a much larger proportion of diverging models of 36.7%.

	large population (100)	small population (14)
	R MSEm	R MSEm
best	1.17	0.98
median	1.32	1.38
mean	1.37	1.26
std	0.21	0.17
diverging	5	11

Table 5.40: X3D quadrotor, mean best and median prediction error computed over 30 coevolutionary runs using two different sizes for the population of models.

	Large Population (100)		Small Population (14)	
	RAEmmean	RAEmsd	RAEmmean	RAEmsd
u	1.26	0.23	1.10	0.19
v	1.16	0.16	1.16	0.23
w	0.37	0.063	0.39	0.061
ϕ	0.54	0.062	0.54	0.12
θ	0.50	0.09	0.43	0.06
ψ	0.77	0.048	0.75	0.062
p	0.27	0.064	0.24	0.034
q	0.22	0.028	0.23	0.042
r	0.26	0.016	0.28	0.041
diverging	5		11	

Table 5.41: X3D quadrotor, *RAEm* error computed over 30 coevolutionary runs using two different sizes for the population of models.

As expected from the statistical comparison of the models' performances, the performance of the two sets of models are very similar at the level of individual variables, with no evidence that one population is better than the other in any specific dimension.

Table 5.42 shows the results for the owl quadrotor. A first glance does not reveal any clear difference between the outcomes of the two variants of the algorithm. Statistical testing (Mann-Whitney U test, $n_1 = 30$, $n_2 = 27$, $P < 0.01$ two-tailed) confirms our intuition, failing to reject the null hypothesis that the medians of the two distributions are the same.

Once again, although no clear difference in performance was found, a higher number of models show convergence for the algorithm using a larger population.

Looking in more detail at the performance in the different dimensions (see Table 5.43) we again see no major difference between the results of the two algorithms.

In these experiments we compared only two different population sizes at opposite ends of the spectrum of what is allowed by what we consider viable computational constraints

	large population (100) RMSEm	small population (14) RMSEm
best	0.49	0.55
median	0.85	0.82
mean	0.84	0.84
std	0.12	0.18
diverging	0	3

Table 5.42: Owl quadrotor, mean best and median prediction error computed over 30 coevolutionary runs using two different sizes for the population of models.

	Large Population (100)		Small Population (14)	
	R AEm m ean	R AEm s d	R AEm m ean	R AEm s d
u	0.9	0.19	0.96	0.25
v	0.33	0.038	0.36	0.15
w	0.44	0.081	0.39	0.07
ϕ	0.23	0.038	0.25	0.079
θ	0.6	0.11	0.58	0.17
ψ	0.53	0.084	0.56	0.13
p	0.20	0.011	0.20	0.11
q	0.2861	0.00095	0.29	0.090
r	0.233	0.0092	0.24	0.032
diverging	0		3	

Table 5.43: Owl quadrotor, *RAEm* error computed over 30 coevolutionary runs using two different sizes for the population of models.

given the current implementation of our algorithm. The results obtained by our limited exploration of sizes confirm the genetic programming folk wisdom that larger populations produce better results. In our application, better results correspond to models with lower errors, and also to a larger number of models converging to a low error at the end of the coevolutionary epoch.

Obviously additional investigation could be conducted on this front. For example, it would be interesting to understand more clearly the relationship between performance and population size, and to see at what point, if any a larger population stops being beneficial. We reserve such questions for future work.

5.9.2 Effect of Length of Training Windows

In Section 2.2.4 we discussed how it was necessary to compute the performance of our models over a training window instead of over independent points in the dataset in order to take into account the long range dependencies inevitably present in the models. All our experiments in Sections 5.5-5.8 used a window length of 300 time steps. In this section we repeat the coevolutionary runs using longer windows (600 time steps) as well as shorter ones (150 time steps), a range of window sizes that it is meaningful to examine given the current setup. Given the size of our dataset, windows much longer than 600 steps will not only require an impractically long evolution time, but will also reduce to a handful the number of possible non-overlapping tests, which will in turn undermine the effectiveness of the coevolution framework. On the other hand having very short training windows goes

against many of the points made in Section 2.2.4 about long range dependencies.

With the aim of providing as far as possible a platform independent analysis, we repeated the coevolution runs using the new window sizes for the Owl, the X3D and the toy car platforms. As in the case of investigating the population size, the ATTAS aircraft was not used due to the limited amount of experimental data available. For each of the platforms the usual population of 100 individuals and a maximum number of epochs of 20 were used. All remaining parameters were kept the same as in all our coevolutionary runs.

Using three different training window sizes is not in any way exhaustive, but it is certainly sufficient to identify any major effects that the training window size has on our coevolutionary system. Given that we use the same size for the population of models and the same number of epochs for all three different window sizes, it is worth noting that the computational requirements of the algorithm will increase proportionally to the number of data points in the training window.

We evaluate the algorithm performance by comparing the distribution of the $RMSE_{m_{total}}$ achieved by the sets of best models produced from independent runs. For each platform and each training window size we repeat the coevolution 30 times.

The $RMSE_{m_{total}}$ metric is computed over each of the consecutive windows into which the validation dataset is split, and therefore depends directly on the training window size (see 4.1.1 for more details about the $RMSE_{m_{total}}$ metric). For this reason, instead of arbitrarily choosing one window size as a reference for each combination of the platform and training window sizes, we will consider the $RMSE_{m_{total}}$ computed with all three different window sizes, namely 150, 300 and 600 data points. The aim is to see if there is a size of training window that gives better performance at all window lengths, or if for instance training with a certain window size guarantees good performance only when tested with the same window size.

We start with the toy car in Table 5.44, which shows the statistics of the $RMSE_{m_{total}}$ metric for each of the 9 training-testing window combinations. Models that diverge are not used to compute the statistics. One point that is immediately clear from the table is that the number of models diverging is much higher in the case of the population of models trained with the window size of 600 steps, independently of the size of the testing window. Additionally we see that, although the $RMSE_{m_{total}}$ metrics take into account the size of the window (i.e. the error is divided by the number of sample points), larger windows are

<i>training size</i>	<i>testing size</i>		
	150	300	600
150	<i>b</i> : 0.41	<i>b</i> : 0.30	<i>b</i> : 0.61
	<i>m</i> : 0.62	<i>m</i> : 0.41	<i>m</i> : 1.08
	<i>M</i> : 0.45	<i>M</i> : 0.34	<i>M</i> : 0.82
	<i>s.d.</i> : 0.50	<i>s.d.</i> : 0.24	<i>s.d.</i> : 0.70
	<i>d</i> : 2	<i>d</i> : 2	<i>d</i> : 3
300	<i>b</i> : 0.38	<i>b</i> : 0.29	<i>b</i> : 0.59
	<i>m</i> : 0.43	<i>m</i> : 0.32	<i>m</i> : 0.78
	<i>M</i> : 0.42	<i>M</i> : 0.31	<i>M</i> : 0.76
	<i>s.d.</i> : 0.030	<i>s.d.</i> : 0.020	<i>s.d.</i> : 0.11
	<i>d</i> : 2	<i>d</i> : 2	<i>d</i> : 1
600	<i>b</i> : 0.37	<i>b</i> : 0.30	<i>b</i> : 0.60
	<i>m</i> : 0.46	<i>m</i> : 0.34	<i>m</i> : 0.86
	<i>M</i> : 0.44	<i>M</i> : 0.34	<i>M</i> : 0.79
	<i>s.d.</i> : 0.076	<i>s.d.</i> : 0.035	<i>s.d.</i> : 0.25
	<i>d</i> : 10	<i>d</i> : 11	<i>d</i> : 11

Table 5.44: Toy car: Performance of the automatically produced models when trained and tested with different sizes of data windows. *training size* indicates the length of the data windows used as tests during coevolution, while *testing size* indicates the size of the consecutive windows used in the computation of the $RMSEm_{total}$. The letters *b*, *m* and *M* indicate the $RMSEm_{total}$ of respectively the best, mean and median models in the set; *sd* indicates the standard deviation, and *d* the number of diverging models.

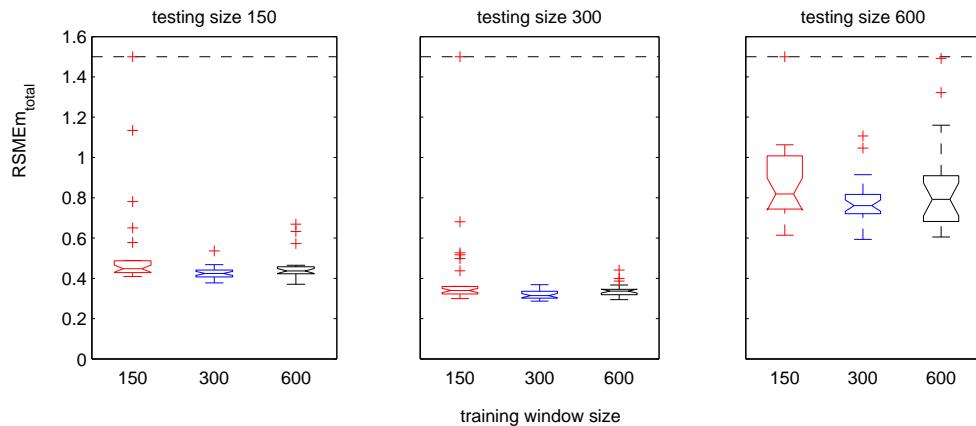


Figure 5.35: Toy car: $RMSEm_{total}$ metric evaluated over different testing window sizes for models trained with different lengths for the training windows.

characterized by a larger error, as expected due to the effects of error integration.

The information in Table 5.44 can be reorganized in a clearer fashion by using box and whiskers plots as in Figure 5.35⁴².

There is no clear difference between models trained with different window sizes, since the notches in the box plot all partially overlap. However, it is clear that the distribution of performance is much narrower for the models evolved with 300 time step windows. This is in agreement with the fact that this population also has the smallest number of diverging models.

No significant difference (Kruskal-Wallis test $P < 0.01$) was found between the medians of the performances of the three sets of models when tested in the 600 time step windows. Applying the same statistical test in the cases of the 150 and 300 time step windows led in both cases to the rejection of the null hypothesis of the equality of the medians. We then tested each pair of populations with the Mann-Whitney U test ($P < 0.01$ two-tailed), which can be interpreted here as a test of the equality of the medians. The only significant differences were that two of the three populations trained with 150 time step windows were worse than the corresponding populations trained with 300 time step windows, and one of the three populations trained with 300 time step windows was better than the corresponding population trained with 600 time step windows.

The results for the X3D quadrotor are shown in Table 5.45. It is immediately clear that for all the test window sizes, the number of diverging models is always smallest for the models trained with a window size of 300. The box and whiskers plots⁴² in Figure 5.36 show no clear difference between the sets of models, suggesting that changing the training window size has little influence. No significant difference (Kruskal-Wallis test $P < 0.01$) was found between the medians of the performance of the three sets of models when tested in the 300 and in the 600 time step windows. Applying the same statistical test in the cases of the 150 time step windows led to the rejection of the null hypothesis of the equality of the medians. Each pair of populations was tested with the Mann-Whitney U test ($P < 0.01$ two-tailed), against the hypothesis of equality of the medians. The only significant differences were that one of the three populations trained with 300 time

⁴²In the box and whisker plots, on each box, the central mark is the median, and the edges of the box are the 25th and 75th percentiles (q_1 and q_3 respectively). The whiskers extend to the most extreme data points not considered outliers; outliers are plotted individually. Points are drawn as outliers if they are larger than $q_3 + 1.5(q_3 - q_1)$ or smaller than $q_1 - 1.5(q_3 - q_1)$. The plotted whisker extends to the adjacent value, which is the most extreme data value that is not an outlier. The notches are comparison intervals, two medians are significantly different at the 5% significance level if their intervals do not overlap.

<i>training size</i>	<i>testing size</i>		
	150	300	600
150	<i>b</i> : 0.87	<i>b</i> : 1.11	<i>b</i> : 1.24
	<i>m</i> : 1.0	<i>m</i> : 1.30	<i>m</i> : 1.46
	<i>M</i> : 0.97	<i>M</i> : 1.29	<i>M</i> : 1.43
	<i>s.d.</i> : 0.10	<i>s.d.</i> : 0.11	<i>s.d.</i> : 0.13
	<i>d</i> : 5	<i>d</i> : 10	<i>d</i> : 10
300	<i>b</i> : 0.86	<i>b</i> : 1.17	<i>b</i> : 1.31
	<i>m</i> : 0.96	<i>m</i> : 1.37	<i>m</i> : 1.58
	<i>M</i> : 0.96	<i>M</i> : 1.31	<i>M</i> : 1.49
	<i>s.d.</i> : 0.067	<i>s.d.</i> : 0.21	<i>s.d.</i> : 0.30
	<i>d</i> : 0	<i>d</i> : 5	<i>d</i> : 5
600	<i>b</i> : 0.76	<i>b</i> : 1.059	<i>b</i> : 1.31
	<i>m</i> : 0.90	<i>m</i> : 1.25	<i>m</i> : 1.72
	<i>M</i> : 0.89	<i>M</i> : 1.22	<i>M</i> : 1.42
	<i>s.d.</i> : 0.05	<i>s.d.</i> : 0.11	<i>s.d.</i> : 1.05
	<i>d</i> : 4	<i>d</i> : 9	<i>d</i> : 14

Table 5.45: X3D quadrotor: Performance of the automatically produced models when trained and tested with different sizes of data windows. *training size* indicates the length of the data windows used as tests during coevolution, while *testing size* indicates the size of the consecutive windows used in the computation of the $RMSE_{total}$. The letters *b*, *m* and *M* indicate the $RMSE_{total}$ of respectively the best, mean and median models in the set; *sd* indicates the standard deviation, and *d* the number of diverging models.

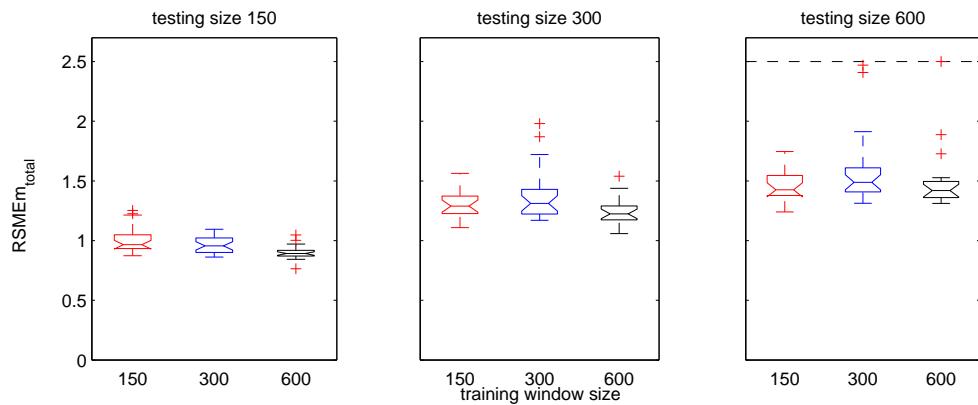


Figure 5.36: X3D quadrotor: $RMSE_{total}$ metric evaluated over different test window sizes for the model training with different length for the data windows.

step windows was worse than the corresponding populations trained with 600 time step windows, and one of the three populations trained with 600 time step windows was better than the corresponding population trained with 150 time step windows.

Results for the Owl quadrotor are shown in tabular (Table 5.46) and in box and whiskers plot form (see Figure 5.37).

Although the spread of the performances appears much larger, the data lead us to the same conclusions as for the toy car and X3D models. No clear difference in performance is visible for any of the three training window sizes investigated, and for testing windows of size 300 and 600 no significant difference (Kruskal-Wallis test $P < 0.01$) was found between the medians of the performance of the three sets of models. Applying the same statistical test in the cases of the 150 time step windows led to the rejection of the null hypothesis of the equality of the medians. We then tested each pair of populations with the Mann-Whitney U test ($P < 0.01$ two-tailed) and the only significant differences were that two of the three populations trained with 150 time step windows were better than the corresponding populations trained with 300 time step windows, and one of the three populations trained with 300 time step windows was worse than the corresponding population trained with 600 time step windows.

Again, there is agreement with what was seen for the toy car and X3D quadrotor since the models trained with windows of size 300 are less likely to diverge, although the difference is not as marked as for the previous two platforms. We have examined only three different sizes for the training windows, but the agreement of results across platforms leads us to conclude that across the windows sizes that are meaningful for our current coevolutionary setup, the exact size of the training window is not a determining factor for model quality. However, there is a clear dependence between the number of models diverging and the size of the training data window.

The training window size of 300 steps appears to be a good compromise between long and short windows, and is the most effective of the three lengths tested. We believe this is a direct result of the effects of error integration. On the one hand, models trained on very short windows are not strongly penalized for the effects of error accumulation, but these effects will become a major cause of divergence as the size of the testing window is increased. On the other hand, over long windows all but the very good models will tend to diverge. As a result the fitness metric will not provide a smooth gradient for the

<i>training size</i>	<i>testing size</i>		
	150	300	600
150	<i>b</i> : 0.40	<i>b</i> : 0.55	<i>b</i> : 0.69
	<i>m</i> : 0.61	<i>m</i> : 0.76	<i>m</i> : 1.0
	<i>M</i> : 0.49	<i>M</i> : 0.74	<i>M</i> : 0.99
	<i>s.d.</i> : 0.60	<i>s.d.</i> : 0.14	<i>s.d.</i> : 0.26
	<i>d</i> : 2	<i>d</i> : 3	<i>d</i> : 3
300	<i>b</i> : 0.37	<i>b</i> : 0.49	<i>b</i> : 0.65
	<i>m</i> : 0.54	<i>m</i> : 0.84	<i>m</i> : 1.19
	<i>M</i> : 0.54	<i>M</i> : 0.85	<i>M</i> : 1.2
	<i>s.d.</i> : 0.071	<i>s.d.</i> : 0.13	<i>s.d.</i> : 0.22
	<i>d</i> : 1	<i>d</i> : 0	<i>d</i> : 1
600	<i>b</i> : 0.38	<i>b</i> : 0.49	<i>b</i> : 0.61
	<i>m</i> : 0.49	<i>m</i> : 0.80	<i>m</i> : 1.13
	<i>M</i> : 0.50	<i>M</i> : 0.84	<i>M</i> : 1.15
	<i>s.d.</i> : 0.056	<i>s.d.</i> : 0.13	<i>s.d.</i> : 0.25
	<i>d</i> : 2	<i>d</i> : 1	<i>d</i> : 2

Table 5.46: Owl quadrotor: Performance of the automatically produced models when trained and tested with different sizes of data windows. *training size* indicates the length of the data windows used as tests during coevolution, while *testing size* indicates the size of the consecutive windows used in the computation of the $RMSE_{total}$. The letters *b*, *m* and *M* indicate the $RMSE_{total}$ of respectively the best, mean and median models in the set; *sd* indicates the standard deviation, and *d* the number of diverging models.

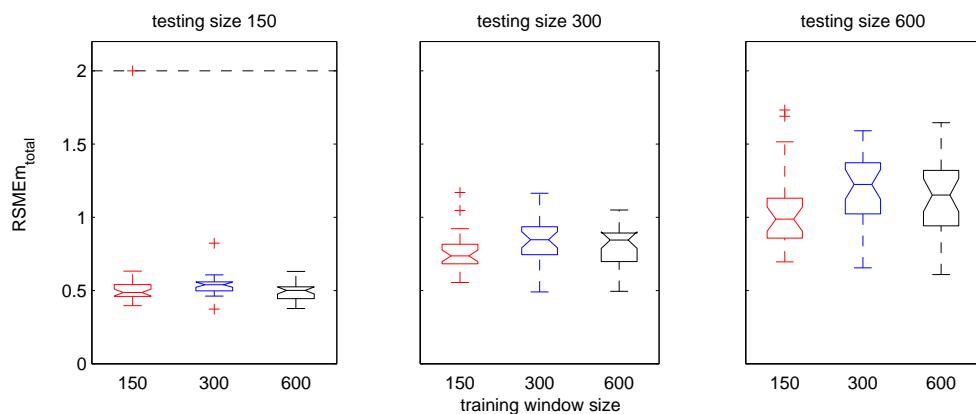


Figure 5.37: Owl quadrotor: $RMSE_{total}$ metric evaluated over different testing window sizes for the model training with different length for the data windows.

evolutionary algorithm, which will hinder the effectiveness of the search. Further work will be necessary to clarify this issue in more detail.

5.10 Summary

This chapter presented and analyzed a novel modelling algorithm based on coevolution; this represents one of the main contributions of the thesis.

In contrast to other work in the literature, the algorithm was specifically designed to make no use of platform specific knowledge; it was aimed at platforms that can be modelled as 6DoF rigid bodies.

As a result of this unconventional choice, we made clear during the design of the algorithm how the more traditional techniques like the one we examined in Chapter 4 become unviable. With our constraints, not only the parameters of the dynamic model, but also its structure, the identification of the relevant inputs and the input delays are part of the modelling task. The lack of platform specific knowledge and the fact that the platforms used are unrecoverable mean that it is not possible to define specific control inputs, nor to actively test the vehicles. Making an intelligent selection from available data becomes therefore an important step in the modelling.

To provide the ability to generate both the structure and the parameters of our systems, we based our algorithm on evolutionary computation techniques, and in particular on the latest findings in the coevolutionary field. The resulting algorithm searches the space of models for good performing candidates while also actively selecting the parts of the dataset that best discriminate between models.

Empirical tests performed on real data from four different vehicles confirmed that the algorithm performs well in practice, although because the methodologies are based on stochastic optimization, we can not *a priori* guarantee the quality and correctness of the models produced. Both in terms of repeatability and of error, the models obtained are satisfactory, although the quality may vary somewhat in each specific coevolutionary run.

Although we are primarily interested in a hands-free methodology, we recognize the usefulness of being able to investigate the models produced, and therefore we have chosen to use a symbolic regression approach to the search for our dynamic models. By focussing on rigid body models we can directly exploit the physics equations that govern translations and rotations in our systems. By combining symbolic regression and the rigid body

assumptions, our method is able to produce dynamical equations that directly predict the body frame accelerations in the vehicle. Since they are physically meaningful, such relationships can give us insight into the dynamics of the platform being studied. In fact for most of the platforms studied we have shown that, after reasonably simple manipulations, the equations produced are meaningful and often approximate exactly the same effects on which the first principles models are based. Importantly this property is shared by the great majority of the evolved models.

Of course, while developing and testing our algorithm we have also seen that the effectiveness of our approach can vary. In particular, the ATTAS platform demonstrated how our approach, which relies solely on data, can be undermined by data that does not contain sufficient information about the system. The direct dependence on the training data was also shown with the Owl platform where, due to the high level of noise in the data the evolved models had a tendency to drift away from the experimental data. Both these situations emphasise how the data collection and pre-processing steps are the keys to our method; this is obviously an implicit characteristic of any model that does not use domain knowledge.

Although we were able to identify general forms for the model equations, the fact that we had to manually rearrange some of the model equations leaves some room for improvement in the autonomy of our method.

The analysis of the effects of the training window size was also revealing in term of emphasizing the trade off that needs to be made between long range predictions and the cumulative effects of model error. Nevertheless, the results obtained show that a compromise can be found under which good models can be produced even in the presence of considerable noise, as in the case of the Owl platform.

An additional aspect related to neglecting the system noise is the fact that the models we evolved are deterministic, while the real platform (as we noted during the tests with the simulated quadrotors) does not have deterministic behaviour. We will see in the next chapter how an additive noise model can be used to complement the deterministic models, and to produce a better fit to the real system.

Chapter 6

Beyond Deterministic Modelling

In the previous chapter we described the abilities of one the main contributions of this thesis, a hands-free identification method based on coevolution. In Section 2.2.1 we discussed the assumption of negligible process noise and justified its adoption; the experimental results in the previous chapter show that this can indeed lead to good deterministic models.

However, real world systems are always affected by process noise, although to different extents, and even for the best of models some degree of difference from the real system will always be present¹. For example, in Section 5.8 after testing the Owl model in simulation, it was clear that although the motion and velocities were very realistic, the deterministic model did not capture the aerodynamic disturbances that characterize the real platform.

We discussed in Section 2.1 how, when evolving a controller with the aim of transferring it to the real platform, not only is a correct model of the platform necessary, but the variability of conditions that characterizes the real world also needs to be considered. In this chapter we look at a pragmatic yet principled way of taking such variability into account by modelling the differences present between the deterministic models obtained so far, and the real platforms under study.

In keeping with the spirit of this thesis, the key requirement of our methodology will be to obtain such representations both automatically, and without using any platform specific knowledge. Again, these are requirements that set this work apart from the overwhelming majority of the work in the literature.

¹As we explained in Section 2.2.1 we define any difference between the model prediction and the measured data as process noise.

6.1 A Stochastic Model

We again start with the Markovian assumption, but in this chapter we shift from the idea of producing a model that, given the current state and inputs at time t , predicts the state at time $t + 1$, to the idea of a model that, given the inputs at time t and a probability distribution for the values of the state at time t , returns a distribution over the values of the state at time $t + 1$. More concretely, using the space state formulation that we introduced in Section 2.2.1, we now consider the case in which the functions f and g of equation 2.13 (repeated here for ease of reading):

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_{t-D}, \boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (6.1)$$

$$\mathbf{y}_t = g(\mathbf{x}_t, \boldsymbol{\nu}_t), \quad (6.2)$$

are stochastic instead of deterministic. To be able to directly relate the measurement \mathbf{y} (i.e. the platform pose) to the model state, the state description of our platforms needs to be *extended* to include position (x, y, z) . See Sections 3.3.3 and 3.3.1 for how *extended* states are defined for the toy car and the quadrotor respectively.

As we saw in the case of a deterministic model, an appropriate way of expressing the functions f and g needs to be chosen. In addition to providing a probabilistic input-output mapping, the model needs to fulfil the same requirements as were set out in Section 5.2 about not using platform specific knowledge, having a reasonably low computational complexity for prediction, and (ideally) model transparency.

In practice we can approach the choice of model representation in two different ways. The first requires the use of representations that deal naturally with distributions, in contrast to those used in Chapters 4 and 5. The second way builds on the results obtained in Chapters 4 and 5, and is based on the idea of augmenting an available deterministic model with a suitable stochastic noise model.

The available stochastic model representation range from global techniques like Gaussian Processes [194] to Bayesian Neural Networks [168][143], to semi-local techniques like the Bayesian mixture of experts [25], and local techniques such as LWR, LWPR or RFWR (respectively Locally Weighted Regression [10], Receptive Field Weighted Regression [200] and Locally Weighted Projection Regression [242]).

All of these popular methods are based on the use of combinations of basis or kernel

functions, very often from the exponential family for mathematical tractability; as a consequence the models obtained are completely opaque and not amenable to interpretation. In addition, as noted and commented in Section 5.2, our setup differs from the standard supervised learning setup since the target outputs for training the models are not directly available; this makes the adoption of Bayesian techniques far from straightforward.

For some of the global techniques (e.g. Gaussian Processes), the computational complexity involved in using the model for prediction scales poorly with the training dataset size, making it unsuitable for situations such as our controller evolution in which there is a sizable dataset, and the models frequently need to be queried.

In the light of these considerations, the second option of augmenting existing deterministic models with a stochastic component sounds much more promising. Because the models obtained from our coevolution methodology are in the same form (differential equations) as the models based on first principles commonly used for dynamic systems, we can readily consider the types of noise models generally used in the system identification and filtering literature.

The first step in this direction is to rewrite equation 6.2 as the sum of the deterministic (f, g) and stochastic components (f_ω, g_ν) to obtain the following:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_{t-D}) + f_\omega(\mathbf{x}_t, \mathbf{u}_{t-D}, \boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (6.3)$$

$$\mathbf{y}_t = g(\mathbf{x}_t) + g_\nu(\mathbf{x}_t, \boldsymbol{\nu}_t). \quad (6.4)$$

With this notation, the functions f_ω, g_ν can be viewed as what in the regression literature are called residuals, since they represent the terms “remaining” after the learning of the deterministic model.

To learn the residuals, we could consider using one of the approaches mentioned in the beginning of this section (see [122] for an interesting example using Gaussian Process), but the problems of computational complexity still make those approaches impractical.

An assumption that helps to simplify the learning is to consider f_ω and g_ν as independent from the input and the model state. In other words the noise sources (i.e. the external disturbances and the difference between the model prediction and the measured data) should not change much in different parts of the dynamic envelope of the vehicle. This assumption is reasonable in the case of external disturbances since effects like air

disturbances can be considered as being unrelated to the vehicle state. In the case of the model error we know that, during evolution, training data is purposely selected to avoid biasing models toward specific parts of the dataset. In our setting this simplifying assumption appears reasonable, and so equation 6.3 becomes the following:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_{t-D}) + f_\omega(\boldsymbol{\omega}_t) \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (6.5)$$

$$\mathbf{y}_t = g(\mathbf{x}_t) + g_\nu(\boldsymbol{\nu}_t). \quad (6.6)$$

Deciding on the form of f_ω and g_ν requires platform specific knowledge, and even when this is available it is not a straightforward problem. In fact, the type of distributions chosen determine the difficulty of the problem of learning the parameters of f_ω and g_ν as well as its computational complexity². In order to simplify the mathematical treatment, it is common to assume f_ω and g_ν to be zero mean Gaussian distributions³; equation 6.5 therefore becomes:

$$\mathbf{a}_t = f(\mathbf{x}_t, \mathbf{u}_{t-D}) + \boldsymbol{\omega}_t \quad \mathbf{x}_{t_0} = \mathbf{x}_0 \quad (6.7)$$

$$\mathbf{y}_t = g(\mathbf{x}_t) + \boldsymbol{\nu}_t. \quad (6.8)$$

The choice of representing both the system noise ($\boldsymbol{\omega}$) and measurement noise ($\boldsymbol{\nu}$) as additive zero mean Gaussian distributions is a suitable compromise in our situation since on the one hand it is not platform specific, and on the other it is computationally cheap for both training and predicting. More explicitly we can write:

$$\boldsymbol{\omega} = \mathcal{N}(0, R) \quad (6.9)$$

$$\boldsymbol{\nu} = \mathcal{N}(0, Q), \quad (6.10)$$

where Q and R are the covariance matrices of the process and measurement noise respectively. Substituting 6.9 and 6.10 into 6.7 and 6.8 respectively, and making use of the

²Different form of f_ω and g_ν will differ in the number of parameters and for being or not tractable analytically, two determinant factor in determining how difficult is the learning.

³We give justification for such assumption later on in the section.

integration equations of the relevant platform as specified in Chapter 3 leads to:

$$p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{u}_t) = \mathcal{N}(\mathbf{x}_t; f(\mathbf{x}_{t-1}, \mathbf{u}_t), R') \quad (6.11)$$

$$p(\mathbf{z}_t | \mathbf{x}_t) = \mathcal{N}(\mathbf{z}_t; g(\mathbf{x}_t), Q), \quad (6.12)$$

in which we have adopted the notation commonly used in the filtering literature (e.g. [227]). Since we assume additive noise only on the predicted accelerations, the covariance matrix R' is chosen as:

$$R' = \begin{bmatrix} R & 0 \\ 0 & 0 \end{bmatrix}.$$

Although we cannot argue that all real world disturbances are Gaussian, it is noteworthy that the independent Gaussian noise assumption is widely adopted in the filtering literature ([227]) and has been shown in practice to be a suitable approximation in many practical engineering applications ([14]).

It is important to note that, since using the proposed model for predictions only requires us to draw a few samples from a Gaussian random number generator, this choice enables the economical use of a stochastic model for controller evolution in which the model is queried very often.

While the use of random noise to improve controller robustness and transferability is certainly not new in evolutionary robotics [175], our novel contribution lies in defining a principled and automatic method by which such a noise distribution is determined. Contrary to common practice, in which this is often a parameter left to the experience of the designer [117], we obtain the noise parameters from real experimental data. In addition, the fact that we followed a principled analysis to arrive at the noise model offers a much clearer understanding of the effects and implications of the choice. The next section concentrates on determining the parameters (i.e. variances) of the noise model ω .

By this point the reader will probably have realized that in this section we have simply given a more detailed explanation of the structure of the noise model introduced in Section 2.2.1; equation 6.7 is in fact equation 2.15.

The measurement function g which maps the extended state onto the predicted measurement \mathbf{y}_t will simply be the identity function in respect of the pose variables, and zero for any other state variable. In the case of the toy car with extended state $\mathbf{x}_t = [u, v, r, x, y, \psi]^T$,

equation 6.8 becomes:

$$\mathbf{y}_t = \begin{bmatrix} \mathbf{0}_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & \mathbf{I}_{3 \times 3} \end{bmatrix} \mathbf{x}_t. \quad (6.13)$$

Since it will be useful in the next section, we also report G , the Jacobian of g , which is obviously:

$$G = \begin{bmatrix} \mathbf{0}_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & \mathbf{I}_{3 \times 3} \end{bmatrix}, \quad (6.14)$$

and which does not depend on t since g is linear.

Since we already obtained the measurement noise parameters from the static experiments described in Section 3.1, we avoid having to learn them. This will reduce the search space and simplify the problem of learning the noise model parameters R .

6.2 The Algorithm

Having decided on the structure of the noise model, we now require a way to estimate the parameters of such a distribution from the collected data. Since we represent the noise as independent and Gaussian, we simply need to estimate the diagonal parameters of the covariance matrix R . We can now see that we are in the context of the FE methodologies of Section 2.2.1.

As explained in Section 2.2.5, we cannot rely on absolute prediction error or square error to train our models as we have done for the techniques presented so far, since those metrics would only gauge the state predicted by the model instead of considering the probability distribution associated with it. Since we are dealing with distributions, a natural metric is the probability

$$p(\mathbf{z}_{0:N} | \mathbf{u}_{1:N}, R), \quad (6.15)$$

which as we saw in Section B.4.2 is the likelihood of the N measured data points being observed given the noise parameters R and the inputs $\mathbf{u}_{1:N}$. If we can compute this likelihood we can search for the parameters that maximize it over the whole of the measurement dataset $t=0:N$:

$$R_{ML} = \arg \max_R p(\mathbf{z}_{0:N} | \mathbf{u}_{1:N}, R). \quad (6.16)$$

While as in Section B.4.2 we apply the Maximum Likelihood principle, we will see that

due to the different assumptions about the process noise, the cost function to be optimized is different, and depends on the uncertainty associated with the model prediction.

In this section, we make novel use of the approach presented in [3] by Abbeel *et al.* to compute R_{ML} . While Abbeel and colleagues were focussing on improving the estimation performance of the filter used in their robotic application, we are directly interested in the noise model obtained by the algorithm. In addition, we will explain how we have also modified the algorithm to make use of the more general Unscented Kalman filter instead of the usual Extended Kalman filter.

Our models (see equation 6.7) allow us to compute a distribution over the values of the state at time $t + 1$ given the inputs and state distribution at time t ; however to compute $p(\mathbf{z}_{0:N}|R)$ we need to be able to propagate this distribution forward in time⁴. By doing so, we will capture the long range input/output effects of our model, as well as any coupling between equations, since all dimensions of the model are used simultaneously for prediction.

Under the stated assumptions, a general algorithm suitable for propagating the extended system state forward in time is a Bayes filter which consists of two fundamental steps: prediction and update (respectively steps 2 and 3 in algorithm 1).

Algorithm 1 General algorithm for Bayes filtering, (adapted from [227]).

Input: $p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1}), \mathbf{u}_t, \mathbf{z}_t$

Output: $p(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t})$

```

1: for all  $\mathbf{x}_t$  do
2:    $p(\mathbf{x}_t|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t}) = \int p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})d\mathbf{x}_{t-1}$ 
3:    $p(\mathbf{x}_t|\mathbf{z}_{1:t}, \mathbf{u}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t})^5$ 
4: end for

```

In the prediction step, the system's model $p(\mathbf{x}_t|\mathbf{u}_t, \mathbf{x}_{t-1})$ and the distribution over the past state $p(\mathbf{x}_{t-1}|\mathbf{z}_{1:t-1}, \mathbf{u}_{1:t-1})$ are combined to predict the distribution over the current state. In the update, the measurement function $p(\mathbf{z}_t|\mathbf{x}_t)$ is used to incorporate the incoming measurement \mathbf{z}_t into the current state estimate.

While the Bayes filter algorithm is straightforward in principle, being no more than a recursive application of Bayes rule, its implementation is far from being so. The integral in step 3 is in general not trivial to evaluate without making further assumptions about the distributions involved and the dependencies between state inputs and measurements. In our case however we have already introduced all the assumptions we need to make the

⁴At this point in the description the reason might not be clear, but this difficulty will be resolved when we introduce the full filtering algorithm.

implementation of the Bayes filter algorithm feasible. These are the Markovian assumption (equation 2.3), which simplifies the dependencies between the current state and past inputs and measurements, and the assumption of an additive Gaussian independent noise model for both process and measurements. In these settings, a Kalman Filter and its variants are well known implementations of the Bayes filter algorithm.

In choosing the most appropriate type of Bayes filter for our setting, we need to take into account that no *a priori* knowledge of our models is available, and therefore the filter must allow for fully non-linear models. In addition, we want as usual to produce an automatic algorithm and therefore to avoid any step that requires intervention (e.g. any manual calculation).

To fulfil our requirements we have chosen to use the unscented Kalman filter algorithm [116] (UKF) since it is suitable for any choice of process model, while not requiring the computation of the Jacobians as would be needed for the EKF used in [3]⁶.

Within this filtering framework, not only is the way of propagating the state distribution forward in time well defined, but so are the relationships that allow the computation of all the distributions in the system.

Applying the chain rule to equation 6.15 we obtain

$$p(\mathbf{z}_{0:N} | \mathbf{u}_{1:N}) = \prod_{t=0}^N p(\mathbf{z}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:N}), \quad (6.17)$$

where, since we are in the process of deriving the expression that allows for the likelihood optimization, we have temporarily dropped the explicit dependency on the noise parameters Q to avoid any misunderstanding. We will reintroduce it later on in the derivation when focussing on its computation.

Now, each term in the product can be written as:

$$p(\mathbf{z}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:N}) = \int_{\mathbf{x}_t} p(\mathbf{z}_t | \mathbf{x}_t, \mathbf{z}_{0:t-1}, \mathbf{u}_{1:N}) p(\mathbf{x}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:N}) d\mathbf{x}_t \quad (6.18)$$

$$= \int_{\mathbf{x}_t} p(\mathbf{z}_t | \mathbf{x}_t) p(\mathbf{x}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:t}) d\mathbf{x}_t. \quad (6.19)$$

To go from 6.18 to 6.19 we used the facts that, given \mathbf{x}_t , \mathbf{z}_t does not depend on $\mathbf{z}_{0:t-1}$, nor on $\mathbf{u}_{1:N}$ as per equation 6.12, and the current state \mathbf{x}_t does not depend on future inputs

⁶For more details on the UKF and on filtering approaches used in robotics , see [227].

$$\mathbf{u}_{t+1:N}.$$

The first term in the integral is our observation model from equation 6.12, while the second term is the current state distribution computed by the UKF (see step 2 of algorithm 1). For the UKF this is a Gaussian distribution:

$$p(\mathbf{x}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:t}) = \mathcal{N}(\bar{\boldsymbol{\mu}}_t, \bar{\Sigma}_t). \quad (6.20)$$

Again we refer to [116] for details of how the predicted state mean $\bar{\boldsymbol{\mu}}_t$ and covariance $\bar{\Sigma}_t$ are computed.

Since in our case the measurement function g is linear, the integral in equation 6.19 can be computed exactly. Although it has fairly standard derivation, (since it is part of obtaining the basic Kalman filter algorithm), the computation of this integral is very lengthy; details can be found in [227] or in any other good textbook on Kalman filtering. The resulting expression is rather simple, and is familiar to those with knowledge of Kalman filtering:

$$p(\mathbf{z}_t | \mathbf{z}_{0:t-1}, \mathbf{u}_{1:N}) = \mathcal{N}(\mathbf{z}_t; g(\bar{\boldsymbol{\mu}}_t), G\bar{\Sigma}_t G^T + Q). \quad (6.21)$$

Remembering the form of g , we see that this expression simply says that the probability of the measurement \mathbf{z}_t given the previous measurements $\mathbf{z}_{0:t-1}$ is a Gaussian with mean and covariance obtained by selecting from the covariance predicted by the filter the rows and columns corresponding to the variables that represent the measurement. Since the quantities $\bar{\Sigma}_t$ and $\bar{\boldsymbol{\mu}}_t$ are obtained from the filter, we actually need to run the filter over the whole dataset; at each step t we will then be able to calculate equation 6.21. This is the exact counterpart of what we did in the case of the deterministic models when integrating the models forward in time.

It is interesting to note that since in step 3 of algorithm 1 the measurement \mathbf{z}_t is used to propagate the state forward in time (as in Figure 2.1 where both \mathbf{u} and \mathbf{z} are used to compute $\tilde{\mathbf{y}}$), the model prediction will not be able to diverge from the collected data. This avoids the problem of the predicted and real data becoming uncorrelated (see Section 2.2.4), which was among the practical factors limiting the maximum length of a training window.

As we saw already in Section B.4.2, working with the negative logarithm of the likelihood instead of using the likelihood directly makes for simpler mathematical expressions. Taking the logarithm (and disregarding any constant factor since it would not change the

parameters resulting from the optimization), we obtain:

$$LL(R) = \sum_{t=0}^N -\log |2\pi\Lambda| - (\mathbf{z}_t - g(\bar{\boldsymbol{\mu}}_t))^T \Lambda^{-1} (\mathbf{z}_t - g(\bar{\boldsymbol{\mu}}_t)). \quad (6.22)$$

Equation 6.17 therefore becomes:

$$R_{ML} = \arg \max_R \sum_{t=0}^N -\log |2\pi\Lambda| - (\mathbf{z}_t - g(\bar{\boldsymbol{\mu}}_t))^T \Lambda^{-1} (\mathbf{z}_t - g(\bar{\boldsymbol{\mu}}_t)). \quad (6.23)$$

where $\Lambda = G\bar{\Sigma}_t G^T + Q$. The dependency on R that we dropped for clarity earlier in the section is in fact still present in this equation through the state and covariance $(\bar{\boldsymbol{\mu}}_t, \bar{\Sigma}_t)$ predicted by the filter. The process noise model is fundamental for computing these quantities.

6.3 Optimization of Parameters

Having obtained the expression for the likelihood, we now can apply a numerical optimization scheme to compute the parameters of R_{ML} . In practice, evaluating the likelihood involves setting a value for R and running the UKF over the whole dataset. Because of the recursive formulation of the likelihood function, computing a closed form expression for its gradient with respect to the parameters (R) is not trivial. This makes any gradient based optimization infeasible and so a direct search method is preferred to obtain R_{ML} .

Any of the standard direct search method is suitable for the optimization of R_{ML} ; we chose a coordinate ascent algorithm (algorithm 2) for its simplicity of implementation and its convergence ability [1].

Empirical investigation on both the toy car and X3D datasets confirmed that the algorithm converges reliably within 50 iterations. Since the likelihood function is not convex, we need to account for the possibility in which the algorithm might converge to a local minimum. Restarting the algorithm from different initial positions in the search space is a pragmatic and effective technique that obviates this problem ([103]). In our case the values of R were randomly drawn from a uniform distribution $\mathcal{U}[0, 1]$. R_{ML} was selected as the matrix that produce the highest likelihood among 10 restarts⁷.

Given the parameters R_{ML} , the complete model (the deterministic part plus the noise

⁷We considered 10 runs sufficient since we empirically found the parameters obtained to be consistent across restarts

Algorithm 2 Coordinate ascent algorithm, (adapted from [3]).

Input: $R, LL(R), \alpha = 0.1$

Output: R_{ML}

```

1: while  $iterations < MAX$  do
2:   for all  $r_{jj} \in R$  do
3:     if  $LL(R) < LL(R\{j, j, (1 + \alpha)r_{jj}\})^a$  then
4:        $\alpha = 1.1\alpha$ 
5:        $r_{jj}^b = (1 + \alpha)r_{jj}$ 
6:     else
7:       if  $LL(R) < LL(R\{j, j, (1 - \alpha)r_{jj}\})$  then
8:          $\alpha = 1.1\alpha$ 
9:          $r_{jj} = (1 - \alpha)r_{jj}$ 
10:      else
11:         $\alpha = 0.5\alpha$ 
12:      end if
13:    end if
14:  end for
15: end while

```

^a $R\{i, j, k\}$ indicates a matrix R for which the element at position (i, j) is substituted by the value k .

^b r_{ij} indicates the element at position (i, j) in the matrix R .

part) can be used to generate sample trajectories from the model's distribution. First, from an initial state at time t , the deterministic equations of the model are used to compute the accelerations resulting from the control inputs. Then we add to those values a sample from the Gaussian distribution representing the process noise ω . Finally, the resulting acceleration is integrated one step forward in time to produce the new state vector at time $t + 1$. Every trajectory generated will be slightly different from any other, and the further time progresses, the more the trajectories will differ. The distribution therefore tends to spread as time passes, and the sampling of the spatial density therefore becomes more and more sparse.

Repeatedly sampling trajectories in this way allows the construction of a distribution that indicates which are the future state values that the learned model indicates are more probable given the state at time t and the control inputs. We will use this approach to build the distributions used in the following sections. Thanks to this feature of the FE methodologies, the model can in practice be propagated over the full length of the dataset.

6.4 Experiments

In this section we see the application of the algorithm just discussed to the toy car and the X3D. This is an interesting choice of platforms since it shows how our noise tuning method

performs when applied to vehicles with different dynamics and degrees of freedom.

Although the approach could be extended without any change to the Owl or ATTAS platforms; due to space limitations we restrict our treatment to the X3D and the toy car since they are the two platforms for which we will automatically produce controllers in Chapter 7.

6.4.1 Toy Car

We start with the toy car platform, using the best of the models developed in the previous chapter, the *carGPbest* model.

We first train our noise parameters using the same dataset (**datasetcar_6**) that was used to evolve the model. Since in the case of the toy car the dynamic model consists of three equations which predict the accelerations a_x, a_y and α_z , three noise parameters are needed to determine the diagonal elements of R . The resultant model is called *carAccGPbestNoise*.

To obtain a qualitative understanding of how our noise model compares with the variability present in the real toy car, we did a simple experiment with the real car that involved repeatedly recording the trajectory resulting from the same sequence of open loop manoeuvres. We chose a short sequence containing a balanced mixture of commands with the car accelerating, slowing down and turning in each direction.

The set of throttle and steering commands were delivered to the car at pre-specified time intervals:

$$\begin{cases} u_{th} = 0, u_{st} = 0 & 0 \leq t < 20 \\ u_{th} = 1, u_{st} = 0 & 20 \leq t < 52 \\ u_{th} = 0, u_{st} = 1 & 52 \leq t < 84 \\ u_{th} = 1, u_{st} = -1 & 84 \leq t \leq 118, \end{cases} \quad (6.24)$$

where t is expressed in time steps⁸.

After each run the car was repositioned as closely as possible to the same location within the tracking area and was also pointed in the same initial direction. To ensure that an identical sequence of commands was used every single time, an automatic software routine was used to issue the commands. The commands were recorded along with the

⁸The uneven lengths in time of the actions is the result of a coding mistake; initially we had planned to carry out every action for 25 consecutive timesteps. Although it looks awkward, the error does not compromise our analysis.

car trajectories so that it was possible to verify post-hoc that the car actually received the same control inputs during each of the 30 repetitions⁹.

For the same starting state¹⁰ and control inputs, we computed 300 instances of the model state trajectory using the *carAccGPbestNoise* model¹¹. A length of 118 steps was used for each trajectory to match the duration of the experiments done with the real car (see equation 6.24). Table 6.1 shows the standard deviations of the noise obtained from the optimization procedure and used in the *carAccGPbestNoise* model.

Parameter	Value
σ_{ax}	0.077 m/s^2
σ_{ay}	9.50 m/s^2
σ_{az}	4.79 rad/s^2

Table 6.1: Noise values used in the *carAccGPbestNoise* model.

To qualitatively understand what types of trajectories the *carAccGPbestNoise* model produced, we randomly selected 30 trajectories out of the 300; they are plotted in Figure 6.1.

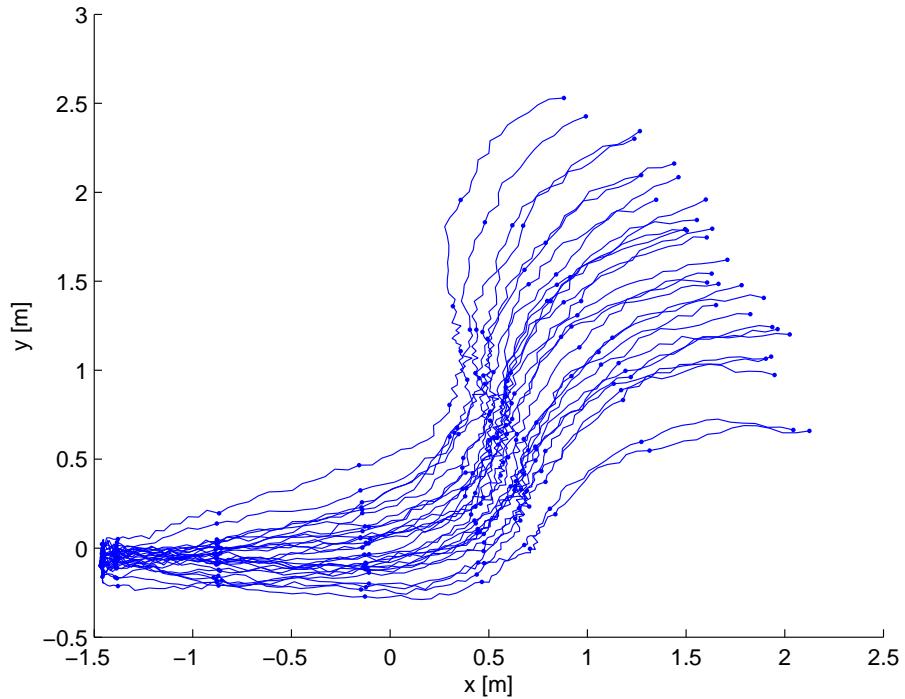


Figure 6.1: Toy car: 30 sample trajectories simulated with the *carAccGPbestNoise* model. The dots on the trajectories are equally spaced in time (52ms).

⁹We chose 30 repetitions as a compromise between producing a representative set of trajectories and restricting the duration of the experiment.

¹⁰The same position and orientation, and with velocities equal to zero.

¹¹The number of instances chosen is not critical, and depends on the duration of the trajectory, since as we explained the distribution gets sparser as time increases. In practice, a few hundred samples showed itself to be sufficient in our experiments.

Although the length of the simulated run is only about $6s$ we can see how the model predicts quite different trajectories which diverge as the time progresses. In particular it is clear how the two turns in the trajectories change in radius. The noise model produces trajectories that are more irregular than one would probably expect. We think that the optimization might be overestimating the level of noise to compensate for the fact that the real noise does not exactly match our assumptions.

To understand which positions in space were occupied more often by the model during the 300 simulated trajectories, we constructed a simple histogram. A regular grid was first overlaid onto the trajectories produced by the model, and then for each of the grid cells the number of trajectories crossing it were counted¹². Using colours to represent the trajectory count for each of the cells, we obtain the occupancy grid plotted in Figure 6.2, on which

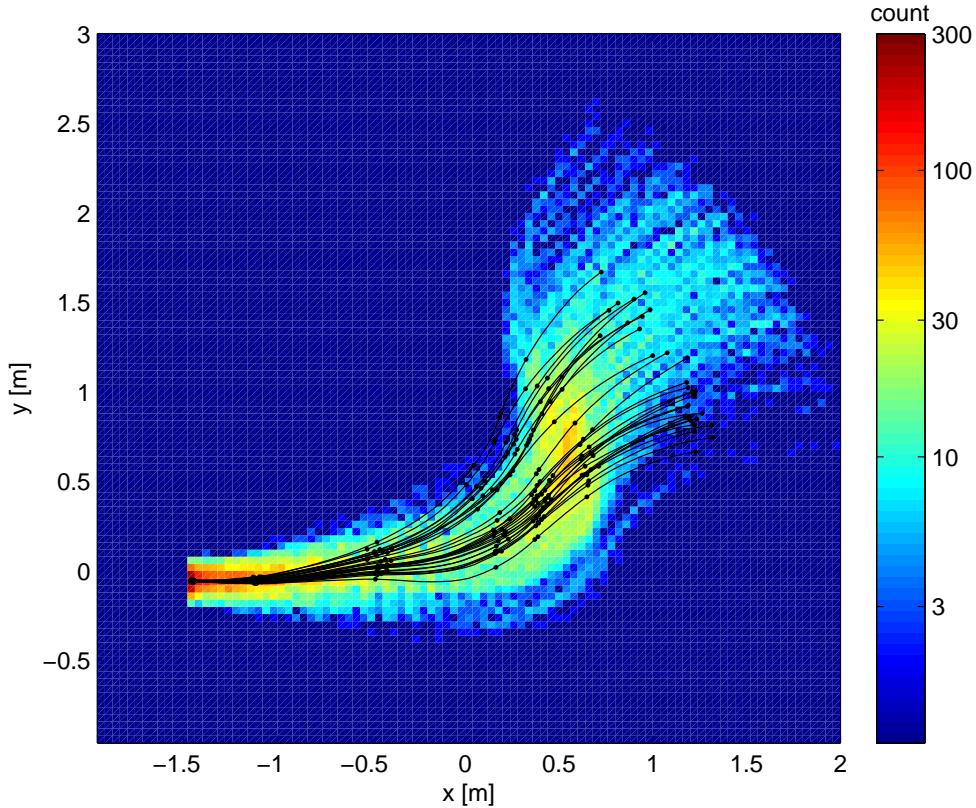


Figure 6.2: Toy car: real and *carAccGPbestNoise* model trajectories. The real trajectories obtained from 30 repetitions of the task (black lines) are superimposed onto the distribution obtained from 300 instances computed from the *carAccGPbestNoise* model subject to the same input and starting state. Each cell of the grid used to compute the distribution has size $0.04m$ by $0.04m$, the colour represents the number of trajectories passing through each specific cell in the grid. The dots on the real car trajectories are equally spaced in time ($52ms$).

¹²For the grid we used a cell size smaller than the car width, namely $0.04m$ by $0.04m$.

are superimposed the real car trajectories in black.

The first thing in the plot requiring comment is certainly the 30 trajectories recorded from the real car. As expected, as the car moves away from the starting point the trajectories diverge as we saw with the simulated trajectories (Figure 6.1). What is most interesting is the fact that two different sets of trajectories appear. The two sets appear to be produced by a different response of the car to the first of the two steering commands, with some trajectories having a larger turning radius than others. We suspect the origin of this to be the electromagnetic steering system of the car which is quite worn and sometimes fails to turn the wheels fully. Since the steering has only three settings, full left, full right and neutral, steering to less than the full extent ends up having quite an impact on the car's behaviour.

Looking now at the distribution of positions generated by the model, we see that the real car trajectories overlap the area of the distribution marked by a high histogram count. The assumption of independent Gaussian noise we made for the process is too restrictive to allow for learning that the car trajectories tend to have a bimodal distribution. We knew already that our assumptions were a simplifying compromise, so what is really important is to see that the distribution of our model covers both sets of trajectories exhibited by the real car. It is also clear that the distribution produced by the model also covers cases in which the model might turn more or less sharply than the real car generally does.

Undoubtedly the simple noise model adopted has its shortcoming but overall produces a distribution that is a superset of the recorded real trajectories. When used to evolve controllers such a model should simulate the variability of behaviour typical of the real world eliciting the evolution of more robust controllers.

6.4.2 X3D

To train the model of the X3D quadrotor we used a method similar to that used in the previous section. Starting from the *X3DAccGPbest* model obtained in Section 5.7, we first trained the parameters of the noise covariance, again using the data from the training dataset `datax3d_3` (as with the coevolutionary modelling). In this case the model is six-dimensional so the number of noise parameters required is also six.

The noise parameters were initialized randomly, as with the toy car, and the same random restart procedure for the optimizer was used. The resultant model is called

Parameter	Value
σ_{a_x}	0.011 m/s^2
σ_{a_y}	0.012 m/s^2
σ_{a_z}	0.435 m/s^2
σ_{α_x}	3.48 rad/s^2
σ_{α_y}	3.41 rad/s^2
σ_{α_z}	1.15 rad/s^2

Table 6.2: Noise values used in the *X3DAccGPbestNoise* model.

X3DAccGPbestNoise. Table 6.2 shows the noise parameters obtained from the optimization procedure and used in the *X3DAccGPbestNoise* model.

In the case of the X3D helicopter, repeatedly running a set of pre-specified open loop control manoeuvres as was done for the toy car is in practice not feasible. Due to the inevitable air disturbances, without closed loop control the quadrotor is unable to maintain either a fixed position or a fixed attitude, and therefore it is not possible to start the helicopter from exactly the same state for each repetition. Running a predefined sequence of open loop manoeuvres with the quadrotor is also very risky since relatively small difference in the starting state (e.g. the attitude) can lead to very different trajectories, some of which might direct the helicopter into the ground or against the wall of the flying arena. Both of these are possibilities that we are keen to avoid, since they are likely to heavily damage the platform.

Since sampling from our model is not a problem, we can instead see how the model compares with the real X3D in randomly selected parts of the already collected flight trajectories. We therefore selected at random three windows of data 50 time steps in length from the validation dataset `datax3d_2`. For each of the windows, we set the model state to the starting state of the window from the dataset, and then we use the model (both the deterministic and stochastic parts) and the input sequence to make acceleration predictions; we then integrate the predictions as was done for the toy car. For each of the three windows, 300 runs were simulated.

The X3D model is six dimensional, so instead of a single histogram we need to build three. For all the three windows the angle ψ is almost zero, and as a consequence a direct association exists between linear positions and angles (i.e. θ influences x and ϕ influences y). For this reason we chose to display in the same plot variables that are associated.

Using the technique, described for the toy car, we produced one histogram for x and θ , one for y and ϕ and one just for z and ψ . Those histograms can be seen in Figure 6.3; the

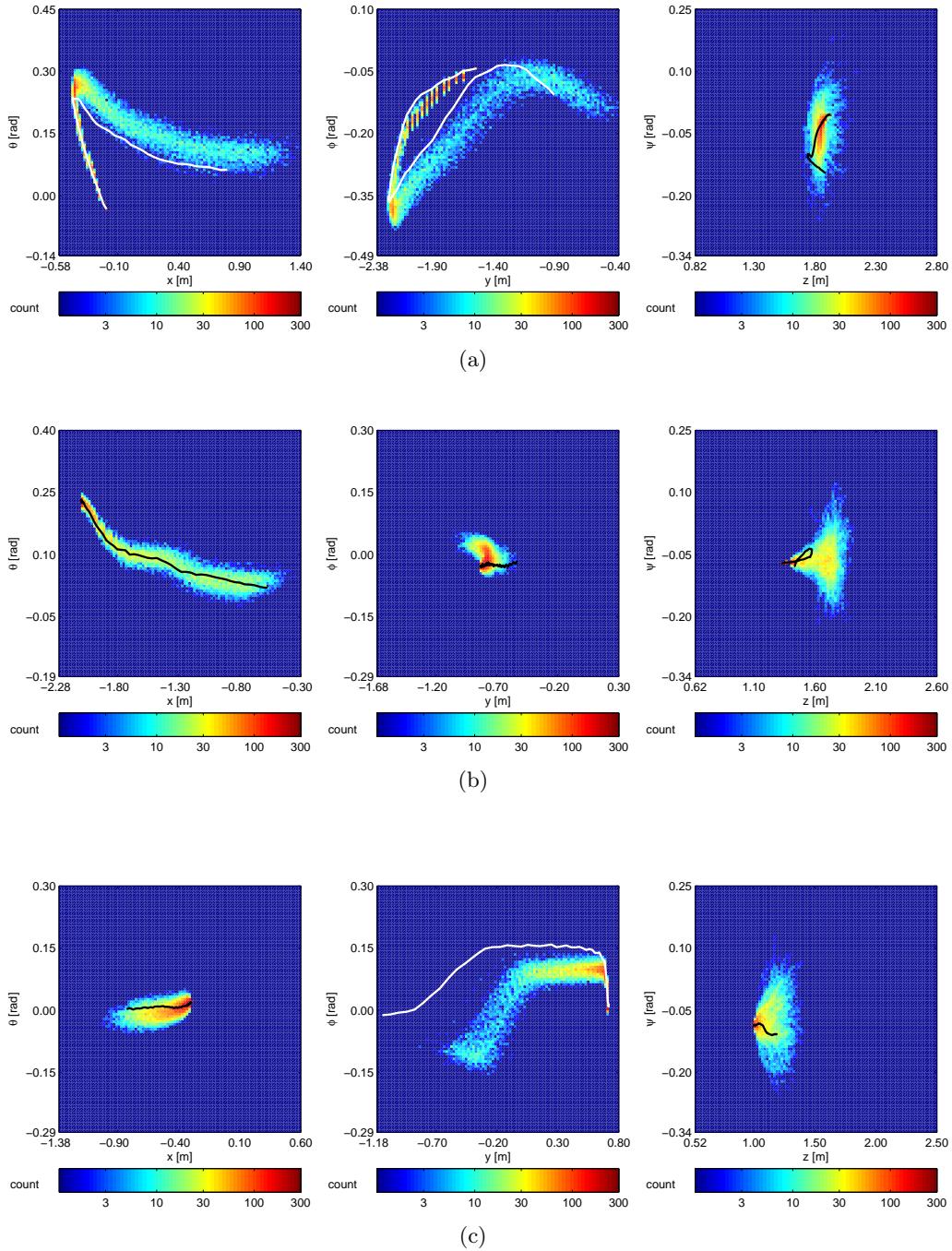


Figure 6.3: X3D, actual versus *X3DAccGPbestNoise* model trajectories. Three short windows 6.3(a), 6.3(b), 6.3(c) are selected at random from the validation dataset and the actual trajectory (continuous line) is superimposed on the state distribution computed from 300 runs of the *X3DAccGPbestNoise* model subject to the same input and starting state. Each cell of the grid used to compute the distribution has size $0.02m$ by $0.006rad$; the colour represents the number of trajectories passing through each specific cell in the grid.

actual trajectory of those variables calculated from the collected data is also superimposed on the obtained distributions¹³.

The three windows depicted in Figure 6.3 show three quite different manoeuvres: in Figure 6.3(b) a pitch input makes the helicopter move forward; in Figure 6.3(c) a roll input produces a sideways translation; and Figure 6.3(a) shows a combined pitch and roll action. In all three windows the heading and altitude of the flying machine do not change appreciably.

As before, the distributions produced from sampling the stochastic model fan out from the starting point as the trajectories progress. Both the windows in Figure 6.3(a) and Figure 6.3(b) show the real trajectory of the X3D overlapping the distribution built from sampling the model trajectories. For window (b) the real trajectory is mostly located in the very probable areas of the model distribution, while the second part of the trajectory in window (a) covers an area deemed less probable by our sampling of the model trajectories.

In the third window in Figure 6.3(c) we see an example in which the real trajectory of the X3D quadrotor deviates noticeably from the trajectories obtained from the model. In particular, it seems that the simulations are underestimating both the angle ϕ and (as a consequence) the motion on the y axis. This is a situation in which the deterministic part of the model behaves very differently from the real platform, producing an error larger than is accounted for by the noise model. Given the good performance that we measured in the previous chapter for the deterministic model, we feel that we should consider this a window in which the model performs unusually poorly.

In general, the augmented model $X3DAccGPBestNoise$ seems able to cover the variability in behaviour demonstrated by the real X3D platform, but nevertheless there are situations in which some of the dimensions are poorly predicted.

6.5 Summary

In this chapter we brought the idea of process noise into our modelling approach. Its function is to model the uncertainty present in the system, and also to allow for the discrepancies that will inevitably exist between our models and the real systems, a feature that we will exploit in the next chapter to produce robust transferrable controllers.

¹³Either a white or a black continuous line was used to denote the real X3D trajectory, depending on whichever ensured better readability.

We have shown how our automatically derived models can be readily extended to include process noise, thanks to the standard state space form in which our models are expressed. With the aim of simplifying the problem of training the noise model, but also, more importantly, of producing a computationally cheap noise description, we made purely pragmatic assumptions of Gaussianity and independence for the process noise distribution.

Through a novel application of results from the filtering literature, we created an algorithm that is both automatic, and does not use platform specific knowledge, thus meeting the two essential requirements of our automatic modelling framework. Contrary to most common practice in evolutionary robotics, our algorithm is able to determine the appropriate level of model noise solely from experimental data, removing the need for human expertise to set those parameters.

Tests on validation data have shown the ability of our enhanced models to reproduce the levels of variability in the dynamics that exists in our real world platforms. However, we identified some situations in which the noise model shows an inability to fully capture the differences between the real platforms and the models.

The full potential of our approach will only be determined by using the evolved model to develop proficient controllers, and then testing the extent to which the controllers transfer successfully to the real platforms.

Chapter 7

Automatic controller design

In this chapter we move to the second part of our investigation and look at the question of automatically designing controllers based on the models obtained automatically in the previous chapters (Chapter 5 and Chapter 6)¹.

Time constraints limited our study of automatic controller design to the toy car, the X3D quadrotor and the Autopilot helicopter simulator. Those platforms were chosen since they represent platforms with very different dynamics and therefore will require different solutions for the problem of control. Based on prior experience in manually designing controllers for aircraft platforms [56] we judged the design of a control system for a linear platform such as the ATTAS to be of little interest, especially given the fact that we would not be able to test the obtained controller on the real aircraft. In the case of the toy car and of the X3D we do have the possibility of testing and analysing how the controllers transfer to the real platform - a type of test that as we saw in Section 2.1.1 was crucial for validating the automatically produced models.

For each of the three platforms considered we will compare a controller designed by an expert (making extensive use of platform domain knowledge) with two or more general controllers which are effectively platform independent. This approach will give us good grounds for understanding how far our idea of limited domain knowledge can be pushed.

In Section 2.1 we gave an overview of the many representations that have been used in the literature for controllers suitable for optimization using evolutionary methods. From the many possibilities, we decided to use controllers based on neural networks since they have been shown to deliver good performance in a variety of real world control problems

¹With the exception of the Autopilot simulator.

([80, 172]). In addition, neural networks are suitable for systems with both discrete (e.g. our toy car) and continuous control inputs (e.g. our quadrotors and Autopilot helicopter simulator) meaning that the same control structure could be used with all our three test vehicles. This generality is a key feature that makes neural networks well suited to our approach. However, we are not suggesting that other representations could not be used; this was not possible at present due to time constraints, but we are likely to consider at least some of the main alternatives to neural network controllers in future research.

Since the different platforms require different test procedures, it is more natural to address each of the platforms separately in this chapter, rather than following the structure used so far of introducing a technique and then applying it to the different platforms.

Section 7.1 examines the toy car, Section 7.2 deals with the X3D quadrotor, and Section 7.3 presents the Autopilot helicopter simulator. The analysis and interpretation of the results is described in Section 7.4.

7.1 The Toy Car

Given its lower number of dimensions and the fact that is the simplest of our platforms to control, we will deal first with the toy car.

7.1.1 Task

As noted in Sections 1.2 and 2.1, we are particularly concerned with the design of controllers that fulfil a well specified task using the same sensor and control inputs commonly used to solve such control tasks using more orthodox techniques.

We note again that we allow the use of task specific knowledge (i.e. what sensor inputs are needed to solve the task), but avoid the use of platform specific knowledge.

In line with this, an interesting and very useful task to be accomplished by a car (or any vehicle) is to be able to follow a path defined in terms of waypoints that are specified in advance. As an example of the usefulness of such a control algorithm we can consider the state of the art approaches to autonomous navigation ([228, 8]), in which a path following module sits at the bottom of a stack of well engineered algorithms designed to drive (or fly) the vehicle intelligently. In such systems, a localization and mapping module provides a map of the environment and the position of the vehicle within it, a deliberative module

determines a goal location to be reached next, and a path planning algorithm finds a feasible path to the goal in the form of a trajectory or a set of waypoints. The type of controller that we aim for here would then be able to translate such a set of waypoints into the appropriate series of command inputs to be given to the vehicle.

While the outputs of the controller are obviously the same control variables as were used for the car in Section 3.3.3, ($\mathbf{u} = [u_{th}, u_{st}]$), the inputs to the controller will be the state of the car and information relating to the next few waypoints in the path. Since we want to obtain a controller for a generic path, the inputs to the controller are distances and angles in the frame of reference of the car. In this way the behaviour of the controller will be invariant to the absolute location and curvature of the path.

In Figure 7.1 we show the car following a path defined by a series of waypoints. Given a waypoint k , the path information is represented by the distance from the waypoint d_k , the cross track distance d_k^\perp and the angle to the track direction δ_k . It is obvious how the first is computed; the second is simply the orthogonal distance of the car's centre from the path segment between waypoints $k - 1$ and k , and measures how much the car is off track; finally, the angle δ_k is the angle between the car's heading and the track segment, and measures the alignment of the car with the predefined path.

The state information is obtained directly from the models, (these were learned automatically in Chapters 5 and 6) and is simply the car's forward, lateral and angular velocities

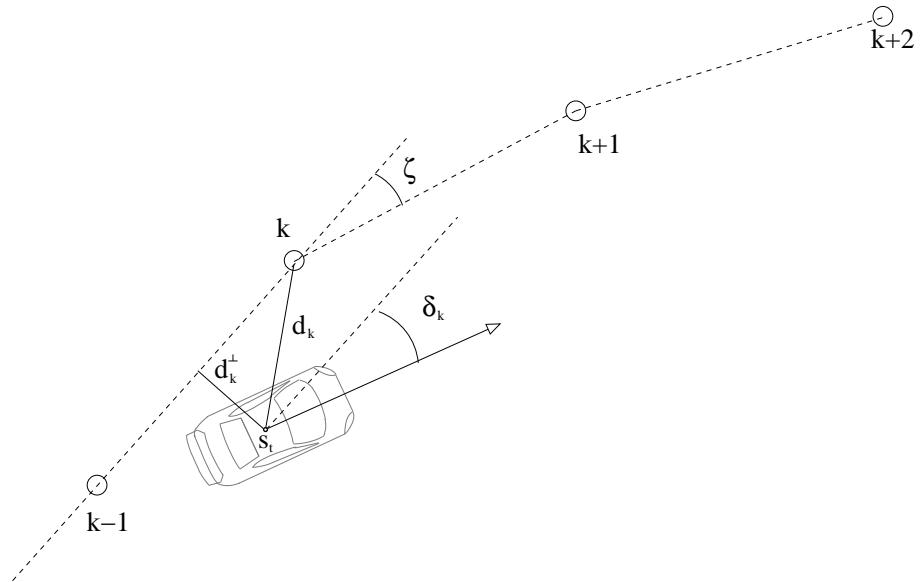


Figure 7.1: Track waypoint information. Distance and angular information from current and up-coming waypoints are considered.

(u, v, r) . As we will see in more detail in Section 7.1.2, we will consider several different type of controllers, and depending on the type of controller, we may pass it information from the current waypoint only (i.e. waypoint k), or may add information about the next two future waypoints.

Since the aim of the exercise is to produce controllers able to follow novel and unseen paths, and not just paths on which they have been trained, we use randomly generated paths when evolving controllers. This will stop the controller from learning to exploit any track specific feature(s).

Tracks are generated as a list of concatenated parts drawn randomly (with replacement) with equal probability from the set $\mathcal{P} = \{\text{straight}, \text{right curve}, \text{left curve}\}$. Each part is made up of a variable number of consecutive fixed length segments which have a waypoint at each end; the distance between two waypoints is therefore fixed. The straight parts are built of an integer number of segments chosen from the set $\mathcal{N}_s = \{1, \dots, 10\}$ to allow for different lengths. For the curved parts, the number of segments is drawn from $\mathcal{N}_c = \{5, \dots, 15\}$ and the angle ζ between the directions of two consecutive segments (see Figure 7.1) is selected randomly for each curve $\zeta = \mathcal{U}[0.02, 0.4]\text{rad}^2$. This allows the production of curves of different length and radius. The total number of parts in each path is fixed at 200, which ensures that the model cannot cover the full length of the path in the allotted time, even in the case of a straight path.

The parameters for the distributions used to generate the random path were selected empirically so that with a segment length of $0.25m$ the paths produced were qualitatively similar to the trajectories covered by the car when collecting the data used for the modelling stage. Five random samples chosen to show examples of right and left curves of different radius as well as (almost) straight trajectories are shown in Figure 7.2.

We noted in Section 3.1.3 that by comparing on-line and off-line recordings of the data coming from the tracking system we were able to measure the time necessary for computing the vehicle's position and making it available to our controller. Since when testing our controllers on the real platform this delay will inevitably be present, we need to model it within the setup of the evolutionary task. To do so, the state produced by the learned car model is buffered, and at every time interval t the controller is fed what is in fact a delayed state. Since the time step used by our model is $\Delta_t = 40ms$ and the

²The angle ζ is chosen positive or negative depending on whether the turn is right or left handed.

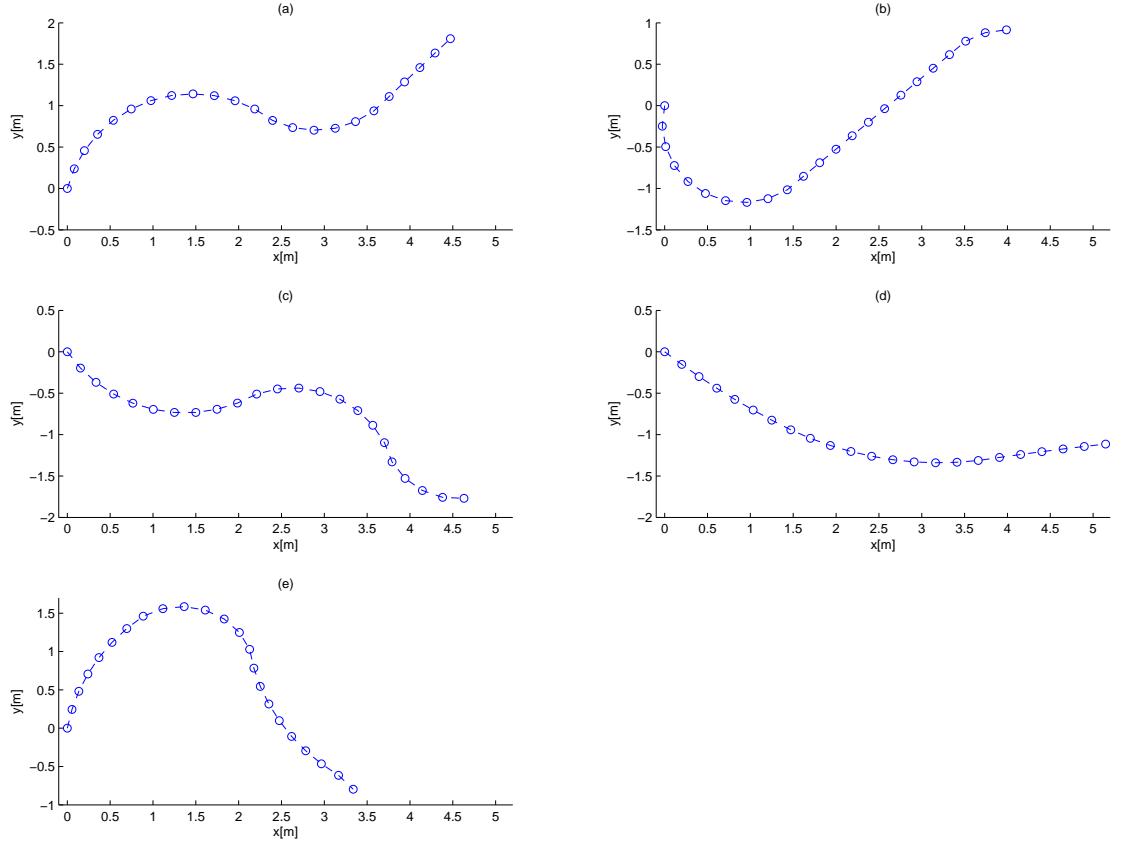


Figure 7.2: Some randomly generated paths of the type used for training controllers.

delay of the tracking system is $\Delta t_{MOCAP} = 140ms$ this corresponds to delaying the state information by 4 time steps³.

7.1.2 Controllers

In the case of the toy car, three different types of controller were considered; the first one was a hand crafted controller based on domain knowledge that would serve as a baseline for our approach, while the remaining two were more general controller structures based on artificial neural networks.

In both the neural network and evolutionary robotics literature, many varieties of neural network topologies and activation functions have been explored (see [158] and [255] respectively for two excellent if somewhat dated summaries). In order to simplify our analysis, we will limit ourselves to the use of relatively simple feed forward and Elman type recurrent networks (see details in the following section). Although we recognize that (time permitting) one could also consider other more exotic neural network variants, we

³140ms corresponds to 3.5 time steps; to obtain an integer number we rounded it up to 4 to avoid making the problem simpler by reducing the delay to 3 timesteps.

do not believe that our analysis is significantly limited since it can be shown that the selected networks, (with appropriate weights) can at least approximate the same functional relationships as are implemented by each of the hand crafted controllers.

The Stanley Domain Specific Controller

Although various control schemes have been used for the control of car-like vehicles, for our test we have chosen a simplified version of the controller used in Stanley, the autonomous car that won the DARPA Grand Challenge described in [228] by Thrun *et al.* We chose this controller since it was explicitly designed to be robust, and is therefore likely to give reasonable results for a toy car which is certainly different from a full size car. A second motivation is the fact that the basic formulation of this controller is particularly intuitive and so can be easily adapted to our situation.

Thrun and colleagues first designed a base controller focussing on a kinematic model of the car, and then augmented it to take into account the effects of dynamic variations. Given a continuous path, the base controller is designed to control the steering and throttle independently. The steering control applied is proportional to the angle between the direction of the car and the path (δ_t), and it is also to a first approximation proportional to the cross track error d_t^\perp divided by the forward speed u_t . These two contributions should respectively align the car with the track and reduce the cross track error to zero. More specifically:

$$u_{st}(t) = \begin{cases} \delta_t + \arctan \frac{K_{st} d_t^\perp(t)}{u(t)} & |\theta_t + \arctan \frac{K_{st} d_t^\perp(t)}{u(t)}| < u_{st}^{max} \\ u_{st}^{max} & (\delta_t + \arctan \frac{K_{st} d_t^\perp(t)}{u(t)}) \geq u_{st}^{max} \\ -u_{st}^{max} & (\delta_t + \arctan \frac{K_{st} d_t^\perp(t)}{u(t)}) \leq u_{st}^{max}, \end{cases} \quad (7.1)$$

where u_{st}^{max} represents the maximum value for the steering command, and the arctan function is used to provide a smooth transition to the maximum steering input. The constant K_{st} allows the influence of the cross track error to be adjusted⁴.

In [228] a target speed was defined⁵ along the continuous path and a PI control scheme was used to adjust the instantaneous forward speed $u(t)$ to the desired speed $u_d(t)$. More

⁴For the meaning of the symbols the reader should refer to Figure 7.1 paying attention to the fact that the subscript k is here replaced by t since the equation is continuous in time instead of discrete

⁵The speed profile was computed based on limits known in advance, and also on the real time assessment of the terrain roughness.

precisely:

$$u_{th}(t) = K_{th,p}(u(t) - u_d(t)) + K_{th,i} \int K_{th,p}(u(t) - u_d(t)) dt, \quad (7.2)$$

where $K_{th,p}$ and $K_{th,i}$ are respectively the proportional and integral constants for tuning the controller.

With an intuitive understanding of this controller we set out to derive from it a version that would be suitable for our car, which differs from Stanley in that it allows only for discrete control inputs. In the case of the steering control, we essentially retained the formulation of equation 7.1, while introducing a constant K_{δ_k} to allow for scaling of the control output. We also introduced a discretization to produce the left, neutral, and right commands (respectively -1 , 0 and 1). We therefore obtained:

$$u_{st,k} = \begin{cases} 1 & (K_{\delta_k} \delta_k + \arctan \frac{K_{std}^\perp}{u_k}) \geq \frac{1}{3} \\ 0 & |K_{\delta_k} \delta_k + \arctan \frac{K_{std}^\perp}{u_k}| < \frac{1}{3} \\ -1 & (K_{\delta_k} \delta_k + \arctan \frac{K_{std}^\perp}{u_k}) \leq -\frac{1}{3}. \end{cases} \quad (7.3)$$

For the throttle we adopted a PD scheme based on the current speed u_k and the distance from the current waypoint d_k . As with the steering control, we also applied a discretization of the output:

$$u_{th,k} = \begin{cases} 1 & (K_{th,d} u_k + K_{th,p} d_k) \geq \frac{1}{3} \\ 0 & |K_{th,d} u_k + K_{th,p} d_k| < \frac{1}{3} \\ -1 & (K_{th,d} u_k + K_{th,p} d_k) \leq -\frac{1}{3}. \end{cases} \quad (7.4)$$

Given such control laws, the evolutionary training should discover the values of the control constants that produce suitable switching between the discrete control commands. However, since this is an adaptation of a continuous controller, we do not expect excellent performance from it but merely a baseline for comparison.

At the start of the optimization process, the parameters of the model are initialized to random values drawn from a zero mean Gaussian distribution $\mathcal{G}(0, 0.1)$.

Monolithic Recurrent Network

While the controller of the previous section is clearly based on a good understanding of the behaviour of a car, our real concern is with more general control approaches that are platform independent.

The dynamics of the car has a significant effect on the way the vehicle can be controlled, in that the actions of changing speed, turning and stopping all require a non-negligible amount of time. For this reason, although the model is Markovian, it might in fact be easier to train a controller that is able to use past state and control values. With this aim, we chose to use a recurrent network instead of a standard feed forward neural network. While still being simple, it offers potentially better performance by being able to retain and exploit past information by means of its recurrent neurons.

Since we assume no platform specific information, we consider a very general fully connected two layer topology (see Figure 7.3) with an arctan activation function. Although such a structure is commonly known as an *Elman Network* since it was introduced by J. Elman [66], we will usually refer to it as a RMLP (Recurrent MLP).

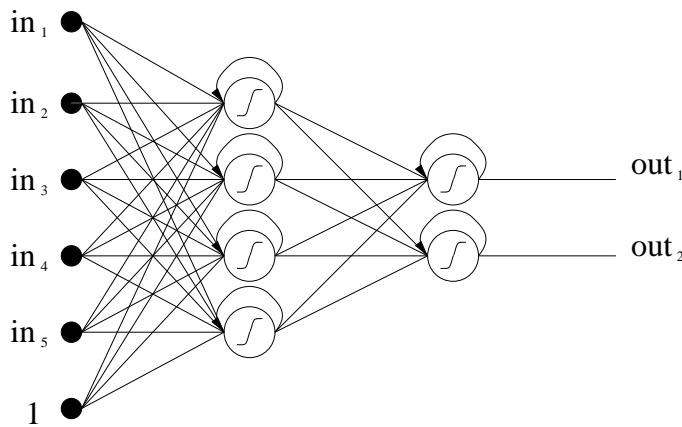


Figure 7.3: Structure of a general fully connected two layer recurrent neural network with 5 inputs, 2 outputs, and a 4 neuron hidden layer.

The inputs to the network are the state of the car (i.e. u, v, r), and the distance (absolute and cross-track) and angular information from the current (k) and the next two waypoints ($k + 1, k + 2$). The aim is to provide the controller with information about its current error, and also about what lies ahead in the path to see if evolution is able to use this information in order to achieve better performance. Including the necessary bias input, the network has 13 inputs:

$$\text{inputs} = [u, v, r, d_k, d_k^\perp, \delta_k, d_{k+1}, d_{k+1}^\perp, \delta_{k+1}, d_{k+2}, d_{k+2}^\perp, \delta_{k+2}, 1].$$

Based on experience with a small number of sample runs, we chose to use 8 neurons for the hidden layer; obviously the number of output neurons is 2 (we will call them u_{th}^{net} and

u_{st}^{net}). The outputs again need to be discretized since the control inputs of the car are discrete. Here we use the same discretization as the domain specific controller:

$$u_{th} = \begin{cases} 1 & u_{th}^{net} \geq \frac{1}{3} \\ 0 & |u_{th}^{net}| < \frac{1}{3} \\ -1 & (u_{th}^{net}) \leq \frac{1}{3}. \end{cases} \quad (7.5)$$

$$u_{st} = \begin{cases} 1 & u_{st}^{net} \geq \frac{1}{3} \\ 0 & |u_{st}^{net}| < \frac{1}{3} \\ -1 & (u_{st}^{net}) \leq \frac{1}{3}, \end{cases} \quad (7.6)$$

where, since the activation function of the neuron is the arctan and its output is limited between -1 and 1, each of the possible outputs is associated with an interval of the same size.

For the process of optimization the weights of the network are again initialized to small random values drawn from a zero mean Gaussian distribution $\mathcal{G}(0, 0.1)$.

Modular Recurrent Network

In Section 2.1.3 we identified modularity as one of the approaches that aims at improving the evolvability and the capabilities of controllers. We therefore believe it will be interesting to test a controller based on this approach in our scenario and compare it to the two already proposed.

In discussing the possible advantages of modularity, we also noted how an understanding both of the problem domain and of the specific platform is usually needed to produce an effective decomposition of the controller into modules. This is obviously in conflict with the spirit of our work, but here we can take a different and more pragmatic stand-point. Without loss of generality we can always choose to use a controller structure that has a separate module for each of the outputs. In this way we can allow for smaller networks in each of the modules, since it is reasonable to assume that the function to be learned within each module will be relatively simple.

The result of this choice is a controller that will, by design, avoid the problem of neural interference that we described in Section 2.1.3. In addition, as we will see in Section 7.1.3, because of this modular decomposition, more effective search techniques based on coevo-

lution can be employed to train the controller, potentially leading to better performing solutions.

To allow for more meaningful comparisons between our two neural network controllers, the same recurrent Elman network topology used for the monolithic network was used for each of the modules. However, while we use the same number of inputs, we use a smaller number of hidden neurons (i.e. 4); obviously there is only one output neuron. The 13 input variables are the same as those used for the monolithic network, and the thresholding applied to the output signal in order to produce the discrete control commands (see equations 7.5 and 7.6) is also the same.

For the process of optimization the weights of the network are again initialized to small random values drawn from a zero mean Gaussian distribution $\mathcal{G}(0, 0.1)$.

7.1.3 Evolutionary Training

Fitness Function

While we will be evolving three different types of controllers, and will consider two different types of evolutionary algorithms, all training and testing will use the same path following task described in Section 7.1.1. Comparisons between controllers will be at the heart of our analysis, and therefore it makes sense to use the same fitness function across all the controllers to guide the evolutionary search.

The process of measuring the performance of a controller begins by generating a random path (see Section 7.1.1), placing the selected car model at the beginning of it and, over a fixed amount of time, measuring how well the controller is able to follow the predefined trajectory. Intuitively, the more ground the controller is able to cover without leaving the path, the better the controller.

To decide when a controller leaves the path, a maximum orthogonal distance from the path d_{max}^\perp is defined; when this is exceeded the trial is terminated, and a fitness of zero is returned for that trial. Considering the scale of the car and its kinematics, 0.3m seemed a suitable initial choice for d_{max}^\perp , in relation to the scale of the test paths.

It is easy to see how such a fitness scheme should lead to the generation of controllers that follow the path within the boundaries defined by d_{max}^\perp . However, such a fitness function does not explicitly reward the controllers for staying in the centre of the path, which would correspond to the safest solution since the car is then at the maximum distance

from the track borders. Instead, a controller can exploit the full width of the track and potentially even learn how to choose trajectories that are faster, so covering more ground within the allotted time. Whether the safer or the riskier behaviour is the most rewarding depends on the accuracy of the controller and on the behaviour of the model; evolution will automatically strike a balance of course, but exactly where it will depend on the details. For example, given the way we are evaluating fitness, the chosen durations of the tasks will have an impact on the controllers obtained. A very short task duration would favour a controller that is very fast but potentially unsafe since being able to reach waypoints quickly (perhaps without performing well on curves) would be more rewarding than keeping away from the track borders. At the other end of the spectrum, in the case of a long task duration, a safe but slow controller would not risk leaving the track, and would eventually end up by covering more ground.

In practice those two represent degenerate situations that are easily avoided. If we ensure that the time allotted is long for a controller to learn to negotiate several curves at least to get a good fitness score, but not unnecessarily long, we should avoid the extremes. We will analyse the effects of task length in more detail in Section 7.1.4.

For a generic trial r we define the fitness as:

$$f_{Fr} = \begin{cases} \frac{k_{end}}{K_{max}} & \forall k \quad d_k^\perp < d_{max}^\perp \quad k \in [0, K_{max}] \\ 0 & \exists k | d_k^\perp > d_{max}^\perp \quad k \in [0, K_{max}], \end{cases} \quad (7.7)$$

where k_{end} is the index of the waypoint reached at the end of the trial when the time expired, while K_{max} is the total number of waypoints in the path (200 in our experiments).

It is interesting to note that, with this type of fitness function there will be a difference in fitness (even at the beginning of the evolution when the controllers are poor), between those controllers that perform well on only a few tracks (i.e. the easy ones) and those that completely ignore the tracks. This immediately provides a fitness gradient that should help the convergence of the training algorithm.

This is not the only way of defining a fitness function to measure the performance of the controller; we could for example have a fitness function made up of two contributions, one directly related to the distance covered, and a second inversely related to the cross track error d^\perp . While the idea behind such a fitness function is the same as that already proposed, in practice, the coefficients used to scale the two contributions will determine

how a controller should strike a balance between speed and track following accuracy. Since this trade-off is not explicit, it is not easy to gauge in advance what weighting would be most appropriate.

Since the paths are generated randomly, the fitness obtained during evolution from a single run is not a good indication of the controller's overall performance, since a given path could be particularly easy or difficult. Standard practice in such cases of noisy fitness measurements is to repeat the task several times, and average the scores obtained across the runs. After a small number of preliminary runs, we set the number of repetitions that are averaged to obtain the controller fitness to 16.

We remember from Section 3.3.3 that our car models expect not only the throttle and steering commands as inputs but also the battery voltage u_{ba} . The most extreme conditions for the battery state are being fully charged and being empty; using both of those during controller evolution should ensure that the controllers obtained can cope with the differences in behaviour resulting from the effects of changing battery state. For 8 of the 16 task repetitions (the even ones) we used the minimum battery voltage u_{ba}^{min} and for the odd ones⁶ we used the maximum u_{ba}^{max} .

Evolution Strategies

For both the domain specific controller and the neural network based controllers, the learning corresponds to the optimization of their parameters. These parameters are the controller's constants in the case of the domain specific controller, and the network weights in the case of the RMLP controllers. In both cases therefore we are dealing with optimizing an array of real numbers.

While an EA similar to that underlying the coevolutionary algorithm analysed in Section 5.3 could be used for the training, since we are dealing with real numbers, a better approach is to use evolution strategies (ES) [22]. This evolutionary approach, introduced by Ingo Rechenberg and Hans-Paul Schwefel in the 1960s to overcome the limitations of conventional parameter optimization algorithms, is specifically designed with real-valued search spaces in mind.

The evolutionary process starts with an initial population of $(\lambda + \mu)$ randomly generated individuals that are evaluated according on the task. In evolution strategies terminology,

⁶The maximum and minimum voltages were derived from the datasets used for the training of the models. We obtained $u_{ba}^{min} \simeq 3.5V$ and $u_{ba}^{max} \simeq 4.2V$.

λ denotes the size of the elite, while μ is the size of the remaining individuals, sometimes called the tail. In our case, each individual is evaluated by testing it on 16 randomly generated tracks, and its fitness is determined by the average of the fitnesses obtained. The population is then sorted according to fitness, and while the λ best individuals are retained, the remaining individuals are replaced by mutated copies of randomly selected (with replacement) members of the elite. In line with the vanilla implementation of evolution strategies that uses only one mutation parameter⁷, ([22]) the mutation operation consists of adding a random value drawn from a Gaussian distribution $\mathcal{G}(0, \sigma)$ to each of the parameters⁸. The newly obtained population forms the next generation. In our experiments we used 400 generations; we will comment on this choice in Section 7.1.4 when analysing the results of the training. The algorithm does not employ recombination ($\rho = 1$ in the terminology used in ES) and since the best λ solutions are actually maintained between generations, this is in all respects a steady state evolutionary algorithm. We chose a size of 50 for both the tail and the elite based on some preliminary runs of the optimization scheme.

Cooperative Coevolution

In Section 5.1 we presented the idea of competitive coevolution, a paradigm in which the fitnesses of different populations of evolving individuals are used in a competitive fashion. In this section we look at the cooperative paradigm, in which individuals work together to solve a task. We described in Section 2.1.3 how cooperative evolution has been often explored in conjunction with modularity; here we focus on the key steps in the implementation of cooperative coevolution which are the credit assignment and the method by which individuals are selected to collaborate.

The credit assignment problem refers to the way in which the performance obtained by a group of cooperating individuals is attributed to each of them to determine their individual fitness. The performance depends on each of the cooperating modules, therefore the way in which modules are chosen from the sub-populations to form a full solution also has an impact.

Different approaches have been proposed to establish the performance score of an indi-

⁷An application of ES with multiple adaptive mutation parameters is presented in Section 3.3.5.

⁸A σ equal to 0.1 proved to be effective for both the domain specific controller and the neural network controllers.

vidual. Among the simplest is the one proposed in [187] in which each individual in turn is coupled with the best individual from each of the other sub-populations and tested on the task. The fitness obtained is then attributed to that individual since, all the other individuals being the same, the difference in performance depends on it alone.

Another possibility is that proposed in the ESP (Enforced Sub-Population [80]) algorithm in the context of evolving neural networks. In ESP a subpopulation is maintained for each neuron in the network. To compute the fitness of an individual, test networks are repeatedly constructed by connecting a randomly selected individual from each of the subpopulations and their fitness is measured. When each individual has participated on average in a predefined number of tests (e.g. 10 for ESP) the average of all the fitnesses recorded for every test in which it took part is attributed to the individual as its fitness.

Other variations are also possible, ranging from those in which fitness is used to bias which individuals are selected in addition to the one under test, to those in which only the outcomes of the most successful tests in which the individual participated are considered.

The second important step is to provide a way of determining how many modules (and therefore sub-populations) should be used to solve the task. For some tasks, an almost natural division will exist, while for others automatic techniques must be used. For example Potters and De Jong ([187]) proposed a system in which early signs of stagnation are used to trigger the introduction of a new sub-population, while the contribution of each sub-population to the solution is monitored in order to determine if a species is superfluous.

The general mechanism of cooperative coevolution does not prescribe the way (if any) in which the individuals of different sub-populations should interbreed during evolution. In practice, many of the proposed systems exploit some understanding of the task or the solution. For example, for a task in which the designer thinks that having several similar modules could be advantageous (e.g. as would be the case for the lateral and longitudinal control of a quadrotor, since the machine is symmetric), it would make sense to employ a limited amount of migration of individuals between sub-populations.

Our coevolutionary implementation is very straightforward and is mostly inspired by the baseline algorithm suggested in [187]. It operates at the macroscopic level (see Section 2.1.3) since the objective is to coevolve the recurrent neural networks that constitute the parts of the modular RMLP controller described in Section 7.1.2. For the modular RMLP controller, the number of modules is dictated by the number of control signals.

This provides a natural number of coevolving sub-populations that coincides with using one species for each of the control signals (u_{th} , u_{st}), i.e. for each of the modules.

In line with the idea of avoiding platform knowledge we decided to keep our setup completely general, by keeping the sub-populations completely isolated. The evolution of each sub-population will therefore be carried out independently. Evolution Strategies will again be our algorithm of choice to make the comparison with the other controller solutions more meaningful.

To restrict the computational complexity, credit assignment was implemented with the simple scheme proposed in [187].

A generic run of our cooperative coevolution algorithm proceeds as follows:

1. A sub-population of networks is created for each of the modules. In the case of the car we have 2 modules (steering and throttle), and we set the number of individuals in each sub-population to 50.
2. The *best individual* from each sub-population is selected to be used in the testing phase. In the first iteration, since none of the networks has been evaluated, a random individual is selected from each sub-population.
3. Each network is tested: a controller is formed using the network under test and the selected *best individuals*; the fitness (equation 7.7) obtained in the waypoint following task is attributed to the module under test.
4. The fitnesses obtained are used to evolve each sub-population by means of a (25+25)ES algorithm of the type described in 7.1.3 which does not use recombination. This boils down to ranking the individuals by fitness, and replacing the worse performing half with mutated copies of the elite.
5. The newly obtained sub-population can then be used for a new iteration of the algorithm (steps 2-5). When the maximum number of generations (300 in our case) has been reached, the algorithm stops and the set of controllers that produced the best fitness over the whole evolutionary run are returned.

A small set of preliminary runs was used to choose the size of the sub-populations and the length of the coevolution.

7.1.4 Training and Testing

We now have all the ingredients needed to train the three different types of controllers and compare their performance on the path following task.

Before doing this however, we are interested in investigating two issues that potentially could have an effect on the overall performance of the controllers. The first concerns the car model used during training, and the second the duration used in the path following task.

Effect of Model

To determine the influence of different automatically generated models on the resulting controllers we used the automatic modelling techniques discussed in Sections 5.6 and 6.4.1 to make the most of the available training data by building models representative of the variability of the real platform.

In the case of the toy car we have three different datasets (`datasetcar_6`, `datasetcar_7` and `datasetcar_8`) and so by applying our methodology we can obtain three complete models (complete in the sense that they include both the deterministic part and the noise model part). The effort required from the user is minimal since the whole process is automatic. We will call the obtained models *carAccGPbestNoise6*, *carAccGPbestNoise7* and *carAccGPbestNoise8* to reflect the dataset from which they originate⁹.

To restrict the number of controllers to be evolved we have chosen to perform our comparison using only one controller type. As we will see in Section 7.1.4 the monolithic network has mid range performance when compared to the other controllers, and is therefore a good candidate for our analysis. Given the three models, we then evolve a monolithic neural network controller for each of them. For the evolution we use a duration of 300 time steps for the task.

To obtain representative results, the evolution was repeated independently 30 times for each of the three models. The fitness progress during the 400 generations of the evolutionary optimization is shown in Figure 7.4 using box and whiskers plots¹⁰ to produce a concise

⁹Since these automatic control experiments were performed some time before the training experiments reported in Section 6.4.1, none of the models used here is strictly identical to the *carAccGPbestNoise* model obtained in that section. We therefore use different names for the models to avoid misunderstandings.

¹⁰In the box and whisker plots, the central mark in each box is the median, and the edges of the box are the 25th and 75th percentiles (q_1 and q_3 respectively). The whiskers extend to the most extreme data points not considered outliers; outliers are plotted individually. Points are drawn as outliers if they are larger than $q_3 + 1.5(q_3 - q_1)$ or smaller than $q_1 - 1.5(q_3 - q_1)$. The plotted whisker extends to the adjacent

but informative representation of the distribution of fitnesses. The evolutionary progress

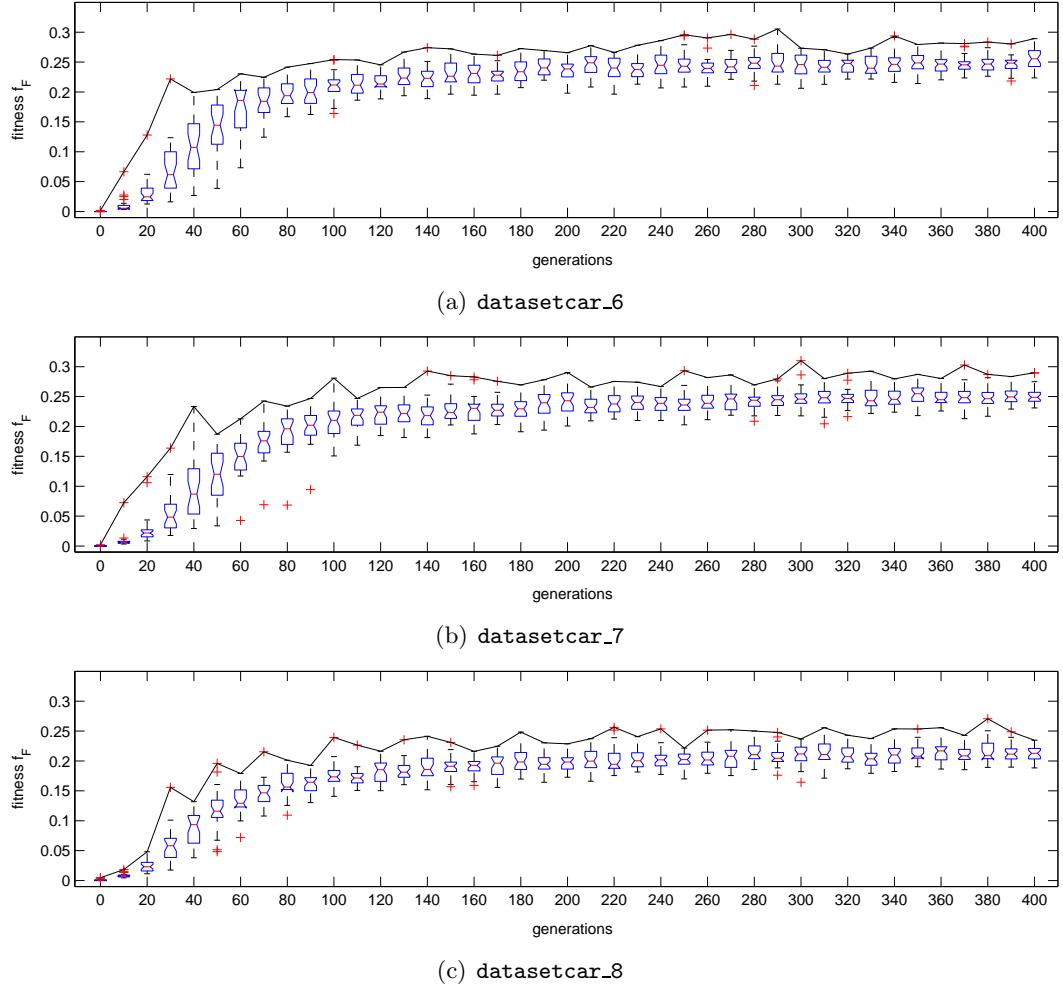


Figure 7.4: Fitness progress during the evolution of monolithic controllers using different automatically generated models.

looks pretty similar for all three models used, with 10 to 20 generations being needed for most of the controllers to become effective. Once that is achieved, smooth progress towards the maximum fitness is apparent. As expected, some of the controllers converge faster than others (as indicated by the whiskers) but in general convergence is achieved within the first 150 generations. All 30 repetitions for all three models consistently reached very similar levels of performance. This indicates that the ES algorithm is performing well in training the RMLP controllers. Our choice of allowing 400 generations for the evolution appears to be quite conservative, but allows us to confirm that the fitness reaches a plateau.

While looking at the fitness progress is a good starting point for our analysis, and gives us some insight, we undoubtedly need a more quantitative comparison of the populations

value, which is the most extreme data value that is not an outlier. The notches are comparison intervals, two medians are significantly different at the 5% significance level if their intervals do not overlap.

of models obtained. To achieve this we tested each of the 90 controllers obtained (30 for each of the 3 datasets) not only with the model it was evolved with, but also with the other two models. The test consisted of running the controller in turn with each of the *carAccGPbestNoise6*, *carAccGPbestNoise7* and *carAccGPbestNoise8* models on 30 randomly generated paths (generated as in Section 7.1.1) for a total of 135 runs. The random number generator was set to the same starting seed for all the controllers to ensure that they would be tested on exactly the same paths.

The performance of each controller was measured by averaging the fitness f_F obtained in each of the runs. The results obtained are shown numerically in Table 7.1 and in the form of a box and whiskers plot¹⁰ in Figure 7.5.

Looking both at the plots and at the table we have the clear impression that the controllers obtained from different models are very similar. To confirm this impression we applied a statistical test to ascertain whether the three populations of controllers are statistically different in performance. The test confirms that the difference between the

dataset	f_F	
	mean	s.d.
datasetcar_6	0.23	0.033
datasetcar_7	0.25	0.018
datasetcar_8	0.24	0.025

Table 7.1: Mean and standard deviation (s.d.) of the fitness obtained by the population of controllers evolved with different car models.

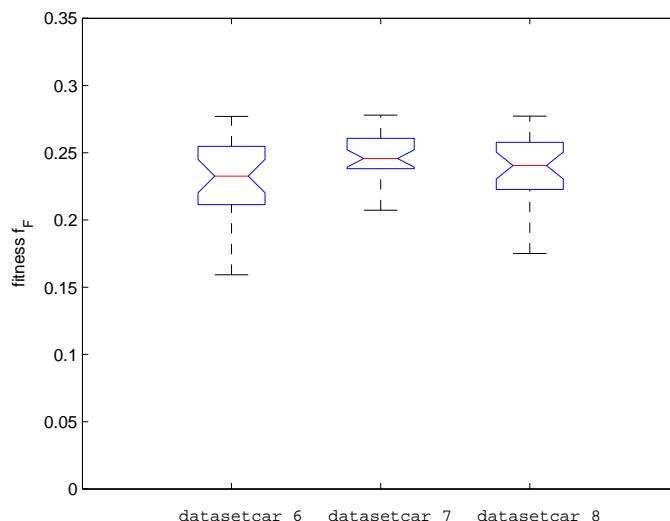


Figure 7.5: Fitness comparison for controllers evolved from different models. The difference in fitness is not statistically significant (Kruskal-Wallis test $P < 0.01$).

task length	f_F	
	mean	s.d.
300	0.22	0.029
600	0.24	0.020

Table 7.2: Mean and standard deviation (s.d.) of the fitness obtained by the population of controllers evolved with different amounts of time allowed to complete the task.

three populations is not statistically significant (Kruskal-Wallis test $P < 0.01$).

In the light of this, we can safely proceed with tests using only one of the car models, namely *carAccGPbestNoise6*.

Effect of Task Length

In Section 7.1.3 we argued that, excluding very degenerate cases, the number of time steps allotted to carry out a task during evolution should not have a significant impact on the controllers obtained. To empirically verify our reasoning we evolved a population of 30 controllers using the *carAccGPbestNoise6* model and the usual task, but this time we doubled the number of time steps allowed to 600. The obtained controllers are then compared to the one trained in the previous section using the same model and a shorter task duration of 300 steps.

The performances of both populations of controllers are assessed by testing them on tasks of different lengths, including durations not used for evolution; we used 150, 300, 450 and 600 time steps, equivalent to 6, 12, 18 and 24 seconds respectively. We averaged the performance obtained across 30 randomly generated paths for each of the task lengths, for a total of 120 independent runs per controller¹¹. The distributions of fitness for each of the controllers in the two populations are summarized in Table 7.2, and in the box and whiskers plot of Figure 7.6.

At first sight both the mean and median performance of the models trained with the longer task duration are higher than the values obtained for the shorter task duration, suggesting that there may be an advantage in allowing more time during training. However, when we statistically tested the two populations of controllers against the hypothesis that they belonged to populations with equal medians (Mann-Whitney U test $P < 0.01$), the

¹¹Before averaging, the fitness was normalized by the length of the task so that short and long runs have in effect the same importance.

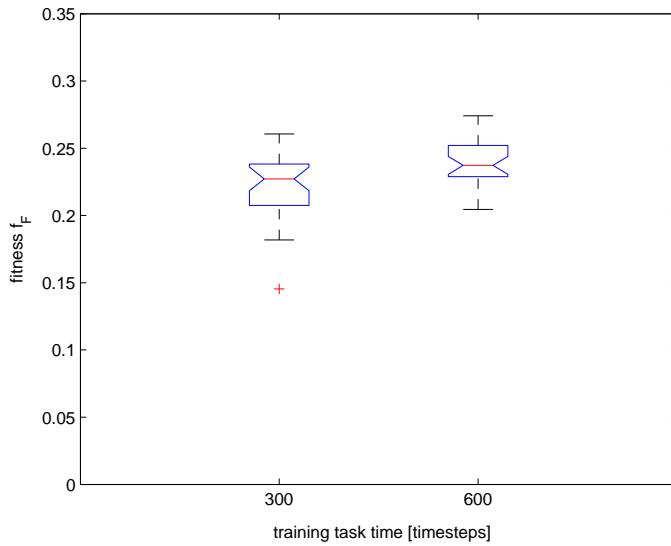


Figure 7.6: Fitness comparison for controllers evolved with different amounts of time allowed to complete the path following task. The difference in fitness is not statistically significant (Mann-Whitney U test $P < 0.01$).

null hypothesis could not be rejected¹². The superiority of the controllers trained on longer tasks is at best questionable; training on a 300 time step task is therefore preferable since less computationally expensive.

Effect of Controller Type

Finally, the last and perhaps most interesting test is the one in which we evolve the three different types of controllers described in Section 7.1.2.

In addition to the monolithic RMLP, modular RMLP and the domain specific controllers trained using the *carAccGPbestNoise6* model, we will also train an additional population of monolithic RMLP controllers using the deterministic version of the *carAccGP-bestNoise6* model (i.e. without the noise model). The main reason for training such a controller is obviously to be able to test it on the real car and compare it with its “noisy” counterpart¹³. In the light of the results obtained in Section 7.1.4, we used a task duration of 300 time steps.

The progress of performance for each of the controller types during the evolutionary runs is shown in Figure 7.7. Immediately we notice that, in the case of the domain specific controller, the spread of performances is worryingly high. To better understand the reason for such a spread, we plotted the fitness progress of each model as a simple line plot in

¹²The null hypothesis could not be rejected even by relaxing the significance level to 5%.

¹³We will do this in Section 7.1.5.

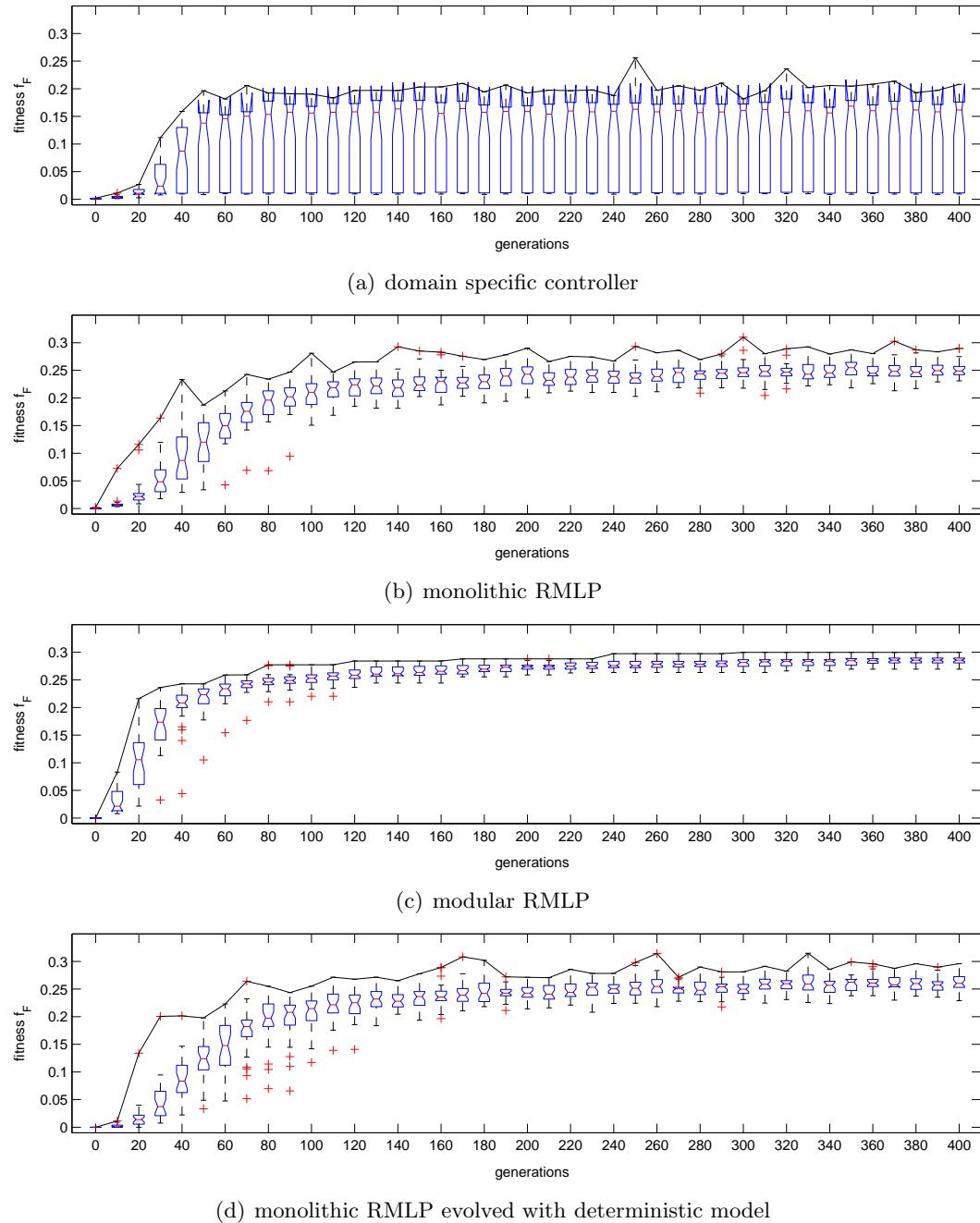


Figure 7.7: Fitness progress during evolution of three different type of controllers.

Figure 7.8. Now it is clear what is happening, as the population is split into two distinct sets: a larger set (17 controllers) which achieved good performance, and a second set (13 controllers) with much lower fitness. Some of the solutions appears to be trapped in a local minimum of the search space, from which the ES algorithm is not able to escape despite the relatively high number of generations.

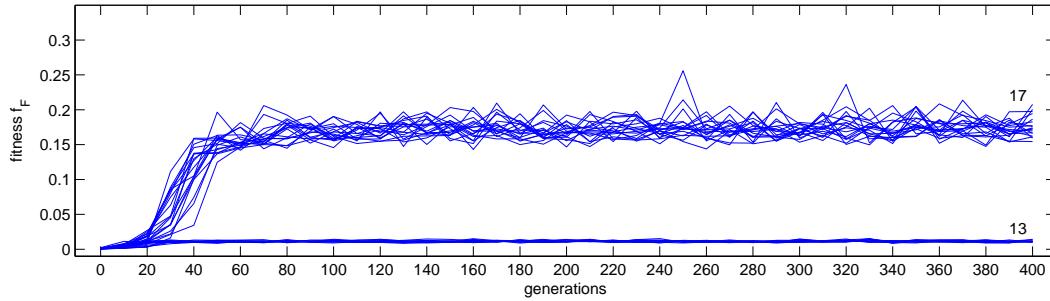


Figure 7.8: Fitness progress during evolution of the domain specific controllers, showing that some of the controllers are poor. The numbers on the plot indicate the number of controllers reaching good or poor fitness.

Continuing the analysis of the remaining controllers, we see that while the monolithic controller shows the same pattern of smooth gradual progress for both the *carAccGPbestNoise6* and for the *carAccGPbest6* models, the modular RMLP controller exhibits much faster convergence. In addition, the final population clearly shows a smaller spread in performance than for the monolithic controllers.

The statistics of the fitnesses of the populations of controllers (see Table 7.3 and Figure 7.9) show what appear to be three distinct sets of performances. The domain specific controller appears inferior to the other controllers even when considering only the set of “good” controllers. The two RMLP controllers trained with the *carAccGPbestNoise6* model show very similar performances, but the controller trained with the deterministic model shows the best performance of all.

controller type	f_F		
	best	mean	s.d.
domain specific controller	0.19	0.10	0.078
monolithic RMLP	0.28	0.24	0.030
modular RMLP	0.27	0.24	0.018
monolithic RMLP from deterministic model	0.31	0.26	0.018

Table 7.3: Fitnesses obtained by different type of controllers. Best, mean and standard deviation (s.d.) are shown.

Given the large difference, there is no need for statistical testing to confirm that the domain specific controllers are indeed the worst, but sound comparisons are needed for the remaining controllers.

We first tested the three populations of neural controllers against the hypothesis that they were samples drawn from the same population (Kruskal-Wallis test $P < 0.01$). The null hypothesis was rejected, suggesting that at least one population median is significantly different from the others. Statistical tests on the three possible pairings of the populations confirmed the first impressions we had by looking at the box and whiskers plot. The difference between the modular and monolithic RMLP controllers is not significant (Mann-Whitney U test $P < 0.01$) while the RMLP controllers trained with the *carAccGPbest6* model are significantly better than both populations of controllers trained on the *carAccGPbestNoise6* model (Mann-Whitney U test $P < 0.01$).

To put these results into context, we can now see that the neural network based controllers are actually superior to the baseline controller that was designed manually. There can be two reasons for this. The first is that the very simple domain specific controller was not designed for our toy car, and had to be adapted to produce discrete control outputs, which may have adversely affected it. The second is that the neural network based controllers are fed more information about the path since they use information about three

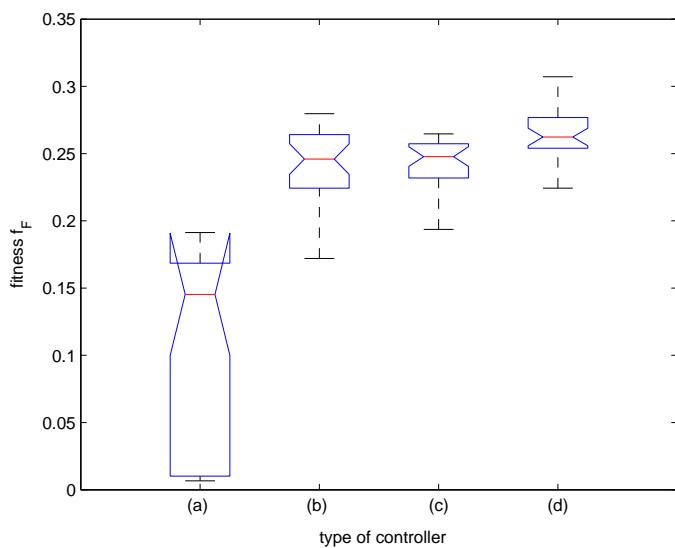


Figure 7.9: Fitnesses of different types of controllers: (a) domain specific controller, (b) monolithic RMLP, (c) modular RMLP, (d) monolithic RMLP trained with deterministic model.

waypoints along the path¹⁴ rather than just one.

The similar levels of performance for both neural network based controllers suggest that the platform independent modularization that we employed is effective, and does not hinder learning. Additionally, the faster and more reliable convergence behaviour demonstrated by the modular RMLP controller shows that, as expected, the modular decomposition allows for a more effective search than the cooperative coevolution is capable of.

In considering the performance of the controller trained on the deterministic model, we have to be very cautious. In fact, both our experience and the evolutionary robotics literature tell us that the evolved controller might have reached this level of performance by exploiting some of the differences that we know to exist between the model and the real car. If this is the case, the controller will not show the same ability when tested on the real car. Being aware of this, we will perform more comprehensive tests in the next section.

7.1.5 Real Car Testing

The evolutionary results gave us considerable insight into the capabilities of the obtained controllers, but the most meaningful measure is of course their ability to control the real car. This final verification is what closes the loop in our novel approach based on limiting the platform knowledge used for developing both modelling and control.

To test the evolved models on the real car we should ideally replicate the task used for the controller evolution; in practice, however, testing each controller with 16 randomly generated paths in order to evaluate its fitness is too time consuming. We therefore concentrated on testing only a small number of selected controllers (the best ones for each controller type¹⁵) on a set of five paths.

We also reduced the number of paths used for the real test to 5 in order to keep the number of trials manageable. In order to cover the space of possible tasks well, the paths used (see Figure 7.2) were generated randomly, as in the training task; paths were selected to have both right and left handed curves with both small and large radii, as well as an almost straight section. To fit within the tracking area of the MOCAP system, the length of each path was limited to 20 waypoints.

¹⁴While it would have been interesting to make a distinction between the effects of these two different contributions, this is in practice not trivial since it is not obvious how to make use of additional waypoints in the domain specific controller.

¹⁵In the case of the monolithic RMLP controller, we also compared the best against the median and the worst controllers in the population in Section 7.1.5.

We tested each of the controllers by running it 30 times (i.e. for 30 repetitions) on each of the five paths; we conducted the same tests on both the real car, and on a simulation based on the *carAccGPbestNoise6* model. A trial lasted until the 20th waypoint was reached or 250 time steps had elapsed, whichever occurred first¹⁶. Since we were interested in analysing the car's trajectories, a repetition was not terminated if the car went outside the path limits.

In the case of the real car, its state was obtained from the MOCAP system, while the control commands generated by the controller were delivered wirelessly using the custom interface described in Section 3.3.3. At the end of each trial the car was manually repositioned at the start location. The car batteries were recharged before testing each of the different controllers, but due to the large number of runs (150, since that we had 5 paths and 30 repetitions of each¹⁷) the battery level inevitably changed during the test. To avoid any correlation between the path type and the battery level, the order in which the repetitions for each path were executed was randomized¹⁸. Recharging the batteries for each new controller ensured that the battery voltage was always between $u_{ba}^{min} \simeq 3.5V$ and $u_{ba}^{max} \simeq 4.2V$) the maximum and minimum voltages present in the training dataset¹⁹.

For the simulated runs the input battery voltage was randomly chosen as in the controller training; for each repetition the voltage was chosen with equal probability between the two values $u_{ba}^{min} \simeq 3.5V$ and $u_{ba}^{max} \simeq 4.2V$.

Controller Types

The first interesting comparison to be made is between the various types of controllers that we described previously.

For our test we selected the best individual from each type; we call the best of the modular RMLP controllers *carbestCoRMLP*, the best of the domain specific controllers *carbestDS*, the best of the monolithic RMLP controllers *carbestRMLP*, and finally *carbest-RMLPno* is the best of the monolithic RMLP controllers trained with a deterministic model. Each controller was run 30 times from the starting position to the 20th waypoint

¹⁶250 time steps is a long time in order to provide a terminating criterion for controllers that get stuck.

¹⁷As a side note, since we are testing 5 different controller structures, the number of runs on the real car necessary for this test adds up to 750!).

¹⁸More specifically instead of executing the 30 repetitions of one path and then moving on to the next, we executed a full set of repetitions using each path once before moving on to the next set. Within the set, the path order was chosen randomly with equal probability (without replacement).

¹⁹We monitored the battery voltage during the run to verify that it was within the limits, and recharged the battery when this was not the case.

of each path and the trajectories were logged to allow performance analysis.

We start with a first qualitative analysis of the controllers' abilities by comparing the 2D plots of their simulated and real trajectories (Figures 7.10-7.13). While such plots show the car trajectory from the origin to the 20th waypoint, they do not provide an understanding of how the controller made progress along the trajectory as a function of time. To provide this information we added to each of the trajectory plots a small inset showing the progress (as a fraction of the path length) made by the car at $t = 80$ time steps. We used a box and whiskers plot to show the progress of all 30 repetitions.

A quick glance at Figures 7.10-7.13 shows all four types of controllers successfully evolved the ability to drive both the real and the simulated car. The controllers are therefore able to "transfer" onto the real car the qualitative behaviour learned through evolution in simulation²⁰.

However, some controllers cannot always maintain the car within the boundaries of the path, and this is especially evident in the case of the more challenging turns.

We start our analysis with the *carbestDS* controller in Figure 7.10. This controller appears capable of maintaining the real car within the path limits when the path is relatively straight (i.e. path 4) but is unable to do so as the path gets more bendy. For paths 3 and 5 the poor ability of the controller in this respect is evident with the car crossing the path limits both in the simulated and in the real trials. For paths 1 and 2 a marked difference is visible between the simulation, for which most of the trajectories are within the limits, and the real car, for which this is clearly not the case.

Overall the test confirms the expectations of poor performance that we built up in Section 7.1.4, but it is unfortunate that from the point of view of the trajectories the behaviour of the real car is only partially predicted by the dynamic simulation.

For all five paths, the progress at the 80 time steps mark is significantly lower (Mann-Whitney U test $P < 0.01$) than that predicted by the simulated trajectories, indicating that the progress (and therefore the fitness) determined in simulation is too optimistic.

The *carbestCoRMLP* controller in Figure 7.11 shows a clear improvement in performance when compared with the *carbestDS* controller. With the exception of path 2, in all the remaining paths in the large majority of repetitions, the controller is able to maintain

²⁰In the evolutionary robotics community, the concept of transferability is defined in a rather loose way. The term is mostly used when a controller evolved in simulation is able to perform the same task it was trained for on the real system (e.g. balancing a pendulum [80] or performing a memory task [110]). In this work we will adopt this qualitative definition.

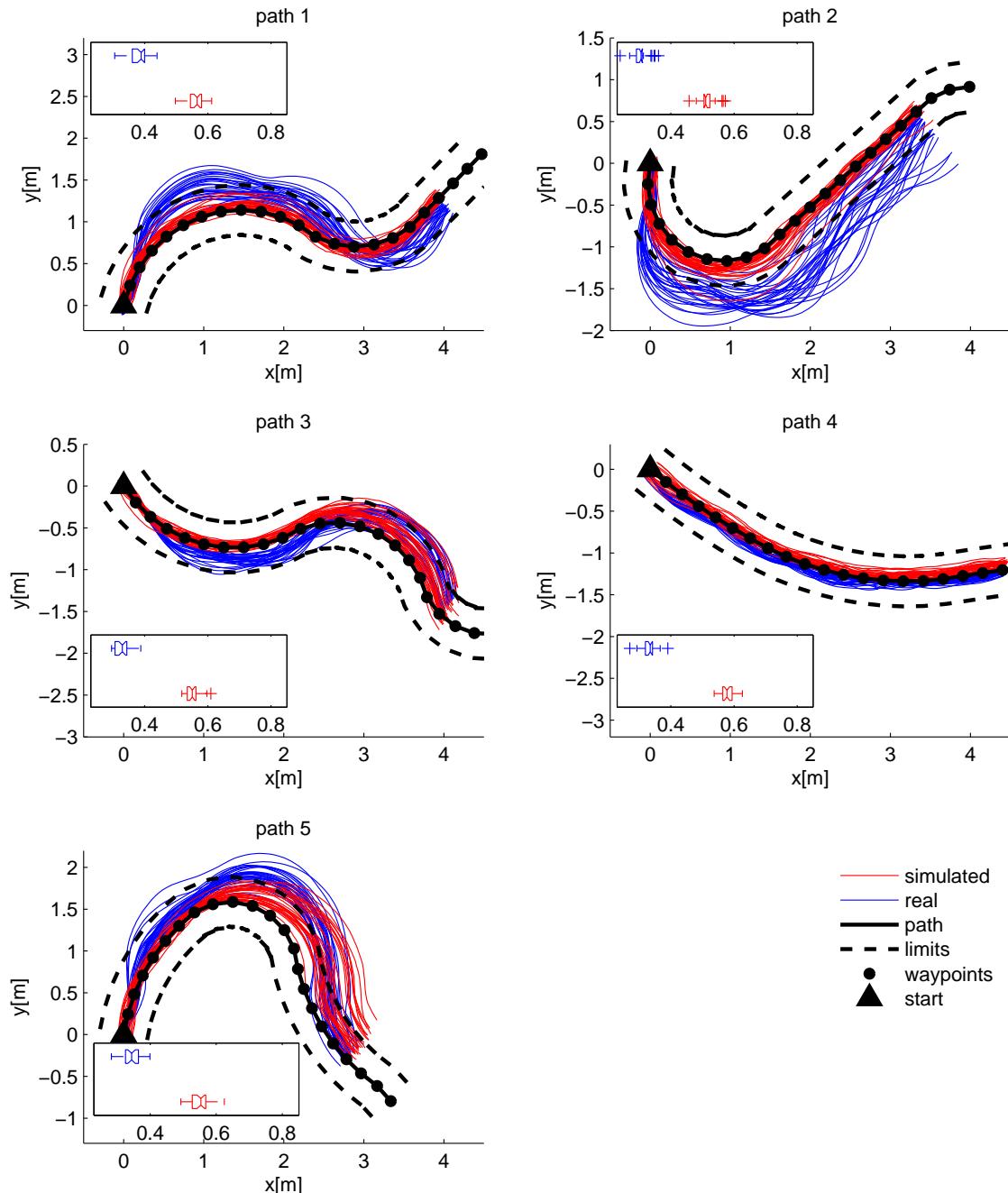


Figure 7.10: Domain specific controller: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length.

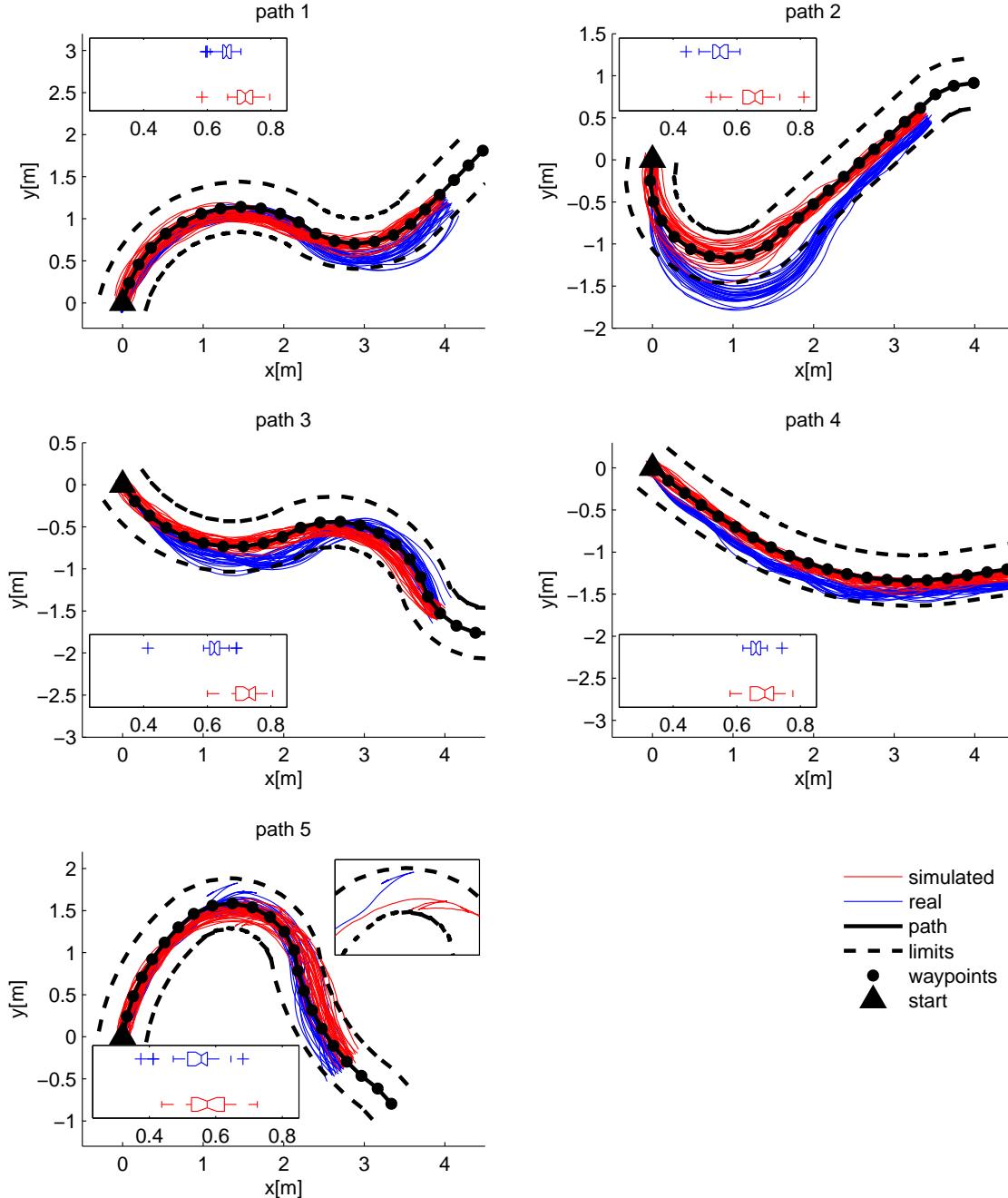


Figure 7.11: Best modular RMLP controller: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length. The top right inset of path 5 shows a situation in which the controller backtracks to reposition itself in the path.

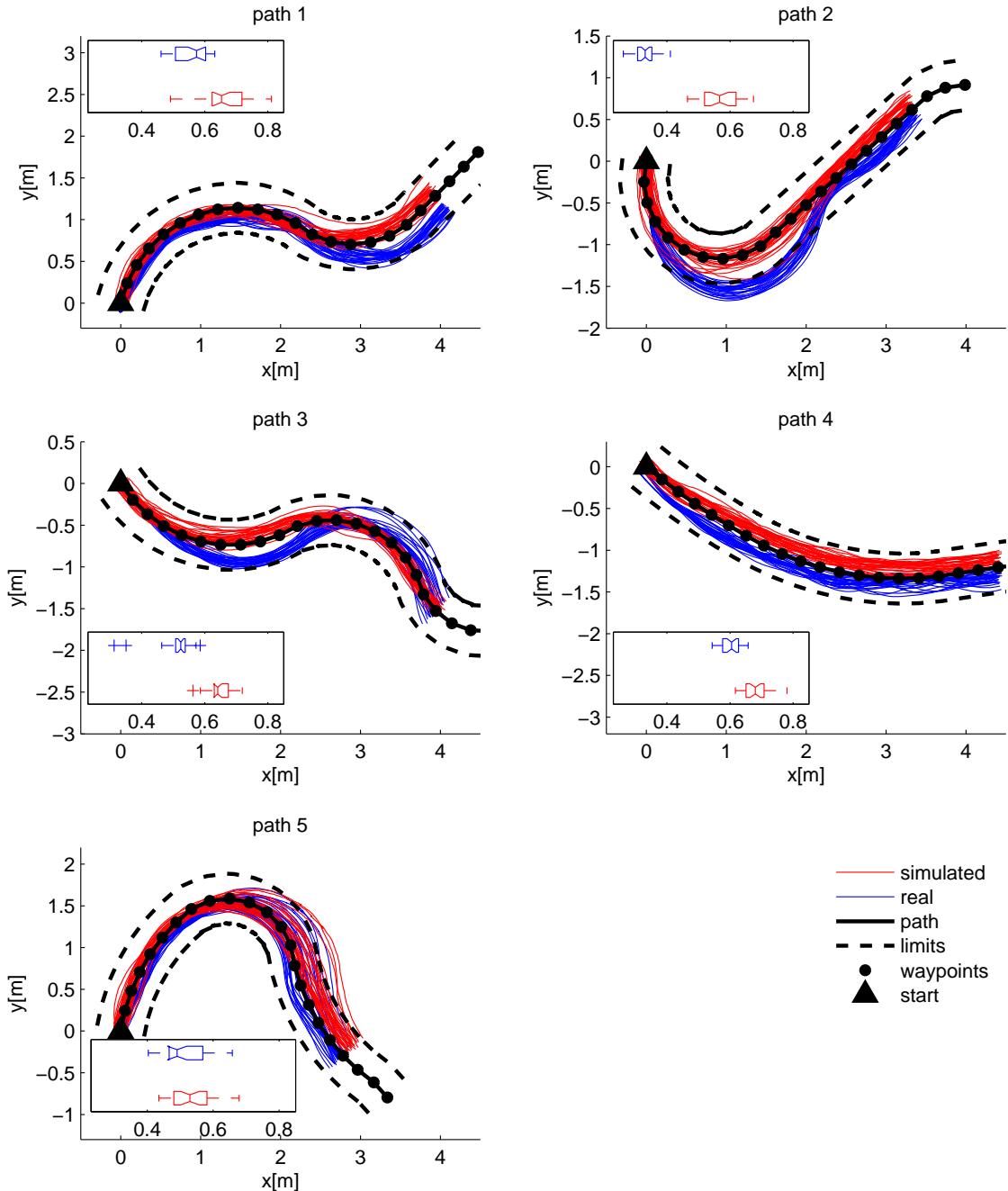


Figure 7.12: Best monolithic RMLP controller evolved with a deterministic model: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length.

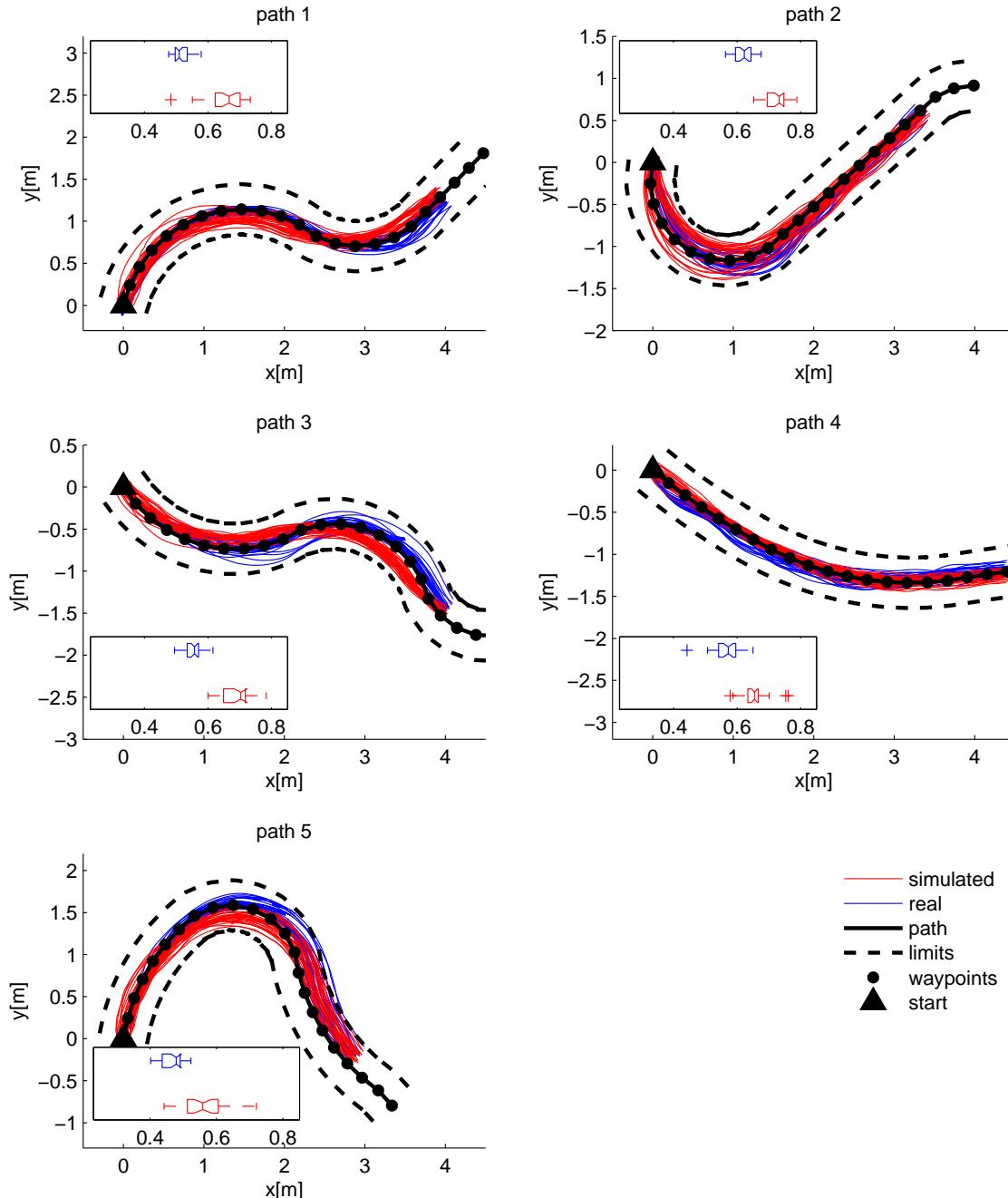


Figure 7.13: Best monolithic RMLP controller: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length.

the car within the path limits similarly to what was the case for the simulated car. Path 2 shows a different scenario, with a clear difference in trajectories at the point of highest curvature in the path. While in the case of the simulated car, the trajectories are very close to, but still within, the limits at such points, most of the real trajectories go outside the path borders in the same area and then return when the trajectory becomes straight again.

In terms of the progress measured at $t = 80$ time steps, the progress computed for the simulated trajectories for two of the 5 paths (i.e. paths 4 and 5) is not significantly different (Mann-Whitney U test $P < 0.01$) from the same measure applied to the real trajectories. For the remaining paths (i.e. paths 1, 2 and 3) the difference is significant and again suggests that the progress predicted in simulation is optimistic.

Looking at the trajectories for path 5 we notice a peculiar ability of the *carbestCoRMLP* controller (more visible in the inset). When the controller is close to the track limit, it stops the car and backtracks in order to reposition itself within the path. Often this behaviour requires a considerable amount of time and prevents the controller from completing the run. Since this behaviour is not shown by any other controller, this may suggest that the modularization is what makes possible the learning of this ability, although is not at all clear how.

We consider now the first controller based on a monolithic RMLP network, the one trained using the deterministic model *carAccGPbest6*. The pattern of performance is very similar to that shown by the *carbestCoRMLP* controller, with good performance on driving within the path limits for all path types except path 2. Here again, the sharpest portion of the turn is the most problematic, where most of the real car trajectories exceed the path limits. As a comparison, only one of the trajectories obtained by the *carbestCoRMLP* controller driving the simulated car is outside the path limits. The progress obtained at the 80 time steps mark shows a significant difference (Mann-Whitney U test $P < 0.01$) between the real and simulated trajectories. For paths 1-4, the simulated progress is overestimated, while for path 5 this it is actually lower than that for the real car.

Finally we look at the trajectories obtained from the *carbestRMLP* controller. It is immediately clear that this controller is capable of better driving performance. For all the paths, all the simulated and real trajectories are within the path limits with the sole exception of one in path 5 which is a few centimetres beyond the limit. The similarity between

the simulated and real trajectories is higher than was seen for the previous controllers. Another noticeable difference is that all the trajectories, independently of the path curvature, are much closer to the path centreline than was the case with the previous controllers. This suggests that the control law learnt by this controller might be substantially different from those applied by the other controllers, and ultimately better.

The comparison in terms of the progress measured at the 80th waypoint again reveals that the simulated trajectories are significantly overestimating the ability of the controller (Mann-Whitney U test $P < 0.01$).

While it would be too time consuming to look in more detail at the behaviour of each of the four different controllers that we tested, we can certainly analyse further the one that, at least qualitatively, appears to be the best of all, the *carbestRMLP* controller. In Figure 7.14 we randomly selected a trajectory produced by the controller for each of the paths and plotted it in a colour coded fashion to visualize the car's instantaneous forward speed. In the insets we also show the control inputs delivered by the controller during the run.

After the expected initial speed up, which is common for all the different paths, we can see that the controller manages the speed differently depending on the path curvature. For path 4, which is almost straight, the controller maintains an approximately constant speed between $0.5m/s$ and $1m/s$. Since the car has only discrete controls, the controller switches between neutral and forward throttle, and occasionally uses some backward throttle. This type of on-off modulation is in practice the only way of controlling the speed in our type of car, and is one of the reasons that makes accurate control of such a car non-trivial.

In curves the controller governs the speed in a rather inconsistent way. On right hand turns (paths 3 and 5) the controller slows down on entering a curve and speeds up at the exit in a reasonably sensible fashion. However, in left hand turns, (paths 1 and 2) the controller speeds up while in the curve and at its exit, a much less understandable strategy. Obviously the speed, mass and tyre friction of our toy car are very different from those of a full sized car, and therefore is not obvious that the idea of slowing down when approaching a curve that we consider natural is indeed the only possible or best choice in the case of our toy car.

Over all the paths we can see that the steering is a result of a delicate mix of left and right steering control inputs. In straight sections of the path the controller tends to

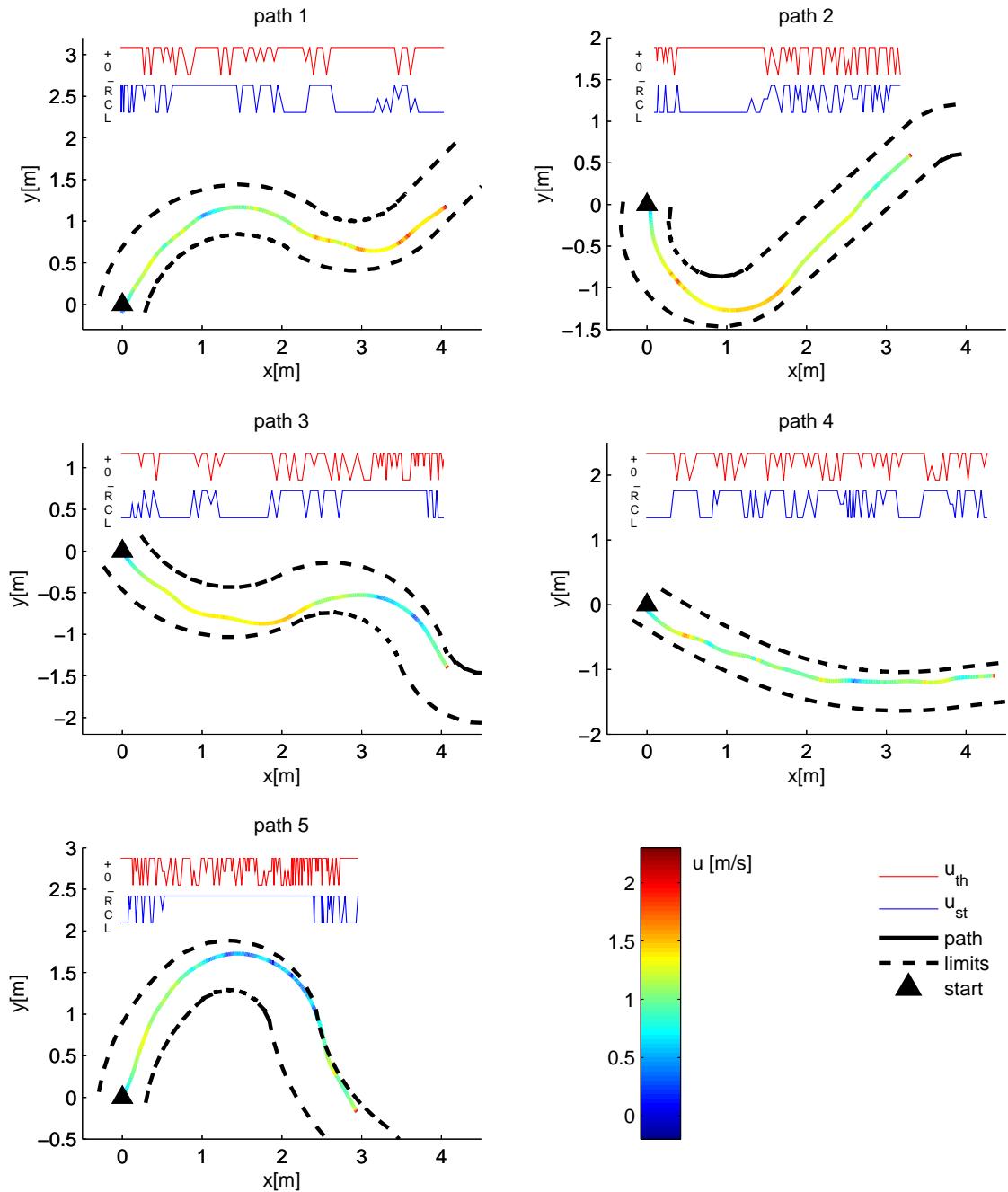


Figure 7.14: Best monolithic RMLP controller: a randomly selected trajectory for each of the five test paths. The colour of the trajectory indicates the forward speed u . At the top of each plot are depicted the control inputs \mathbf{u} applied by the controller.

switch between these two control signals, while in the turns the correct steering command is applied consistently throughout the curve. As expected, neutral steering is used mostly in situations where the path is straight.

Is now time to use the real car trajectories recorded in our tests to quantitatively compare the controllers.

As in Section 7.1.4, we will measure controller performance using its fitness computed using equation 7.7. With the aim of obtaining a single fitness that reflects the general ability of a controller, we average the fitnesses obtained from one of the trajectories for each of the 5 paths²¹ to form a set that we call a repetition. Since we have 30 trajectories for each of the paths, we can obtain 30 different repetitions for each of the controllers. The mean and standard deviation for each of the controller types is shown in Table 7.4, and in the box and whiskers plot of Figure 7.15.

controller	f_F mean	f_F s.d.
<i>carbestCoRMLP</i>	0.47	0.085
<i>carbestRMLPno</i>	0.46	0.037
<i>carbestRMLP</i>	0.57	0.017
<i>carbestDS</i>	0.23	0.047

Table 7.4: Fitnesses of the best controller of each type calculated from the real car trajectories.

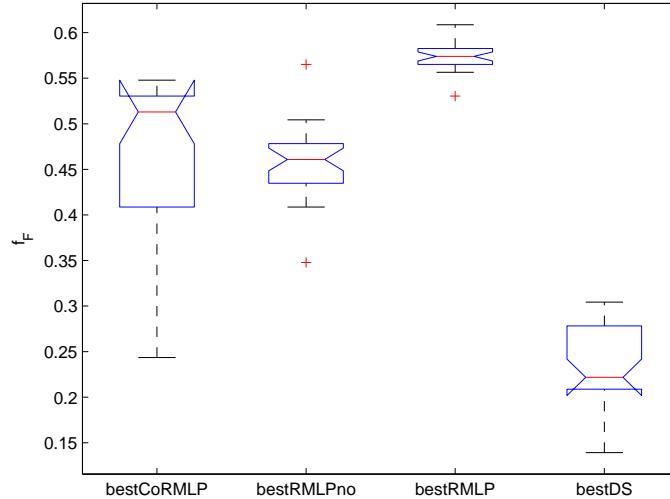


Figure 7.15: Fitnesses of the four different controllers *carbestCoRMLP*, *carbestRMLPno*, *carbestRMLP* and *carbestDS* computed from the real car trajectories.

As in the case of the performance computed using the simulated model in Section 7.1.4, the performance of the *carbestDS* controller is markedly inferior to that of the other

²¹Selected with equal probability without replacement.

controllers. A statistical comparison of the three remaining controllers (Kruskal-Wallis test $P < 0.01$) shows that their performance is significantly different. Proceeding with a series of pair-wise comparisons, we establish that the performances of the *carbestCoRMLP* and *carbestRMLPno* controllers are not significantly different (Mann-Whitney U test $P < 0.01$); but both are significantly worse than the *carbestRMLP* controller (Mann-Whitney U test $P < 0.01$).

The *carbestRMLPno* model that was the better performing one in Section 7.1.4 is not any more. This is not surprising since as we explained its fitness was evaluated using a deterministic model.

Overall the monolithic RMLP controller appears the best at controlling the real car, in terms of both qualitative and quantitative performance.

Best, Mean, and Worst RMLP Controllers

When testing the different controller structures, it was feasible to test only the best of the controllers of each type on the real car. Understanding how the remaining 29 controllers compare to the best one is of obvious interest, and this is the focus of this section. Again we were limited by the number of tests we could actually run on the real car, so we focussed on only one type of controller, the monolithic RMLP network, and analysed the worst, median and best controllers obtained using evolutionary optimization (we call them *carbestRMLP*, *carmedianRMLP* and *carworstRMLP* respectively.).

The testing proceeded as described in the previous section with 30 repetitions for each of the controllers, and for each of the paths, for a total of 450 independent runs.

The trajectories obtained for the *carmedianRMLP* controller are shown in Figure 7.16. When compared with the *carbestRMLP* controller analysed in Figure 7.13, we see that the two controllers produced very similar types of trajectories. Overall however, the *carmedianRMLP* controller shows a less precise driving style, and tends to produce a more dispersed set of trajectories. On paths 2 and 5, which we know to be the most difficult due to the sharp bends, a limited number of trajectories actually exceed the path limits. On path 5 (see top right inset) a limited number of repetitions brings the car to a safe halt instead of allowing it to travel outside the defined path; this is due to the fact that the controller learned to slow down when reaching the track limits.

Figure 7.17 shows the trajectories obtained using the *carworstRMLP* controller. Here

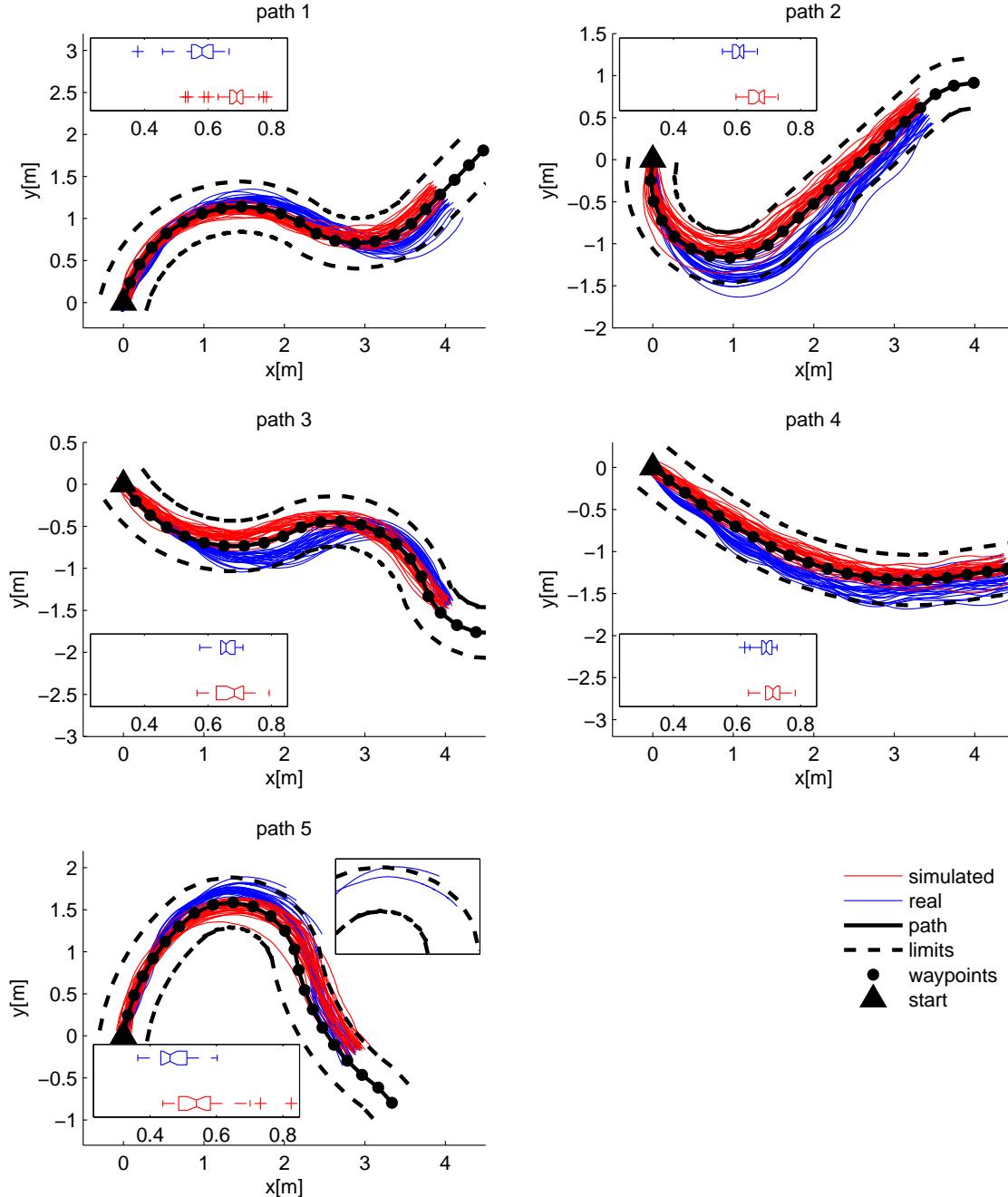


Figure 7.16: Median monolithic RMLP controller: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length.

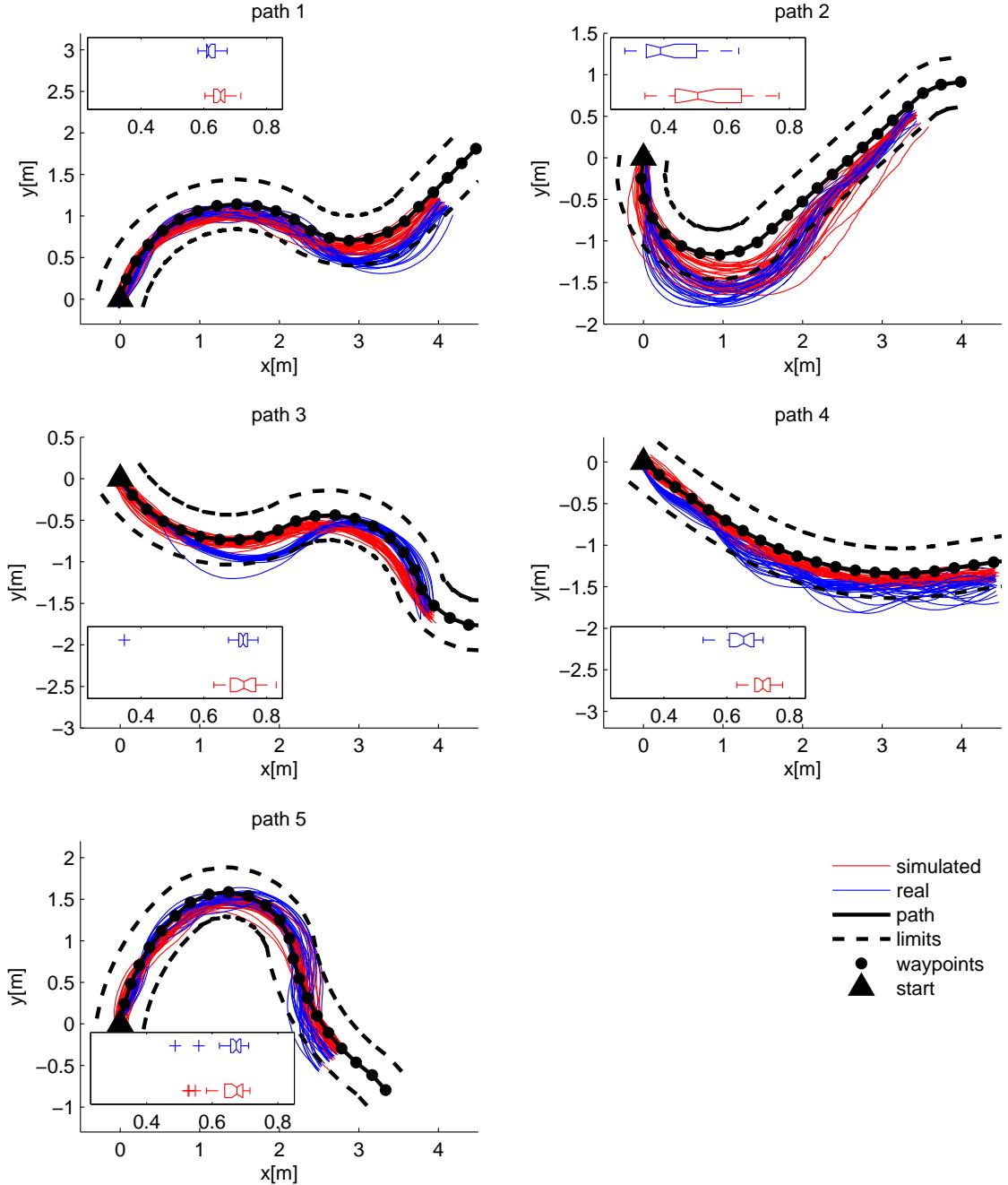


Figure 7.17: Worst monolithic RMLP controller: 30 simulated versus 30 real trajectories on five selected test paths. The first 20 waypoints are shown. The insets show the box and whiskers plot of the controllers' progress along the path at $t = 80$ time steps measured as a fraction of the total path length.

controller	f_F mean	f_F s.d.
<i>carbestRMLP</i>	0.57	0.017
<i>carmedianRMLP</i>	0.50	0.089
<i>carworstRMLP</i>	0.37	0.11

Table 7.5: Fitnesses of the best, median and worst RMLP controllers calculated from the real car trajectories.

a different scenario is visible. While the controller seems to perform well on left hand turns (path 5, second part of path 3 and first part of path 1), maintaining the car within the path limits, the same is not true for right hand turns. This problem shows up in paths 1, 3 and 4 but is most prominent in path 2, where the point of highest curvature of the path sees the controller failing, similarly to the *carbestCoRMLP* and *carbestRMLPno* controllers.

Comparing the best, median and worst controller by computing their fitnesses (see Table 7.5 and box plot in Figure 7.18), reveals a statistically significant difference in performance (Kruskal-Wallis test $P < 0.01$). The difference is significant even when the performances of the controllers are compared pair-wise using two Mann-Whitney U tests ($P < 0.01$), which confirms that the *carworstRMLP* controller is worse, and the *carbestRMLP* is better, than the *carmedianRMLP* controller.

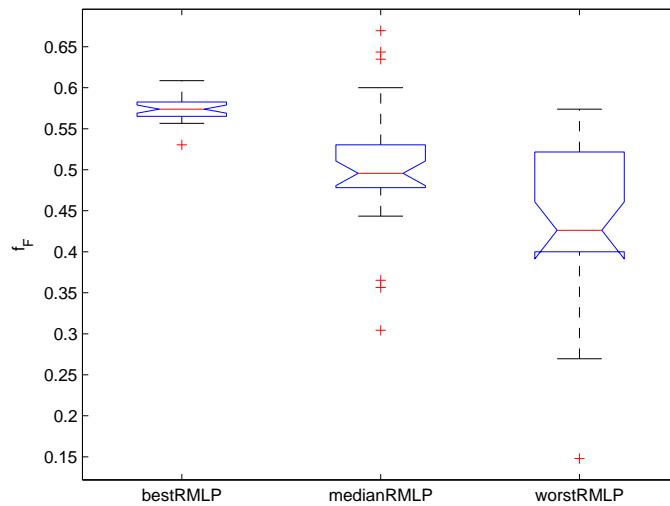


Figure 7.18: Fitnesses of the three different controllers *carbestRMLP*, *carmedianRMLP* and *carworstRMLP* computed from the real car trajectories.

In summary, the qualitative analysis showed good performance for the best and median controllers, suggesting that at least half of the evolved controllers are indeed able to drive the car within the path boundaries in the large majority of situations. However this result also confirms that even when the controller structure is capable of good driving

performance, the training is a critical component in the automatic design of controllers.

Conclusions

The toy car has proved to be a very valuable test bed to explore our idea of automatically designing controllers without making use of platform specific knowledge. It was valuable both because of the familiarity we have with this kind of platform, and also because it gave us the possibility of executing almost a thousand real world runs to test the evolved controllers.

Qualitatively, all the controller structures tested showed an ability to transfer to the real car and were able to drive it successfully, but differences were present between controllers.

Our adaptation of a domain specific controller that had demonstrated excellent performances on a real size car, showed its shortcomings when tested on the toy car. This highlights how important the dynamic characteristics of a vehicle and the nature of its control inputs are for the performance of a controller, and how platform specific knowledge can make a difference when a controller is manually designed.

Controllers based on neural networks were able to provide the necessary flexibility to learn better control strategies, which in the case of the monolithic network reached very proficient levels. While on paper the modular RMLP controller should have been as good as the monolithic controller it is hard to ascertain the origin of its poor performance. It is entirely possible that with a single network it might be easier to evolve the coordinated control of throttle and steering that is needed in very sharp turns, but this remains just a speculation.

So far we have been commenting on the controllers, but obviously the model used to train them plays an integral role in their abilities. As we know, both the deterministic and the stochastic parts of the model are equally important for the controller training.

Since the model used was successfully tested against an independent dataset in Section 5.6, and more than one of the RMLP controllers delivered very good performance, the probability that the *carAccGPbest* model is the origin of the problems in the other types of controllers is quite small. However, improving the quality of the model, and in particular improving its stochastic component, might be a suitable way of reducing the difference between the real and simulated controller performance, as well as increasing the controllers' ability.

Substituting the independent noise model that we currently use with one in which the state and inputs condition the noise description would deliver a model better suited to replicate situations such as that, for example, in which the steering radius of the real car is slightly different from that of the model. Continuing with this example, such a model, when used to generate trajectories during the training, could produce a noise component that consistently increases (or decreases) the steering angle during the time in which the steering command is applied. With the current model this is obviously not possible since in consecutive time steps the noise is independent. A noise model of this type has also the potential to elicit the evolution of controllers that are more accurate in tight turns by being able to deal with variations in the car's turning radius.

Unfortunately, verifying how effective a better noise model could be would require the reformulation of the noise model (with all the drawbacks already discussed in Section 6.1) as well as repeating the training and the testing. The sheer amount of work means that we are forced to defer this to sometime in the future.

We should also point out that several weeks passed between the collection of the data used for the modelling, and the testing of the controllers. During this time the car was used for other experiments, and although we do not have evidence of a drastic change in behaviour (e.g. due to damage) the wear and tear incurred in this time might have had subtle consequences for the car's behaviour.

Finally, although time did not permit the investigation of different types of fitness function, we cannot exclude the possibility that a different way of rewarding the evolving controllers might produce better results. Remembering the fact that the *carbestRMLP* controller showed a tendency to maintain the centre of the path, it would be interesting to formulate a fitness function that took into account the distance from the path centre. Unfortunately, as we already commented in Section 7.1.3, it is not obvious how to mix the two contributions of overall progress and distance from the centre line.

7.1.6 Different Types of Tasks

While the experimental results presented in the previous section are clear evidence of the soundness of our approach in the case of evolving controllers for the path following task, this is certainly not the only task to which our modelling-and-control concept can be applied. In fact the opposite is almost true: our approach can in principle be applied to any task

for which is possible to provide a clear measure of controller progress. The idea of using artificial evolution easily allows the incorporation of information from different types of sensors, and is sufficiently flexible to accommodate very diverse types of tasks; however, we are not claiming that this is a silver bullet since depending on the context at least some of the limitations discussed in 2.1 will come into play.

In our experience however, given a good model, a controller can be automatically designed for different tasks. In the following two sections we will briefly review some of our published work in which we automatically developed controllers for what we call point to point car racing as well as for creating interesting tracks for car racing games.

Point to Point Car Racing

Although it used less refined techniques, a preliminary version of our work on automatic modelling and control of the toy car was presented in [234]. In that paper we considered different types of models: a parametric model, a model based on a neural network trained with backpropagation, a model based on an evolved neural network and a model based on a nearest neighbour classifier. All those were simple deterministic models. The models were trained to reproduce the car's behaviour and were then used to train controllers based on a recurrent network topology, the inputs to the controller being the car state, and the distance and angle to the next waypoint.

The task we chose to evolve controllers for was point-to-point car racing, a task in which a controller is allowed to control the car for 500 time steps, or 25 seconds of simulated time. During this time, the car has to reach as many waypoints as possible, and the fitness is equal to the number of waypoints passed by the end of the trial. A waypoint is considered passed when the centre of the car is within 30 centimetres of the centre of the waypoint; when a waypoint is passed, a new waypoint immediately pops up at a random position within a radius of 1.5 metres from the centre of the test area. Only the information about the next waypoint is available at any given time.

The evolution strategy explained in Section 7.1.3 was used to train the controllers²².

The evolved controllers showed a variety of behaviours which depended on the peculiarities of the model used to train them. In fact the only truly proficient controllers, those consistently able to reach waypoints without getting stuck, were those trained with more

²²We refer the reader to our original publications for more details of the training of models and controllers.

than one model, and it was the effectiveness of the training based on multiple models that guided us towards the idea of having a noise model, as presented in Chapter 6. The differences that were expressed in Chapter 6 by the different models in the paper were instead expressed by the noise distribution, an approach that has the advantage of affording a straightforward way of tuning the noise parameters and ensuring that the whole noise envelope is covered.

The controller evolved with two different models had a rather peculiar behaviour; when it missed a waypoint, which happened quite often, it usually braked to a full stop just afterwards. When it accelerated again it would be able to return and reach the waypoint, since the car has a narrower turning radius at low speed than at full speed. Another strategy was displayed by the controller evolved using three different models; the controller would drive forwards most of the time, aiming for the waypoint, but if it missed it, the controller would back off some distance and do a full repositioning, and was then able to reach the waypoint on its second try. The behaviour of these two controllers when tested on the real car can be seen in Figure 7.19.

The point to point car racing is simple example of how controllers with quite different abilities can be evolved with limited modifications to our approach.

Automatic Generation of Interesting Car Racing Tracks

Further examples of using car models to train specialized controllers were presented in [232] and [233].

In these applications a different scenario was considered, in which we desired to automatically generate tailor made content for the players of a car racing video game. We looked at the idea of generating racing tracks to present just the right amount of challenge for each player, with the aim of increasing the satisfaction of the player²³. The method behind our approach is relatively straightforward. After recording a user playing on a test track, we would evolve a controller with the same driving style of the user. The trained controller can then be used as a proxy for the player's ability in order to search the space of racing tracks to obtain tracks tailored to the driving ability of the player. It turns out that emulating the player's driving style it is far from trivial, and our experiments showed that learning the behaviour of the player directly (i.e. training a controller in a supervised

²³We refer to the original publications and the references therein for a more in-depth discussion about the idea of player satisfaction and the concept of an engaging playing experience.

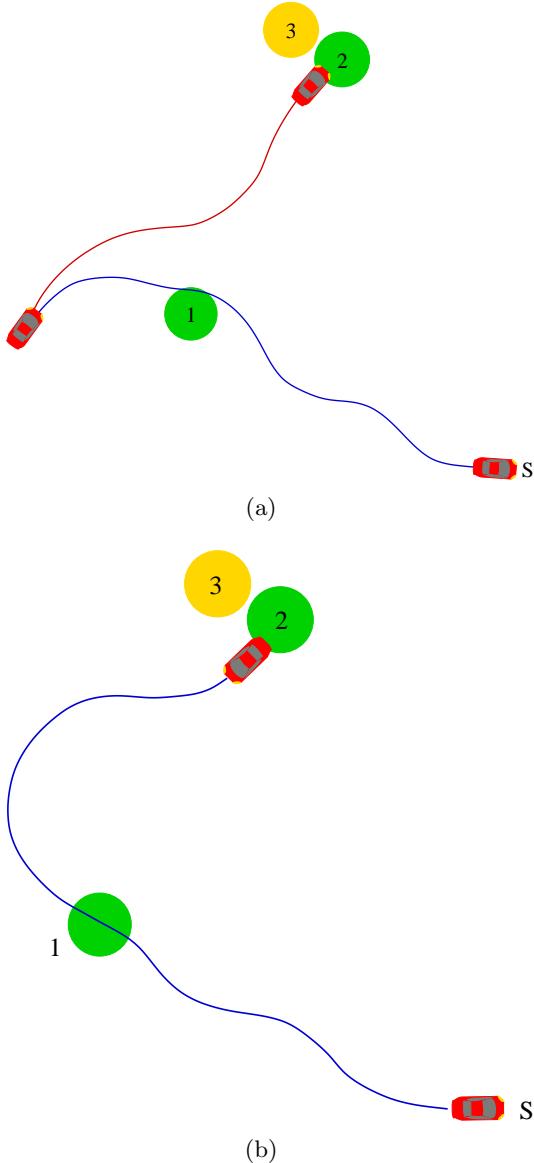


Figure 7.19: Trajectories of RMLP controllers evolved on two (a) and three (b) models completing the first two waypoint of the point to point racing task after starting from the position “S”. A red trajectory indicates that the car was driving forwards, and a blue one that the car was driving backwards.

learning fashion to produce the same commands as the player given the state of the car) is not an effective approach.

A more effective strategy is that of identifying measurable characteristics of the human player’s behaviour, and adapting an evolved controller to reproduce these characteristics. After experimenting more or less successfully with different metrics, we decided to use the total progress along the track (f_1), the speed at which each waypoint was passed (f_2), and the orthogonal distance to the path when passing a waypoint (f_3) as the measures of fitness .

The performances of the human players were captured on a track specifically designed to embed a variety of driving challenges, from straight segments, to very sharp turns to large radius curves (see Figure 7.20). In this case we have three different objectives used to

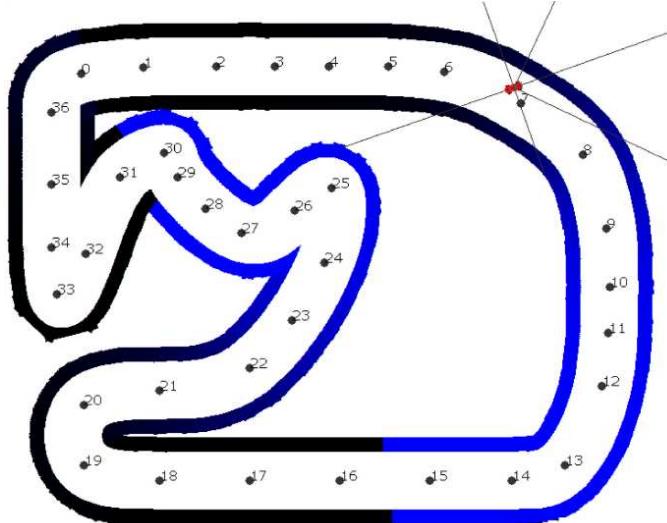


Figure 7.20: Path containing segments of different type used to capture the driving stile of human players. Note the waypoints placed along the track and the wall sensors protruding from the car.

train our controllers, and in order to concentrate on emulating the driving style, we started with good automatically designed controllers²⁴.

While evolutionary multi-objective optimisation is a rich and active research field, what we need here is just a simple way of handling more than one fitness function. We are not interested in Pareto fronts; what we are interested in is specifying which fitness measures have higher priorities than others. A simple solution to this problem is to use an evolution strategy with multiple elites. In the case of three fitness measures, it works as follows: out of a population of 100, the best 50 genomes are selected according to fitness measure f_1 . From these 50, the 30 best according to fitness measure f_2 are selected, and finally the best 20 according to fitness measure f_3 are selected. Then these 20 individuals are copied four times each to replenish the 80 genomes that were selected against, and finally the newly copied genomes are mutated. This algorithm, which we call Cascading Elitism, was inspired by an experiment by Jirenhed *et al.* [114].

The inputs to the controller were the speed of the car, the angle to the next waypoint and the readings from six wall sensors²⁵ (see Figure 7.20).

²⁴See the original paper for more details on how the controllers were initially evolved.

²⁵Each wall sensor is located at the centre of the car, and has two parameters under evolutionary control:

In our experiments, five different players' driving was sampled on the test track. After 50 generations of the Cascading Elitism algorithm with a population of 100, controllers whose driving bore an acceptable degree of resemblance to the modelled humans had emerged. The total progress varied considerably between the five players - between 1.31 and 2.59 laps in 1500 time steps - and this difference was faithfully replicated in the evolved controllers, which is to say that some controllers drove much faster than others.

Progress was made on the two other fitness measures as well, and though there was still some numerical difference between the real and modelled speed, and the orthogonal deviation when passing most waypoints, the evolved controllers did reproduce qualitative aspects of the modelled players' driving. For example, the controller modelled on the first author drives very close to the wall in the long smooth curve, very fast on the straight paths, and smashes into the wall at the beginning of the first sharp turn. Conversely, the controller modelled on the anonymous and very careful driver who scored low total progress crept along at a steady speed, always keeping to the centre of the track.

Again this is an additional example in which controllers can be tailored to very specific characteristics which, as in this case, might be expressed by conflicting objectives.

For reasons of brevity we have reported only the most relevant parts of the experiments in [232] and [233]; the papers give more details of how the obtained models are used to optimize tracks for specific player driving styles.

7.2 The X3D Quadrotor

The second platform that we address is the X3D quadrotor, which with its 6DoF dynamics and four different controls is a more challenging problem than the toy car. Even in this case our controllers will be developed around the model automatically learned in Chapter 5 and Chapter 6.

7.2.1 Task

Even in the case of a quadrotor helicopter, the problem of autonomous navigation can be solved using a hierarchical approach like the one we discussed in Section 7.1.1. In this case,

its direction (expressed as the angle it makes with the axis of the car) and its maximum range, which may be anything between 0 and 200 pixels. A wall sensor returns a value between 0 and 1, depending on whether it detects a wall within its maximum range, and on the distance of the wall as a proportion of the maximum range.

the whole system will work in three dimensions, but conceptually the same decomposition into layers can be applied. The ability to follow a series of defined waypoints is again a useful part of a UAV system ([100]) making the automatic development of controllers with such an ability an interesting task.

A straightforward three-dimensional extension of the concept of a waypoint path that we used for the toy car (see Section 7.1.1) is a suitable way of formulating the task for a flying machine, and we will also use it for the autopilot helicopter simulator in Section 7.3.1. However, in this section we are keen to show how evolution allows us the freedom to structure the controller training in other ways. Given their intuitive and widespread use, we have chosen to use some of the same metrics used in classic control theory.

In control theory terminology, the quadrotor becomes the plant which has to be controlled so that its response (i.e. the positions) matches a reference signal (i.e. the waypoints). Measuring the performance of a controller in terms of its response characteristics is at the root of many of the classic control design methods ([241]), and in particular in [126] Koza shows how it is well suited to the evolution of proficient controllers using genetic programming²⁶.

The metrics of overshoot, settling time and error are readily adaptable to our scheme. For simplicity we will first consider the single dimensional case, and will examine a typical system response to a step control input (Figure 7.21). We can define the following:

- *absolute error* e_a : integral of the absolute error between the response and the commanded value from time t_e (defined as a fraction of the total window length) to the end of the time window (see shaded area in Figure 7.21);
- *normalized overshoot* o_r : initial increase of the controlled variable over the final steady state level for a step input (see x_o in Figure 7.21), defined as a fraction of the final steady state level;
- *normalized settling time* t_{sr} : time needed for the response to settle to within 20% of its final value, defined as a fraction of the overall window length.

These are computed on the trajectory produced by the helicopter whenever a new waypoint

²⁶We made some preliminary attempts at using genetic programming to design our controllers as suggested in [126], but none of them produced results good enough to justify any more concrete work in this direction. As explained in Section 2.1 the MIMO nature of our system makes it substantially different from the examples considered by Koza.

is issued²⁷. We can use these metrics to measure how closely the controller approaches a

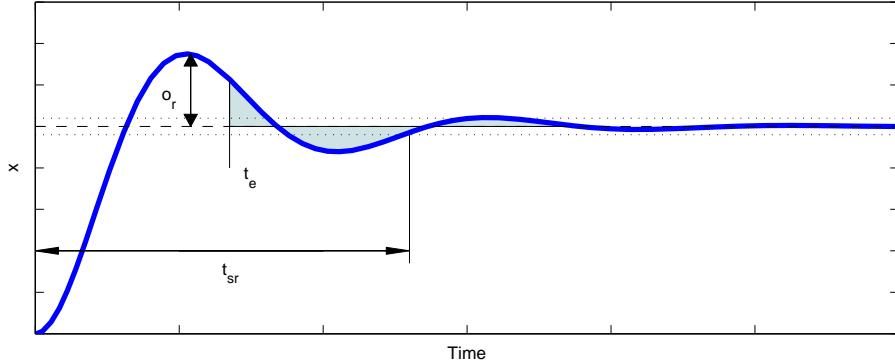


Figure 7.21: System response to a step input, relative overshoot o_r , relative settling time t_{sr} and absolute error (shaded area) are marked.

waypoint as well as how fast it reaches it, while also accounting for any oscillation. The error before time t_e is not considered in the absolute error computation, because as soon as a new reference is issued, a large error will necessarily be present²⁸.

Using a 3D waypoint with coordinates x, y, z we can compute the three metrics discussed above, but we also need a way to combine them. For settling time we can simply select the maximum from the three dimensions since we want to control oscillations in all dimensions. Similarly we can select the minimum value for overshoot. For the absolute error, it makes sense to use the Euclidean distance from the waypoint $d(k)$.

The fitness measure for the task of reaching a single waypoint therefore becomes:

$$f_{Wk} = -\min(\mathcal{O}) - \min(\mathcal{S}) + k_e \sum_{t=t_e}^{t_{end}} (d(k)_t + |\psi_k - \psi_t|), \quad (7.8)$$

where $\mathcal{O} = \{o_r^x, o_r^y, o_r^z\}$ and $\mathcal{S} = \{t_{sr}^x, t_{sr}^y, t_{sr}^z\}$ and $|\psi_k - \psi|$ is the absolute heading error²⁹.

The fitness function parameters ($k_e = 0.3$, $t_e = 0.5$) were based on the results of a few preliminary runs to give a balance between fast responses and limited oscillations. To allow the controller sufficient time to reach even the most distant waypoints, the window length t_{end} was set to 32 seconds.

The fitness function f_W measures the ability of a controller to reach a single waypoint. Since we are interested in producing a controller able to reach waypoints at arbitrary

²⁷In standard control theory the rise time (time to go from the 10% to the 90% of the target value) is usually considered. We omitted it since the evolutionary pressure of increasing t_{sr} is generally sufficient to ensure a low rise time.

²⁸This is akin to the ITAE error criterion [126] where the absolute error is scaled by the time index.

²⁹We used the exponent to indicate to which of the dimensions the overshoot and settling time refer to.

locations, the target waypoint is selected randomly during the training. The locations of the waypoints are chosen as follows:

$$\begin{aligned} x_k &= a(k) + \mathcal{U}[-1, 1] \\ y_k &= -a(k) + \mathcal{U}[-1, 1] \\ z_k &= \mathcal{U}[-2, 2] \end{aligned} \quad (7.9)$$

$$a(k) = \begin{cases} 0 & k = 0 \\ 1 & k = 1, 2 \\ -1 & k = 3, 4, \end{cases} \quad (7.10)$$

This ensures that the points chosen lie successively in front of, behind, to the left, and to the right of the helicopter's starting position. Since the aim is to fly the helicopter indoors, all the waypoints are generated within a cube with a 4m edge centred at the helicopter's starting position.

The fitness of a controller is computed by averaging the fitness f_{Wk} obtained by testing the controller on five different waypoints (i.e. $k = 0..4$):

$$f_W = \frac{1}{5} \sum_{k=0}^4 f_{Wk}. \quad (7.11)$$

The way the waypoints are chosen, and the use of fitness averaging, ensure a reasonably robust measure of controller capabilities.

To train the controller in a setup that is as close as possible to real scenario, the delay of the tracking system is simulated by delaying the simulated state as explained in Section 7.1.1.

7.2.2 Controllers

Controller structures using varying degrees of platform knowledge have been developed for the X3D quadrotor. Essentially they are straightforward extensions of the ones we tested for the toy car. The idea of doing this came from our focus on avoiding platform knowledge: if a controller design is truly platform independent, the same controller should be usable for different platforms. Our tests aim at finding out to what extent this is true.

PID

We begin with a multi loop controller based on a mixture of P, PD and PID loops. Such a controller is inspired by [89] and takes advantage of the stabilization controller already present onboard the X3D quadrotor (details in Section A.4).

The controller (in Figure 7.22) exploits the knowledge that the platform is symmetric about its x and y axes, and uses identical control structures for the longitudinal and lateral axes. Each of these controllers has a nested structure with an inner P loop and an outer PD loop. The inner proportional controller governs the attitude of the flying machine, and uses only proportional feedback since the stability augmentation system already provides velocity based control. The outer PD loop translates the relative distance to the waypoint (i.e. Δ_x and Δ_y), and the rate at which the distance changes, into a required attitude command so that the flying machine tilts and drifts towards the waypoint.

The thrust control uses a standard PID structure driven by the altitude error. In addition to the proportional term we use a derivative term to improve the speed³⁰ of the response since the vertical dynamics of the quadrotor is particularly slow. The integrative portion of the control compensates for any steady state error, a key feature since as we know (see Section A.3) the thrust associated with a throttle value depends on the battery level. As with the pitch and roll control, a proportional controller is sufficient for the yaw control. Each controller output is limited to the maximum and minimum values (i.e. $[-1, 1]$) allowed by the serial interface connected to the RC transmitter that we use to fly the machine (see Section 3.3.2).

While developing the controller we empirically determined a set of coefficients and limits that allowed the helicopter to be controlled adequately (see Table B.2).

In Section 7.2.3 we will discuss how the PID feedback constants were evolved, but in this case we start the training with all the constants equal to zero.

Monolithic Network

A monolithic Elman recurrent neural network is the simpler of the two platform agnostic controllers that we will consider for the X3D. The structure of the controller is identical to the one considered in Section 7.1.2, and is based on two recurrent layers, the second of which forms the output units of the network. Obviously the number of inputs and outputs

³⁰Since it allows us to use a higher P constant.

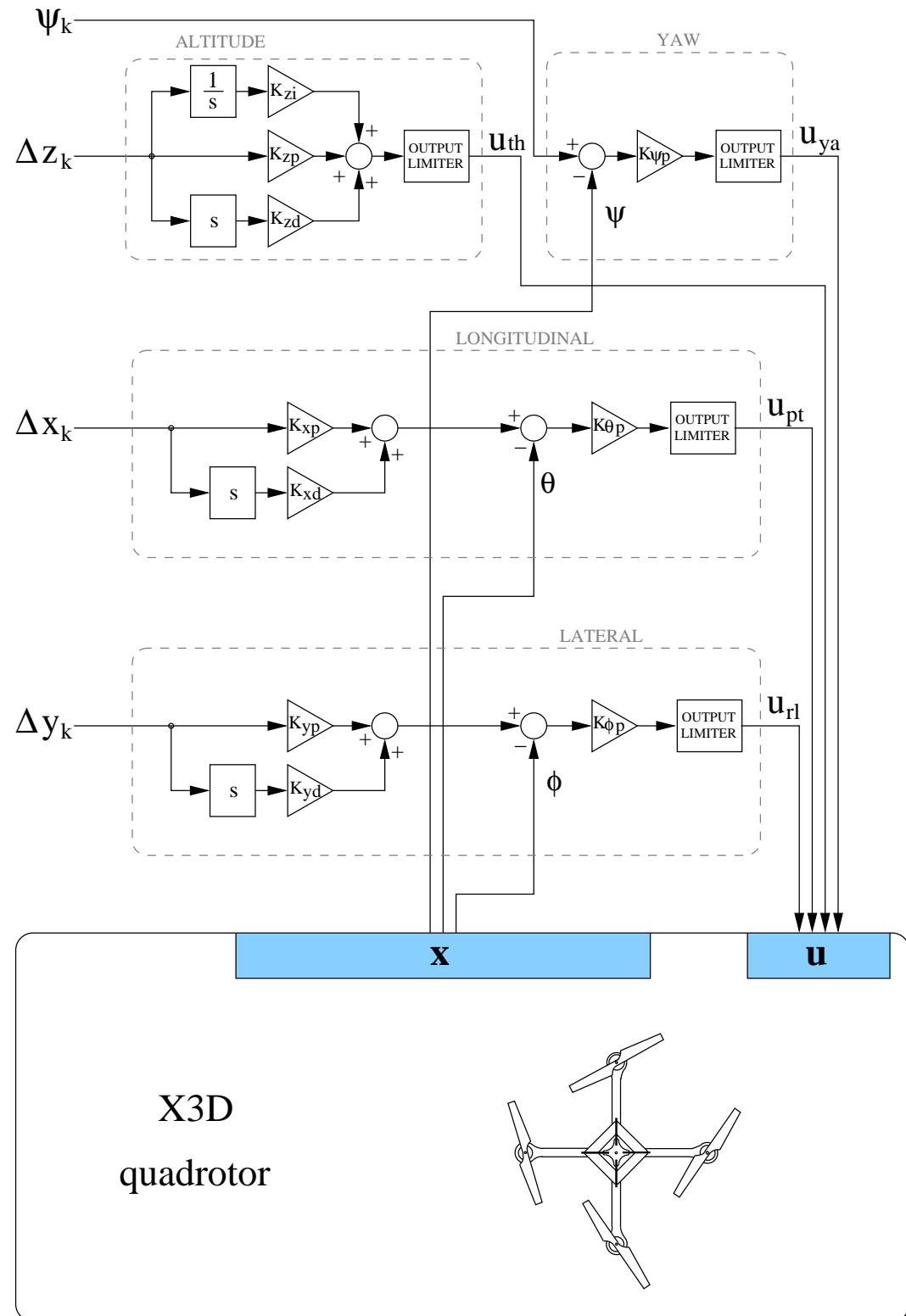


Figure 7.22: Hand crafted multi loop PID controller for the X3D quadrotor helicopter.

is different from the case of the car.

The inputs to the controller were the state of the quadrotor (\mathbf{x}), and its position relative to the waypoint k expressed in body coordinates $(\Delta_{x_k}, \Delta_{y_k}, \Delta_{z_k})$. The full set of inputs is:

$$\text{inputs} = [u, v, w, \phi, \theta, \psi, p, q, r, \Delta_{x_k}, \Delta_{y_k}, \Delta_{z_k}].$$

This simple controller therefore has 13 inputs (12 variables plus bias) and four outputs (u_{ya} , u_{th} , u_{pt} , u_{rl}). For the process of optimization the weights of the network are initialized to small random values drawn from a zero mean Gaussian distribution $\mathcal{G}(0, 0.1)$.

Modular network

In Section 7.1.2 we proposed a platform independent way to modularize a neural controller based on the idea of having one module for each one of the control outputs. In the case of the X3D this means four different networks, each of which controls one of the quadrotor inputs. For each module we used the same networks with recurrent topology (see Figure 7.3) that were used for the toy car modular controller. The inputs are the same as those used for the monolithic network (Section 7.2.2) and are passed to each of the RMLP modules, giving networks with 13 inputs, 4 recurrent units in the first layer and one recurrent unit in the second layer.

At the start of the optimization, the network weights are initialized as for the monolithic networks.

7.2.3 Training and Testing

The research reported in this thesis did not unfold chronologically in the order in which it is presented here. In particular, when we developed and tested the techniques for automatically designing quadrotor controllers, we had evolved deterministic models using genetic programming but we had not yet developed the method presented in Chapter 6 for estimating the parameters of the additive noise model. Unfortunately time constraints prevented us from being able to go back and repeat the whole procedure of training and testing the controllers using the *X3DAccGPbestNoise* model, primarily because testing on the real platform is very time consuming.

The noise parameters of the model used for the evolution of controllers had to be

manually set, and so for each dimension the noise standard deviation was chosen to have the same order of magnitude as the mean absolute value of the respective acceleration (Table B.3). We call this model *X3DAccGPSetNoise*. In the remainder of this section, and in the next, we will present results based on the *X3DAccGPSetNoise* since for those we can provide results on the real platform in the case of the PID controller and so test the transferability of the controller.

The different types of controllers we will examine require different training algorithms.

We trained the PID controller using the steady state algorithm based on Evolution Strategies presented in Section 7.1.3, since this is particularly well suited to optimize real valued parameters. The parameters of the PID were initialized to zero, and we used the same Gaussian mutation operator (with mutation rate $\sigma = 0.1$) that was used for the toy car controllers.

Based on preliminary trials, the size of the evolving population was set to 400 individuals, while the size of the elite was set to 100. Each of the controllers was evolved for 100 generations, and 30 independent evolutionary runs were executed to analyse the repeatability of the training.

The fitness progress of the best controller for each of the 30 repetitions of the training is shown in Figure 7.23. For the large majority of the repetitions the initial fitness is poor, but a fairly rapid improvement distinguishes the first few tens of generations. By the 70 generation mark most of the repetitions have converged to a similar fitness, but there are a small number of repetitions for which this is not the case.

While in general the ES algorithm produces good results, the behaviour visible in Fig-

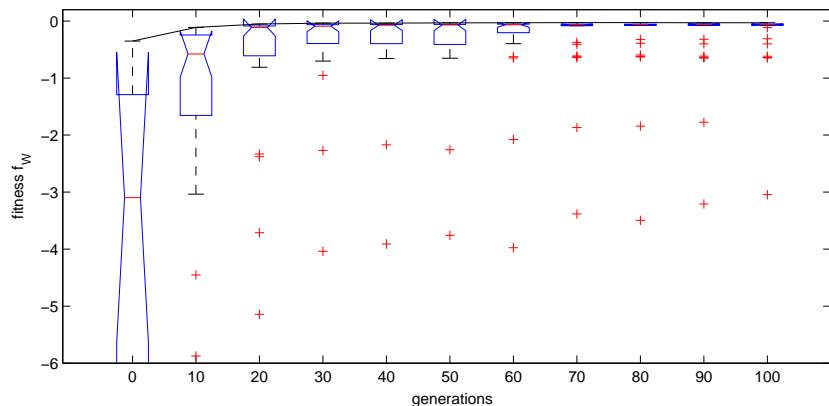


Figure 7.23: Fitness progress during evolution of the X3D PID controllers. The distribution is obtained from the fitness of the best model for each of the 30 repetitions.

ure 7.23 suggests that the fitness landscape produced by the task and controller structure combination is not trivial and might present several local minima in which evolution can get trapped.

Using the same settings for the task and the model, we then tackled the problem of evolving monolithic RMLP controllers using the standard ES algorithm. Initially we used the same population size and number of iterations found to be effective for the PID controllers.

In this case results were very poor; most of the controllers were barely able to keep the helicopter level, and were not at all able to maintain altitude or move towards the waypoint. Our attempts to improve the situation by increasing the size of the population used by the evolutionary algorithm, or increasing the maximum number of generations or the network size, did not produce any clear improvement. Although negative this is an interesting result since it is not difficult to show that, with the correct weights, the RMLP network we used can be mapped to a PID controller of the type used above. This points directly at the inability of evolution to find a solution, and relates directly to the scaling up problem discussed in Section 2.1.3.

It was then the turn of the modular RMLP controller, the second controller structure that does not depend on platform knowledge. The training adopted for this controller is the cooperative type described in Section 7.1.3. In this case, since the controller is formed by four modules, there are four populations of RMLP networks that cooperate. A population of 200 individuals is used, with an elite size of 20. As usual 30 repetitions of the whole training procedure were performed, and the fitness progress during evolution was recorded. In Figure 7.24 the fitness of the best individual for each repetition is displayed as a function of the generation number. In this case each evolutionary run lasted a total of 300 generations.

After some exploratory runs in which we varied the numbers of individuals and generations, we settled for larger values than those used for the PID controller, since the search space for the network parameters is much larger than that for the PID constants. Figure 7.24 shows that the fitness of the initial population is markedly lower than that obtained for the PID controller in Figure 7.23; this is not unexpected since the PID controllers already have a suitable structure, while the modular controllers might have structures totally inappropriate to the task. Despite the low initial fitness, in all 30 repetitions the coevolutionary

algorithm is able to make quick improvements, and after 180 generations all the repetitions converge to fitness values comparable to those of the majority of the PID controllers.

The single waypoint task we defined in section 7.2.1 allows us to have direct control over the different aspects of the system response, but of course our ultimate goal is to use the controllers for waypoint following. For this reason we set out to compare the performance of the PID and modular controllers on a more general waypoint following task. A 3D path is created as a sequence of waypoints at a fixed distance l from each other. Given a current waypoint, a new one (k) is created as a point with spherical coordinates α_k , β_k and l with origin at the current waypoint³¹. In order to produce smooth trajectories we chose a distance between waypoints of the same order of magnitude as the helicopter size ($l = 0.2m$), and to avoid rapid changes of direction we chose the elevation (α_k) and azimuth (β_k) angles between consecutive waypoints to be dependent:

$$\alpha_k = \alpha_{k-1} + \mathcal{U}(0, \pi/6) \quad (7.12)$$

$$\beta_k = \beta_{k-1} + \mathcal{U}(0, \pi/8). \quad (7.13)$$

where for smoother changes in altitude a smaller variance than the one for α was chosen for the increments of β . These parameters deliver trajectories (see waypoints in Figure 7.25) roughly similar to those flown during data collection.

During flight, the quadrotor will receive a new waypoint when within a certain distance (switching distance) from the new one. This distance was set to l . When moving from the single waypoint task to the waypoint following task, we have to remember that, by

³¹The azimuth axis of all the local frames are parallel to the global earth-fixed x axis

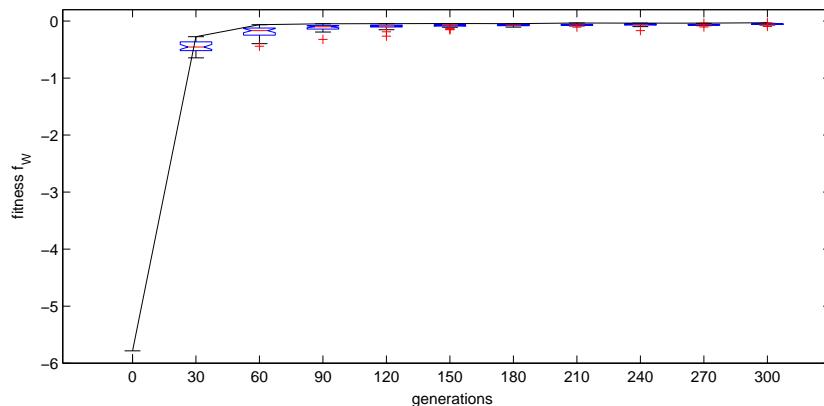


Figure 7.24: Fitness progress during evolution of the X3D modular RMLP controllers. The distribution is obtained from the fitness of the best model for each of the 30 repetitions.

construction, the maximum offset between the quadrotor and a waypoint never exceeds $2m$ in each of the three coordinates. Since the waypoint chain does not imply any such limit, we limit the values Δ_x , Δ_y and Δ_z delivered to the controller³² to $2m$.

The definition of fitness obviously changes for this type of path following task, and so, as with the toy car, the fitness f_P is simply proportional to the number of waypoints traversed in the allotted time³³. The fitness of ten repetitions of the task is averaged to give a more robust estimation of performance than a single trial.

The statistics for the fitness of the 30 controllers in each of the modular and PID populations are shown in Table 7.6 and in the box and whiskers plot of Figure 7.25. Both

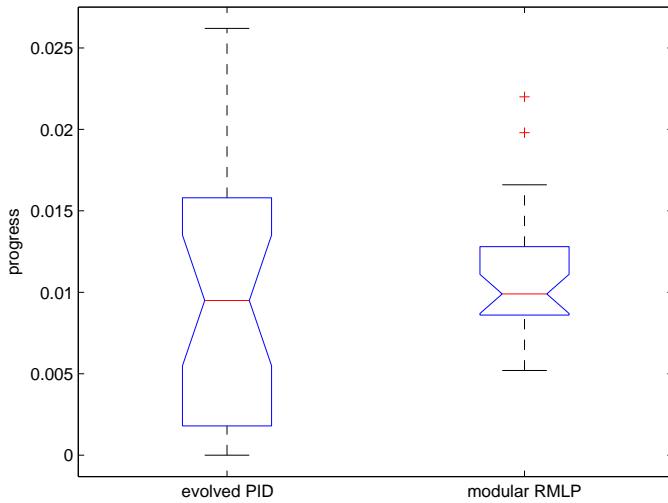


Figure 7.25: Comparison of performance between PID and modular controllers on the path following task. The means of the two sets of controller are not significantly different.

Controller type	$max f_P$	$mean f_P$	$s.d.$
evolved PID	0.026	0.010	0.0077
modular RMLP	0.022	0.011	0.0039

Table 7.6: Performance of the various controllers on the path following task.

populations of controllers have very similar mean performances, but as was for the case of the single waypoint fitness, a larger standard deviation is seen in the performances of the PID controllers. The set of evolved PID controllers also contains the best controllers. However, a statistical comparison between the two populations (Mann-Whitney U test

³²This is particularly important for the real world test to avoid the helicopter receiving large inputs as a consequence of wandering away from the path.

³³Since this fitness is used only for testing, a controller is not penalized for not staying within a certain distance of the path.

$P < 0.01$) reveals that overall the means of the two populations are not statistically different.

Finally we take a qualitative look at the trajectories followed by the best examples of each type of controller ($X3DbestPID$ and $X3DbestCoCoRMLP$ respectively for the PID and the modular network controllers). Figure 7.26 shows six trajectories randomly generated in the same way as was used to assess the controller fitness. As expected given its higher fitness, the PID controller appears to have better path following abilities. In particular the $X3DbestCoCoRMLP$ appears less able to control its altitude, and as a consequence performs less well on trajectories with larger altitude variations (e.g. Figure 7.26 (e) and (f)). This was also confirmed by manually inspecting several other trajectories. Manual inspection also revealed that the pitch and roll behaviour of the two controllers is different, the $X3DbestPID$ delivering a predominantly level flight while the $X3DbestCoCoRMLP$ tending to pitch and roll more when moving to a new waypoint.

7.2.4 Real X3D Testing

The controllers with good performance on the waypoint task could now be tested on the real platform. Unfortunately, due to time constraints, we were only able to test the $X3DbestPID$ controller on the real helicopter. To do this, we first created a trajectory to be followed in the form of a series of waypoints. To simplify the experimentation, a simple constant altitude U shape was chosen for the trajectory (see Figure 7.28), making it easy to ensure that during the test the quadrotor would always remain well within the tracking volume³⁴. It is worth noting that this does not constitute an easy trajectory to follow since the trajectory presents two very sharp turns (i.e. 90 degrees), and the series of aligned waypoints allows the helicopter to build up considerable speed during the run.

To carry out the experiment, the code that feeds the next waypoint to the controller was modified so that the current helicopter position could be used as a waypoint, and so we could have the helicopter hovering in any place we needed. To execute the tests we started with the helicopter hovering in close proximity to the first waypoint; and fed the controller with the first waypoint, the helicopter would start travelling along the defined path until reaching the last waypoint where it would stop, and after a few seconds safely land.

³⁴We used the same spacing l used for testing the controller performance.

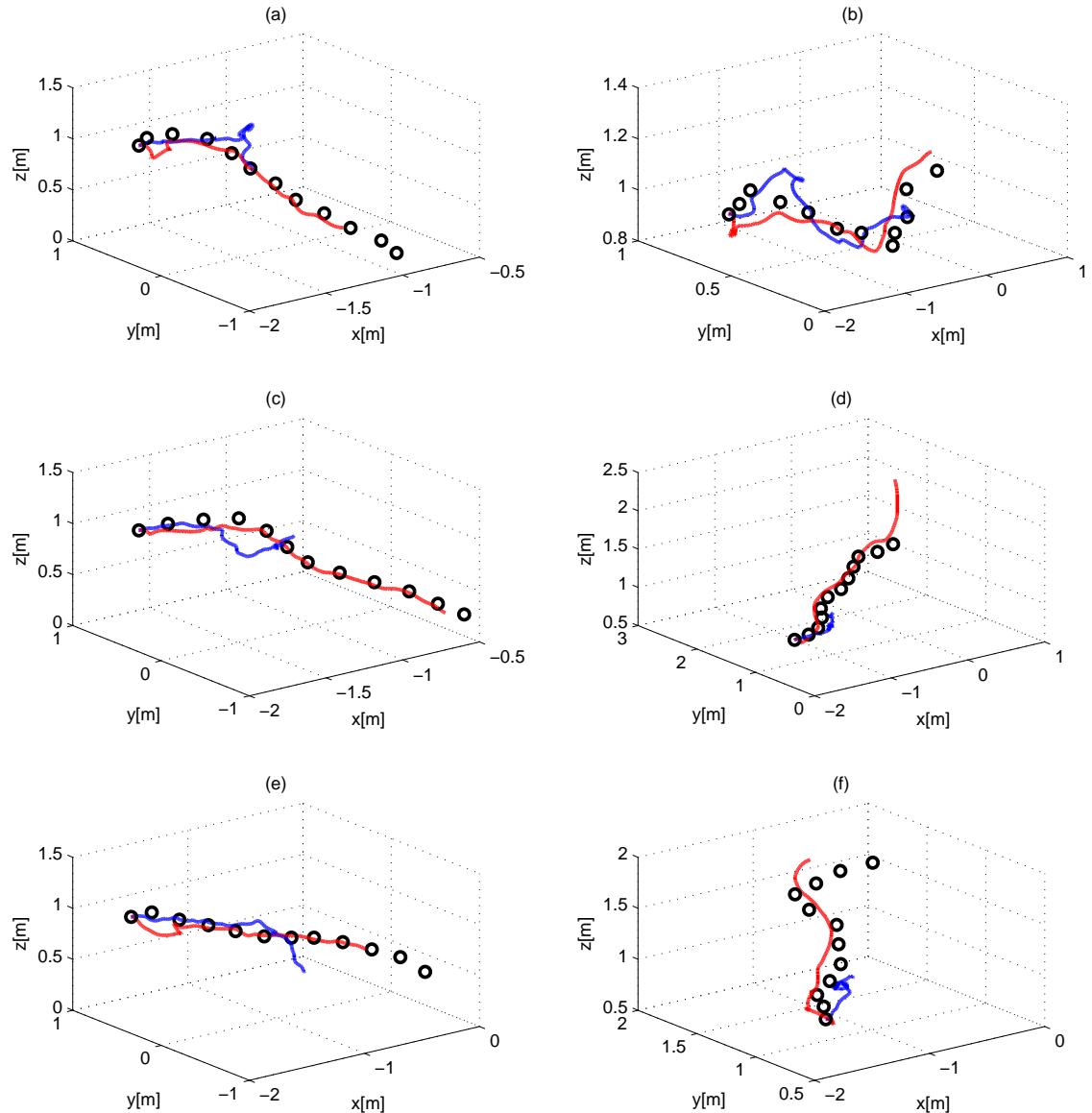


Figure 7.26: Trajectories followed by the *X3DbestPID* and *X3DbestCoCoRMLP* controllers on 6 randomly generated paths.

The test was repeated ten times; the trajectories obtained are plotted in Figure 7.27³⁵.

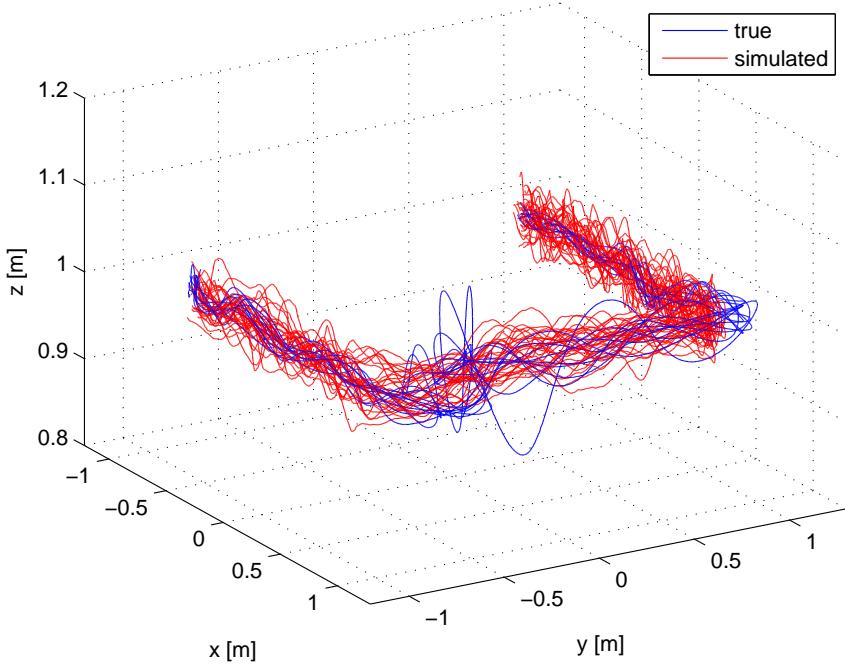


Figure 7.27: Actual versus simulated trajectories for the *X3DbestPID* controller.

In all ten repetitions the *X3DbestPID* controller successfully flew the quadrotor from the first to the last waypoint, and was able to maintain a stable altitude. We can therefore say that the controller successfully transferred to the real platform. It is interesting to note that, at the time of publication, this was to the best of our knowledge the first successful transfer to a real platform of a quadrotor controller evolved exclusively in simulation.

By computing the deviation from the predefined trajectory³⁶ (Table 7.7) we obtained a mean error in position and altitude of $0.08m$. The maximum error was an order of magni-

	<i>mean</i>	<i>max</i>	
x-y deviation	0.08	0.34	<i>m</i>
altitude deviation	0.08	0.11	<i>m</i>

Table 7.7: Maximum and mean deviation from the reference trajectory computed on the 10 repetitions of the real task.

tude larger at $0.34m$ for the x - y position and $0.11m$ for the altitude. Such a performance is certainly respectable especially if we consider the level of automation of our procedure.

With the help of separate plots for the x - y and z axes (Figures 7.28 and 7.29) we can

³⁵A video recorded during one of the runs is also available from the author's web page.

³⁶The deviation is computed as the orthogonal distance from the reference trajectory.

look in more detail at the behaviour of the controller, and can also compare its trajectories with the trajectories obtained in simulation.

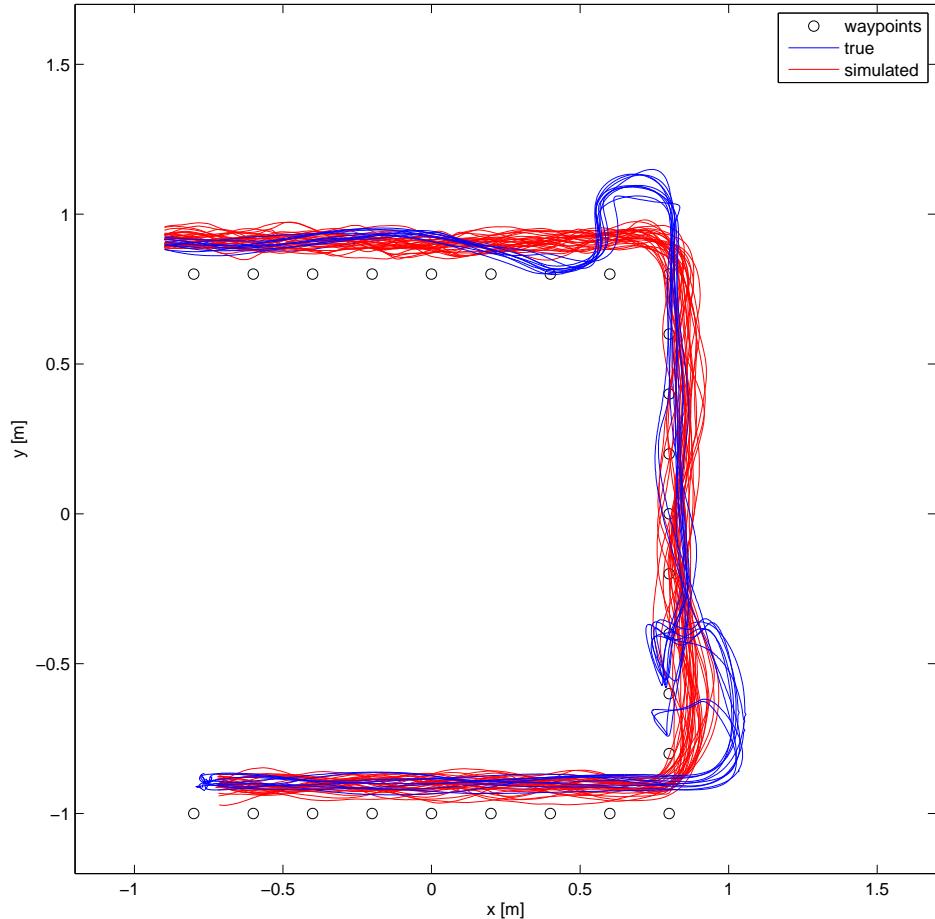


Figure 7.28: Actual versus simulated trajectories for the *X3DbestPID* controller (x and y coordinates only).

Looking at the x - y plots we see that all the trajectories seem to have a positive offset in the y direction, but this is not the case for the x direction³⁷. It is clear however that the sharp changes of direction are the most problematic points for the controller, which is not able to prevent the platform from proceeding too far in the previous direction of travel as a result of inertia. As a consequence, when the direction changes the controller has to make up for the error while also proceeding in the new travel direction, resulting in a localized tracking error.

The altitude control shows good abilities, with lower performance at the locations that are also problematic for the lateral and longitudinal control. The fast change in orientation of the flying machine affects the direction of the propeller thrust, with obvious effects on

³⁷This type of error is compatible with the type of controller we are using since our PID based controller does not employ an integral component for the lateral and longitudinal loops.

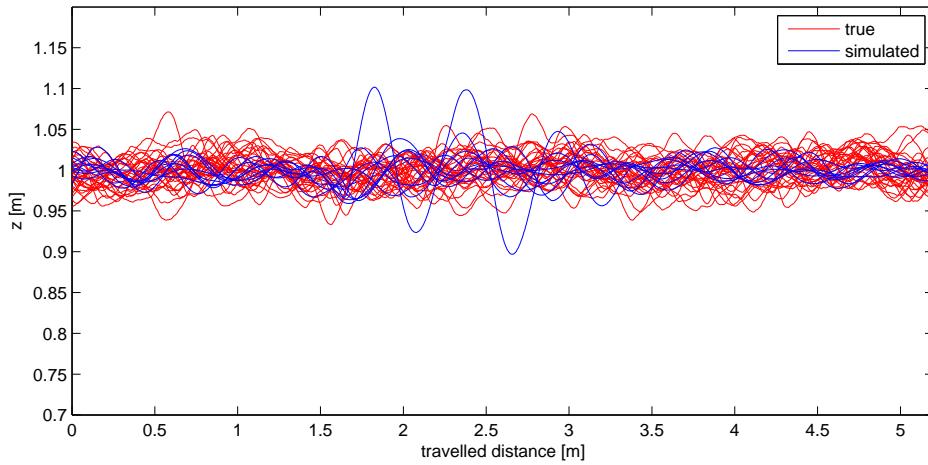


Figure 7.29: Actual versus simulated trajectories for the *X3DbestPID* controller (z coordinate only).

the balance between lift and gravitational force.

Comparing the real and simulated x - y plots we see very good agreement on the straight stretches of the trajectory. Interestingly, the simulated trajectories have the same y offset that the real trajectories have; the spread of the two sets of trajectories is also similar. The similarity breaks down at the turning points where the simulated trajectories show a smooth transition; this is a situation in which the model³⁸ does not fully mimic the real machine. Similar considerations also apply to the altitude, where it is clear that for one of the runs the error was particularly large.

To determine in a more quantitative fashion the level of agreement between the real and simulated trajectories, we compared the normalized number of traversed waypoints (progress) at $t = 20s$ ³⁹. This metric is interesting since it introduces the time dimension into the comparison, an element that is not present when comparing trajectories spatially. The fitnesses computed across 10 repetitions with the real helicopter, and over 30 simulations with the model, are shown in Figure 7.30 and in Table 7.8. Despite the localized

	<i>real</i>	<i>simulated</i>
median	0.75	0.71
s.d.	0.075	0.017

Table 7.8: Median and standard deviation (s.d.) of the fitness progress obtained on the U-shaped trajectory in simulation and in reality.

³⁸Deterministic and stochastic parts.

³⁹We have chosen this instant in time since it is the furthest for which the helicopter did not reach the end point in any of the repetitions.

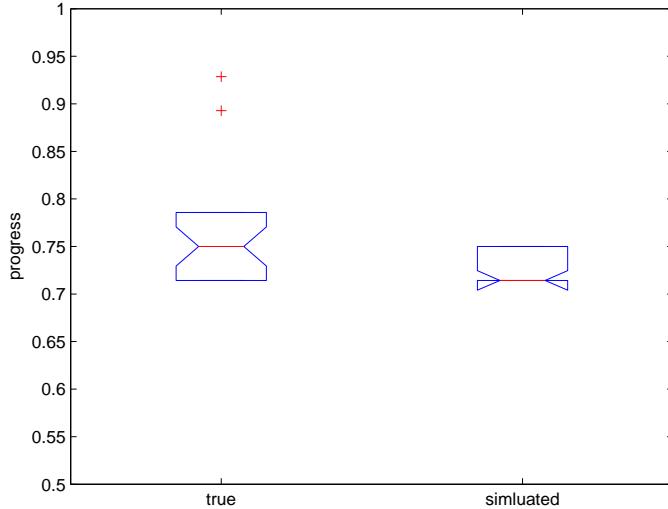


Figure 7.30: Distribution of progress reached by 30 simulated and 10 real repetitions of the waypoint following task on the U-shaped trajectory.

differences in trajectories, in terms of progress the simulated and real controllers appear very similar, and in fact a statistical test (Mann-Whitney U test) reveals no significant difference. Observing the helicopter while executing the task revealed that the localized instances of large error were also characterized by a faster motion of the helicopter. Of course in a PID the control inputs are mainly proportional to the error, so this is not surprising.

Conclusions

In these interesting sections on the X3D quadrotor we tested our controller design methodology on a more complex platform. As we expected, handcrafted controllers (PID) can be successfully trained, but our results suggest that the fact that the structure of the controller is fixed does not necessarily lead to a well behaved search space.

The failure in evolving a basic platform agnostic controller consisting of a single RMLP underlined the fact that the dynamics of the quadrotor platform makes for a non trivial control problem. Although a successful controller can be formed from such a network, evolution was simply unable to reach a good solution. The problems of neural and genetic interference that we discussed in Section 2.1 are possible, and indeed probable explanations.

Although based only on knowledge of the number of outputs, the modularization of the controller into separate networks, combined with the use of cooperative coevolution, showed itself able to lead to effective controllers with performances generally comparable to

handcrafted controllers, at least in simulation. This is in agreement with the line of research that suggests modularity as a way of scaling up evolution to tackle complex problems (see Section 2.1.3).

The PID controller showed an ability to transfer to the real platform and deliver the desired behaviour. Quantitatively the tracking performance was good, but specific circumstances (i.e. sharp turns) can lead much larger errors. Simulated and real trajectories are still not in full agreement and local discrepancies can appear, giving rise to the feeling is that in situations in which the behaviour of the platform is highly dynamic, our model might underestimate its error in comparison to the real flight machine. This problem might be solved with a more sophisticated noise model that is dependent on the state and input of the model. Such a model could potentially capture the information that the deterministic model performs worse in specific circumstances, leading to the evolution of more robust controllers.

If we compare the performance of our evolved controller with what has been published in the literature on manually designed controllers [89, 100], we find that in terms of maximum error our results are similar but not as good as the best among the published examples. [89] claims an accuracy of $\pm 0.1m$ in position and $\pm 0.04m$ in altitude, while [100] claims $\pm 0.35m$ and $\pm 0.10m$ for position and altitude respectively. Having said that, we stress again that the main aim of this research is to investigate if it is at all possible to produce useful models and controllers without using platform knowledge, rather than to produce models and controllers better than those using domain knowledge. Having shown that this is indeed possible, is not difficult to imagine that with the availability of more powerful computers and improved evolutionary algorithms, in forthcoming years it will be possible to not only match but to exceed the performance of controllers designed with more traditional methodologies.

7.3 The Autopilot Helicopter Simulator

The last, but not the least challenging of the platforms that we investigated, is the Autopilot helicopter simulator (see Section 3.3.5).

As we saw in Section 2.3, a large body of work and in depth understanding has been developed dealing with the modelling of single rotor helicopters, and miniature single rotor

model helicopters in particular. For this reason we do not consider the modelling of this kind of platform to be of primary interest, especially given the resources required to work with this type of flying machine⁴⁰. In this section we tackle only the more interesting problem of automatic controller design. We are confident that the approach of Chapters 5 and 6 could be applied to the modelling of a single rotor helicopter, and might possibly achieve good results. Our confidence is based on the fact that other authors have shown that, for a single rotor helicopter, even a linear model ([5]) can be trained from real data and used to design a controller. As we know, our modelling methodology can handle both linear and nonlinear systems, making it well suited for the task. Additionally our method has proved to be effective with systems (i.e. the quadrotors) with the same number of dimensions and the same number of inputs that a single rotor helicopter would require.

Although the simulator is not a real platform, and we will not be able to test the transferability of the obtained controller, it remains an interesting platform since it is dynamically unstable, a quality that has strong implications for the design of control systems. The dynamic model used in the simulation is an open source implementation of a single rotor helicopter model based on first principles which replicates many of the characteristics that make single rotor helicopters notoriously difficult to fly (see Section 3.3.5 for more details).

7.3.1 Task and Fitness

From the point of view of its flying abilities in near-hover, a single rotor helicopter is similar to a quadrotor helicopter⁴¹ and so a hierarchical approach to the problems of navigation and control is often used. Paths and waypoints are the building blocks from which more complex trajectories are built ([210, 41, 253, 211]), and so it is relevant to develop controllers for a waypoint following task.

The task is similar to the trajectory following task used for the X3D quadrotor in Section 7.2.3. At the start of the trial an ordered chain of waypoints is randomly generated. The position of each new waypoint is generated by drawing a random angle $\gamma_k \in \mathcal{U}[0, 2\pi]$ and a radius $l_k \in \mathcal{U}[10, 35] \text{ft}$ ⁴². Those represent the position of the new waypoint k

⁴⁰In the case of a single rotor helicopter is not just a matter of purchasing a platform but also of having a skilful pilot who can actually fly the machine with the precision needed when working indoors.

⁴¹In more advanced flight regimes, single rotor helicopters are also capable of aerobatics and inverted flight, feats that are not possible for a quadrotor.

⁴²The autopilot simulator uses imperial units, and so in this section we will use ft and $\frac{\text{ft}}{\text{s}}$ for positions

in polar coordinates relative to the last waypoint $k - 1$ (see Figure 7.31). In contrast,

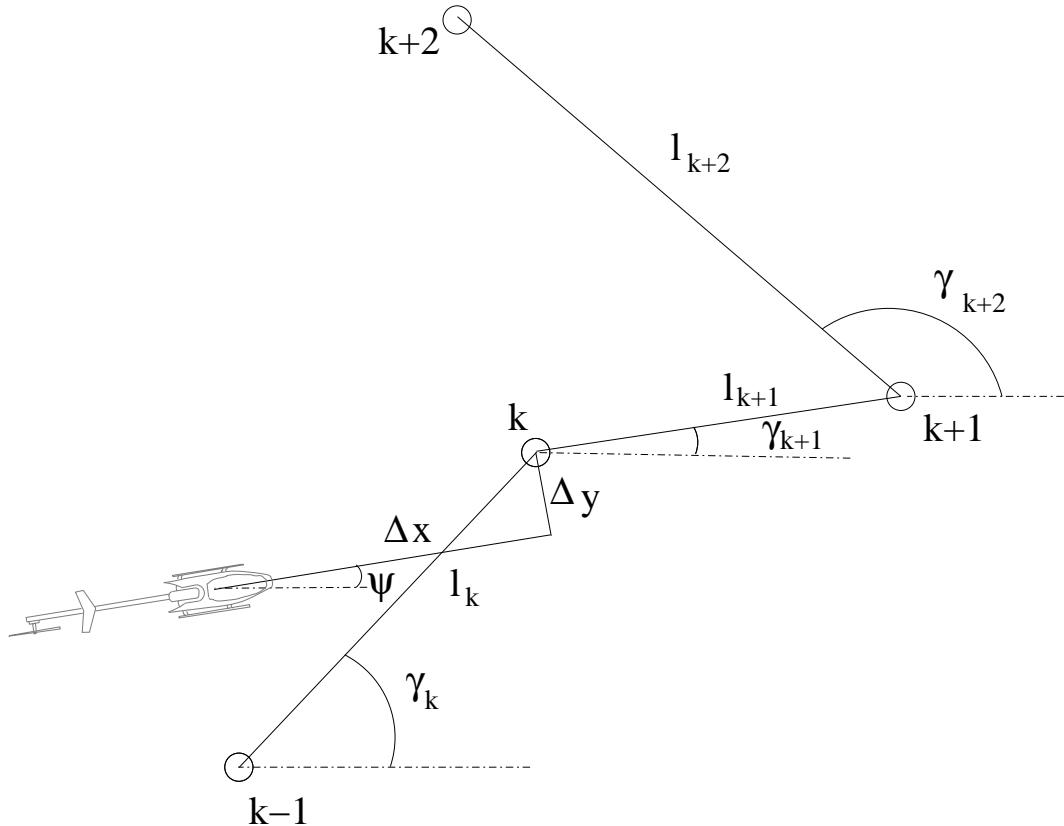


Figure 7.31: Typical waypoint chain automatically generated for controller evolution. The waypoints are in the 3D space and have different altitudes.

the altitude is independently generated for each of the waypoints by drawing it from the uniform distribution $\mathcal{U}[25, 30] \text{ ft}$. Since in a helicopter the heading can be controlled independently of the direction of travel, a target heading ψ_k was also generated randomly for each waypoint, the idea being that the helicopter should have this heading when reaching the waypoint ($\psi_k \in \mathcal{U}[0, 2\pi]$)⁴³. The parameters used for the generation of waypoints were chosen manually taking into consideration the size of the helicopter platform represented by the simulator. The choice $\gamma_k \in \mathcal{U}[0, 2\pi]$ makes possible drastic changes of direction in the path, a feature that makes the control even more challenging.

The fitness function used for the evolutionary algorithms is based on the number of waypoints along the path visited in the correct order. In order not to impose unrealistic precision requirements, a waypoint is deemed to be visited if the centre of the helicopter and velocities respectively.

⁴³This has useful applications. For instance in a UAV surveying the location specified by the waypoints, we would be able to choose the direction at which to aim our sensors (i.e. a camera).

approaches within 1 foot of it⁴⁴. The full fitness function (f_H) depends not only on progress along the track, but also on the orthogonal distance from the shortest path between the waypoints, and on the ability of the controller to maintain the correct altitude and heading:

$$f_H = \frac{\sum_{t=0}^N (w_h P_{chain} |z - z_k| - |\psi - \psi_k|)}{N}. \quad (7.14)$$

Where N is the number of time steps allowed to execute the task ($N = 1000$ was used for the evolution since it was empirically found to be a good compromise between allowing the helicopter enough time to reach several waypoints, and not making the task too computationally expensive) and z_k , ψ_k are respectively the altitude of the next waypoint and the desired heading at the waypoint. The factor w_h is equal to one if the helicopter is on the shortest path between two waypoints and decays as the cube of the orthogonal distance from it, thus penalizing controllers that do not follow the shortest path. In the early stages of development, crashes into the ground were frequent, so if this happened the fitness was set to a negative value equal to the difference between N and the number of elapsed time steps. As a consequence of this choice, in the beginning, the controllers that spent the longest time aloft were selected as members of the elite even if they were initially unable to follow the path, thereby helping the early stages of the evolutionary process.

Since the waypoint path is generated randomly for each trial, the fitness in one trial alone is not fully representative of the controller's ability; to obviate this we compute the fitness as the average of 5 independent trials⁴⁵.

7.3.2 Controllers

As with the toy car and the quadrotor, several different controller structures will be considered; some will be based on platform knowledge, and others will be completely platform agnostic. However, we will also mix the two approaches in an attempt to identify the key features that differentiate their evolvability.

⁴⁴For comparison, the main rotor radius is 2.25 ft (see other physical characteristics in Table 3.5)

⁴⁵We empirically found 5 trials to be a good compromise between having a representative measure of fitness and reducing computational requirements.

PID

The simulator comes with a hand crafted PID controller, tuned to perform waypoint following. The controller (see Figure 7.32) is designed as four separate parts: two single loop PID modules for controlling yaw and altitude respectively, and two modules controlling longitudinal and lateral motion. The lateral and longitudinal modules each contain two nested PID loops, the inner controlling the roll angle ϕ (pitch angle θ in the case of the longitudinal module), and the outer controlling the distance from the waypoint in terms of body coordinates y (x). The controller is completely specified by the values of the feedback constants; there are eighteen in our case (i.e. K_{xi} , K_{xp} , K_u , K_{yi} , K_{yp} , K_v , K_{zi} , K_{zpi} , K_w , $K_{\theta i}$, $K_{\theta p}$, K_q , $K_{\psi i}$, $K_{\psi p}$, K_r , $K_{\phi i}$, $K_{\phi p}$, K_p).

The inputs to the controller were the helicopter state:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r],$$

plus the vector to the next waypoint k in body coordinates (Δ_{xk} , Δ_{yk} , Δ_{zk}), and the desired heading associated with the next waypoint ψ_k (refer to Figure 7.31 for a graphical depiction of the input quantities).

This controller, although not optimal, provides a good baseline against which to compare our evolved solutions. In addition, its modular structure inspired the topology of the network that we discuss in Section 7.3.6, and also proved useful during the training phase (see Section 7.3.5).

In Section 7.3.6 we will also evolve the feedback constants of this controller structure so that a fair comparison with the network based controller can be made.

Neural Networks

As controllers that employ limited or no knowledge about the platform, we chose various types of controllers based on artificial neural networks.

The controllers are based on feed-forward neural networks, using the tanh activation function at all nodes. Some of the networks are organized as multi-layer perceptrons (MLPs) with two layers of weights (see Figure 7.33), but others use less connected topologies (see Figure 7.35). We will discuss the topologies and the rationale behind our choices in detail in the forthcoming sections.

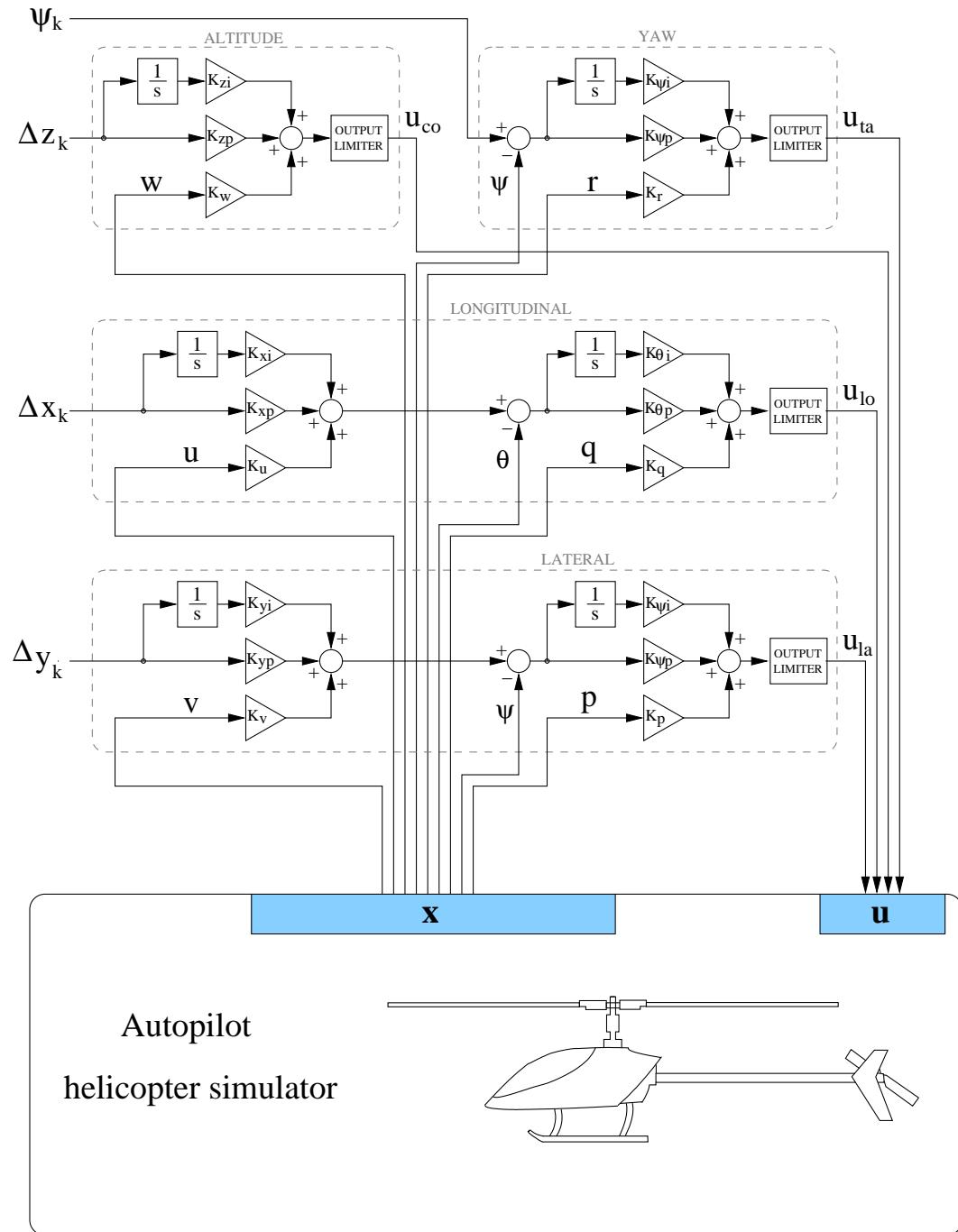


Figure 7.32: Hand crafted multi loop PID controller for the single rotor helicopter modelled by the Autopilot simulator.

The networks are fed the same information that is delivered to the PID controller: the helicopter state \mathbf{x} and the information about the position of the current goal (the next waypoint) relative to the position of the helicopter ($\Delta_{xk}, \Delta_{yk}, \Delta_{zk}, \psi_k$). Since the output of a neuron is limited to be between -1 and 1, the control outputs were appropriately scaled in order to guarantee that the network could span the full range of the control values if needed. The outputs of the neural network were then treated as some or all of the actuator commands to be sent back to the simulator.

7.3.3 Controller Training

The connection weights of the neural networks and the feedback constants of the PID controller are trained by an evolutionary algorithm. The algorithm used is essentially a (10+23) Evolution Strategy of the type described in Section 7.1.3. Again we use a steady state algorithm that does not employ recombination.

In some of the experiments self-adaptive mutation was used ([22]). With this technique, the mutation magnitude D becomes part of the parameter set of an individual and is therefore under evolutionary control. When self-adaptive mutation is in use, each mutation operation is preceded by a step in which, following standard practice ([64]), we mutate D as follows:

$$D = De^\eta \quad \eta \in \mathcal{G}(0, 1).$$

After this step the remaining parameters of the individual are mutated using the newly generated mutation magnitude.

The idea behind this technique is that if an individual repeatedly achieves good fitness, not only its parameters but also its mutation magnitude will be better than those of the other elements in the population.

7.3.4 A Partial Overview of Preliminary Non-Working Approaches

In this section we survey some of the more noteworthy unsuccessful approaches that eventually led to the successful methods presented in Sections 7.3.5 and 7.3.6. Discussing this work is relevant since it will shed some light on the key aspects that make the learning difficult.

(Not) Evolving Monolithic Networks

In the first approach, we started by avoiding the use of any platform specific knowledge and tried to evolve a full MLP with two layers of weights, using the standard version of the evolution strategy algorithm presented in Section 7.1.3, and the fitness function f_H (equation 7.14) based on progress along the waypoint chain.

Results were not encouraging. In some cases, no goal-directed behaviour at all was observed, save that of maintaining enough altitude not to crash into the ground. In some other cases, the helicopter drifted slowly towards the first waypoint, but rarely reached it. In all cases, the helicopter was continuously spinning around its z axis, though the speed with which it did this varied.

The reason for this behaviour might not seem obvious at first, but a single rotor model helicopter is not a stable system, and to maintain a fixed heading the tail rotor has to be constantly adjusted in order to counteract exactly the torque generated by the main rotor. Without the right tail control input, any helicopter of this kind will always tend to spin; this fine control is not an easy operation, and in practice an electronic feedback system based on a gyroscopic sensor has to be used to fly such machines manually.

Guessing that the failure to learn any waypoint following behaviour was due to the continuous z -rotation, we decided to test this hypothesis. Our primary aim is to automatically develop controllers without using platform specific knowledge, but more importantly, since we are addressing a research question, we also want to understand what the limiting factors of our approach are. Using part of the original PID controller as a well specified and limited amount of platform knowledge, we removed yaw control from the neural network and let part of the original PID handle the yaw. This made sure that the helicopter heading was always stable for the duration of the task. The neural network retained control over the three other actuator dimensions. However, controllers evolved with this setup were only marginally better than those with neurally controlled yaw, suggesting that something more complex than yaw control was preventing the learning.

As we discussed in Section 2.1.3, when a single network is required simultaneously to learn different functions (which in our case correspond to the four controls), the interactions between the different components of the controller can affect learning to the point of hindering it. Modularization has been proposed ([40]) to obviate such problems by splitting the network into separated functional blocks. Following these findings, we decided to divide

up the monolithic MLP controller into modules. Still using the PID to control the yaw, we tried a variety of network modularizations and hybrid network-PID solutions which ultimately led us to the networks used in Section 7.3.6 and 7.3.5, and which proved able to control the flying machine.

(Not) Evolving Yaw Control Together with The Rest of The Task

Our next step was to evolve the yaw control at the same time as the control of the other three actuator dimensions. With this aim, a neural network module (see Sections 7.3.6 and 7.3.5) was substituted into the PID yaw block. In our experiments we were indeed able to evolve stable yaw, but only if the trials lasted for 50 time steps or less. Evolutionary runs using trials longer than 50 time steps did not even achieve yaw stabilization, and consequently waypoint-following did not appear. We had the clear feeling that the fitness reward for moving forward in the waypoint chain was working against the yaw stability as the length of the trials increased. A variety of alternative fitness functions were tested, always including progress along the track, but also trying out such factors as rotational speed, angular error in orientation, and so on. None of these proved to be really successful.

These findings, along with those of the previous section, suggested that learning to control the simulated single rotor helicopter is complex because of the necessity of operating the four controls simultaneously, combined with the intrinsic instability of the flight machine.

The first issue can be addressed by modularization, but for the second we need to stabilize the machine before moving onto learning waypoint following. As explained in Section 2.1.2, incremental evolution is a technique often suggested for scaling the evolutionary algorithm for tasks that are too difficult to be solved directly. The evolution is split into different phases of increasing complexity; in our situation the split was almost natural, as the first short evolution dealt with yaw stabilization, and the second phase tackled waypoint-following. For the first phase the standard fitness function (equation 7.14) was replaced by the following:

$$f_Y = \sum_{t=1}^N |\psi - \psi_k|, \quad (7.15)$$

where N is the number of time steps, ψ is the instantaneous heading and ψ_k is the heading required by waypoint k . When evolving waypoint-following, the fact that the fitness

includes a factor for maintaining the correct heading ensures that good yaw control is preserved.

We now describe in detail the modular controllers developed during our experimentation, and consider their performance. In the next four sections we will mainly focus on technical details and comparisons between types of controllers, and will defer to Section 7.3.8 the extremely important in-depth analysis of the amount and type of knowledge used in our approach.

7.3.5 First Working Approach: Incrementally Substituting a PID with Neural Networks

Given the poor results of the first attempt to evolve a controller based on a full MLP (see Section 7.3.4), we decided to use the PID controller delivered with the simulator to enforce functional separation in the network. The first phase was the evolution of the yaw controller. For this purpose a very simple neural network, with four connections in total (see the yaw module in Figure 7.33), was evolved to stabilize the yaw using the fitness function of equation 7.15. Each trial lasted for only twenty time steps, and evolution produced a good solution within several tens of generations. During the second phase, the yaw network was free to evolve along with the rest of the controller.

The second phase was divided into three steps; in all steps the full fitness function (equation 7.14) was used. In step 1, a three layer MLP was substituted for the PID controller's guidance layer; it took as its input the distances from the waypoint (Δ_x , Δ_y , Δ_z), and the velocity of the helicopter (u , v , w , p , q , r) and had the desired pitch and roll attitude as outputs to the longitudinal and lateral PIDs.

In step 2, the two inner PID loops (the inner parts of the longitudinal and lateral PID modules - see Figure 7.32) were replaced by two separate MLPs. These were evolved to act on the information given by the neural outer layer, and they output the helicopter motor commands (u_{la} , u_{lo} and u_{co}). After this step, we had a working helicopter controller consisting solely of neural networks (Figure 7.33).

In step 3, the outer network layer, which was frozen during step 2, was allowed to evolve further, along with the inner network layer. In this way, the networks could co adapt to each other, potentially allowing them to exploit modes of cooperation not possible for the purely linear PID controllers.

The controller based on the inner network reached a good fitness level within 200 generations in all ten repetitions of the experiment. The evolution of the outer network showed more variability, but within 500 generations an outer network had been evolved in all replications which gave the controller a reasonable, if not good, performance. When both networks were evolved further together, however, very good performance was reached in every repetition (see Figure 7.34).

7.3.6 Second Working Approach: Incremental/Simultaneous Evolution of Modular Networks

The first phase, evolving a simple yaw stabilizer, was repeated exactly as in the previous working approach. For the rest of the controller, three relatively simple custom-topology neural networks were evolved simultaneously using the standard fitness function. The three networks output longitudinal cyclic control, lateral cyclic control, and the collective pitch; the topologies of the networks are depicted in Figure 7.35. The longitudinal network had the following inputs: longitudinal distance to waypoint (Δx), u , q and θ . Similarly, the lateral network made use of the lateral distance to the waypoint (Δy), v , p and ϕ . The collective pitch network used the difference in altitude of the waypoint (Δz) and w .

The experiment was replicated ten times. The variability in fitness within the first few

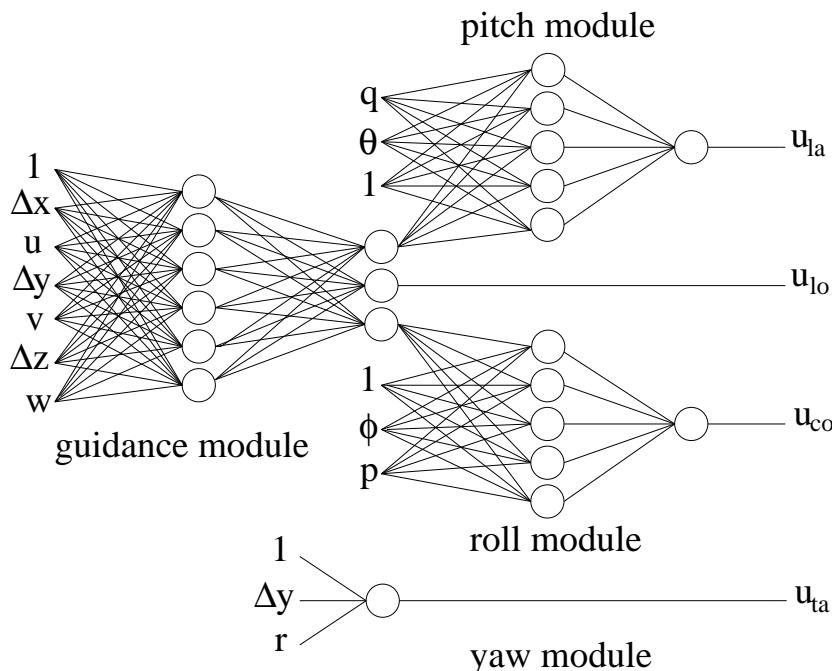


Figure 7.33: Topology of the substitution network.

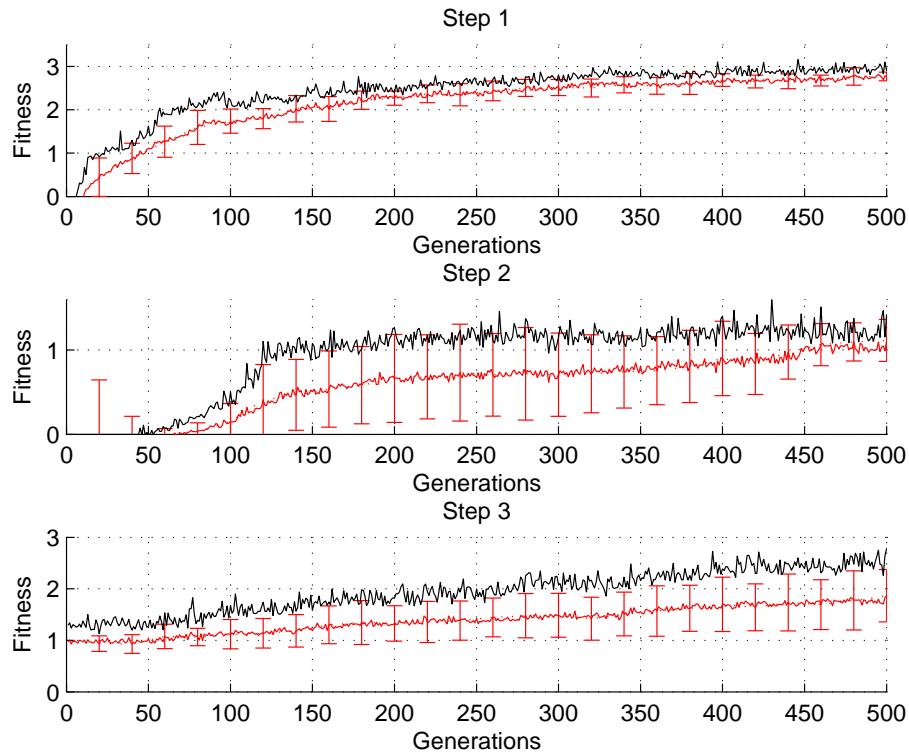


Figure 7.34: Evolving the substitution network in three steps: in step 1 the fitness penalizes only the yaw error, in steps 2 and 3 the fitness rewards progress and penalizes yaw error. Best fitness (black line) and average of the best fitnesses (gray line) over 10 repetitions of the evolution are shown. Error bars show the standard deviation calculated every 20 generations.

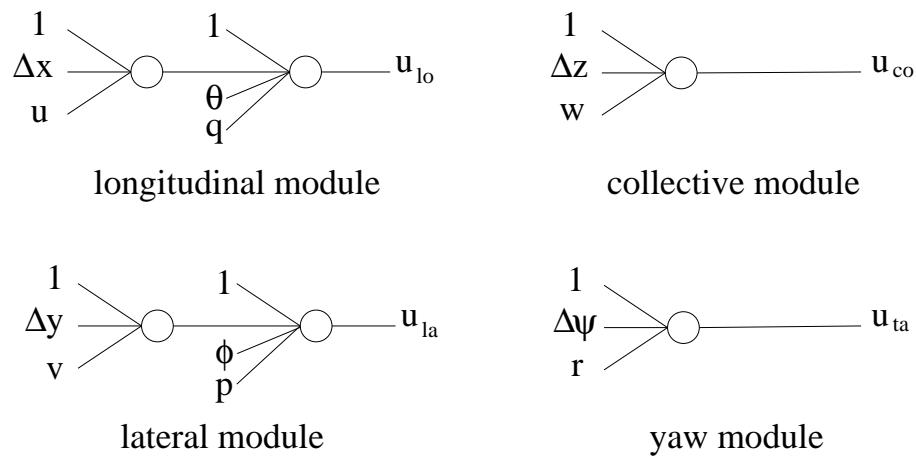


Figure 7.35: Topology of the modular network.

hundred generations was quite high, but within 500 generations very good performance was reached in all ten repetitions, as shown in Figure 7.36.

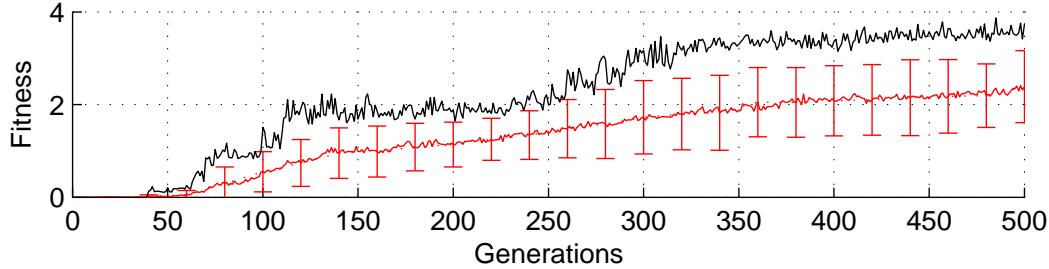


Figure 7.36: Evolving the modular network. Best fitness (black line) and average of the best fitnesses (grey line) over 10 repetitions of the evolution are shown. Error bars show the standard deviation calculated every 20 generations.

Third Working Approach: Evolving PID Gains

In order to obtain a controller to serve as a meaningful standard of comparison in our performance tests, we evolved the gains of PID controllers structurally identical to the hand crafted controller shipped with the simulator. The evolutionary runs proceeded by setting all the gains of the controllers to 0, and then first evolving the yaw control component of the PID. The fitness function and trial length were as described in Section 7.3.5 for the yaw stabiliser. Evolution produced fairly good solutions within several tens of generations. Without this preliminary step, the continuous z -rotation of the helicopter prevented evolution from obtaining successful controllers as reported in Section 7.3.4. The rest of the PID was then unfrozen, and all the gains were evolved together to perform the full task. Reasonable performance was usually reached within the first two hundred generations (see Figure 7.37).

7.3.7 Performance Analysis

The fitnesses reached by the evolved controllers during the evolution process can serve as a useful first comparison among the solutions obtained. However, to better understand the peculiarities of the different controllers, additional tests were performed on four tasks differing from that used in the evolutionary process:

- *Task 1*, Sparse waypoints

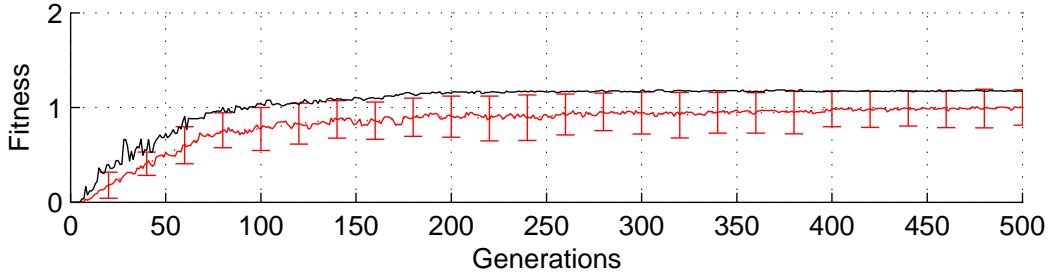


Figure 7.37: Evolving PIDs. Best fitness (black line) and average of the best fitnesses (gray line) over 10 repetitions of the evolution are shown. Error bars show the standard deviation calculated every 20 generations.

The first test was simply a repetition of the task used for the controller evolution, but over a much longer time span. The extra time allows for values of P_{chain} bigger than 1.0 since the helicopter was able to fly the whole waypoint chain more than once⁴⁶.

- *Task 2, Close waypoints*

Closer waypoints were chosen for the second task, with the average distance between the waypoints now set to 6 feet.

- *Task 3, Sparse waypoints in the presence of wind*

The waypoints were chosen with the same criteria used for task 1, but an external disturbance in the form of wind gusts was added to the simulation in order to test the robustness of the controller. The velocity of the simulated wind is time varying ($[0, 10] \text{ ft/s}$) and influences the aerodynamic forces acting on the helicopter.

- *Task 4, Sparse waypoints with varying gross weight*

This task provided an understanding of the controllers' ability to handle variations in the helicopter's weight (and mass). The maximum random weight variation⁴⁷ was set to fifty percent since the payload of a small helicopter can in extreme cases reach this limit.

In addition, since the compound fitness used for evolution (equation 7.14) gives only limited insight into the specific abilities of the controllers, three more specific performance indices were developed:

⁴⁶Once the last waypoint is reached the helicopter will find the first waypoint of the chain as its next waypoint.

⁴⁷At the beginning of each trial the weight variation is drawn from the uniform distribution $\mathcal{U}[0, 50]$.

- progress along the waypoint chain P_{chain} (higher is better),
- mean deviation from the shortest path $e_p = \sum_{t=0}^N |w_h|$ (lower is better),
- mean heading error $e_h = \sum_{t=0}^N |\psi - \psi_k|$ (lower is better).

Table 7.9 shows the results of the tests; the best values are printed in bold. The waypoint layout was randomly generated at the start of every evaluation, and each task was repeated 20 times; the average value (and standard deviation) of the performances obtained is shown (in parentheses). The penalty for crashing into the ground was a fitness of 0, rather than a negative fitness as during evolution. On the wind task, the number of crashes in 20 trials is given; this is a measure of the ability of the controller to maintain stable flight. In all the test tasks the performance was evaluated during a predefined period of 3000 time steps (corresponding to 60s of flight time).

Figures 7.38 and 7.39 show the trajectories of an evolved PID controller and an evolved modular neural network controller respectively, performing the same task (task 1) for 1100 time steps. The PID exhibited a very conservative strategy, slowing down when still far away from the waypoint and almost stopping when close to it. The network-based controller showed better control over the helicopter's speed, which produced a more linear trajectory and better progress along the waypoint chain.

Generally, the modular network evolved in Section 7.3.6 performed best on all four variations of the task, as it always progressed the furthest of the four controllers along the waypoint chain, and always moved in a more or less straight line to the waypoint. The substitution network was a close second as far as progress along the chain was concerned, but had a more mixed performance when it came to deviation from the shortest path, and was the only controller that had any significant problems keeping the desired heading.

The performance of the PID controllers was much worse than the neural network controllers when it came to progress along the waypoint chain, except in the situation with the waypoints closer together, where its performance was comparable to (if slightly lower than) the neural networks. This suggests that the reason for the good performance of the neural network controllers was not only the superiority of evolutionary tuning over manual tuning, but also the lack of non-linearity in the PID controllers. It simply does not seem possible to evolve a purely linear controller that is robust enough to perform well over the task variations and performance measures we used here.

	Hand crafted PID	Evolved PID	Substitution Network	Modular Network
Close waypoints				
P_{chain}	0.96 (0.077)	1.85 (0.091)	2.27 (0.182)	2.38 (0.106)
e_p	0.95 (0.216)	0.40 (0.048)	1.04 (0.21)	0.34 (0.044)
e_h	0.005 (0.003)	0.010 (0.002)	0.457 (0.002)	0.013 (0.001)
Sparse waypoints [10, 25ft]				
P_{chain}	0.46 (0.121)	0.74 (0.273)	1.35 (0.127)	1.64 (0.081)
e_p	1.61 (0.265)	0.84 (0.192)	2.5 (0.748)	0.63 (0.242)
e_h	0.007 (0.001)	0.015 (0.004)	0.459 (0.003)	0.018 (0.002)
Sparse waypoints with wind [0, 10ft/s]				
P_{chain}	0.27 (0.231)	0.49 (0.330)	0.95 (0.434)	1.07 (0.565)
e_p	1.99 (3.705)	2.58 (2.880)	3.54 (3.097)	1.33 (1.651)
e_h	0.069 (0.083)	0.160 (0.216)	0.430 (0.146)	0.062 (0.049)
Crash	7	2	2	4
Sparse waypoints with variable weight [$\pm 50\%$]				
P_{chain}	0.54 (0.128)	0.83 (0.235)	1.3 (0.170)	1.53 (0.137)
e_p	1.68 (0.271)	0.80 (0.153)	2.28 (0.503)	0.618 (0.112)
e_h	0.012 (0.002)	0.017 (0.005)	0.48 (0.02)	0.031 (0.017)

Table 7.9: Performance of the various controllers on different tasks.

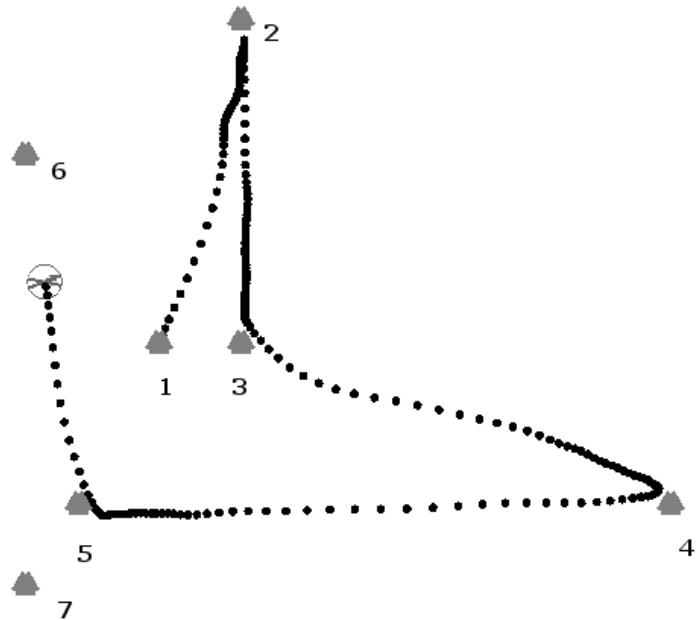


Figure 7.38: Trajectory of an evolved PID after completing 1100 time steps of a typical sparse waypoint task. In this particular run the progress along the path (after 3000 time steps) was 1.14, the mean trajectory error 0.39 and the mean heading error 0.019. The dots mark the position of the helicopter every 10 time steps.

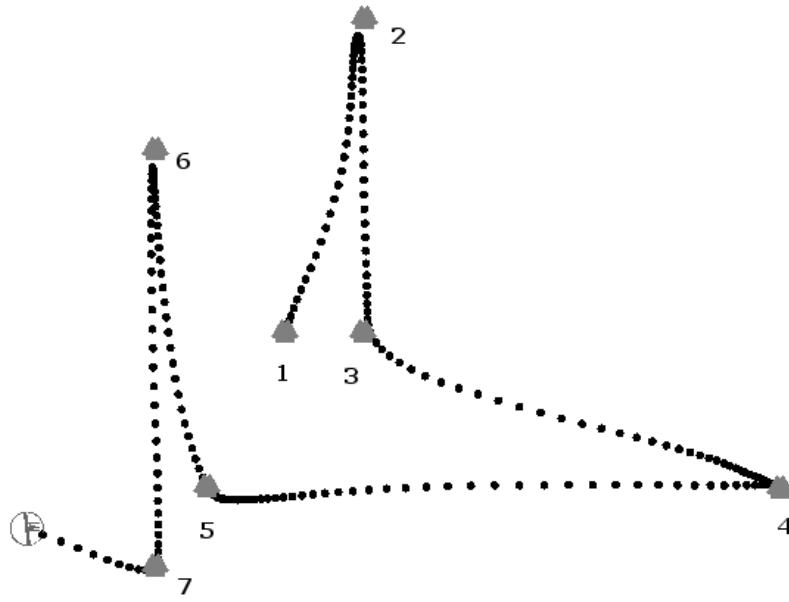


Figure 7.39: Trajectory of a modular network controller after completing 1100 time steps of the sparse waypoint task used in Fig.6. At the end of the run (3000 time steps) the progress along the path was 1.70, mean trajectory error 0.38 and mean heading error 0.016.

7.3.8 Considerations

Both of the approaches we tried for evolving neural networks generated very capable controllers that significantly outperformed the human-designed PID controller. They also outperformed the attempts by the authors of the article to control the vehicle manually. It is interesting to note that the evolved neural networks were quite robust when the parameters of the task, vehicle and environment were varied, something that could not be said about the PID controllers. This may be due to two factors: evolution is better at tuning weights and gains than are humans; and the non-linearity of neural networks makes them better suited for handling such variations than linear controllers. Such robustness is obviously important when transferring controllers to real vehicles.

Without incorporating some domain knowledge in the evolutionary process, we have not been able to evolve a controller for doing anything more advanced than not crashing. However, the most relevant question that we want to address is this: what type of domain knowledge was actually used? Let us analyse the three types of controller and the training mechanisms we used.

The multi loop PID controller is without doubt an example of the use of platform knowledge. The way the longitudinal and lateral modules are configured as nested loops,

and the choices made for the inputs of each of the 6 PID modules, are tailored for a single rotor helicopter as well as for the task of waypoint following. The fact that an essentially independent module (simple or nested) is used for each of the control outputs might also at first sight look like a platform specific choice, but in fact this is a rather standard practice in control engineering regardless of the platform. For complex MIMO (multi input multi output) systems, separating the controller into independent units for each of the inputs to be controlled yields simpler and easier to tune controller designs, although the overall performance might be poorer than that of a monolithic controller.

The substitution network was obtained by replacing parts of a PID controller structure, but to a first approximation the effect was simply to split the controller into four modules, one for each of the control outputs. The modules themselves are generic MLP networks and do not have a tailor-made topology. This modular decomposition is very similar to what we have shown to be effective both for the toy car (see Sections 7.1.2) and the quadrotor helicopter (Section 7.2.2), and again is standard practice in the control system community. We argue therefore that this decomposition is quite general and not very platform dependent. The choice of the inputs that are passed to each module in our substitution network is certainly based on knowledge about the platform, since in our case it was suggested by the structure of the PID controller.

For the modular network we followed the platform independent idea of using one module for each of the control inputs, but platform specific knowledge was used both in deciding on the topology of the network (which was inspired by the PID controller), and in selecting which input was used for each of the neurons. The good results obtained with both the networks and the PIDs suggests that, rather than the representation used by the controller, the key to enabling the learning of waypoint following might be the modularization, since this was a common feature in all three architectures. In this context it is worth mentioning the recent results of Koppejan *et al.*, who also tackled the problem of controlling a simulated single rotor helicopter. In [123] they reported that even using well established techniques that can evolve both the structure and the parameters of neural network controllers, they were not able to produce monolithic controllers able to fly the helicopter. This is a further suggestion that something more than a powerful representation is needed.

Because of time constraints we were, unfortunately unable to use the simulator to test the modular architecture based on RMLP and trained with coevolution that we had

successfully evolved for both the toy car (7.1.2) and quadrotor platforms (7.2.2). However, given that the number of dimensions and the number of control inputs are the same for the X3D and the Autopilot helicopter, we are inclined to think that the same approach would also be successful for a single rotor helicopter⁴⁸. The added flexibility of a recurrent network and the better training ability of a cooperative coevolutionary approach would allow us to provide all the state variables and waypoint information to each of the controller modules, leaving the identification of the relevant information to the learning process, and thus removing any use of platform knowledge from the process.

The choice of an incremental decomposition of evolution also deserves some discussion. Certainly the way in which we arrived at the incremental decomposition of the task, and the fact that both neural and PID approaches were unsuccessful without prior yaw stabilization, makes our choice appear very platform dependent. We should also remember that the tasks considered when training controllers for both the toy car and the quadrotor platforms were pretty similar to the task considered here, but for those platforms there was no need for any such decomposition.

However, we do know that a fundamental difference exists between the Autopilot simulator and the other platforms considered so far: its instability. We are very much inclined to attribute the need for decomposing the training task to this characteristic of the single rotor helicopter platform more than to any other. This is for the simple reason that the first of our two evolutionary phases is devoted to achieving stabilization. Once stability is achieved, the approaches that were successful for the car and the quadrotors provide a viable solution to the waypoint following problem.

7.4 Summary

In this chapter we investigated the second of the two concepts that this thesis is putting forward, the idea of automatic controller design without the use of platform knowledge. The same basic design approach based on various types of evolutionary algorithms has been tested and shown to be effective on three platforms differing in dynamics and number of degrees of freedom.

Both controllers based on and independent of platform knowledge were tested; the first

⁴⁸This is obviously conditional on first allowing the network to learn the stabilization of the helicopter heading.

took the form of manually designed controllers (often PIDs) while the second used modular and non modular neural networks. In line with our idea of avoiding the use of platform knowledge, very similar (if not identical) networks employing very general structures were used for all the controllers.

In the car and quadrotor, the neural network controllers that did not use domain knowledge performed at least as well as, if not better than, their handcrafted counterparts. Trained networks performed better even in the case of the Autopilot simulator, but in that case a limited amount of platform knowledge was used in the selection of the controller inputs.

The exceptions to this were the monolithic networks used for the single rotor and X3D helicopters which we were simply unable to evolve successfully. Our results indicate that the number of dimensions and the peculiar dynamics of our helicopters make for a hard problem that cannot be tackled with a vanilla network structure and an evolutionary algorithm. We can clearly not exclude the possibility that some of the more exotic types of networks that we have not considered could be evolved successfully.

Drawing on the idea of modularity, we suggested a platform independent method for splitting the controller into decoupled modules that were able to achieve satisfactory control of the platforms, most probably thanks to reduced interference during training and better performing search based on cooperative coevolution. Showing that this idea works consistently across very different platforms is a clear contribution of this chapter, and one that is likely to apply to other platforms and tasks.

Part of the reason for our general result is obviously the choice of the evolutionary computation techniques. Their primary strength is the robustness and consistency across optimization problems as different as the ones represented by our platforms. As confirmation we note the fact that the same mutation operators and magnitudes were used for all controllers. Any specific platform dependent adaptation would obviously have been against the spirit of our work.

Throughout the chapter we showed that evolution is able to accommodate not only different types of models but also different types of tasks. That said, limitations related to the size and type of the search space, or to specific characteristics of the platform (i.e. instability) mean that more work needs to be carried out in improving the performances of search algorithms and in devising principled ways to tackle the problem of task decomposition.

On the basis of the results obtained with all our models we can certainly say that stability is a key factor in determining the difficulty of the automatic design of a controller. If the platform at hand is not stable, the primary focus of the automatic controller design has to be stability.

The few works in the literature specifically dedicated to evolving controllers for unstable systems [250, 82] have concentrated on systems with a smaller number of dimensions (e.g. inverted pendulums) and with dynamics simpler than that of our single rotor helicopter. Whether and how artificial evolution can be made to overcome the challenging features of helicopter dynamics to produce a suitable decomposition of the task without a human in the loop is still an open research question.

Both in the case of the car and the X3D quadrotor the overall trajectories produced by the controllers using the automatically designed model are qualitatively similar to those of the real platforms. Both the car and quadrotor controllers were able to transfer to the real platforms without modifications; to the best of our knowledge this was the first reported case for a quadrotor.

Localized discrepancies between the modelled and the real behaviours still however exist. A more refined noise model in which the noise depends on the state and inputs of the system appears to be a suitable candidate for closing the remaining gap between real and simulated systems. How exactly such noise models could be defined and learned without the use of platform specific knowledge still needs to be answered.

Chapter 8

Conclusions and Future Work

In Chapter 1 we set ourselves a simple question: how far can we get in automatically building models and controllers of 6DoF rigid body vehicles without using any platform specific knowledge? We now know that the short and simple answer to the question is this: far enough to be both interesting and useful.

8.1 Research Achievements

For a longer and more technical answer, it will be convenient to consider the many aims that were identified in Section 1.3. In examining the extent to which the aims were achieved, we will also point out some of the novel content of the thesis. However, perhaps the major novelty is conceptual, and resides in the over-riding aim of investigating what can be achieved by ignoring, rather than exploiting, knowledge of the target system. This is strongly counter-intuitive, but we believe that the single minded pursuit of this aim has led us to a radical position where the real value of acquiring and using such knowledge can now be questioned. The fact that much of our work (but not all of it) has consisted of the exploitation and adaptation of techniques already available in the literature is a strength rather than a weakness, in that it demonstrates that it is a change of attitude rather than a single technical breakthrough that has enabled us to reach this position.

- *Proposing a generic model structure that can cope with 6DoF rigid body models with different levels of complexity, different numbers of inputs and both linear and nonlinear dynamics.*

The symbolic expression representation developed in Chapter 5 possesses all these

capabilities. It can cope with very generic models, both linear and nonlinear, and it is very flexible in terms of the number and types of inputs. The addition of delay terms to the symbolic expression representation allows it to deal neatly with delays on the inputs, a problem often encountered in real systems. This is a particularly novel aspect of our representation, in that it ensures that the same delay is used for each occurrence in the expression of the same input node.

Our tests showed the representation to be effective in modelling platforms having different levels of complexity, different types of dynamics and different numbers of inputs. In particular, it was shown that both continuous and discrete control signals can be handled. Although the maximum depth for the expression trees and the maximum allowed input delay had to be limited in our tests, this was a pragmatic choice dictated by the need to limit the search space, and was not a limitation of the representation.

- *Devising a general training method for determining the structure and parameters of a model, a method that relies solely on data collected from the real platform and, more importantly, relies on training data that has not been examined and selected by a human expert.*

The coevolutionary method developed in Chapter 5 uses recent findings in evolutionary computation research to carry out the steps of both data selection and of modelling. Our choice of using genetic programming allows the joint handling of the search for model structure, model parameters and input delays, optimizing for the combination that produces the most accurate predictions. The method of coevolving the data used for training is designed to automatically deal with the situation in which not all the training data are equally informative. Our tests indicate that during a typical coevolution run (Section 5.4) the parts of the dataset characterised by more activity are preferred, as would be expected. The tests carried out with real data from different vehicles (Sections 5.5-5.8) confirmed that the training algorithm is effective across different platforms as long as the data is sufficiently rich in information. The selection of inputs and the optimization of the input delays carried out automatically by genetic programming also appeared effective.

The algorithm constitutes a novel solution to the problem of coevolving models

and tests in that it is specifically geared towards rigid body dynamic systems which are computationally expensive to propagate. This affects the genetic operators used, the way the model state is propagated by using 6DoF rigid body dynamics and the way the evolutionary selection is applied at two levels to contain computation.

While our methodology does not involve any explicit limitation on the types of rigid body platform that can be tackled, its stochastic nature means that it cannot be proven formally that no such limitation exists. However, the ability of the method has been confirmed on several platforms that differ in nature and complexity, a clear and strong indication of the generality and of the potential of the algorithm.

- *Ideally obtaining performance at least as good as the models produced by experts, and by using more conventional identification techniques.*

In Chapter 4 a significant amount of effort was put into developing and using models based on first principles, and also on conventional black-box techniques, in order to provide good grounds for comparison with our coevolutionary technique. To produce a comparison that was as far as possible unbiased, in Section 4.1.1 we selected platform independent metrics that did not depend on the data used for training. In addition, the same data and inputs were used for computing the performance of both the evolved and the non-evolved models. This allowed us to attribute any performance differences directly to the modelling technique employed.

The results of our tests (Sections 5.6.2 and 5.7.2) showed that, for both the toy car and the X3D platforms, the automatically evolved models had accuracies comparable with or better than those of the models produced either with expert knowledge, or with the more orthodox techniques. This was not the case for the ATTAS aircraft (Section 5.5); but in this case there were strong indications that the lack of performance could be due to an insufficiently rich training dataset.

The development of the X3D model based on first principles is to the best of our knowledge the first example of applying fully fledged standard system identification methodologies to a miniature quadrotor helicopter. For the first time the parameters of a model based on first principles have been determined on the basis of real flight data, and the accuracy of the model obtained has been demonstrated against unseen validation data. This model constitutes a valuable contribution to the research in

the field of quadrotor modelling in its own right.

Our comparisons between the different models are also of general interest since in the literature it is rare to find examples of such diverse techniques tested as methodically and as completely as in this thesis.

- *Generating models that are transparent and understandable by an expert engineer who could use them to gain insight into the dynamic behaviour of the platform.*

In our modelling algorithm we address the creation of interpretable models in three ways. Firstly, we choose models that predict physically sound quantities (i.e. accelerations); secondly we use a representation that can be directly interpreted as an analytic expression; and thirdly, we explicitly simplify the models during model evolution to produce more concise (and hopefully more interpretable) solutions (Section 5.3.1).

The stochastic nature of genetic programming does not guarantee the interpretability of the resulting expressions in any formal way. However, we argue that this apparent deficiency is immaterial since no formal definition exists of what constitutes an interpretable expression. In fact such a concept would depend to a great extent on the expertise of whoever attempts the interpretation. More importantly, we have shown that, after some limited and straightforward manipulations, most of the models produced in our examples were easily understandable (Sections 5.6.3, 5.7.3 and 5.8.3). In addition we also showed that it was not too difficult to see similarities between the evolved models and those designed manually (Section 5.7.3).

- *Generating models that are physically consistent i.e. that produce meaningful accelerations and velocities making them usable in place of the equations devised by a human expert.*

We designed our algorithm to evolve equations that predict accelerations, and then used the error in the prediction of the system state as our measure of accuracy. Since we propagated the state forward in time using standard rigid body dynamics, the equations evolved are in effect rewarded for predicting the accelerations produced by the actual platform. In Section 5.6.2 we showed that the models obtained accurately predict the state of the platforms, and also that they predict qualitatively and quantitatively accurate accelerations.

Our way of forcing the solution of the modelling algorithm to have a tangible meaning is innovative in the way that it uses the constraints imposed by the rigid body dynamics while allowing complete freedom in the model expressions.

- *Encoding in the model the uncertainty usually present on the behaviour of real world systems.*

The evolved models are expressed in state space form. We showed (in Chapter 6) that is relatively straightforward to introduce a platform independent formulation for representing the uncertainties present in the system. The formulation we chose expresses the platform noise in terms of independent Gaussian additive terms which complement the equations obtained using coevolution. This very simple noise model assumes a unimodal distribution of the noise, and neglects the influence of the system state and the inputs on the noise variance. However, it has the very real advantage of being relatively straightforward to estimate and cheap to compute; the second of these characteristics is fundamental for the evolution of controllers.

The type of noise model proposed does not differ in substance from what is often used in the evolutionary computation field; our novelty lies instead in the way the problem of modelling the noise was approached, by borrowing from the well established filtering and estimation literature. Rather than being a change of form, this is another important change of attitude that opens up new and interesting possibilities.

- *Devising a way of automatically determining the uncertainty characteristics of the model from real data.*

Within the assumptions of our simple noise model, we showed in Section 6.2 that a sound, efficient and platform independent algorithm for the estimation of the noise parameters exists. In particular, we extended previously published work by using a UKF that allows the handling of both linear and nonlinear models, a crucial requirement for our approach.

Our tests showed that for both the toy car and the X3D helicopter (Sections 6.4.1 and 6.4.2 respectively) the augmented model (i.e. the evolved equations plus the noise model) can reproduce the levels of variability found in the real world platforms. Some

situations were also identified in which the noise model clearly showed its limitations, clearly a consequence of its simplicity.

The method of estimating the noise parameter is innovative in its use and also in its form. In contrast with current practice in the evolutionary computation literature, we propose a principled way of estimating the noise. The method proposed is novel since completely general nonlinear dynamic models can now be handled thanks to the use of a UKF.

- *Proposing a generic controller structure (or set of structures) that can cope with the number of control inputs that a typical rigid body system could have.*

In Chapter 7 we proposed the use of controllers based on recurrent neural networks, both for their generality, and for their ability to handle both discrete and continuous inputs as our systems require. In most cases RMLP were preferred for their ability to retain and exploit past information, a potentially useful trait for a controller.

Different topologies were proposed: monolithic types in which all the control inputs are computed by a single network, modular types in which each network is associated with a separate control output, and hybrid topologies in which several outputs are associated with a single module.

Our experiments showed that, while in general the performance of the two types of topologies are not significantly different for simpler problems like that of the toy car (Section 7.1.4), monolithic solutions are not effective in controlling vehicles with a larger number of DoFs and more coupled dynamics (i.e. the X3D quadrotor Section 7.2.3 and the Autopilot simulator Section 7.3.7). In these situations, modular topologies proved to be effective and to be able to learn the prescribed task. While these positive results do not strictly imply that modular controllers will be effective with other types of 6DoF rigid body vehicles, the fact that the proposed approach was found to be able to cope with such diverse platforms is an indication of its potential.

Our efforts to produce a general and platform independent solution pointed us towards a novel way of handling the complexity posed by multi-input platforms, a method in which each output is taken care of by one module, and more complex platforms simply require more modules. Our algorithms adapt cooperative coevolution

to this simplifying concept to exploit its benefits in terms of search.

- *Devising training techniques for controllers that can scale up to the complexity posed by different platforms and tasks.*

For training controllers (Chapter 7) we again employed an evolutionary approach both for its flexibility in terms of the types of controllers that can be handled, and also in terms of the types of tasks that can be tackled. A specific algorithm based on cooperative coevolution was developed to exploit the structure of the modular network controller, and to allow a more effective search of the solution space. Its effectiveness was demonstrated by being able to produce solutions for problems not amenable to monolithic controllers and standard ES training.

We showed how this training methodology can be applied to very diverse tasks, from waypoint following for a quadrotor platform to imitating the driving style of a player in the case of a car. While the technique used does not place any explicit limitations on the types of task that can be tackled, the specific details of the task implicitly define the nature of the solution space that needs to be searched. Although we did not attempt to identify different types of solution spaces our results show that our methodology can be really effective in practice in at least some very different tasks.

- *Ideally obtaining controllers with performances as good or better than those produced by experts.*

In Sections 7.1.2 and 7.2.2 we manually designed two controllers for the toy car and for the X3D respectively. The first was based on an existing controller that exploited an understanding of car dynamics, but this had to be adapted to the discrete control inputs of our car, potentially affecting its capabilities; the second was a more or less standard multi-loop PID setup. In the case of the toy car, both types of evolved controllers based on neural networks performed significantly better than the manually designed model, both when tested in simulation and on the real car. In the case of the X3D, our tests showed no significant difference between a controller based on modular neural networks, and the standard PID controller automatically tuned using evolution. Both these results show clearly that the technique we proposed can design controllers with performances comparable to that of manually designed

controllers.

- *Generating controllers in simulation that can transfer to the real platforms and control them successfully.*

In the cases both of the toy car and of the X3D platforms (Sections 7.1.5 and 7.2.4 respectively) our tests showed that controllers trained in simulation can transfer successfully to the real platforms. In the case of the toy car, several different types of controllers were evolved to follow a set of predefined waypoints, and all of them were able to drive the car successfully, although with different levels of accuracy. For the X3D quadrotor we were able to test only the handcrafted PID controller tuned by evolution; this was also able to fly the helicopter on a predefined course successfully.

The approach of learning a noise model of the platform to facilitate the transfer of controllers trained in simulation proved to be effective for both platforms. To the best of our knowledge, this is the first demonstration of a controller evolved wholly in simulation that has transferred successfully to a flying machine.

- *Ideally obtaining controllers that control the real platform with the accuracy shown in simulation.*

The qualitative performances demonstrated by different controllers in simulation were also reflected in tests on the real car, with controllers significantly better in simulation also performing better on the real vehicle (Section 7.1.5).

Models, and as a consequence also controllers, are created only on the basis of the training data, and therefore differences present in the training datasets could lead to controllers with different performances. For both the X3D quadrotor and the real car, our tests did not find any significant effect on the obtained controllers of the specific datasets used.

For the X3D, the evolved PID showed no significant difference in performance between the simulated and the real tests in terms of progress along the trajectory. In the case of the toy car the results were more mixed, with some trajectories and controllers performing similarly in simulation and in reality, and some others being significantly different.

Comparisons between trajectories showed that both in the case of the X3D and of

the toy car there tend to be specific sections in which a difference is apparent between the simulated and real trajectories, a problem that we attribute to the simplicity of the noise model.

8.2 Methodological Critique

Both in the beginning and in the course of this work, many choices were made on the basis of the knowledge and understanding available at the time. The very experimental nature of our approach meant that the research did not always develop as expected due to technical difficulties and limitations, and also time pressure. All these factors had to some extent an effect on our results. With 20/20 hindsight we recognize that some of our choices were inevitably not ideal, and that some aspects of our research could be improved. These points need to be reviewed, and their influence on the outcome of our investigation needs to be critically evaluated. Given the structured nature of the thesis, this is best done chapter by chapter.

- *Chapter 1:* At the outset of this work we decided to restrict our analysis to 6DoF rigid body vehicles, a decision that could be questioned because it neglects vehicles with a higher number of DoF. In support of our decision, we explained in the introduction that our choice was well founded because many interesting and non-trivial vehicles belong to the class considered. For vehicles with a higher number of DoF, it might be possible to scale up the technique we proposed, but due to the increase in size of the model space, it is not clear to what extent this would be effective.

A decision that might have appeared restrictive is that we only used a batch modelling approach based on pre-collected datasets. However since some of the chosen platforms are not recoverable, it would be difficult to reconcile the idea of the generation of test inputs for with our central idea of avoiding platform specific knowledge.

- *Chapter 2:* In our review of the literature on modelling we explored the disciplines most directly connected with the type of vehicles used in this study. However, since modelling is also used in disciplines outside our survey, it is at least possible that other approaches compatible with our requirements might have been neglected.

In response, we can say that in our treatment we made it very clear that we are

interested in a proof of concept system that shows the feasibility of the concept of avoiding platform knowledge, rather than in discovering some optimal solution.

We have certainly achieved this aim in our work, and while some other suitable and superior modelling technique might be seen as a welcome improvement, neglecting it did not affect the core output of our investigation.

- *Chapter 3:* Our methodology was developed using the data produced by a Vicon MOCAP system, and it is therefore relevant to analyse if it could be extended to other type of data sources. Our methodology requires logged data for all the variables present in the system state, therefore the type of data commonly used in robotic vehicles, and produced for example from the fusion of inertial and absolute position data (i.e. GPS or localization data from SLAM), is in principle potentially suitable.

It could be argued that to better test the abilities of the modelling algorithm, more aggressive speeds and manoeuvres should have been attempted with the X3D¹. A first reply to this is that, given the space available, more elaborate manoeuvres were simply not possible in practice. It is also the case that the speeds and types of motion chosen were those that a pilot would use if following a predefined waypoint path; sampling other parts of the helicopter envelope was therefore unnecessary in this context.

- *Chapter 4:* The assumption of negligible process noise made in this and the following chapter also calls for comment. Strictly speaking, only an extensive knowledge of the platform and of the experimental condition can give assurance that the conditions required by this assumption are met². In the case of evolving the structure and parameters of a model, the question is meaningless (since no model is available at the start of the identification process). The assumption of negligible process noise has to be treated as a working assumption, our results both in Chapter 4 and Chapter 5 show that this assumption is clearly effective.

When optimizing the parameters of both the ATTAS and of the X3D models based on first principles, we had to rely to a great extent on our understanding to split the parameter optimization into successive steps in order to produce the

¹The toy car was driven to the maximum speed during data collection.

²Again we have to remember that the unmodelled dynamics is considered part of the noise.

full model. Other optimization algorithms could possibly have circumvented this problem, but this was not attempted because we considered it more relevant to our aims to regard these problems as examples of the limitations faced when using conventional modelling techniques.

The nearest neighbour and the MLP networks were selected because they are among the most commonly used black-box models, but it can be perhaps argued that better performing black-box techniques could be found. Our response is that a key advantage of the method lies in the fact that the models are potentially interpretable; this is most important than their absolute level of performance. The use of better performing black-box models would not change substantially our analysis.

- *Chapter 5:* Two main criticisms can be made of the choices in this chapter.

The first is that, due to its stochastic nature the modelling technique we chose does not allow the derivation of formal proofs of convergence, nor of the optimality of the solutions. As noted in the previous section, is also impossible to guarantee that the obtained solutions are understandable.

The second criticism is that in designing the algorithm (Section 5.3) a great many design decisions were taken, or parameters were set, on the basis of trial runs executed during the development of the algorithm. Since our main concern, as previously explained, was in demonstrating proof of concept, we do not feel that this criticism is especially damaging. We do present some results or more extensive work on finding suitable parameters values (Section 5.9.2).

- *Chapter 6:* A questionable choice made in this chapter is that of using an independent Gaussian noise model. As was expected, our tests showed its inability to reproduce some aspects of the uncertainty in our models (Section 6.4.2). In order to use more sophisticated noise models, the challenge of computational limitation would need to be addressed, and appropriate ways of estimating the noise density would need to be devised; neither of those tasks would be straightforward to solve.

In spite of its shortcomings, our tests showed that even this simple noise model enables the evolved controllers to transfer successfully to the real platforms; this practical success perhaps explains why the model is often adopted in the literature.

- *Chapter 7:* This chapter has similar weaknesses to those discussed for Chapter 5. Again, in several situations we had to resort to brief empirical investigations to determine several of the parameters employed in our evolutionary setup. However, we would claim that the fact that the same parameters have been shown to be effective for very different platforms and tasks is a clear indication that their precise values are very probably not critical.

In general the experiments carried out all three platforms (i.e. toy car, X3D, Autopilot simulator) pointed to the modularization of the controller as being a key factor in enabling the learning of complex and coupled models. Clearly we cannot exclude the possibility that some more sophisticated scheme might be found that could automatically develop both the structure as the parameters of a good controller. However, we would expect the end result to show at least some aspects of the separation provided by our modular solution.

Lastly we need to comment on the fact that, in the case of the X3D helicopter, the amount of experimentation carried out on the real platform was particularly limited, in that we were unable to test the *X3DbestCoCoRMLP* controller on the real platform, and only one type of trajectory was examined. This was mainly due to the time and resource needed to set up and carry out the experiments.

8.3 Future Work

There are several directions in which the current work could be taken; some are more immediate, and address the shortcomings discussed in the previous section, and others are instead avenues for longer term research.

In the short term it would be interesting to look at more complex representations for the noise model in order to avoid the limitations of our current approach. Local regression techniques are potentially suitable, but a training scheme compatible with our setup would need to be developed. A further possibility in the same direction is that of integrating the learning of the noise model with the learning of the dynamic model, the former providing the mean prediction and the latter providing an uncertainty envelope that depends on the current state and inputs. The challenges to be faced in this case are the development of a principled way of training the models, containing the computation required.

An additional possibility, which is however in conflict with our primary idea of avoiding domain knowledge, is that of seeing how our method could be used to extend and complement existing models based on first principles. Since the models developed by our system are in state space form, and are written as symbolic equations, it seems quite natural to think of using known models or known partial models to seed the search carried out by genetic programming. An interesting research question would be to seek effective ways of seeding the initial population.

Continuing our investigation of controllers and increasing the complexity of the tasks that can be learned would also be of interest. From the engineering perspective it would be useful to incorporate standard concepts of control into our automatic design approach in order to be able to assess the robustness of the solutions produced.

The author is currently working on the EPSRC founded project SUAAVE³ (Sensing, Unmanned, Autonomous Aerial VEHicles) which aims at creating swarms of quadrotors that collaboratively self-organise in order to sense the environment. Within the scope of the project there will be the possibility of applying some of the techniques developed in this thesis to the challenging scenario of flying outdoors; this would also entail the use of on-board sensors. More interestingly, since several identical platforms will be available, the possibility of using data from several helicopters could be explored.

In the long term, our work is part of a research strand that embraces the possibilities opened by the increasing availability of computing power. The latest incarnation of this trend is represented by what is called GPU computing; the use of graphics processor units to perform general purpose scientific and engineering computing. The inherent parallelism of evolutionary algorithms is particularly well suited to the multi-core architecture offered by GPUs, and as a result recent years have seen a tremendous increase in work using this technology in the domain of evolutionary computation. Speed-ups of an order of magnitude (in relation to a standard CPU implementation) have been reported by various authors ([186]). The future is not too distant in which the type of approach we have developed, could be used on much more complex and demanding problems. Only time will tell if concepts like modularity and incrementality are the keys to mastering the higher levels of complexity and difficulty.

³<http://www.suaave.org>

Appendix A

Derivations of the Parametric Quadrotor Model

In this section we look at deriving the full model of a quadrotor, including its stability augmentation (or stabilization) system and its propulsion group¹, with the aim of obtaining a set of relationships that predicts the evolution of the system state. In doing this we take inspiration from the work of Bouabdallah ([34]) who models the dynamics of a quadrotor from first principles in terms of forces and moments. Our efforts will lead to a model in acceleration space of the type introduced in equation 3.4, reproduced here for convenience:

$$[\mathbf{a}, \boldsymbol{\alpha}]_t^T = f(\mathbf{x}_t, \mathbf{u}_t). \quad (\text{A.1})$$

We need therefore to develop a model that, given the control inputs \mathbf{u} from the RC remote, predicts the linear and angular accelerations of the system. From those, we will then be able to update the full state of the vehicle using equations 3.5-3.7. Additionally we will show how the formulation of the stability augmentation system and the propulsion group model leads to the introduction of additional state variables for which we also need to derive appropriate update equations.

We start by looking at the aerodynamic forces generated by each rotor, estimating the parameters (Section A.1.1), and then investigating how those forces and moments affect the flying machine (Section A.2). We then look at the propulsion group (Section A.3), and at the stability augmentation system (Section A.4), to formulate the relationships that

¹We call propulsion group the combination of motor, blade and electronic speed controller.

translate the control inputs into propeller speeds.

A.1 Rotor Forces

The equations used to compute the aerodynamic forces acting on a rotor were originally proposed by Fay [68]; we report them in this section for completeness. We accompany the equations with a simplified explanation intended to aid the understanding; we refer to [68] and a specialized text-book [115] for a more detailed explanation and a justification for the approximations adopted in what follows.

A helicopter blade is essentially an airfoil, and as it cuts through the air each of its sections will produce an amount of lift, and at the same time will be affected by a drag force. The extents of these forces depend on the characteristics of the blade (i.e. the lift slope a , and the coefficient of drag C_d), on the speed of the incoming air flow, and on the orientation of the relevant section with respect to the incoming air flow (i.e. the angle of attack α).

Using momentum theory and assuming a constant inflow velocity ν_1 for the air across the rotor disk, we obtain an equation for thrust in hover which has the form:

$$T = 2\rho A \nu_1^2, \quad (\text{A.2})$$

where ρ is air the density and A is the rotor disc area. Since in hover the thrust has to be equal to the lifted weight W , we can easily compute the inflow velocity ν_1 as:

$$\nu_1 = \sqrt{\frac{W}{2\rho A}}. \quad (\text{A.3})$$

When moving through the air, equation A.3 also has to account for the horizontal velocity $V_h = \sqrt{u^2 + v^2}$ and it therefore becomes:

$$\nu_1 = \sqrt{-\frac{V_h^2}{2} + \sqrt{\left(\frac{V_h}{2}\right)^2 + \left(\frac{W}{2\rho A}\right)^2}}. \quad (\text{A.4})$$

We now define and derive some useful quantities:

- solidity ratio:

$$\sigma = \frac{N\bar{c}}{\pi R}, \quad (\text{A.5})$$

This is the ratio between disc area and blade area; following standard conventions it is approximated by considering the blade area as simply equal to $A_b = \bar{c}R$. N is the number of blades, R is the rotor radius and \bar{c} is the average blade chord;

- *inflow ratio:*

$$\lambda = \frac{\nu_1 + w}{\Omega R}, \quad (\text{A.6})$$

This is the ratio between the inflow speed and the blade tip speed;

- *rotor advance ratio:*

$$\mu = \frac{V_h}{\Omega R}, \quad (\text{A.7})$$

This is the ratio between the horizontal speed V and the blade tip speed.

Two forces and two moments are a direct consequence of the aerodynamic forces of lift and drag acting on the blades. To simplify the derivation of forces and moments, it is usual to assume [68] that the local velocity² of the blade section is much larger than the total inflow through the blade, that blade twist increases linearly with the distance from the blade root and that the coefficient of lift varies linearly with the angle of attack.

The lift force integrated over the whole surface of the blade produces the thrust; this force is directed upward and is perpendicular to the plane in which the blade rotates. It can be written as:

$$\frac{C_T}{\sigma a} = \left(\frac{1}{6} + \frac{\mu^2}{4} \right) \theta_0 - (1 + \mu^2) \frac{\theta_{tw}}{8} - \frac{1}{4} \lambda \quad (\text{A.8})$$

$$T = C_T \rho A (\Omega R)^2. \quad (\text{A.9})$$

The total drag is expressed as a force on the plane in which the blade rotates:

$$\frac{C_H}{\sigma a} = \frac{1}{4a} \mu C_d + \frac{1}{4} \lambda \mu \left(\theta_0 - \frac{\theta_{tw}}{2} \right) \quad (\text{A.10})$$

$$H = C_H \rho A (\Omega R)^2. \quad (\text{A.11})$$

The drag force acting on the blades also manifests itself as a moment about the blade shaft that expresses the resistance to the blade being spun:

$$\frac{C_Q}{\sigma a} = \frac{1}{8a} (1 + \mu^2) C_d + \lambda \left(\frac{1}{6} \theta_0 - \frac{1}{8} \theta_{tw} - \frac{1}{4} \lambda \right) \quad (\text{A.12})$$

²Velocity at which the section of blade is advancing through the air as result of its spinning motion.

$$Q = C_Q \rho A (\Omega R)^2 R. \quad (\text{A.13})$$

As the whole rotor advances through the air, the advancing side of a blade will experience a higher airspeed than the retreating side, and this will produce a rolling moment:

$$\frac{C_R}{\sigma a} = -\mu \left(\frac{1}{6} \theta_0 - \frac{1}{8} \theta_{tw} - \frac{1}{8} \lambda \right) \quad (\text{A.14})$$

$$R = C_R \rho A (\Omega R)^2 R. \quad (\text{A.15})$$

A.1.1 Blade Parameters

Since technical data about the X3D blades was not available, we had to estimate the needed parameters as best we could from measurements and an experimental thrust test. The radius and average blade chord \bar{c} can simply be measured with calipers; similarly, we can also compute the actual twist of the blade from geometrical measurements. Figure A.1

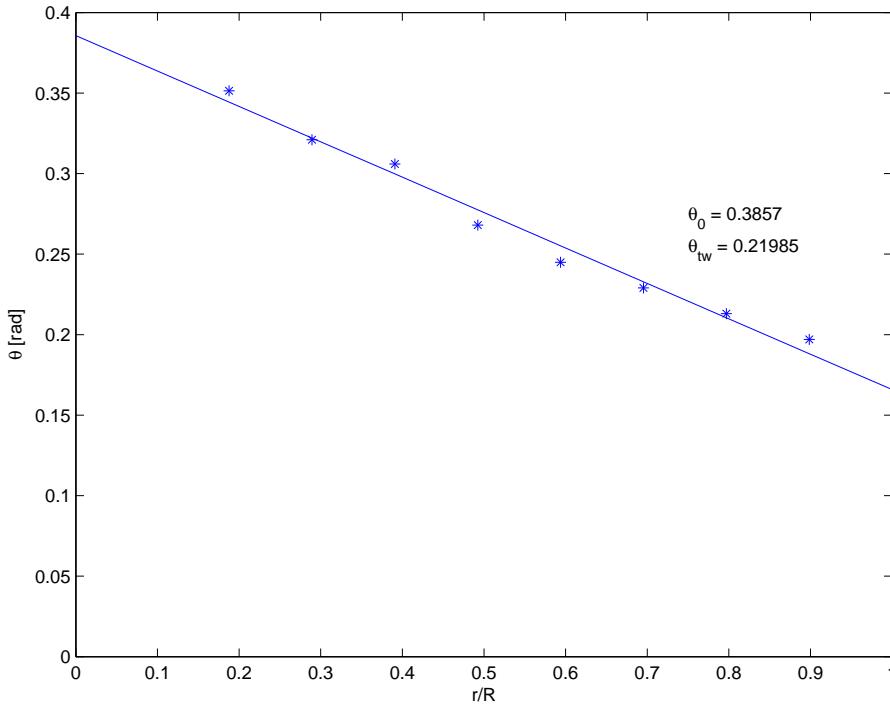


Figure A.1: Twist along the X3D blade: actual measurement (*), linear twist model (-). A least squares fit produced the parameters $\theta_0 = 0.385$ and $\theta_{tw} = 0.218$.

shows the results of our measurements, which are in good agreement with the standard linear twist model.

$$\theta(r)^3 = \theta_0 + \theta_{tw}(r/R), \quad (\text{A.16})$$

³r is the distance from the blade root at which the twist is computed.

for which a least squares fit produced the parameters $\theta_0 = 0.385$ and $\theta_{tw} = 0.218$.

Given those geometric parameters, a thrust test can be carried out using the stand shown in Figure A.3 in order to estimate the lift slope coefficient a . The stand allows the simultaneous recording of the remote throttle input, the input given to the motor controller, the rotational speed of the propeller, and the thrust produced. Because of the way the test stand is constructed, the thrust at any rotational speed is counterbalanced by the reaction force of the electronic scale, and so the rotor is effectively in hover. Equation A.3 can therefore be used to obtain the inflow velocity. It is easy to see that since the thrust and the rotational speed are measured experimentally, the only unknown in equations A.8 and A.9 is the lift slope coefficient a . Combining equations A.8 and A.9 and solving for a we obtain:

$$a = \frac{T}{\rho A (\Omega R)^2 \sigma \left[\left(\frac{1}{6} + \frac{\mu^2}{4} \right) \theta_0 - (1 + \mu^2) \frac{\theta_{tw}}{8} - \frac{1}{4} \right]}. \quad (\text{A.17})$$

Several measurements⁴ were collected experimentally, and we computed the lift slope coefficient by means of a least squares fit, obtaining a value of 6.65. The coefficient used for the several parameters of equation A.17 are shown in Table A.1. Figure A.2 shows the data

Parameter	Value	Description
R	0.08 m	rotor radius
a	6.65	estimated lift slope
ρ	1.184 Kg/m ³	air density
θ_0	0.385 rad	pitch angle
θ_{tw}	0.21 rad	twist angle
\bar{c}	0.0155 m	average root chord
C_d	-0.172	drag coefficient (from identification)

Table A.1: Rotor model parameters.

collected during the thrust experiment along with the model's predictions. Obviously since the inflow velocity for the model prediction (continuous line) is computed from the thrust itself, any outlier in the measurement of T will manifest itself in the output of the model.

The value we obtained for the lift slope is in line with values reported in the literature [34, 188]; however, it is important to note that the approximation used in the derivation has a considerable impact on the result. For example, if instead of using the approximation $A_b = \bar{c}R$ for the blade area, we compute it based on the actual area of the blade⁵, we obtain

⁴Fourteen different measurements for each of four different motor voltages were collected for a total of 56 measurements. The motor voltage is irrelevant at this stage but in Section A.3 data from the same tests will be used to determine the motor speed as a function of the input voltage.

⁵Our blade has a trapezoidal shape, and as a consequence computing a good approximation of it is easy.

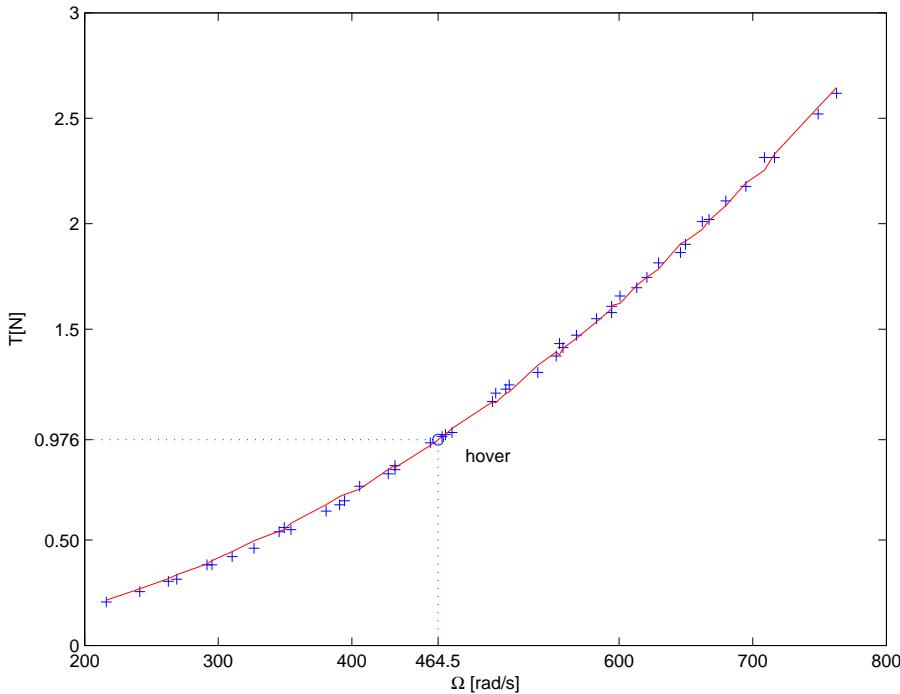


Figure A.2: Relationship between the blades' rotational speed and the generated thrust: experimental data (+), model based on momentum theory and blade elements theory (—). The hover point corresponds to a rotational speed of 465.4 rad/s.

a value for α which is 8.27, more than 25% higher.

Unfortunately our test stand does not offer the possibility of measuring the reaction torque produced by the drag force acting on the blade, and for this reason a procedure similar to that just developed cannot be used for determining C_d . The drag coefficient will be considered as an unknown parameter, and will be determined by fitting the full motion of the aircraft as explained in Section 4.1.2.

A.2 Forces and Moments

We model the quadrotor as a 3D rigid body with mass m , and moments of inertia I_{xx} , I_{yy} and I_{zz} that is affected by the gravitational force g . As in Section A, each of the four rotors produces a thrust force T_n (where n is the rotor index).

The thrust force is perpendicular to the blade's plane of rotation; assuming the blades are rigid, the thrust force is aligned with the rotor shaft and is therefore always aligned with the positive direction of the z axis in the quadrotor reference frame

$$F_{zT} = T_0 + T_1 + T_2 + T_3. \quad (\text{A.18})$$

The hub force H_n lies in the plane swept by the rotating blades; we denote the decomposition of the hub force along the x and y axes of the helicopter as H_{nx} and H_{ny} :

$$\begin{aligned} F_{xH} &= H_{0x} + H_{1x} + H_{2x} + H_{3x} \\ F_{yH} &= H_{0y} + H_{1y} + H_{2y} + H_{3y}. \end{aligned} \quad (\text{A.19})$$

The translational motion of the flyer is inevitably subject to the effect of viscous friction with the surrounding air. We model this as a drag force proportional to the surface area of the central body through the viscous coefficient C_{xy} :

$$\begin{aligned} F_{xD} &= \frac{1}{2}C_{xy}A_c\rho u|u| \\ F_{yD} &= \frac{1}{2}C_{xy}A_c\rho v|v|. \end{aligned} \quad (\text{A.20})$$

Combining these effects, and projecting the gravity vector appropriately according to the attitude of the quadrotor, the linear accelerations a_x, a_y, a_z acting on the system can be computed in body coordinates as:

$$\begin{aligned} a_x &= -(H_{0x} + H_{1x} + H_{2x} + H_{3x}) - \frac{1}{2}C_{xy}A_c\rho u|u| - \sin(\theta)g \\ a_y &= -(H_{0y} + H_{1y} + H_{2y} + H_{3y}) - \frac{1}{2}C_{xy}A_c\rho v|v| + \sin(\phi)\cos(\theta)g \\ a_z &= \frac{T_0 + T_1 + T_2 + T_3}{m} + \cos(\phi)\cos(\theta)g. \end{aligned} \quad (\text{A.21})$$

Each thrust force acts along its rotor shaft which is located at a distance (l in the x or y axis) from the centre of gravity; as a consequence, these forces will produce rotational moments that can roll and pitch the quadrotor:

$$\begin{aligned} M_{xT} &= l(T_3 - T_1) \\ M_{yT} &= l(T_2 - T_0). \end{aligned} \quad (\text{A.22})$$

Similarly, since the plane of rotation of each blade is displaced in the vertical direction from the centre of gravity by an amount h , hub forces will also produce rotational moments. The projection of the hub force along the x axis will produce a pitching moment, while the

projection on the y axis will originate a rolling moment:

$$\begin{aligned} M_{xH} &= h(H_{0y} + H_{1y} + H_{2y} + H_{3y}) \\ M_{yH} &= h(H_{0x} + H_{1x} + H_{2x} + H_{3x}). \end{aligned} \quad (\text{A.23})$$

In the xy plane, the hub forces are applied at a distance l from the centre of the quadrotor; they will therefore also produce a yaw moment:

$$M_{zH} = l(-H_{1x} + H_{3x} + H_{0y} - H_{2y}). \quad (\text{A.24})$$

The differences in the lift forces in forward flight (see Section A) produce a moment (R_n) that lies in the plane of the blades' rotation. Reprojecting this moment along the quadrotor's axes allows the computation of the pitching and rolling moments (R_{nx} and R_{ny} respectively). In Section A we also calculated the torque produced by the drag of each rotor (Q_n). These moments are parallel to the z axis of the flying machine and therefore produce only yaw motion:

$$M_{zQ} = (Q_0 - Q_1 + Q_2 - Q_3). \quad (\text{A.25})$$

Finally, we need to take into account the gyroscopic effects resulting from the rigid body rotations:

$$\begin{aligned} M_{xG} &= qr(I_{yy} - I_{zz}) \\ M_{yG} &= pr(I_{zz} - I_{xx}) \\ M_{zG} &= pq(I_{xx} - I_{yy}), \end{aligned} \quad (\text{A.26})$$

and the rotation of the propellers:

$$\begin{aligned} M_{xP} &= J_r q \Omega_r \\ M_{yP} &= J_r p \Omega_r \\ M_{zP} &= J_r \dot{\Omega}_r. \end{aligned} \quad (\text{A.27})$$

The combination of all the obtained moments (equations A.22-A.28) allows us to obtain

the angular accelerations expressed in body coordinates:

$$\begin{aligned}\alpha_x &= \frac{qr(I_{yy} - I_{zz}) + J_r q \Omega_r + l(T_3 - T_1) - h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jx}}{I_{xx}} \\ \alpha_y &= \frac{pr(I_{zz} - I_{xx}) + J_r p \Omega_r + l(T_2 - T_0) + h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jy}}{I_{yy}} \\ \alpha_z &= \frac{pq(I_{xx} - I_{yy}) + J_r \dot{\Omega}_r + \sum_{j=0}^3 (-1)^j Q_j + l(-H_{1x} + H_{3x} + H_{0y} - H_{2y})}{I_{zz}},\end{aligned}\quad (\text{A.28})$$

where Ω_r is defined as:

$$\Omega_r = \Omega_1 - \Omega_2 + \Omega_3 - \Omega_4. \quad (\text{A.29})$$

A.3 Propulsion Group Model

The model developed in the previous section accounts for the dynamics and aerodynamics of the quadrotor flying machine, but models the rotors as simple thrust and drag forces. We now need to create a model for the X3D system composed of the motor controller, the electric motor and the blade which maps the helicopter's control inputs into thrust and drag forces. Since the X3D is a commercial product, no documentation or firmware source is available for its electronic stabilization system nor for the ESC (electronic speed controllers) that drive the brushless motors. Fortunately, since the communication between the X3D central controller and the motor controllers is through an I^2C bus, we can disconnect the propulsion group from the flying machine and conduct specific experiments to model it independently.

As we explained in Section 3.3.1, the X3D quadrotor is equipped with an electronic stability augmentation system, therefore the relationship between the control input and the I^2C command sent to the ESC is not straightforward. However the throttle is an exception to this since the augmentation system does not control the mean speed of each rotor (see details of the PID control in Section A.4). By logging and comparing the throttle value and the command sent to each motor controller over the I^2C bus, we confirmed that the throttle value u and the command sent to the motor controller u_{I^2C} are linearly proportional:

$$u_{I^2C} = K_u u + K_b, \quad (\text{A.30})$$

where with a simple least squares fit we find that $K_u = 303$ and $K_b = -46.33$.

We then went on to identify the relationship between the input given to the motor controller and the speed achieved by the propeller (and therefore the thrust generated). In order to do so, we built a test rig (see Figure A.3) in which a motor with its propeller is mounted at the extremity of a low friction cantilever mechanism. At the opposite side a precise⁶ electronic scale connected to a computer registers the thrust. An infrared encoder mounted on the motor and connected to a digital storage scope allows the measurement and recording of the instantaneous rotational speed. Since the motor controller operates

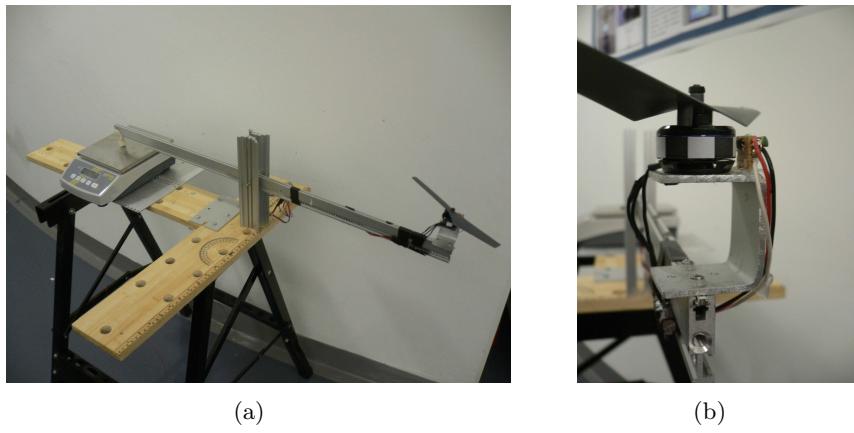


Figure A.3: Purpose built stand used to measure motor speed and thrust produced (a), and a detail of the infrared encoder (b). Both rotor speed and thrust variables are measured simultaneously using the IR encoder and the electronic scales.

by pulse width modulating the battery supply, the battery voltage (V_b)⁷ will also have an effect on the rotational speed achieved. During this test the voltage is precisely controlled by powering the motor from a regulated power supply.

At first we conducted a series of steady state tests in which a specific input was given to the motor controller and, the speed was recorded once it was stable. The command range 1-200 was sampled with 13 different measurements for each of 6 different voltage levels; for each of the throttle/voltage combinations a few seconds' worth of readings were collected and averaged. Figure A.4 shows a plot of the data obtained. For input voltages higher than 11 volts and control inputs higher than 16 the relationship u_{I^2C} to Ω is quadratic. But, for control inputs lower than 16 or lower than 32 and input voltage lower than 11 volts, the relationship has a plateau at the value 214. Fortunately, during normal flight

⁶Kern PCB electronic scale with 1gram resolution.

⁷We remind the reader that the control input u_{ba} is simply a “centered” version of V_b , or more precisely

$$u_{ba} = V_b - 10.$$

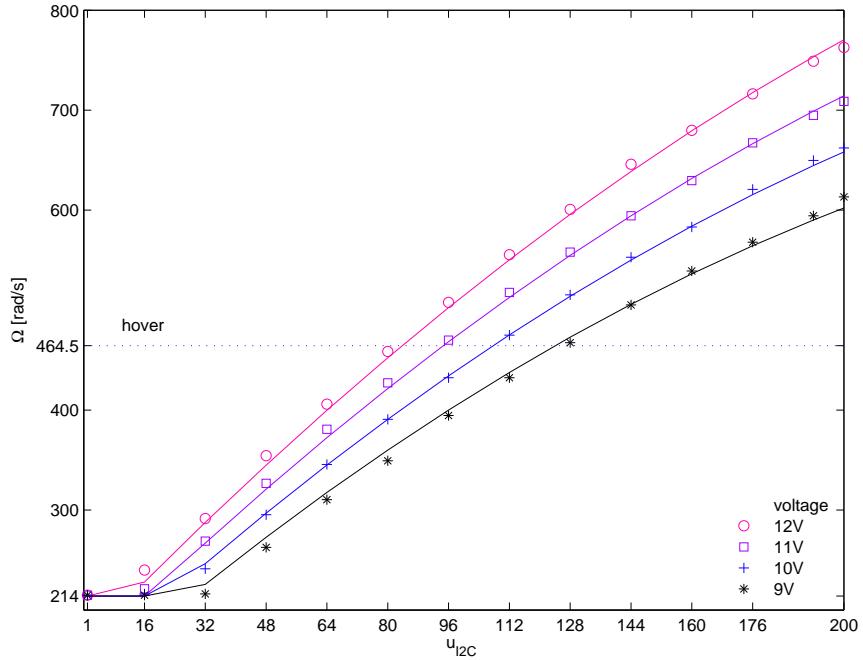


Figure A.4: Propulsion group identification: control input u_{I2C} versus measured angular speed Ω . The hovering throttle value 82.6 corresponds to a rotational speed of 464.5 rad/s. The rotational speed has an inverse square relationship to the control input and exhibits a plateau for $\Omega < \Omega_{plateau}$ at the value 214.

the control input is never as low as 16 and given that as the battery discharges higher and higher values of the throttle are needed, an input lower than 32 with a low battery voltage is also unlikely. For example, while a throttle value of about 83 is sufficient to maintain hover when the battery is at 12 volts, at 9 volts an input value of 129 is needed. Under these circumstances we can discard those points from the dataset⁸ and try to approximate the now simpler relationship by introducing an artificial cut-off at the value $\Omega_{plateau} = 214$. Optimizing the coefficient of a second order relationship using the standard Nelder-Mead Simplex we obtain the following:

$$\Omega_f = K_{\Omega_0} + K_{\Omega_u} u_{I2C} + K_{\Omega_{u^2}} u_{I2C}^2 + K_{\Omega_{uV}} (uV_b) + K_{\Omega_V} V_b, \quad (\text{A.31})$$

where $K_{\Omega_0} = -1.21$, $K_{\Omega_u} = 1.42$, $K_{\Omega_{u^2}} = -0.00465$, $K_{\Omega_{uV}} = 13.92$ and $K_{\Omega_V} = 0.210$. The nonlinear cut-off for values of Ω_f lower than 214 can be expressed formally as:

$$\Omega = \begin{cases} \Omega_f & \Omega_f > \Omega_{plateau} \\ 214 & \Omega_f \leq \Omega_{plateau}, \end{cases} \quad (\text{A.32})$$

⁸The measurements discarded were all those corresponding to an input value lower than 32, a total of 8 measurements out of 56.

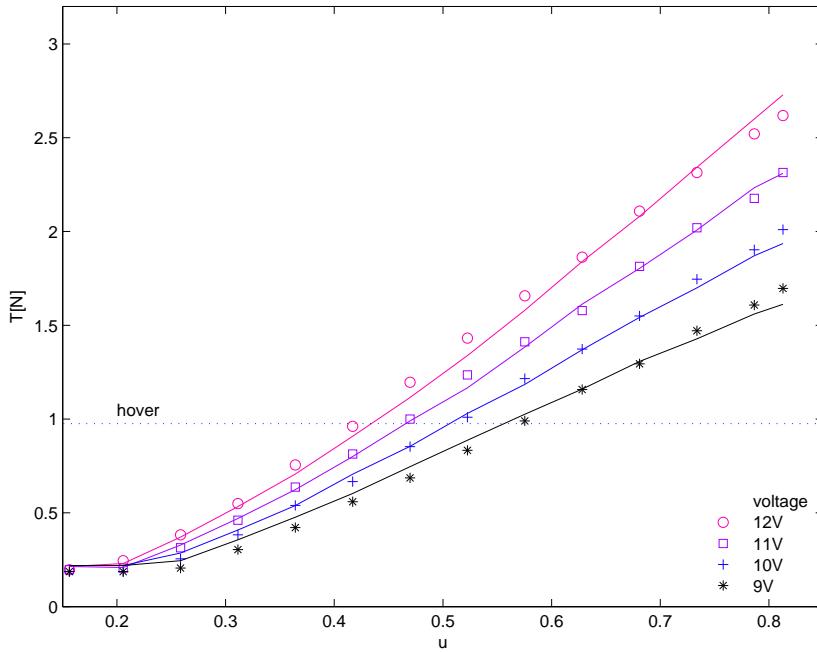


Figure A.5: Propulsion group identification: control input u versus measured thrust at different voltages. The hovering thrust of $0.976N$ corresponds to a throttle value between 0.42 and 0.55 depending on the state of the battery. For throttle values higher than 0.3 the thrust generated is approximately linear with respect to the control input.

Although in reality u_{I^2C} can take only integer values, for simplicity we neglect the discretization in our simulation. Substituting equation A.30 into equation A.32 we obtain the function that relates the throttle input directly to the rotational speed Ω :

$$\begin{aligned} \Omega_f &= K_{\Omega_0} + K_{\Omega_u}(K_u u + K_b) + K_{\Omega_{u^2}}(K_u u + K_b)^2 + K_{\Omega_{uV}}((K_u u + K_b)V_b) + K_{\Omega_V}V_b \\ \Omega &= \begin{cases} \Omega_f & \Omega_f > \Omega_{plateau} \\ 214 & \Omega_f \leq \Omega_{plateau}, \end{cases} \end{aligned} \quad (\text{A.33})$$

Figure A.5 plots the relationship between the throttle input and the measured thrust force. Substituting equation A.33 into equation A.9, which allows us to compute the thrust force generated by a single propeller, we obtain an expression for thrust generated as a consequence of a specific control input u :

$$T = \begin{cases} C_T \rho A R^2 (\Omega_f)^2 & \Omega_f > \Omega_{plateau} \\ C_T \rho A R^2 (\Omega_{plateau})^2 & \Omega_f \leq \Omega_{plateau}, \end{cases} \quad (\text{A.34})$$

where u appears within the expression of Ω_f (equation A.33). The relationship defined by equation A.34 is plotted in Figure A.5; it is interesting to notice that for throttle values

higher than 0.3 the relationship is close to linear, suggesting that a simpler model than the one we derived could perhaps be sufficient to describe the data.

The brushless motors of our flyer can generate a considerable torque, but the effect of the inertia of the blades, and more importantly the effect of drag, constrains the ability to change the rotor speed. Although the speed controller constantly monitors the voltage in each of the motor's phases and uses them as a feedback signals to maintain the correct commutation timing, a lag exists between changes in the control input and the corresponding changes in rotational speed. A simple and effective way to represent the dynamic behaviour of the whole propulsion group is to model it as a first order system ([34, 245]):

$$\mathcal{L}(\Omega_i) = \frac{1}{1 + \tau_\Omega s} \mathcal{L}(\Omega_i) \quad i \in [0, 1, 2, 3], \quad (\text{A.35})$$

where $\mathcal{L}(\Omega_i)$ and $\mathcal{L}(\Omega_i)$ are the Laplace transforms of the real and commanded rotational speed functions of the control input. To estimate the system's time constants we devised a simple experiment in which a series of step inputs were generated and the rotational speed of the rotor was measured using the rig of Figure A.3.

Figure A.6 shows the comparison between the speed measured during the test and that

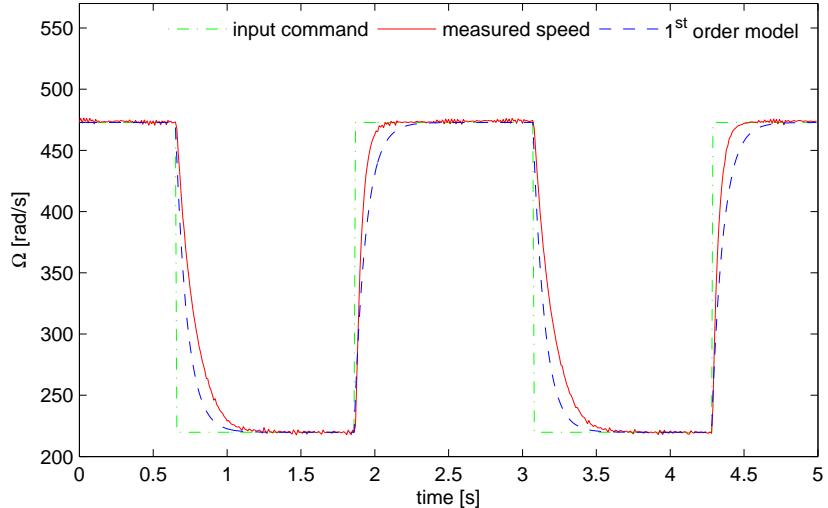


Figure A.6: Propulsion group step input response: commanded angular speed (dash-dotted), real angular speed (solid), first order model (dashed).

obtained using the first order model in equation A.35. The plot shows how the system's time constant for step increases and decreases in the throttle are different; for simplicity we do not account for this in our model, but instead set the time constant to be the average of the two ($\tau_\Omega = 0.084$).

A.4 Stability Augmentation System

The motor controllers are able to drive each of the rotors to a commanded speed, but even when running at the same speed, small differences between the blades and complex aerodynamic disturbances usually mean that each rotor will generate a slightly different thrust. This variability makes a quadrotor intrinsically unstable and practically impossible to fly without a stabilization system. Our quadrotor employs a PD stabilization system based on the feedback produced by three electronic gyros. Gurdan et al. in [89] describe the structure of the electronic stabilization used in the X3D (see Figure A.7). For each of the three axes (roll, pitch and yaw) the speed command sent to the motors is proportional (K_P) to the integral of the difference between the gyro output and a scaled version of the control input. This constitutes what is commonly called an attitude hold controller; the stabilization will compensate for any sudden rotation that causes the integrator to deviate from zero. Since a scaled (K_{SP}) version of the control input is also subtracted before the integration, the control input biases the integration, thus determining the absolute attitude that the controller will maintain. As noted in [89], this control scheme is sensitive to any sort of bias error present in the gyros' output (since a bias will integrate to a non zero value); fortunately, in the X3D this is relatively small, and as a result the pilot can easily compensate for these small changes in attitude. To improve the responsiveness of the attitude hold loop, both the control input and the gyro readings are also used as direct

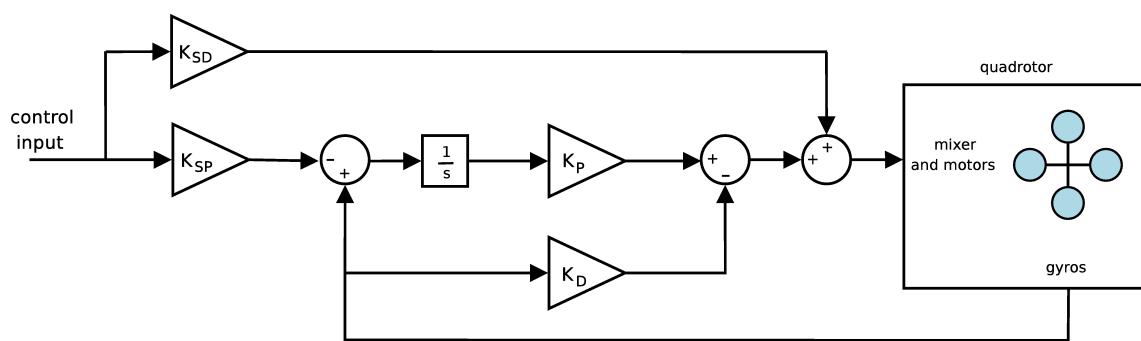


Figure A.7: PID stabilization loop.

scaled inputs to the motors (K_{SD} and K_D respectively), yielding the following control laws:

$$\mathbf{u}_p = K_{SDpr}u_{pi} - K_{Dpr}q + K_{Ppr} \int (q - K_{Spr}u_{pi}) \quad (\text{A.36})$$

$$\mathbf{u}_r = -K_{SDpr}u_{rl} - K_{Dpr}p + K_{Ppr} \int (p + K_{Spr}u_{rl}) \quad (\text{A.37})$$

$$\mathbf{u}_y = K_{SDy}u_{ya} - K_{Dy}r + K_{Py} \int (r - K_{Sy}u_{ya}) \quad (\text{A.38})$$

$$\mathbf{u}_t = u_{th} \quad (\text{A.39})$$

where $\mathbf{u}_p, \mathbf{u}_r, \mathbf{u}_y, \mathbf{u}_t$ are the computed controls, and u_{th}, u_{rl}, u_{pi} and u_{ya} are the pilot's control signals from the RC remote. Since the quadrotor is symmetrical, the controller constants for the pitch and roll sections are identical.

In Section A.2 we saw how the pitch and roll motions are determined by the differences in thrust ($T_0 - T_2$) and ($T_1 - T_3$); to achieve this, the outputs of equations A.36-A.38 need to be mixed. Similarly, to produce yaw motion we will need to vary the difference $((Q_0 + Q_2) - (Q_1 + Q_3))$.

As explained in Section A.3, the rotational speed obtained is related to the control input through the second order relationship f_{mot} (equation A.33). The main component of the rotor speed is given by the throttle input, to which all the other controls are added as appropriate. This yields the following mixing equations, which form the basis of all conventional control schemes for quadrotors:

$$\begin{aligned} \Omega_0 &= f_{mot}(\mathbf{u}_t + \mathbf{u}_p + \mathbf{u}_y) \\ \Omega_1 &= f_{mot}(\mathbf{u}_t + \mathbf{u}_r - \mathbf{u}_y) \\ \Omega_2 &= f_{mot}(\mathbf{u}_t - \mathbf{u}_p + \mathbf{u}_y) \\ \Omega_3 &= f_{mot}(\mathbf{u}_t - \mathbf{u}_r - \mathbf{u}_y). \end{aligned} \quad (\text{A.40})$$

The stability augmentation system will form an integral part of the dynamic equation of our model and will be integrated with the same time step we use for our dynamic model (0.04s corresponding to an update rate of 25Hz). In reality the augmentation system onboard the X3D runs at a frequency of 1KHz, but implementing such a rate of control would require us to integrate the dynamics of the system at the same speed, leading to a 40 fold increase in the computational requirements. If we look at the bandwidth B_m of the

propulsion group, we see that this is much smaller than 1KHz:

$$B_m = \frac{1}{\tau_\Omega} = 11.9 \text{Hz}, \quad (\text{A.41})$$

this led us to conclude that such a high control speed in the real platform was not chosen because of the dynamics of the system. We are inclined to think that a high speed was chosen because it would reduce the delay between sensing and actuation as well as increasing the rate at which the gyros were sampled; as a result, the control system would be more robust and less sensitive to gyro noise. Neither noise nor delay is included in our model, so in the light of these considerations we can comfortably assume that our 25Hz control will be sufficient to approximate the functionality of the real controller.

A.5 Full Model

The relationships presented above account for the full dynamic behaviour of our quadrotor, from the control inputs sent using the remote control, to the variables representing the state. For the implementation of the PID controllers and of the dynamic equations of the motors we will use discrete time equations; we therefore need to augment our state by adding 10 additional variables $h_{pt1}, h_{pt2}, h_{rl1}, h_{rl2}, h_{ya1}, h_{ya2}, \Omega_1, \Omega_2, \Omega_3, \Omega_4$. The first six variables represent the value of the PID integrators⁹ while the remainder store the instantaneous speed of each single rotor. With these additions the state vector becomes:

$$\mathbf{x} = [u, v, w, \phi, \theta, \psi, p, q, r, h_{pt1}, h_{pt2}, h_{rl1}, h_{rl2}, h_{ya1}, h_{ya2}, \Omega_1, \Omega_2, \Omega_3, \Omega_4].$$

Is now useful to collect in a single place all the equations that dictate the state evolution

⁹The reason why we need six variables instead of just three, is made clear later in this section.

of our quadrotor model. First, the acceleration prediction equations:

$$a_x = -(H_{0x} + H_{1x} + H_{2x} + H_{3x}) - \frac{1}{2}C_{xy}A_c\rho u|u| - \sin(\theta)g \quad (\text{A.42})$$

$$a_y = -(H_{0y} + H_{1y} + H_{2y} + H_{3y}) - \frac{1}{2}C_{xy}A_c\rho v|v| + \sin(\phi)\cos(\theta)g \quad (\text{A.43})$$

$$a_z = \frac{T_0 + T_1 + T_2 + T_3}{m} + \cos(\phi)\cos(\theta)g \quad (\text{A.44})$$

$$\alpha_x = \frac{qr(I_{yy} - I_{zz}) + J_r q \Omega_r + l(T_3 - T_1) - h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jx}}{Ix} \quad (\text{A.45})$$

$$\alpha_y = \frac{pr(I_{zz} - I_{xx}) + J_r p \Omega_r + l(T_2 - T_0) + h \sum_{j=0}^3 H_{jy} + \sum_{j=0}^3 (-1)^j R_{jy}}{Iy} \quad (\text{A.46})$$

$$\alpha_z = \frac{pq(I_{xx} - I_{yy}) + J_r \dot{\Omega}_r + \sum_{j=0}^3 (-1)^j Q_j + l(H_{0y} - H_{1x} - H_{2y} + H_{3x})}{Iz} \quad (\text{A.47})$$

and the state update equations:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{C}_{b_t+1}^{b_t} \mathbf{a}_t \Delta t \quad (\text{A.48})$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \boldsymbol{\alpha}_t \Delta t \quad (\text{A.49})$$

$$\boldsymbol{\Phi}_{t+1} = \boldsymbol{\Phi}_t + H(\boldsymbol{\Phi}_t) \boldsymbol{\omega}_t \Delta t \quad (\text{A.50})$$

$$h_{pt\ t+1} = h_{pt\ t} + [p - K_{Spr}(u_{pt} - b_{pt})] \quad (\text{A.51})$$

$$h_{rl\ t+1} = h_{rl\ t} + [q - K_{Spr}(u_{rl} - b_{rl})] \quad (\text{A.52})$$

$$h_{ya\ t+1} = h_{ya\ t} + [r - K_{Spr}(u_{ya} - b_{ya})] \quad (\text{A.53})$$

$$\Omega_{0\ t+1} = \Omega_{0\ t} + \left[\frac{1}{\tau_\Omega} (f_{mot}(\mathbf{u}_t + \mathbf{u}_p + \mathbf{u}_y) - \Omega_0) \right] \quad (\text{A.54})$$

$$\Omega_{1\ t+1} = \Omega_{1\ t} + \left[\frac{1}{\tau_\Omega} (f_{mot}(\mathbf{u}_t + \mathbf{u}_r - \mathbf{u}_y) - \Omega_1) \right] \quad (\text{A.55})$$

$$\Omega_{2\ t+1} = \Omega_{2\ t} + \left[\frac{1}{\tau_\Omega} (f_{mot}(\mathbf{u}_t - \mathbf{u}_p + \mathbf{u}_y) - \Omega_2) \right] \quad (\text{A.56})$$

$$\Omega_{3\ t+1} = \Omega_{3\ t} + \left[\frac{1}{\tau_\Omega} (f_{mot}(\mathbf{u}_t - \mathbf{u}_r - \mathbf{u}_y) - \Omega_3) \right] \quad (\text{A.57})$$

where $\mathbf{a} = [a_x, a_y, a_z]^T$, $\boldsymbol{\alpha} = [\alpha_x, \alpha_y, \alpha_z]^T$, $\mathbf{v} = [u, v, w]^T$, $\boldsymbol{\omega} = [p, q, r]^T$, $\boldsymbol{\Phi} = [\phi, \theta, \psi]^T$, $H(\boldsymbol{\Phi}_t)$ are the Euler kinematics equations defined in equation B.5 and $\mathbf{C}_{b_t}^{b_{t+1}}$ is the DCM between the body frame at time t and that at time $t+1$.

We also report again the stability augmentation system PID equations, to which the

input bias parameters $b_{pt}, b_{rl}, b_{ya}, b_{th}$ have been explicitly added:

$$u_p = K_{SDpr}(u_{pi} - b_{pt}) - K_{Dpr}q + K_{Ppr} \int q - K_{SPpr} \int (u_{pt} - b_{pt}) \quad (\text{A.58})$$

$$u_r = K_{SDpr}(b_{rl} - u_{rl}) - K_{Dpr}p + K_{Ppr} \int p - K_{SPpr} \int (b_{rl} - u_{rl}) \quad (\text{A.59})$$

$$u_y = K_{SDya}(u_{ya} - b_{ya}) - K_{Dy}r + K_{Py} \int r - K_{SPy} \int (u_{ya} - b_{ya}) \quad (\text{A.60})$$

$$u_t = u_{th}. \quad (\text{A.61})$$

The constants K_{Py} and K_{Sy} are effectively multiplied in equation A.36, which makes them numerically difficult to identify¹⁰. This is a well known problem ([113]) and is easily solved by rearranging the equations, obtaining equation A.58. The same problem also affects equations A.37 and A.38 so, again exploiting the linearity of the integration, we also rearranged these two equations. As a result of the rearrangement, each PID controller now has two different integrators, and this is why we need to have six separate state variables in our state (equation A.57) instead of only three.

A.6 Model Parameters

All the static parameters of the model were either directly measured on the real vehicle, or estimated using a precise CAD model of the quadrotor (Table A.2).

Parameter	Value	Description
m	0.392 Kg	mass
l	0.17 m	arm length
g	9.805 m/s^2	gravity
I_{xx}	0.00252 $Kg m^2$	moment of inertia x axis
I_{yy}	0.00252 $Kg m^2$	moment of inertia y axis
I_{zz}	0.00488 $Kg m^2$	moment of inertia z axis
J_r	0.0000095 $Kg m^2$	inertia of the propeller

Table A.2: Quadrotor analytical model parameters.

¹⁰Very different values of K_{Py} and K_{Sy} can give rise to the same product.

Appendix B

Algorithms and Definitions

B.1 List of Symbols

x	north position coordinate
y	west position coordinate
z	up position coordinate
ϕ	Euler roll angle
θ	Euler pitch angle
ψ	Euler yaw angle
u	body frame linear speed x axis
v	body frame linear speed y axis
w	body frame linear speed z axis
p	body frame angular speed about x
q	body frame angular speed about y
r	body frame angular speed about z
a_x	body frame linear acceleration x axis
a_y	body frame linear acceleration y axis
a_z	body frame linear acceleration z axis
α_x	body frame angular acceleration about x axis
α_y	body frame angular acceleration about x axis
α_z	body frame angular acceleration about x axis
\mathbf{p}	position vector in world coordinates
Φ	attitude vector in world coordinates

\mathbf{v}	linear velocities vector in body coordinates
$\boldsymbol{\omega}$	angular velocities vector in body coordinates
\mathbf{a}	linear accelerations vector in body coordinates
$\boldsymbol{\alpha}$	angular accelerations vector in body coordinates
\mathbf{C}_b^w	matrix transformation from the body reference frame to the world reference frame
$H(\Phi)$	Euler kinematic equations
Δt_{MOCAP}	processing delay of the motion capture system
V	airspeed
V_h	horizontal speed
β	sideslip angle
Δt	integration time step
d_{max}	maximum input delay
D	vector of input delays
\mathbf{x}	state vector
\mathbf{u}	input vector
\mathbf{y}	prediction vector
\mathbf{z}	measurement vector
$\boldsymbol{\nu}$	measurement noise vector
$\boldsymbol{\omega}$	process noise vector

B.2 Mathematical Definitions

For compactness in all of the following definitions we used the abbreviations $c()$, $s()$ and $t()$ in place of the trigonometric functions $\sin()$, $\cos()$ and $\tan()$.

Coordinate Transformations

X3D and Owl 6DoF transformation matrix from body frame at time t to body frame at time $t + 1$:

$$\mathbf{C}_{b_{t+1}}^{b_t} = \begin{bmatrix} c(q\Delta t)c(r\Delta t) & s(r\Delta t)s(q\Delta t)c(r\Delta t) - c(p\Delta t)s(r\Delta t) & c(p\Delta t)s(q\Delta t)c(r\Delta t) + s(p\Delta t)s(r\Delta t) \\ c(q\Delta t)s(r\Delta t) & s(p\Delta t)s(q\Delta t)s(r\Delta t) + c(p\Delta t)c(r\Delta t) & c(p\Delta t)s(q\Delta t)s(r\Delta t) - s(p\Delta t)c(r\Delta t) \\ -s(q\Delta t) & s(p\Delta t)c(q\Delta t) & c(p\Delta t)c(q\Delta t) \end{bmatrix} \quad (\text{B.1})$$

Toy car 3DoF transformation matrix from body frame at time t to body frame at time $t + 1$:

$$\mathbf{C}'_{b_{t+1}}^{b_t} = \begin{bmatrix} c(r\Delta t) & -s(r\Delta t) \\ s(r\Delta t) & c(r\Delta t). \end{bmatrix} \quad (\text{B.2})$$

Toy car 3DoF transformation matrix from body frame at time t to world frame at time t :

$$\mathbf{C}'_{b_t}^{w_t} = \begin{bmatrix} c(\psi) & s(\psi) \\ -s(\psi) & c(\psi). \end{bmatrix} \quad (\text{B.3})$$

ATTAS 3DoF transformation equation from body frame at time t to body frame at time $t + 1$:

$$\mathbf{C}''_{b_{t+1}}^{b_t} = \begin{bmatrix} c(p\Delta t)c(r\Delta t). \end{bmatrix} \quad (\text{B.4})$$

Euler kinematic equations

X3D and Owl Euler kinematic equations:

$$H(\Phi) = \begin{bmatrix} 1 & \frac{t(\theta)}{s(\phi)} & \frac{t(\theta)}{c(\phi)} \\ 0 & c(\phi) & -s(\phi) \\ 0 & \frac{s(\phi)}{c(\theta)} & \frac{c(\phi)}{c(\theta)} \end{bmatrix} \quad (\text{B.5})$$

where $\Phi = [\phi, \theta, \psi]^T$.

ATTAS Euler kinematic equations:

$$H''(\Phi) = \begin{bmatrix} 1 & 0 \\ 0 & c(\phi) \end{bmatrix} \quad (\text{B.6})$$

where $\Phi = [\phi, \theta]^T$.

B.3 Parameters

The following are the parameters that were empirically determined during the development of the PID controller used on the X3D quadrotor platform (see section 7.2.2).

parameter	value
K_{zi}	0.00185
K_{zp}	0.20
K_{zd}	8
limit Δ_{z_k}	[-1.2,1.2]m
K_{xp}	0.215
K_{xd}	11
limit Δ_{x_k}	[-1.2,1.2]m
K_{yp}	0.215
K_{yd}	11
limit Δ_{y_k}	[-1.2,1.2]m
$K_{\phi p}$	1
$K_{\theta p}$	1
$K_{\psi p}$	-0.345

Table B.2: Parameters of the X3D PID controller.

The following noise parameters were manually set for the *X3DGPSNoise* model (see section 7.2.3).

parameter	value
σ_{a_x}	0.5 m/s ²
σ_{a_y}	0.5 m/s ²
σ_{a_z}	1.0 m/s ²
σ_{α_x}	0.8 rad/s ²
σ_{α_y}	0.8 rad/s ²
σ_{α_z}	0.8 rad/s ²

Table B.3: Noise values used in the *X3DGPSNoise* model.

B.4 Algorithms

B.4.1 Numerical Differentiation

The numerical differentiation used to compute velocities and accelerations from pose data is simply the inverse of the operations of integration introduced in Sections 3.3.1, 3.3.3 and 3.3.4. We write it here for completeness.

$$\mathbf{v}_t = \mathbf{C}_{w_t}^{b_t}{}^T (\mathbf{p}_{t+1} - \mathbf{p}_t) / \Delta t \quad (B.7)$$

$$\boldsymbol{\omega}_t = H(\boldsymbol{\Phi}_t)^T (\boldsymbol{\Phi}_{t+1} - \boldsymbol{\Phi}_t) / \Delta t \quad (B.8)$$

$$\mathbf{a}_t = \mathbf{C}_{b_{t+1}}^{b_t}{}^T (\mathbf{v}_{t+1} - \mathbf{v}_t) / \Delta t \quad (B.9)$$

$$\boldsymbol{\alpha}_t = (\boldsymbol{\omega}_{t+1} - \boldsymbol{\omega}_t) / \Delta t \quad (B.10)$$

In the case of the ATTAS aircraft $H''(\boldsymbol{\Phi}_t)$ and $\mathbf{C}''_{w_t}^{b_t}$ are substituted by $H(\boldsymbol{\Phi}_t)$ and $\mathbf{C}_{w_t}^{b_t}$ respectively.

B.4.2 Maximum Likelihood Estimation

Maximum likelihood is a general principle used in machine learning as well as in many engineering disciplines. Although following [113] we apply the principle in a very standard form, we will briefly sketch the logic behind its derivation, and the approximation applied when it is used in combination with a nonlinear optimization algorithm to estimate the parameters of a model as we do in section 4.1.

The likelihood is defined as the probability density over the observed data $\mathbf{z}_{0:N} = \{\mathbf{z}_0, \dots, \mathbf{z}_N\}$ given a defined set of parameters $(\boldsymbol{\Theta})$ and obviously the control inputs:

$$p(\mathbf{z}_{0:N} | \boldsymbol{\Theta}, \mathbf{u}_t). \quad (B.11)$$

In the specific case of an analytical model, the maximum likelihood criterion corresponds to selecting $\boldsymbol{\Theta}$ so as to maximize the likelihood:

$$\boldsymbol{\Theta}_{ML} = \arg \max_{\boldsymbol{\Theta}} p(\mathbf{z}_{0:N} | \boldsymbol{\Theta}, \mathbf{u}_t). \quad (B.12)$$

If we can compute $p(\mathbf{z}_{0:N} | \boldsymbol{\Theta}, \mathbf{u}_t)$, then we can apply one of the well known optimization methods in order to obtain the parameter vector $\boldsymbol{\Theta}_{ML}$.

We can rewrite the observed data in terms of the data predicted by the model \mathbf{y} and its error $\boldsymbol{\nu}$. This form is widely used in the system identification community and as we have already seen in section 2.2.4 it is called the OE form:

$$\mathbf{z}_t = \mathbf{y}_t(\boldsymbol{\Theta}, \mathbf{u}_t) + \boldsymbol{\nu}_t. \quad (B.13)$$

It is worth emphasising that since the parameters Θ and the inputs \mathbf{u}_t are fixed, the output of the model $\mathbf{y}(\Theta)$ is deterministic.

For mathematical tractability it is standard practice to approximate the error (ν) as a Gaussian distribution, and it is also assumed that the errors at different time points are statistically independent:

$$E\{\boldsymbol{\nu}_i \boldsymbol{\nu}_j^T\} = R\delta_{ij}, \quad (\text{B.14})$$

where δ_{ij} is the Kronecker delta¹ and R is the covariance matrix of the error. Under this assumption it can be shown ([159]) that the likelihood used in equation B.12 can be written as

$$p(\mathbf{z}_{0:N} | \Theta, R, \mathbf{u}_t) = \frac{1}{\sqrt{(2\pi)^n |R|^N}} \exp\left(-\frac{1}{2} \sum_{t=1}^N (\mathbf{z}_t - \mathbf{y}_t)^T R^{-1} (\mathbf{z}_t - \mathbf{y}_t)\right), \quad (\text{B.15})$$

where $|R|$ denotes the determinant of the covariance matrix R and n is the dimensionality of the measurement vector \mathbf{y} . Instead of maximizing the likelihood, it is usually preferred to minimize its negative logarithm since this will produce the same Θ_{ML} while leading to a simpler and better behaved optimization problem.

Unfortunately the covariance matrix R is unknown. A way of dealing with this ([113]) is to apply a relaxation strategy in which an estimation of R given θ is first produced by:

$$R = \frac{1}{N} \sum_{t=1}^N (\mathbf{z}_t - \mathbf{y}_t) (\mathbf{z}_t - \mathbf{y}_t)^T, \quad (\text{B.16})$$

and this is then used to minimize the log-likelihood; iterating these two steps produces a better and better estimation of R as well as a better estimation of Θ_{ML} . As we have explained in section 2.2.4, it is necessary in practice to split whole the dataset into W smaller data windows (of length T)²; in that situation, under the independent error assumption that we made in equation B.14 the covariance matrix computation becomes

$$R = \frac{1}{WT} \sum_{w=1}^W \sum_{t=1}^T (\mathbf{z}_t - \mathbf{y}_t) (\mathbf{z}_t - \mathbf{y}_t)^T, \quad (\text{B.17})$$

where with a slight abuse of notation we have reused the symbol R since this cumulative form is the only one we will actually use.

¹ $\delta_{ij} = 1$ for $i = j$, and 0 for $i \neq j$

²The size of the dataset therefore becomes $N = WT$.

Taking the logarithm of equation B.15 and substituting R with equation B.17 we obtain the expression of the log-likelihood:

$$LL(\Theta) = \frac{1}{2}nWT + \frac{WT}{2}\ln(|R|) + \frac{WTn}{2}\ln(2\pi). \quad (\text{B.18})$$

where we can notice that for a given dataset, n and WT will not change. Ultimately our optimization problem therefore reduces to finding the parameters Θ that minimize $|R|$; in our implementation we assume R to be diagonal, a very common approximation.

Bibliography

- [1] T. Abatzoglou and B. O'Donnell. Minimization by coordinate descent. *Journal of Optimization Theory and Applications*, 36(2):163–174, 1982.
- [2] P. Abbeel. *Apprenticeship Learning and Reinforcement Learning with Application to Robotic Control*. PhD thesis, Stanford University, 2008.
- [3] P. Abbeel, A. Coates, M. Montemerlo, A. Y. Ng, and S. Thrun. Discriminative training of Kalman filters. In *Proceedings of Robotics: Science and Systems (RSS)*, 2005.
- [4] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In *Neural Information Processing Systems*, 2006.
- [5] P. Abbeel, V. Ganapathi, and A. Y. Ng. Modeling vehicular dynamics, with application to modeling helicopters. In *Neural Information Processing Systems*, 2006.
- [6] P. Abbeel and A. Y. Ng. Learning first order markov models for control. In *Neural Information Processing Systems*, 2005.
- [7] P. Abbeel, M. Quigley, and A. Y. Ng. Using inaccurate models in reinforcement learning. In *Proceeding of the 23rd International Conference on Machine Learning*, 2006.
- [8] M. Achtelik, A. Bachrach, R. Heb, S. Prentice, and N. Roy. Autonomous navigation and exploration of a quadrotor helicopter in GPS-denied indoor environments. In *First Symposium on Indoor Flight Issues*, 2009.
- [9] M. S. Alam and M. O. Toki. Dynamic modelling of a single-link flexible manipulator system : A particle swarm optimisation approach. *Journal of low frequency noise, vibration and active control*, 26(1):57–72, 2007.
- [10] C. Atkeson, A. Moore, and S. Schaal. Locally weighted learning. *AI Review*, 11:11–73, 1997.
- [11] D. A. Augusto and H. J. C. Barbosa. Symbolic regression via genetic programming. In *Proceedings of the Brazilian Symposium on Neural Networks*, page 173. IEEE Computer Society, 2000.
- [12] Autopilot. Do it yourself UAV.
<http://autopilot.sourceforge.net>, 2010.
- [13] J. Bagnell and S. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1615–1620, 2001.

- [14] A. V. Balakrishnan. *Stochastic System Identification Techniques*, volume Stochastic Optimization and Control. John Wiley & Sons, London, 1968.
- [15] C. Balas. Modelling and linear control of a quadrotor. Master's thesis, School of Engineering, Cranfield University, 2007.
- [16] W. Banzhaf. Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In *Proceedings of Parallel Problem Solving from Nature III (PPSN)*, volume 866 of *LNCS*, pages 322–332. Springer-Verlag, 1994.
- [17] W. Banzhaf, F. D. Francone, and P. Nordin. The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In *Proceedings of Parallel Problem Solving from Nature IV (PPSN)*, volume 1141 of *LNCS*, pages 300–309. Springer Verlag, Berlin, 1996.
- [18] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, 1998.
- [19] G. J. Barlow, C. K. Oh, and E. Grant. Incremental evolution of autonomous controllers for unmanned aerial vehicles using multi-objective genetic programming. In *Proceedings of 2004 IEEE Conference on Cybernetics and Intelligent Systems*, volume 2, pages 689–694, 2004.
- [20] B. Basappa and R. V. Jategaonkar. Aspects of feed forward neural network modeling and its application to lateral-directional flight data. Technical report, NAL Institutional Repository (India), 1995.
- [21] P. Bendotti and J. C. Morris. Robust hover control for a model helicopter. In *Proceedings of the American Control Conference*, volume 1, pages 682–687, 1995.
- [22] H. G. Beyer and H. P. Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.
- [23] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.
- [24] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York, NY, USA, 2008.
- [25] C. M. Bishop and M. Svensén. Bayesian hierarchical mixtures of experts. In *Proceedings Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 57–64. Morgan Kaufmann, 2003.
- [26] J. C. Bongard. Behavior chaining: Incremental behavior integration for evolutionary robotics. In *Proceedings of Artificial Life XI*, Cambridge, MA, USA, 2008. MIT Press.
- [27] J. C. Bongard and H. Lipson. Automated robot function recovery after unanticipated failure or environmental change using a minimum of hardware trials. In *Proceedings of the NASA/DoD Conference on Evolvable Hardware*, 2004.
- [28] J. C. Bongard and H. Lipson. Automating genetic network inference with minimal physical experimentation using coevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 333–345. Springer, 2004.

- [29] J. C. Bongard and H. Lipson. ‘managed challenge’ alleviates disengagement in co-evolutionary system identification. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 531–538, 2005.
- [30] J. C. Bongard and H. Lipson. Nonlinear system identification using coevolution of models and tests. *IEEE Transaction on evolutionary computation*, 9(4):361–384, 2005.
- [31] J. C. Bongard and H. Lipson. Automated reverse engineering of nonlinear dynamical systems. *PNAS*, 104(24):9943–9948, 2007.
- [32] J. C. Bongard, V. Zykov, and H. Lipson. Automated synthesis of body schema using multiple sensor modalities. In *Proceedings of the Tenth International Conference on Artificial Life, (ALIFE X)*, pages 220–226, 2006.
- [33] J. C. Bongard, V. Zykov, and H. Lipson. Resilient machines through continuous self-modeling. *Science*, 314:1118–1121, 2006.
- [34] S. Bouabdallah. *Design and control of quadrotors with application to autonomous flying*. PhD thesis, EPFL, 2007.
- [35] S. Bouabdallah, A. Noth, and R. Siegwart. PID vs LQ control techniques applied to an indoor micro quadrotor. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.
- [36] M. F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, New York, 2007.
- [37] G. Buskey, J. Roberts, P. Corke, and G. Wyeth. Helicopter automation a using a low-cost sensing system. *Computing & Control Engineering Journal*, 15(2):8–9, 2004.
- [38] R. H. Byrne, C. T. Abdallah, and P. Dorato. Experimental results in robust lateral control of highway vehicles. *IEEE Control Systems Magazine*, 18(2):70–76, 1998.
- [39] R. Calabretta, A. Di Ferdinando, G. P. Wagner, and D. Parisi. What does it take to evolve behaviorally complex organisms? *BioSystems*, 69:254–262, 2001.
- [40] R. Calabretta, S. Nolfi, D. Parisi, and G. P. Wagner. Duplication of modules facilitates functional specialization. *Artificial Life*, 6:69–84, 2000.
- [41] C. L. Castillo, W. Moreno, and K. P. Valavanis. Unmanned helicopter waypoint trajectory tracking using model predictive control. In *Proceedings of the Mediterranean Conference on Control and Automation (MED07)*, pages 1–8, 2007.
- [42] P. Castillo, A. Dzul, and R. Lozano. Real-time stabilization and tracking of a four-rotor mini rotorcraft. *IEEE Transaction on Control Systems Technology*, 12(4):510–516, 2004.
- [43] R. Chartrand. Numerical differentiation of noisy, nonsmooth data. Submitted, 2007.
- [44] Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, 1997.
- [45] Y. Chen, B. Yang, J. Dong, and A. Abraham. Time-series forecasting using flexible neural tree model. *Quality Control and Applied Statistics*, 52(2):221–224, 2007.
- [46] D. Cliff and G. Miller. *Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations*, pages 200–218. Springer, Berlin, 1995.

- [47] D. T. Cliff. Computational neuroethology: A provisional manifesto. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 29–39, Cambridge, MA, USA, 1991. MIT Press-Bradford Books.
- [48] A. Coates, P. Abbeel, and A. Y. Ng. Apprenticeship learning for helicopter control. *Communications of the ACM*, 52(7), 2009.
- [49] D. A. Cohn. Neural network exploration using optimal experiment design. In *Neural Networks*, pages 679–686. Morgan Kaufmann, 1994.
- [50] CompuLab. CM-iGLX computer-on-module.
<http://www.complab.co.il/iglx/html/iglx-cm-datasheet.htm>, 2010.
- [51] O. Cordon, F. Herrera, and M. Lozano. A classified review on the combination fuzzy logic-genetic algorithms bibliography: 1989-1995. Technical report, Deptartment of Computer Science and A.I., University of Granada, 1995.
- [52] I. D. Cowling, O. A. Yakimenko, J. F. Whidborne, and A. K. Cooke. A prototype of an autonomous controller for a quadrotor UAV. In *Proceedings of the European Control Conference*, 2007.
- [53] Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Proceedings of Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.
- [54] DARPA. DARPA grand challenge rulebook.
http://www.darpa.mil/grandchallenge05/Rules_8oct04.pdf, 2004.
- [55] H. de Garis. Building artificial nervous systems using genetically programmed neural network modules. In *Proceedings of the 7th International Conference on Machine Learning*, pages 132–139. Morgan Kaufmann, Palo Alto, CA, 1990.
- [56] R. De Nardi. Flocking of UAVs. Software model and limited vision simulations. Master's thesis, University of Padua, Italy, 2004.
- [57] R. De Nardi and O. E. Holland. UltraSwarm: A further step towards a flock of miniature helicopters. In *SAB Workshop on Swarm Robotics*, 2006.
- [58] R. De Nardi, O. E. Holland, J. Woods, and A. Clark. SwarMAV: A swarm of miniature aerial vehicles. In *Proceedings of the 21st Bristol International UAV Systems Conference*, 2006.
- [59] Y. Dodge. *The Oxford Dictionary of statistical Terms*. Oxford University Press, USA, 2003.
- [60] B. Dolin, F. H. Bennett, and E. G. Rieffel. Co-evolving an effective fitness sample: Experiments in symbolic regression and distributed robot control. In *Proceedings of the 2002 ACM symposium on Applied computing (SAC)*, pages 553–559, New York, NY, USA, 2002. ACM.
- [61] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, NY, 2000.
- [62] A. Dzul, P. Castillo, and R. Lozano. Real-time stabilization and tracking of a four-rotor mini rotorcraft. *IEEE Transaction on Control System Technology*, 3(14):510–516, 2004.

- [63] P. Eggenberger, A. Ishiguro, S. Tokura, T. Kondo, and Y. Uchikawa. Toward Seamless transfer from simulated to real worlds: A dynamically-rearranging neural network approach. In *Proceedings of the 8th European Workshop on Learning Robots (EWLR-8)*, pages 44–60. Springer Berlin / Heidelberg, 2000.
- [64] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer-Verlag, Berlin, 2003.
- [65] A. Ekárt and S. Z. Németh. A metric for genetic programs and fitness sharing. In *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP)*, volume 1802 of *LNCS*, pages 259–270. Springer-Verlag, Berlin, 2000.
- [66] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.
- [67] W. E. Faller and S. J. Schreck. Neural network: Applications and opportunities in aeronautics. *Progress in Aerospace Sciences*, 32:433–456, 1996.
- [68] G. Fay. Derivation of the aerodynamic forces for the mesicopter simulation. Technical report, Stanford University, 2001.
- [69] R. Feldt. Using genetic programming to systematically force software diversity. Technical Report 296L, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1998.
- [70] S. G. Ficici and J. B. Pollack. Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states. In *Proceedings of the Sixth International Conference on Artificial Life (ALIFE VI)*, Cambridge, MA, USA, 1998. MIT Press.
- [71] D. Floreano and F. Mondada. Evolution of plastic neurocontrollers for situated agents. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, USA, 1996. MIT Press-Bradford Books.
- [72] Lawrence J. Fogel. *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*. Wiley Series on Intelligent Systems. John Wiley and Sons, New York NY, 1999.
- [73] A. S. Fukunaga and A. B. Kahng. Improving the performance of evolutionary optimization by dynamically scaling the evaluation function. In *Proceedings of the IEEE Congress on Evolutionary Computation, (CEC)*, pages 182–187. IEEE Press, 1995.
- [74] P. J. Gage and J. S. Drobik. Automatic discovery of novel flight dynamic equations. In *Proceedings of the AIAA Atmospheric Flight Mechanics Conference*, pages 13–17, 1997.
- [75] N. Garcia-Pedrajas and D. Ortiz-Boyer. A cooperative constructive method for neural networks for pattern recognition. *Pattern Recognition*, 40(1):80–98, 2007.
- [76] V. Gavrilets, I. Martinos, B. Mettler, and E. Feron. Control logic for automated aerobatic flight of miniature helicopter. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2002.
- [77] M. Gevers. Identification for control : From the early achievements to the revival of experiment design. *European Journal of Control*, 11:1–18, 2005.

- [78] Ascending Technologies GmbH. Asctec X3D-BL.
<http://www.asctec.de/main/index.php>, 2010.
- [79] Microdrones GmbH. MD4-200.
http://www.microdrones.com/en_md4-200.php, 2010.
- [80] F. Gomez. *Robust Non-Linear Control through Neuroevolution*. PhD thesis, University of Texas, Austin, 2000.
- [81] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.
- [82] F. Gomez and R. Miikkulainen. Transfer of neuroevolved controllers in unstable domains. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2004.
- [83] F. J. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, pages 654–662, Berlin, Germany, 2006.
- [84] J. Grauer, J. K. Conroy, J. E. Jr. Hubbard, and J. S. Humbert. System identification of a miniature helicopter. *Journal of Aircraft*, 46(4):1260–1270, 2009.
- [85] S. Grzonka, S. Bouabdallah, G. Grisetti, W. Burgard, and R. Siegwart. Towards a fully autonomous indoor helicopter. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Nice, France, 2008.
- [86] S. Grzonka, G. Grisetti, and W. Burgard. Towards a navigation system for autonomous indoor flying. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009.
- [87] D. Gu and H. Housheng. Neural predictive control for a car-like mobile robot. *Robots and Autonomous Systems*, 39(2):73–86, 2002.
- [88] N. Guenard, T. Hamel, and E. Laurent. Control laws for the tele operation of an unmanned aerial vehicle known as an X4-flyer. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3249–3254, 2006.
- [89] D. Gurdan, J. Stumpf, M. Achtelik, K. M. Doth, G. Hirzinger, and D. Rus. Energy-efficient autonomous four-rotor flying robot controlled at 1kHz. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2006.
- [90] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 2:159–195, 2001.
- [91] I. Harvey, E. A. Di Paolo, R. Wood, M. Quinn, and E. Tuci. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life*, 11(1-2):79–98, 2005.
- [92] I. Harvey, P. Husbands, and D. Cliff. Seeing the light: Artificial evolution, real vision. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 392–401, Cambridge, MA, USA, 1994. MIT Press.
- [93] S. Hauert, J. C. Zufferey, and D. Floreano. Reverse-engineering of artificially evolved controllers for swarms of robots. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 55–61, 2009.

- [94] R. He, S. Prentice, and N. Roy. Planning in information space for a quadrotor helicopter in a GPS-denied environment. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2008.
- [95] D. Higdon. *Automatic Control of Inherently Unstable Systems with Bounded Control Inputs*. PhD thesis, Department of Aeronautics and Astronautics, Standford University, 1963.
- [96] H. Hjalmarsson, M. Gevers, and F. De Bruyne. For model-based control design, closed loop identication gives better performance. *Automatica*, 32(12):1659–1673, 1996.
- [97] F. Hoffmann, T. J. Koo, and O. Shakernia. Evolutionary design of a helicopter autopilot. In *3rd On-Line World Conf. on Soft Computing (WSC3)*, 1998.
- [98] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2007.
- [99] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun. Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In *Proceedings of the 26th American Control Conference*, New York, NY, USA, 2007.
- [100] G. M. Hoffmann, S. L. Waslander, and C. J. Tomlin. Quadrotor helicopter trajectory tracking control. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, New York, NY, USA, 2008.
- [101] O. E. Holland, J. Woods, R. De Nardi, and A. Clark. Beyond swarm intelligence: The UltraSwarm. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*. IEEE, 2005.
- [102] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.
- [103] X. Hu, R. Shonkwiler, and M. Spruill. Random restart in global optimization. Technical Report 110592-015, Georgia Tech. School of Mathematics, 1997.
- [104] P. J. Huber. *Robust Statistics*. Wiley, New York, NY, USA, 1981.
- [105] P. Husbands, T. Smith, M. O’Shea, N. Jakobi, J. Anderson, and A. Phillipides. Brains, gases and robots. In *Proceedings ICANN’98*, pages 51–63. Springer-Verlag, 1998.
- [106] K. W. Iliff. *Identification and Stochastic Control with Applications to Flight Control in Turbulence*. PhD thesis, University of California Los Angeles, 1973.
- [107] J. H. Imada and B. J. Ross. Using feature-based fitness evaluation in symbolic regression with added noise. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 2153–2158. ACM, New York, NY, USA, 2008.
- [108] Digi International Inc. Xbee-pro 802.15.4 oem rf modules.
www.digi.com/products/wireless/point-multipoint/xbee-series1-module.jsp, 2010.
- [109] Miniature Aircraft Usa inc. X-cell 60.
www.x-cellrc helicopters.com/fury_kit.htm, 2010.

- [110] N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325–367, 1997.
- [111] N. Jakobi. Harnessing morphogenesis. In *Proceedings of Information Processing in Cells and Tissues*, 1997.
- [112] H. Jansson. *Experiment Design with Applications in Identification for Control*. PhD thesis, Department of Signals, Sensors and Systems Royal Institute of Technology (KTH), 2004.
- [113] R. V. Jategaonkar. *Flight Vehicle System Identification, A Time Domain Methodology*. AIAA, 2006.
- [114] D. A. Jirenhed, G. Hesslow, and T. Ziemke. Exploring internal simulation of perception in mobile robots. In *Proceedings of the Fourth European Workshop on Advanced Mobile Robots*, pages 107–113, 2001.
- [115] E. N. Johnson and S. K. Kannan. Adaptive flight control for an autonomous unmanned helicopter. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2002.
- [116] S. J. Julier and J. K. Uhlmann. New extension of the Kalman filter to nonlinear systems. In *Proceedings of SPIE, Signal Processing, Sensor Fusion, and Target Recognition VI*, pages 182–193, 1997.
- [117] Y. Kassahun, J. Schwendner, J. de Gea, M. Edgington, and F. Kirchner. Learning complex robot control using evolutionary behavior based systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 129–136, New York, NY, USA, 2009. ACM.
- [118] R. E. Keller and W. Banzhaf. Explicit maintenance of genetic diversity on genospaces. Technical report, Computer Science Department, Dortmund University, 1994.
- [119] AirRobot GmbH & Co. KG. AR100-B.
http://www.airrobot.com/deutsch/produkt_01_daten.php, 2010.
- [120] H. J. Kim, D. H. Shim, and S. Sastry. Nonlinear model predictive tracking control for rotorcraft-based unmanned aerial vehicles. In *Proceedings of the American Control Conference*, volume 5, pages 3576–3581, 2002.
- [121] J. Ko, D. Klein, D. Fox, and D. Haehnel. GP-UKF: Unscented Kalman filters with Gaussian process prediction and observation models. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007.
- [122] J. Ko, D. J. Klein, D. Fox, and D. Haehnel. Gaussian processes and reinforcement learning for identification and control of an autonomous blimp. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2007.
- [123] R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized helicopter control. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 145–152, New York, NY, USA, 2009. ACM.
- [124] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1990.
- [125] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1992.

- [126] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2005.
- [127] D. H. Kraft, F. E. Petry, W. P. Buckles, and T. Sadasivan. The use of genetic programming to build queries for information retrieval. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 468–473. IEEE Press, 1994.
- [128] M. La Civita, G. Papageorgiou, W. C. Messner, and T. Kanade. Design and flight testing of a high-bandwidth \mathcal{H}_∞ loop shaping controller for a robotic helicopter. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2002.
- [129] W. B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Kluwer Academic Publishers Norwell, MA, USA, 1998.
- [130] W. B. Langdon. Size fair and homologous tree crossovers for tree genetic programming. *Genetic Programming and Evolvable Machines*, 1(1-2):95–119, 2000.
- [131] J. G. Leishman. *Principles of Helicopter Aerodynamics*. Cambridge University Press, New York, NY, 2000.
- [132] S. Leven, J. C. Zufferey, and D. Floreano. A simple and robust fixed-wing platform for outdoor flying robot experiments. In *International Symposium on Flying Insects and Robots*, pages 69–70, 2007.
- [133] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *The Quarterly of Applied Mathematics*, 2:164–168, 1944.
- [134] L. M. Li and S. A. Billings. Continuous time non-linear system identification in the frequency domain. *International Journal of Control*, 74(11):1052–1061, 2001.
- [135] Hirobo Limited. XRB Lama helicopter.
<http://model.hirobo.co.jp/products/0301-905/index.html>, 2010.
- [136] L. Ljung. *System Identification: Theory for the User*. Prentice Hall, 1999.
- [137] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmer-son, D. Perez, R. G. Reynolds, and Y. Saez. The WCCI 2008 simulated car racing competition. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.
- [138] R. Lozano, P. Castillo, and A. Dzul. Stabilization of a mini rotorcraft having four rotors. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2693–2698, 2004.
- [139] S. Y. Lu. The tree-to-tree distance and its application to cluster analysis. *IEEE Transaction on Pattern Analysis Machine Intelligence*, 1(2):219–224, 1979.
- [140] S. Lupashin, A. Schöllig, M. Sherback, and R. D’Andrea. A simple learning strategy for high-speed quadrocopter multi-flips. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2010.
- [141] D. J. C. MacKay. Information-based objective functions for active data selection. *Neural Computation*, 4:590–604, 1992.

- [142] D. J. C. MacKay. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472, 1992.
- [143] D. J. C. MacKay. Probable networks and plausible predictions - A review of practical bayesian methods for supervised neural networks. *Network: Computation in Neural Systems*, (6):469–505, 1995.
- [144] T. Madani and A. Benallegue. Backstepping sliding mode control applied to a miniature quadrotor flying robot. In *Proceeding of the 32nd IEEE Annual Conference on Industrial Electronics (IECON)*, pages 700–705, 2006.
- [145] T. Madani and A. Benallegue. Control of a quadrotor mini-helicopter via full state backstepping technique. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 1515–1520, 2006.
- [146] S. W. Mahfoud. *Niching Methods for genetic algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1995.
- [147] R. E. Maine and K. W. Iliff. *Identification of Dynamic Systems - Applications to Aircraft. Part 1: The Output Error Approach*. Number AGARDograph No. 300 in AGARD Flight Test Techniques Series. NASA Dryden Flight Research Center, 1986.
- [148] M. Marco Lando, M. Battipede, and P. A. Gili. Neuro-fuzzy techniques for the air-data sensor calibration. *Journal of Aircraft*, 44(3):945–953, 2007.
- [149] A. McGrew, J. S. Valenti, M. Levine, D. Frank, and J. How. Hover, transition, and level flight control design for a single-propeller indoor airplane. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2007.
- [150] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 303–309. Morgan Kaufmann, San Francisco, CA, 1995.
- [151] R. K. Mehra. Identification of stochastic linear dynamic system using Kalman filter representation. *AIAA*, 9:28–31, 1971.
- [152] B. Mettler, M. B. Tischler, and T. Kanade. System identification of a model-scale helicopter. Technical Report CMU-RI-TR-00-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.
- [153] T. Miconi. Why coevolution doesn't work: Superiority and progress in coevolution. In *Proceedings of the 12th European Conference on Genetic Programming (EuroGP)*, pages 49–60, 2009.
- [154] O. Miglino, H. Hautop Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2:417–434, 1995.
- [155] O. Miglino, H. Hautop Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. Technical report, Institute of PsychologyArtificial Life C.N.R, Rome, Rome, Italy, 1995.
- [156] Mikrokopter. Homepage.
<http://www.mikrokopter.de/>, 2010.
- [157] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

- [158] W. T. Miller, R. S. Sutton, and P. J. Werbos, editors. *Neural networks for control*. MIT Press, Cambridge, MA, USA, 1990.
- [159] T. M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.
- [160] MOD. The MOD Grand Challenge.
http://www.science.mod.uk/engagement/grand_challenge/grand_challenge.aspx, 2010.
- [161] A. Mokhtari and A. Benallegue. Dynamic feedback controller of Euler angles and wind parameters estimation for a quadrotor unmanned aerial vehicle. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2004.
- [162] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturization: A tool for investigation in control algorithms. In *Third International Symposium on Experimental Robotics*. Springer Verlag, 1994.
- [163] A. Moraglio. *Towards a Geometric Unification of Evolutionary Algorithms*. PhD thesis, Department of Computer Science, University of Essex, UK, 2007.
- [164] A. Moraglio and R. Poli. Geometric landscape of homologous crossover for syntactic trees. Technical Report CSM 430, Computer Science Department, University of Essex, Wivenhoe Park, Colchester, CO4 3SQ, UK, 2005.
- [165] D. E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997.
- [166] P. Muren. The proxflyer.
<http://www.proxflyer.com>, 2010.
- [167] K. Nakamura. Mr. Kimio NAKAMURA's Coaxis Micro Helicopter.
<http://liaison.ms.u-tokyo.ac.jp/agusta/coaxis/nakamura.html>, 2010.
- [168] R. M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, New York, 1996.
- [169] O. Nelles. *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models*. Springer, 2001.
- [170] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Proceedings of the International Symposium on Experimental Robotics*, 2004.
- [171] A. Y. Ng, H. J. Kim, M. I. Jordan, S. Sastry, and S. Ballianda. Autonomous helicopter flight via reinforcement learning. *Advances in Neural Information Processing Systems*, 2004.
- [172] S. Nolfi. Evolving non-trivial behaviors on real robots: A garbage collecting robot. *Robotics and Autonomous System*, 22(3):187–198, 1997.
- [173] S. Nolfi. Evolutionary robotics: Exploiting the full power of selforganization. *Connection Science*, 10(3-4), 1998.
- [174] S. Nolfi and D. Floreano. How co-evolution can enhance the adaptive power of artificial evolution: Implications for evolutionary robotics. In *Proceedings of the First European Workshop on Evolutionary Robotics (EvoRobot98)*, pages 22–38. Springer, 1998.

- [175] S. Nolfi and D. Floreano. *Evolutionary robotics*. MIT Press, Cambridge, MA, USA, 2000.
- [176] J. Oliveira, Q. P. Chu, H. M. N. K. Balini, and W. G. M. Vos. Output error method and two step method for aerodynamic model identification. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, pages 15–18. AIAA, 2005.
- [177] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.
- [178] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, 2 edition, 1999.
- [179] G. D. Padfield. *Helicopter Flight Dynamics: The Theory and Application of Flying Qualities and Simulation Modeling*. AIAA, 1996.
- [180] L. Pagie and P. Hogeweg. Evolutionary consequences of coevolving targets. *Evolutionary Computation*, 5:401–418, 1998.
- [181] A. Palomino, S. Salazar-Cruz, and R. Lozano. Trajectory tracking for a four rotor mini-aircraft. In *Proceedings of the 44th IEEE Conference on Decision and control, and the European Control Conference*, pages 2505–2510, 2005.
- [182] A. C. Paris and M. Bonner. Nonlinear model development from flight test data for F/A-18E super hornet. *Journal of Aircraft*, 41(4):692–702, 2004.
- [183] S. Park, D. H. Won, M. S. Kang, T. J. Kim, H. G. Lee, and S. J. Kwon. RIC (robust internal-loop compensator) based flight control of a quad-rotor type UAV. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.
- [184] P. A. Pérez and A. Sala, editors. *Iterative identification and control: Advances in theory and applications*. Springer, 2002.
- [185] M. G. Perhinschi. A modified genetic algorithm for the design of autonomous helicopter control system. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, pages 1111–1120, 1997.
- [186] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008.
- [187] M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.
- [188] P. Pounds, R. Mahony, and P. Corke. Modelling and control of a quad-rotor robot. In *ACRA*, 2006.
- [189] R. W. Prouty. *Helicopter Performance, Stability, and Control*. Krieger Publishing Co, 1995.
- [190] O. Purwin and R. DAndrea. Performing aggressive maneuvers using iterative learning control. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1731–1736, 2009.

- [191] S. C. Raisinghani and A. K. Ghosh. Parameter estimation of an aeroelastic aircraft using neural networks. *Aeronautical journal*, 192(1011):25–29, 1998.
- [192] S. C. Raisinghani and A. K. Ghosh. Parameter estimation of an aeroelastic aircraft using neural networks. *Sadhana*, 25(2):181–191, 2000.
- [193] M. V. Rajesh, Archana R., A. Unnikrishnan, R. Gopikakumari, and J. Jacob. Evaluation of the ANN based nonlinear system models in the MSE and CRLB senses. In *Proceedings of the World Congress on Science, Engineering and Technology (WC-SET)*, pages 211–215, 2008.
- [194] C. E. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, USA, 2006.
- [195] J. F. Roberts, J. C. Zufferey, and D. Floreano. Energy management for indoor hovering robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2008.
- [196] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.
- [197] D. E. Rumelhart and J. L. McClellan. *Parallel Distributed Cognition: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge, MA, USA, 1986.
- [198] S. Samarasinghe. *Neural Networks for Applied Sciences and Engineering*. Auerbach Publications, Boston, MA, USA, 2006.
- [199] L. C. Sandoval Goes, E. Moreira Hemerly, B. C. De Oliveira Maciel, W. Rios Neto, C. Braga Mendonca, and J. Hoff. Aircraft parameter estimation using output-error methods. *Inverse problems in science and engineering*, 14(6):651–664, 2008.
- [200] S. Schaal and C. G. Atkeson. Constructive incremental learning from only local information. *Neural Computation*, 10(8):2047–2084, 1998.
- [201] S. Schaal, C. G. Atkeson, and S. Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 288–293, 2000.
- [202] J. Schmidhuber. Simple algorithmic theory of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. *Journal of SICE*, 48:21–32, 2009.
- [203] M. Schmidt and H. Lipson. Comparison of tree and graph encodings as function of problem complexity. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1674–1679, New York, NY, USA, 2007. ACM.
- [204] M. D. Schmidt and H. Lipson. Co-evolution of fitness maximizers and fitness predictors. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2005.
- [205] M. D. Schmidt and H. Lipson. Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, 12(6):736–749, 2008.
- [206] M. Schoenauer, M. Sebag, F. Jouve, B. Lamy, and H. Maitournam. Evolutionary identification of macro-mechanical models. In *Advances in Genetic Programming 2*, chapter 23, pages 467–488. MIT Press, Cambridge, MA, USA, 1996.

- [207] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
- [208] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Workshop on Computational Learning Theory*, pages 387–294. Morgan Kauffman, 1992.
- [209] D. H. Shim, H. J. Kim, and S. Sastry. Control system design for rotorcraft-based unmanned aerial vehicles using time-domain system identification. In *IEEE International Conference on Control Applications*, 2000.
- [210] D. H. Shim, H. J. Kim, and S. Sastry. A flight control system for aerial robots: Algorithms and experiments. In *Proceedings of the IFAC Control Engineering Practice Conference*, 2003.
- [211] B. Singh and R. Bhattacharya. Near time-optimal waypoint tracking algorithm for a 3-dof model helicopter. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2007.
- [212] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Deylon, P. Y. Glorennec, H. Hjalmarsson, and A. Juditsky. nonlinear black-box modeling in system identification: A unified overview. *Automatica*, 31, 1995.
- [213] W. Smart and M. Zhang. Applying online gradient descent search to genetic programming for object recognition. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation (CRPIT)*, pages 133–138. Australian Computer Society, Inc., 2004.
- [214] T. Soule and J. A. Foster. Code size and depth flows in genetic programming. In *Proceedings of the Second Annual Conference on Genetic Programming 1997*, pages 313–320. Morgan Kaufmann, San Francisco, CA, USA, 1997.
- [215] K. O. Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, Department of Computer Sciences, University of Texas, Austin, TX, 2004.
- [216] K. O. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen. Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2005.
- [217] B. L. Stevens and F. L. Lewis. *Aircraft Control and Simulation*. Wiley-Interscience, 2003.
- [218] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.
- [219] K. Swingler. *Applying Neural Networks, A Practical Guide*. Academic Press Limited, London, 1996.
- [220] Swarm System Ltd. Homepage.
<http://www.swarmsys.com>, 2010.
- [221] R. Tan, H. S. Law, B. K. Rajamani, and W. B. Zhang. Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons. *IEEE Transactions on Control Systems Technology*, 8(4):695–708, 2000.
- [222] I. Taney, M. Joachimczak, H. Hemmi, and K. Shimohara. Evolution of the driving styles of anticipatory agent remotely operating a scaled model of racing car. In *Proceedings of the IEEE Congress on Evolutionary Computation, (CEC)*, 2005.

- [223] I. Tanev, H. Yamazaki, T. Hiroyasu, and K. Shimohara. Evolution of general driving rules of a driving agent. In *From Animals to Animats 10: Proceedings of the Tenth International Conference on Simulation of Adaptive Behavior*, pages 488–498. Springer-Verlag Berlin, Heidelberg, 2008.
- [224] A. Tayebi and S. McGilvray. Attitude stabilization of a VTOL quadrotor aircraft. *IEEE Transaction on Control System Technology*, 3(14):562–571, 2006.
- [225] M. E. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1321–1328, 2006.
- [226] H. Theil. *Economic Forecast and Policy*. New Holland, 1961.
- [227] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, MA, USA, 2005.
- [228] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, and G. M. Hoffmann. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, 2006.
- [229] M. B. Tischler and J. Kaletka. Modeling XV-15 tilt-rotor aircraft dynamics by frequency and time-domain identification techniques. Technical report, NASA, 1986.
- [230] M. B. Tischler and R. K. Remple. *Aircraft System Identification, Engineering Methods with Flight Test Examples*. AIAA, 2006.
- [231] J. Togelius. *Optimization, Imitation and Innovation: Computational Intelligence and Games*. PhD thesis, Department of Computing and Electronic Systems, University of Essex, Colchester, UK, 2007.
- [232] J. Togelius, R. De Nardi, and S. M. Lucas. Making racing fun through player modeling and track evolution. In *Proceedings of the SAB Workshop on Adaptive Approaches to Optimizing Player Satisfaction*, 2006.
- [233] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *Proceedings of the IEEE Computational Intelligence and Games Symposium (CIG07)*, 2007.
- [234] J. Togelius, R. De Nardi, H. Marques, R. Newcombe, S. M. Lucas, and O. E. Holland. Nonlinear dynamics modelling for controller evolution. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO)*, 2007.
- [235] J. Togelius, S. M. Lucas, H. D. Thang, J. M. Garibaldi, T. Nakashima, C. H. Tan, I. Elhanany, S. Berant, P. Hingston, R. M. MacCallum, T. Haferlach, A. Gowrisankar, and P. Burrow. The 2007 IEEE CEC simulated car racing competition. *Genetic Programming and Evolvable Machines*, 9(4):295–329, 2008.
- [236] J. Togelius, T. Schaul, J. Schmidhuber, and F. Gomez. Countering poisonous inputs with memetic neuroevolution. In *Proceedings of Parallel Problem Solving from Nature VII (PPSN)*, 2008.
- [237] TORCS. The open racing car simulator.
<http://torcs.sourceforge.net/>, 2010.
- [238] U-blox. LEA-5 module series.
<http://www.u-blox.com/>, 2009.

- [239] M. Valenti, B. Bethke, G. Fiore, J. How, and E. Feron. Indoor multi-vehicle flight testbed for fault detection, isolation, and recovery. In *Proceedings of the AIAA Guidance, Navigation and Control Conference (GNC)*, 2006.
- [240] A. Van de Rostyne. The pixelito.
<http://pixelito.reference.be>, 2007.
- [241] J. V. D. Vegte. *Feedback Control System*. Prentice Hall, 3rd edition, 1994.
- [242] S. Vijayakumar and S. Schaal. Locally weighted projection regression : An $O(n)$ algorithm for incremental real time learning in high dimensional spaces. In *Proceedings of the 17th International Conference on Machine Learning*, pages 288–293, 2000.
- [243] J. Walker, S. Garrett, and M. Wilson. Evolving controllers for real robots:a survey of the literature. *Adaptive Behavior*, 11(3):179–203, 2003.
- [244] Z. Wang and A. C. Bovik. Mean squared error: Love it or leave it? - A new look at signal fidelity measures. *IEEE Signal Processing Magazine*, 26(1):98–117, 2009.
- [245] S. L. Waslander, G. M. Hoffmann, J. S. Jang, and C. J. Tomlin. Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.
- [246] R. Watson and J. Pollack. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 702–709. Morgan Kaufmann, 2001.
- [247] R. A. Watson. Embodied evolution: Embodying an evolutionary algorithm in a population of robots. In *Proceedings of the IEEE Congress on Evolutionary Computation, (CEC)*, pages 335–342. IEEE Press, 1999.
- [248] M. F. Weilenmann, U. Christen, and H. P. Geering. Robust helicopter position control at hover. In *Proceedings of the American Control Conference*, volume 3, pages 2491–2495, 1994.
- [249] P. A. Whigham. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, 1995.
- [250] A. P. Wieland. Evolving controls for unstable systems. In *Connectionist Models: Proceedings of the 1990 Summer School*, pages 91–102. Morgan Kaufmann, 1990.
- [251] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.
- [252] P. Wong and M. Zhang. Algebraic simplification of gp programs during evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, volume 1, pages 927–934. ACM Press, New York, NY, USA, 2006.
- [253] H. Xiang and L. Tian. Development of autonomous unmanned helicopter based agricultural remote sensing system. In *Proceedings of the 2006 American Society of Agricultural and Biological Engineers Annual Meeting*, 2006.
- [254] R. Xu, G. K. Venayagamoorthy, and D. C. Wunsch. Modeling of gene regulatory networks with hybrid differential evolution and particle swarm optimization. *Neural Networks*, 20(8):917–927, 2007.

- [255] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.
- [256] Z. Zhigang and L. Tiansheng. GA-based evolutionary identification of model structure for small-scale robot helicopter system. In *Proceedings of the 7th International Workshop on Embedded Systems - Modeling, Technology and Applications*, pages 149–158, 2006.
- [257] A. Zimmerman and J. P. Lynch. A parallel simulated annealing architecture for model updating in wireless sensor networks. *IEEE Sensors Journal*, 9(11):1503–1510, 2009.