Development of a Competition Web Service for Attitude Estimation Algorithms

Sam Osenieks
u4677151

Supervised by Dr. Jochen Trumpf

November 2013

A thesis submitted in part fulfilment of the degree of
Bachelor of Engineering
Department of Engineering
Australian National University

Australian National University

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of the author's knowledge, it contains no material previously published or written by another person, except where due reference is made in the text.

Sam Osenieks
1 November 2013

ABSTRACT

Attitude estimation algorithms can vary greatly, being based around various sensor measurements which relate to the intended application of an algorithm. As such, it can be difficult to find relevant attitude estimation algorithms to benchmark newly developed methods against, or to find an ideal existing solution for a given application.

The goal of this work was to develop a competition web service which would allow researchers and engineers to more easily compare attitude estimation solutions. The system was designed to be robust and secure. It aims to supply a variety of functions and features to aid comparisons, all of which will be built about the concept of a leaderboard where attitude estimation algorithms are ranked against each other for given datasets and quality measures.

This paper focuses on the design of such a system, along with a web proof of concept for some important features.

CONTENTS

List of Figures

List of Tables

Glossary of Terms

Attitude: The orientation of a body with respect to an inertial reference frame.
Attitude Estimation: The process of applying filtered measurements to past results in order to estimate the state of attitude over time.
Backend: The driving components of a web service that the user does not directly interact with.
Competition Web Service: A web service focusing on a ranked competition, used in this thesis specifically to refer to algorithmic competitions centred about a leaderboard.
CSS: The language which defines how HTML looks and generic elements to be included in HTML using it.
Datasets: Files containing measurements or information that can be processed by appropriate algorithms to produce results.
Django: The web development framework used for implementation and considered in design of the thesis project. It is focused on minimising repetition and providing strong bases for common web server functions.
Forms: Web elements used to receive user data and pass it to the web service, and if needed eventually a database.
Frontend: The parts of a web service the user views or interacts with directly.
HTML: The language which defines the content and formatting of static webpages. It can be modified by external functions, but is in itself static.
JavaScript: A language used to create both simple and complicated dynamic elements for webpages. It links with HTML to produce dynamic content.
Models: Django's database definition functions, capable of generating databases from within Django.
Tables: The units of a database, which contain consistent information of the same structure.
URL: A web address.
Views: Django's primary communication functions which facilitate database communication in both directions.
Web Development Framework: A package designed to make the creation of web content easier than manually scripting everything.
Web Service: An online application which provides some useful functionality to its users.
Frontend

Chapter 1 Introduction

Attitude estimation is the process of estimating the orientation of an object with respect to an inertial reference frame. It is an important part of robotics with applications ranging from aerial vehicle control to spacecraft tracking to human pose estimation. The process is widely varied and can take in a selection of many possible input measurements, such as gyroscope angular rate measurements or point motion tracking values, and apply various filtering techniques, which can be based on methods such as the Extended Kalman Filter such as the MARG Extended Kalman Filter considered in Section 2.1.1 or particle filtering such as the TAM Quaternion Particle Filter considered in Section 2.1.3, to create output estimates of various formats. A considerable amount of research goes into developing attitude estimation algorithms for different purposes, and into discovering which approach best suits a certain application [1] [2].

Because of the high variation between different attitude estimation algorithms, and the subsequent difficulty of comparing new or existing algorithms to other established solutions, the community of researchers and engineers would benefit from an efficient way in which to find algorithms well suited to a specific purpose and benchmark their own algorithms against existing work.

Algorithmic competition web services are a platform often used to facilitate a certain level of information sharing between authors of advanced algorithms and engineers working with such algorithms. They can publicise high quality results and make comparison of results easier for anyone in the field. These can take the form of prize-driven contests, such as the ChaLearn Gesture Challenge considered in Section 2.3.1, or simply platforms allowing free comparison of different results, as the NIPS Feature Selection Challenge considered in Section 2.3.2 has become. They make use of pre-recorded datasets as an input to user-made algorithms to rank and compare algorithms according to desirable quality measures.

The creation of such a competition web service focused on the highly diverse area of attitude estimation algorithms has potential to greatly simplify the process of comparing and selecting algorithms. As such, the overall aim of the project is to design and implement one. Creating a platform for simplified comparison of complex and highly diverse attitude estimation algorithms catering to various applications aims to benefit anyone working on aerial robotics projects or in related areas. Such a service could be used to efficiently benchmark new algorithms against existing solutions, or to select an algorithm which performs well in a certain system.

1.1 PROJECT DESCRIPTION

The project aims to develop a modern web service. Developing interactive web applications involves many layers of interaction which can vary depending on the complexity of the application and what is required of it. This project will focus on a web service consisting of a frontend web page which a user can interact with, a database backend for storing content associated with the application, a platform to facilitate interaction between the frontend and backend as well as performing complex functions, and additional static functions and layouts attached to areas of the web page.

Selection of tools to aid web development is within the scope of the project. An efficient and widely used way to develop web content is through the use of web development frameworks. These bind together all the previously mentioned elements of a web application and reduce both repetition and time required to implement basic functionality. The wide variety of available web development frameworks necessitates a strict selection process. In Section 3.2 a formal benchmarking of popular frameworks is completed to rank and select frameworks well suited to the project. Other tools such as the development environment and database backend which are considered in Section 3.2 are chosen less rigorously due to the limited available options and clear differences present between the options, as well as the time constraints present in the project.

The central system will focus on the interaction between attitude estimation algorithms, aerial vehicle datasets, and measures for results. Each of these elements will have varied content which determines which elements are compatible with each other, and as such must interact and check compatibility in a non-trivial way. The service aims to allow algorithms and datasets to be easily added by users and stored in the database backend, as well as to facilitate user-driven comparison of compatible algorithms and datasets. An existing module created as part of a previous student project is used to facilitate data extraction and comparison is built on by the web service. The leaderboards display for datasets themselves is also a complex system, as there is no one definitive attribute to assess results and information must be presented in a useful way.

Additional elements of the design include security and user control. As the web service will interact with sensitive intellectual property, security of the uploaded data is a concern of the project. This depends on user account control also, and these points will be assessed by a preliminary design of user-group based control in Section 4.6.

The system design is the core focus of this thesis project, with implementation of a base prototype also within the scope. The prototype will primarily serve as a proof of concept of designed features, and as such extensions such as web hosting and exhaustive testing will not be considered.

1.2 THESIS OUTLINE

The thesis is divided into three major chapters which cover the content of the project, as well as the introduction in Chapter 1 and conclusion and future work sections of Chapter 5.

Chapter 2 provides a more detailed context for the project including relevant background information on attitude estimation, web services, and competition web services. This includes a detailed literature review of each topic covering distinct examples to demonstrate the nature of existing products and to provide orientation for the reader. It also describes how the thesis problem relates to these areas.

Chapter 3 details the design methodology and preliminary concepts of design for the system, including analysing the given problem specification, formulating required subsystems for the overall design, and the scope of work determined for the project. It also includes a formal benchmarking selection process for web development frameworks and the reasoning behind selected supporting systems including the base development environment, the database backend, and the asynchronous operation package for the project. Finally, the methodology used for detailed design is investigated.

Chapter 4 covers the design and partial implementation of central systems within the thesis project. This includes a high level overview of the system and its requirements, and then goes on to consider the database backend, the web frontend, the interaction with the attitude estimation package, and other elements such as user control and the compatibility testing system. Additionally there is some consideration of testing, and the security risks and mitigation associated with subsystems.

Finally Chapter 5 presents the results of the project, and future work which could be done on the system in order to take it from a proof of concept to a fully functional competition web service.

Chapter 2 Background

This chapter provides a detailed description of the three central elements of the thesis project and a literature review of each. Section 2.1 explains attitude, attitude estimation, the complexities inherent in this process, and how they relates to the thesis project. Section 2.2 covers web services, how they are made, and the subsystems necessary to create a complex web service. Section 2.3 explains the fundamentals of competition web services, their benefits to the development and demonstration of algorithms, and their general structure. Literature reviews follow each of these sections.

2.1 ATTITUDE ESTIMATION

Attitude estimation concerns converting sensor measurements into information useful for controlling or studying an aircraft. The attitude of an aircraft is its orientation with respect to some inertial reference frame, usually set to be attached to the ground. It is most commonly represented as roll, pitch, and yaw; three angular measurements taken about three axes attached to the aircraft. There are many alternative representations for attitude such as quaternions, rotational matrices, and Rodrigues parameters [1]. These representations can often be converted to one another easily, however singularities exist in certain representations such as 3-angle representations [3]. Because of such singularities and the vastly different available representations, algorithms to calculate attitude are capable of having highly varied output formats.

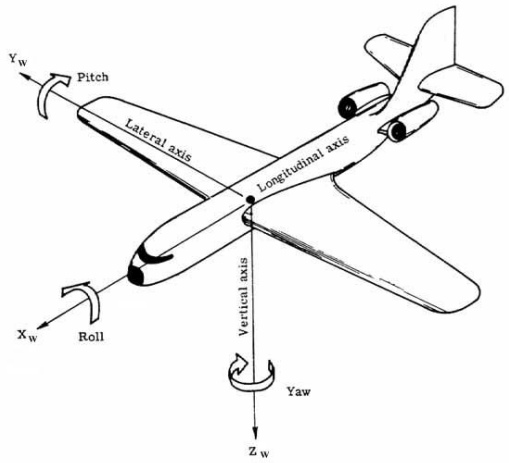

Figure 1: Roll, pitch, and yaw illustrated for an aeroplane.

It is desirable to be able to know the attitude of an aircraft in real time, as this allows for control over its flight and knowledge of its journey. Knowledge of attitude can be useful for a wide range of applications; such as aircraft control on Earth, spacecraft tracking, and even

tracking the movements of a person's limbs for 3D reconstruction [4]. In order to achieve high quality control it is necessary to have highly accurate and smooth attitude measurements, as small errors can lead to overreactions in certain control systems and thus create further error in flight[5]. Individual sensors can be prone to inaccuracy. For example, over time mechanical gyroscopes tend to drift away from accurate values [6], and various other sensors such as cameras and accelerometers are insufficient to determine attitude on their own. This is where attitude estimation comes in.

It is difficult to accurately estimate the attitude of a body in continuous three dimensional motion, as individual sensors mounted on an aerial vehicle can only take somewhat accurate measurements at discrete time steps. Additionally, aerial motion is in itself a complex process due to the multitude of forces acting on an aircraft, as well as the unpredictable nature of the wind acting on an aircraft on Earth. Instead of relying exclusively on one or two sensors, measurements can be combined to form a most likely state for the attitude of the aircraft. As such, attitude estimation techniques apply complex algorithms to some set of measured values in order to estimate the most probable state based on the information available.

Attitude estimation algorithms take the forms of filters, which take in measurements and produce estimates for attitude and possibly other parameters based on the input and past estimated states. These filter use a dynamic model to predict the state at a given point based on propagated past states and corrections to the state which are based on sensor measurements [1]. It should be noted that this is distinct from attitude determination, where only the currently available sensor information is used and previous states do not influence the result [7]. The inputs and outputs of attitude estimation algorithms can vary enormously, not just from differing sensor types, but also from the format of the data they produce. For example some datasets could have gyroscope orientation data in a three-angle representation that leaves them susceptible to singular points [8], and some filtering algorithms may be unable to deal with this.

The mechanics of attitude estimation itself is a relatively minor part of this project, and the most important piece of background knowledge is what form the data required for attitude estimation takes. For the competition web service, the inputs should be as robust and extensible as possible. In general a dataset will consist of sensor readings, ground truth values representing the approximately correct attitude of the vehicle measured from ground, and also possibly certain numerical values such as covariances for sensor noise processes or ground truth values [1]. Additionally, accuracy measures will be required to work on several output formats, such as quaternions or rotational matrices [1].

In order to better understand how attitude estimation algorithms can vary, the subsections below look at examples of filters and how they differ.

2.1.1 MARG Extended Kalman Filter

The Extended Kalman Filter (EKF) is a commonly used probability-based filter that adds to the Kalman Filter, a probabilistic filter capable of making statistically optimal estimates of states for linear systems, in order to address non-linear components [9]. It can be used to predict the states of systems when either the state of the system or the observations associated with predicting the state are non-linear, which makes it useful for real-world problems such as attitude estimation.

The real-time MARG EKF outlined in *An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors* [10] applies the EKF to measurements taken from sensors composed of three-dimensional magnetometers, measuring the Earth's magnetic field relative to the aircraft; angular rate sensors; and accelerometers. The angular measurements are converted into quaternion form before filtering in order to avoid singularities.

Using these sensor inputs, a quaternion convergence algorithm is applied and the designed EKF is applied to produce both angular rates and quaternion angular components.

2.1.2 Star-tracking Based Extended QUEST Filter

Quaternion estimation (QUEST) filters use a matrix minimisation problem to minimise error in state estimation. They can take many forms and apply various matrix-based methods [1].

Extended QUEST filters can be used as an alternative to EKF filters for attitude estimation. Although generally suited only for simpler problems, they provide advantages in that the estimated state is the solution to an eigenvalue problem and does not depend heavily on having a first guess of attitude. This means that it is will not diverge [11].

The extended QUEST filter example presented in *Extended QUEST Attitude Determination Filtering* [11] takes in two sensor measurements; vector angular rate measurements from a gyroscope and additional measurements from a star tracking system. Combining these using extended QUEST, it arrives at a quaternion attitude estimate.

This example demonstrates a case where somewhat accurate results can be arrived upon using only two sensors. The angular rate sensors are used to determine the attitude's evolution over time, while convergence to true values is supported by the star tracker measurements which can be used to affect estimates independent of their past readings.

2.1.3 TAM Quaternion Particle Filter

Particle filtering is generally more computationally complex than other filtering methods because rather than applying a mathematical model to directly find an estimate, its estimate is composed of often thousands of individual possibilities each drawn from the same model, represented as particles [12]. The estimate is obtained by considering the positions of the particles as a probability density. There are an endless number of possible filters that can be derived using this methodology.

A particular particle filter for spacecraft applications is considered in Chapter 4 of *Sequential Monte Carlo Methods for Spacecraft Attitude and Angular Rate Estimation from Vector Observations* [2]. This filter is a unique design taking in gyroscope measurements and three-dimensional magnetic field (TAM) readings. Applying several algorithms yields a quaternion attitude estimate.

Simulations in *Section 4.5* of the paper [2] suggest that the filter performs better than alternate EKF solutions in the case considered, however not many filters are available for comparison.

This section demonstrates the difficulty of finding algorithms to compare one's work to. The comparisons simulated require the author to manually find datasets and implement alternative algorithms, and the difficulty inherent in doing this may be a factor in limiting his comparison to two EKF methods.

Overall this section illustrates the need to compare attitude algorithms. With tools to meaningfully compare a wider array of other solutions, more comprehensive comparisons could be drawn.

2.2 WEB SERVICES

The way in which web content is made has evolved since the creation of the world wide web in 1989 [13]. Over time the requirements of internet services have become more and more demanding as the internet grew and became more complex. In order to develop feature-rich, robust web applications, programmers have strived to create efficient, easy-to-use systems that can aid their work on the web.

Modern web applications are developed using sets of tools that can achieve common functions without the author being required to manually code every element. These come in a huge variety, with the more recent development tools supporting an array of inbuilt security systems, error-handling, and better communication with external systems. These toolsets are called *web development frameworks*, and are generally freely accessible and based around common programming languages.

There are an abundance of potential frameworks from which to create content, and while some excel in certain situations, the majority aim to generally increase the speed at which a programmer can create web content by simplifying the creation of core web application functions to aid database interaction, code reuse, and content generation. Choosing an ideal platform for a task can be difficult when presented with the dozens of similar frameworks that vary from still popular systems originally created many years ago and used to create thousands of webpages, to up-and-coming frameworks promising increased simplicity and better functionality. Web applications typically require a great deal of repetition and standard functions so a framework aiming to minimise the time required to implement these is essential for efficient programming.

The structure of web applications vary by their function. A simple web page can be composed of static data along with files that describe how to format and present the data to a viewer. This functionality is typically handled by two scripting languages: HTML (Hypertext Markup Language) which describes the content and the general sequence in which it appears, and CSS (Cascading Style Sheets) which describes how the information presented in HTML is visually displayed [14]. However, many web services that are used to store and display non-static data require the use of additional systems to support this functionality. Data storage is typically handled by a database system such as MySQL, which can be interacted with by code associated with the web service to store and retrieve data for the user as required. Additionally, more complex interactions between the user, the web page, and the database can be facilitated through supporting code attached to the HTML, usually written in JavaScript [15]. The information that relates to what is displayed to the user and the corresponding interface can be referred to as the *frontend* of the web service, while the database structure and supporting code can be referred to as the *backend* [16]. The management and control of these elements can be done through the web development framework, the role of which is to bind them together and simplify working with them.

Various elements of modern web services are illustrated by examining relevant examples in the following subsections.

2.2.1 HotScripts

HotScripts [17] is a website offering the distribution of scripts for a wide array of web development applications. The website is structured around a large database of its available scripts, which can be searched and sorted by their properties. The user can find listings of the most popular products, or look through databases of scripts for a particular language and application.

Each script within the database has many associated properties, such as a general description, a set of supporting screenshots, and statistics like views and average user rating [18]. Additionally there is linked information that would likely be stored in separate *tables* such as user reviews.

A website like this also requires some specialised security elements. As many of the available scripts cost money, it would be essential to ensure that only people who have purchased scripts can access them, and that there are no exploitable methods available to bypass this. User account security would also be a major concern.

The website is written using the popular PHP framework, CakePHP, and the database backend is controlled using MySQL. This illustrates that frameworks can be used to produce even web applications that require strong security and database interactivity.

2.2.2 Bitbucket

The code repository service Bitbucket [19] supports collaborative version control for large programming projects. It has a great deal of users and handles sensitive information, so it requires strict user control and security. It allows users to form groups together and control what other users can see and do within their repositories. This is handled with a user-group based approach.

The repositories are controlled by the owner and a set of administrators who determine the rights of users within the user group. Users have associated permission levels, with the owner who has the unique power to create and delete repositories, the administrator users who have the power to add other users to their group and change their rights, people who can read the data, and people who can write data to the repository [20]. These user groups and permission levels can be unique for each repository.

Bitbucket was created using the Python framework Django, and is written almost entirely in it [21]. One of the administrators mentions that Django eliminated almost all of the repetitive work of creating the website by providing users with the tools to create simple functions easily.

2.2.3 Dell Shop

The shopping section of the Dell website [22] provides an example of an implementation of complex database interaction using a web frontend. Similar to HotScripts, this section of the website is heavily influenced by the underlying database structure. The data elements are electronics and computers with various associated data, yet in addition to simply inspecting the data, one can also perform comparisons across multiple products.

The product comparison tool can take in selected data elements, for example laptop computers, and print out a comparison table[23]. In addition, the user can easily add other computers to the comparison by dragging and dropping them onto the list, or remove

elements entirely. This does not require reloading the page, instead changing inputs into a querying of the database backend that is then updated in real time.

The comparison system also integrates seamlessly with the rest of the web service, allowing users to revisit and add to the comparison while viewing the database of products.

2.3 COMPETITION WEB SERVICES

It is common that complex algorithms with real-world applications can have their merit assessed by running them on data only, independent of any physical testing environment. This adds a certain degree of freedom to the creation and improvement of such algorithms, as no hardware needs to be purchased in order to develop them if public datasets are available.

As anyone can work on such algorithms anywhere in the world, it is often desirable to create a platform to support cooperation and keep researchers informed. One method to encourage comparison of developing algorithms and increase publicity solutions is with a *competition web service*. This usually takes the form of a website where users can upload their algorithms or results and be compared with other users on a leaderboard. Competition web services can provide a central hub for people interested in certain types of algorithms and can present information on their quality and function in a concise way. Additionally, the users of such websites can in some cases interact by uploading their own datasets which can then be checked against the available algorithms to provide them with an ideal solution suited to their needs.

Such a system can allow a direct comparison of algorithms across many different datasets reflecting different desirable qualities. For example, a researcher may wish to see which algorithms work well for their particular hardware application, and a competition web service would allow them to see which algorithms are suitable for them. Additionally such web services help authors improve their algorithms by comparing functionality to existing solutions. If the resulting solution is of sufficient quality it will also become more publicised within the competition community.

To highlight some of the common features of competition web services, several examples are investigated in the following subsections.

2.3.1 ChaLearn Gesture Challenge

The ChaLearn 2011/2012 one-shot-learning challenge for hand gestures [24] provides an example of a competition where datasets are clear in their structure and algorithms clear in their function. This challenge is about an algorithm to learn enough from seeing only one example for each of large set of unique gestures that it can correctly identify these gestures in the future when processing different videos of them.

The challenge involves datasets composed of colour videos and depth videos taken via a Microsoft Kinect. The provided datasets include a set of training videos taking the form of many gestures with associated truth values for every given gesture, a set of self-assessment videos which have provided only one example gesture with associated truth value per unique gesture, and final assessment videos set which are used for leaderboard assessment, again with only one example gesture with a truth value identified per unique gesture [25]. There are three distinct datasets in this problem because it is somewhat original and computationally complex, benefiting from an extra set of data where everything is known so that algorithms can be trained and tested in various ways during preliminary development.

The algorithms are run by the user on the supplied datasets offline [25] and results are uploaded to an external leaderboard website for comparison [26]. As this competition was timed and had prizes, rather than running indefinitely, the leaderboard was published all at once and is not dynamically updated.

2.3.2 NIPS Feature Selection Challenge

The Neural Information Processing Systems Conference ran a competition of data feature selection algorithms as part of a workshop on feature extraction [27]. A description of the workshop and associated competition states that "The purpose of the workshop is to bring together researchers of various application domains to share techniques and methods" [28].

This particular type of algorithm aims to find patterns within a large set of numerical data in order to break it down into a representation of its features that can provide insight into the makeup of the data [29]. Feature selection algorithms can have many goals, and the data that is required to be analysed can vary greatly as well, so the challenge provides five datasets which attempt to emphasise different properties. The datasets vary in size, sparsity, and the number of features involved [30]. Each dataset is further divided into training data for using however the user sees fit, validation data for comparing solution quality before the end of the competition, and testing data for comparing final solutions publicly after the competition ends. In order to assist users, the instructions specify the format of the data and provide a sample program to read it into algorithms and write output in a format useful for the competition.

When a user uploads a solution, they are supplying output data from their algorithm that is generated on their own computer using the provided datasets. The user has the option to upload either development submissions which can include results attained from any combination of the five datasets and can be submitted anonymously, or to upload final solutions which have to contain results for all datasets and have the user's name attached publicly. The competition took into account each user's five most recent final submissions for the purpose of ranking. Although it has finished, the competition still allows users to upload

their results. This is simply for the purpose of comparison and these submissions are clearly marked as post-competition.

Rank	Method	Balanced Error			Area Under Curve		
		Train	Valid	Test	Train	Valid	Test
1	Optimized svc	0.0096	0.0213	0.0606	0.9978	0.9867	0.9738
2	Optimized svc/TP	0.0094	0.0213	0.0607	0.9978	0.9867	0.9738
3	Optimized svc/TP wcv	0.0094	0.0213	0.0607	0.9978	0.9867	0.9738
4	Normalization, Signal 2 Noise (Feature Ranking), S	0.0094	0.0213	0.0607	0.9978	0.9867	0.9738
5	update	0.0159	0.0176	0.0614	0.9736	0.9703	0.9429
6	final_the2nd	0.0111	0.0120	0.0622	0.9963	0.9965	0.9680
7	BayesNN-DFT-combo+v	0.0037	0.0034	0.0648	0.9999	0.9997	0.9720
8	Aggregate 2	0.0208	0.0238	0.0652	0.9911	0.9937	0.9718
9	final_submission_2	0.0211	0.0221	0.0653	0.9930	0.9930	0.9660
10	final_submission	0.0211	0.0221	0.0661	0.9930	0.9930	0.9660
11	mixed	0.0025	0.0007	0.0675	0.9982	0.9993	0.9290
12	normalize-pacbank-svm-cv	0.0069	0.0203	0.0681	0.9979	0.9868	0.9702
13	BayesNN-DFT-combo	0.0039	0.0526	0.0684	0.9998	0.9827	0.9722
14	BayesNN-DFT-combo	0.0037	0.0523	0.0687	0.9998	0.9826	0.9721
15	Aggregate 1	0.0188	0.0330	0.0690	0.9922	0.9870	0.9722

Figure 2: Several columns of the NIPS challenge leaderboard, showing evaluation metrics and ranking.

The leaderboard is broken down into a global leaderboard for combined measures over all five datasets, as well as individual leaderboards for each dataset. Each dataset result is broken down into three measures including the primary balanced error rate measure, an area under curve measure, as well as statistics about the fraction of features and probes used by the solution. These results are further subdivided by sub-datasets, with the testing dataset being the official measure of result quality. By default the leaderboards are ranked by lowest balanced error rate for testing datasets. The leaderboard can be sorted as desired by clicking the desired column heading, and clicking result names leads to a description of the method and a summary of the results [31].

The NIPS competition has several security features in place to ensure that submissions are done as they desire. In order to avoid overloading the web server by exploiting its most resource intensive components, there is a limit of 5 submissions per day from one origin [30]. Additionally, there are measures in place to discourage cheating within the competition. It is stated that the organisers will, after the competition closes, run the results of final leaderboard submissions through supporting algorithms to detect deviations between the results uploaded and the accuracy achievable with the set of features associated with the results [30]. There is also a requirement that the user supply their real name to use the web service, and publicise it for graded submissions [30].

2.4 Problem Specification

The aim of the project is to design and build a competition web service to simplify comparison of attitude estimation algorithms and promote information sharing within the aerial robotics community. The project will aim to be robust such that the highly varied inputs and outputs of attitude estimation algorithms can be supported. In addition it will utilize web development frameworks to make constructing the web service efficient and bind together the

web frontend with the database backend. A structure similar to existing competition websites is to be aimed for, where results are uploaded to the website and compared on a central leaderboard. High level analysis of the problem including scope is given in more detail in Section 3.1.

Chapter 3 Design Methodology

In this chapter the project aims are analysed, tools for the project are selected, and the methodology for detailed design is considered. Section 3.1 considers the requirements of the project by analysing the given problem and provides in detail the scope of the project. Section 3.2 explains the methodology and results of selecting resources for web service development. Section 3.2.3 considers the methodology for the detailed design process.

3.1 Problem Analysis and Scope

Given the problem of developing a robust competition web service for attitude estimation, the first step is to understand the problem and its complexities. One of the most important considerations before thinking in detail about the project is the presence of existing code written by another student which already handles some of the functions that would be required of the web service.

The competition web service fundamentally aims to supply information about the interactions of algorithms and datasets, and the actual production of a system to attain information about this is outside of the scope of the project. Another student, Conor Horgan, has already produced the fundamental components of a system to run simulations of algorithms on data. This system will be called by the web service to deal with these functions. It includes algorithm-dataset compatibility tests, strict formats for these inputs, and several other functions which can be used to add functionality to the thesis project. The details of this system and how it relates to the project are considered in Section 4.4. With this supporting attitude estimation package in mind, the first element of the problem to be considered was what the overall system would end up being made of, and what should go into it.

The system would be required to store user data, including datasets, actual algorithms, and the results to be published on the leaderboard. Based on the consideration of complex web services in Section 2.2 and these data storage requirements, it was clear that the system would require a non-trivial database backend. This database would have to be used to generate content for the web frontend, and due to the nature of Conor's simulation system there would have to be some processing of the data, such as files of output points being converted into quantitative measures suitable for the leaderboard. This functionality would be expressed in modules that interact with both the database backend and the web frontend.

The web frontend for this system would be required to allow users to upload their own data to the system and to perform some processing of the data so that it can be displayed in meaningful ways. As the project is time-limited and technical in nature, the textual and visual

content of the website will not be focused on. In the simplest cases of information processing, it may just have to read data from the database and order it by name or date, such as displaying a list of available datasets. Basic functionality of the web frontend such as this is considered in Section 4.3. However, as the web service would be required to potentially store a large amount of datasets and algorithms, it would be necessary to have a system for sorting these such that only results relevant to the user's needs are presented. For example, someone wishing to compare their algorithm that takes in specific inputs may wish to compare to only other algorithms compatible with the same inputs. The web service could ideally be queried to present algorithms with similar inputs and also datasets which both algorithms would be able to run on. Performing a complex query like this would require the user to make multiple selections, and making such a process easy for the user goes beyond what can be achieved by simply clicking on links to other pages. As such, asynchronous database communication would be required in this situation. This advanced functionality is considered in Section 4.5.

In addition to these tasks which relate primarily to the attitude estimation competition, the product must also have some elements common to most advanced user-driven web services. These include primarily a secure user system and general security. The challenge web services investigated in Section 2.3 generally had somewhat basic user systems, focusing on just making users log in to interact with their services. However, in the case of the project the system would likely be used primarily for research and comparison rather than short-term prize-driven competitions. As such, a more robust user system will be considered. This system will aim to allow for team-based contributions and sharing within small communities, similar to the user-group based code repository web service Bitbucket investigated in Subsection 2.2.2. With a sufficiently advanced user-group driven system it would be possible for researchers to choose whether they share information with a small group, exclusive other parties, or the whole user-base. The same customisability could in turn be applied to modification and control of content. This goal also relates heavily to security, as such a system if well implemented can give users much more control over their content and its security.

Security must also be handled by ensuring that the system as a whole lacks exploitable weaknesses, especially ones that could be used to extract or affect private data. Secondarily, the system should also be robust to attempts to overload its processing with repeated commands or requests. Although security flaws can mostly avoided by choosing well-established web design concepts and keeping the system simple, certain complexities within it and within web design in general make this a challenging consideration. As such, it will be explored where applicable, but some of its complexities lie beyond the scope of the project.

3.2 RESOURCE SELECTION

Before construction of the web service could begin, appropriate supporting software had to be chosen. Due to time limitations and the scale of the project, formal benchmarking was only applied to the choice of the web development framework as show in Subsection 3.2.2. This

was by far the most critical choice, which would determine a lot of the other systems to be used and how the system would be developed in general. Some choices such as the development environment and database backend required only minor comparisons, while selecting an appropriate web development framework from a pool of dozens of viable options required more thought.

3.2.1 Development Software Package Selection

Early in the project the decision was made to use some form of the well-established LAMP (generally Linux, Apache, MySQL, PHP) software bundle to create the prototype. This software bundle has the advantage of being free, open source, and very well documented. It has been heavily used by web developers for years, and the majority of web services and servers make use of some form of it [32]. The bundle can have various similar systems substituted into it, but the majority of variations focus on similarly robust and free open source software [33].

The chosen Linux system was Debian 7.1. This system was selected because it is very focused on stability and the elimination of bugs from its modules by including only verified versions of software which have been released for sufficiently long to reveal any flaws [34]. This property makes it popular for web development, where stability is crucial [35]. Between the focus on stability and popularity for web development it became a clear choice for development operating system.

Similarly, Apache 2 was chosen as the web service hosting software as it is still the most well documented and widely used software package for this purpose [36]. This decision was not analysed heavily as the main focus of the project was development of a prototype, and not long-term hosting. Many

The remaining two software packages, which consisted of the database system and the scripting language (referenced as MySQL and PHP in the most common version of LAMP), presented more complex decisions, as these would affect how the development went heavily, and more options were available. The database system is considered in Subsection 3.2.3 and the scripting language in 3.2.2.

3.2.2 Web Development Framework Benchmarking

The purpose of a web development framework when developing a web service is to supply basic functionality and reduce the effort required to add new content. This is an essential step in web development, as to work from the ground up on a time-limited project is unfeasible and unnecessary. Doing this in a structured way is important, as certain frameworks may lack features essential to the project or be unnecessarily difficult to begin working with [37]. The wide variety of similar web frameworks means that this is not a simple choice. As such, benchmarking each framework against all relevant characteristics was completed to narrow down the choice and select the ideal framework for the project.

For this particular decision several tables of weighted qualities were derived from research and qualitative ideas of the importance of different attributes on product development. Attaining reasonable measures for the performance of complex software systems that are updated and modified frequently required the use of approximation based on opinions expressed in books, on online forums, and as part of various other internet documents. Due to the sheer number of options, this was simplified by only assessing the most popular web development languages and frameworks that can be used to create complex web services, and also by breaking the benchmarking process into levels where each eliminated more results to avoid overcomplicating each comparison.

To begin the process, commonly used base languages for the framework were ranked based on several desirable attributes. Documentation and popularity ranked highly as they directly impact the ease of finding resources, examples, and troubleshooting problems online. The author's familiarity with the language was weighted heavily as well, but not so much as to invalidate any languages I did not know. Additional criteria were chosen based on the context of the web service project at the ANU. The first of these was a bonus score for Python on account of it being the language which the attitude estimation algorithm package associated with this project is written (see Section 4.4 for details); thus being a language the web service must interact with initially, as well as a language other people working within this area are likely to be familiar with. The second criterion was based on the likelihood of ANU students possibly working to extend this project in the future being familiar with the language. Java and Python received high ratings here as they are favoured in the coursework provided at the ANU.

Table 1: Benchmarking scores for selecting the web development programming language.

Language	Popular Frameworks [37]	Popularity for Web Servers [38] [39] [40] [41]	Apparent Documentation [38] [39] [42]	Scripting Language Familiarity	Same as Supporting Code	Ease for Future Students	Total
ASP.NET	ASP.NET MVC	7	7	0	0	0	**35**
CFML	Fusebox	5	6	0	0	0	**27**
Java	JavaServer Faces	6	8	4	0	5	**55**
JavaScript	Backbone.js	6	8	1	0	0	**38**
Perl	Catalyst	8	7	1	0	0	**42**
PHP	CakePHP	10	10	1	0	0	**54**
Python	Django	8	8	3	1	3	**60**
Ruby	Ruby on Rails	8	8	0	0	0	**40**
Weights		**3**	**2**	**4**	**5**	**1**	

It should be noted that many sources were used to determine fair approximations of benchmarking values, and for practical reasons only the most important or informative of these are referenced.

The three highest scoring programming languages were selected to be evaluated in more detail. The top languages were clearly Python, Java and PHP. These margin by which these came out on top suggests that even selecting slightly different weightings or scores would provide the same results.

For the next step, three popular and widely used web frameworks for each language were chosen to be benchmarked against each other. The score associated with the programming language remained, and added to by high level characteristics of the frameworks. It should be noted that these are complex systems and the majority of information on them is from subjective online sources often based on the experiences of individuals. As such, this process is not absolute, but it is adequate for selecting a viable framework.

On this level one of the most important characteristics was chosen to be feature completeness, regarding how many functions can be achieved easily within the framework. Simplicity was chosen as equal to this, measuring how easily learned and applied the framework was considered. Additionally, the community of the framework was considered. This measure is a combination of documentation, forums, examples, and guides. Finally the performances of the frameworks in completing many operations were compared.

Table 2: A high-level benchmarking of important features for popular web development frameworks of selected programming languages.

Language	**Framework** [43]	**Inherited Score**	**Feature Completeness** [37] [44]	**Simplicity** [45]	**Community** [46] [43]	**Speed** [47] [48]	**Total**
PHP [49] [43] [50]	**Zend Framework**	54	4	2	4	1	**92**
	CakePHP	54	4	3	5	1	**99**
	Symphony	54	4	2	3	1	**89**
Python [51] [52] [53] [54] [55] [56]	**Django**	60	5	4	4	3	**114**
	Pylons	60	4	3	3	2	**101**
	web2py	60	5	5	3	1	**111**
Java [57] [58]	**JSF**	55	3	1	2	4	**85**
	Struts 2	55	4	1	3	4	**92**
	GWT	55	2	2	2	4	**85**
	Weights	**1**	**4**	**4**	**3**	**2**	

From this level three frameworks were selected for detailed comparison. Although the three Python frameworks came out on top, due to the relatively low score of Pylons compared the other Python frameworks the third framework was selected as the highest scoring non-Python framework. Thus Django, web2py, and CakePHP continued to the next level of comparison.

For the final comparison the previous characteristics were expanded into several where possible. These were examined in greater detail to select what would seem to be the ideal framework for the basis of the attitude estimation competition web service. Most of the characteristics chosen are self-

explanatory. Maintainability and future proofness were added to represent the likely ease of other people understanding and changing functionality of the product at a later date, and the probability that the framework will continue to be popular and widely used in the future.

Table 3: A multipart table showing the detailed benchmarking of selected frameworks. Many of these values are based partially on information used in the previous step of benchmarking.

	Language				**Features**		
Framework	**Familiarity**	**Simplicity** [59]	**Same Language as Supporting Code**	**Ease for Future Students**	**Database Interactivity** [60] [61]	**Code Execution**	**User Account Management**
CakePHP [62] [63] [64]	1	3	0	2	5	4	4
Django [62] [63] [65] [66]	3	3	1	3	4	3	5
web2py [65] [67]	4	4	1	5	4	3	3
Weights	**2**	**3**	**5**	**1**	**3**	**4**	**2**

Community			**Future**			**Speed**	
Internal Documentation	**Community Forums**	**External Sources**	**Feature Completeness**	**Maintainability**	**Future Proof** [60]	**Operations** [47]	**Total**
4	4	5	5	3	5	1	**132**
4	4	5	5	4	5	3	**146**
4	4	3	4	5	4	1	**135**
3	**4**	**3**	**3**	**3**	**2**	**3**	

In the final stage Django came out of the process quite notably ahead of the other two options. Although web2py appears to be simpler and easier to work with, Django is more established and would seem to be heavier on features, extensions, and supporting resources. This gives it an advantage

when used to create a complicated system, as reflected by the benchmarking values. CakePHP would seem quite similar in many ways to the other two frameworks, except that PHP is a slightly less desirable language for the project as confirmed earlier, and it also seems rather complicated and poor in performance.

With the benchmarking process completed, sensitivity to small changes in weighting was briefly tested and this confirmed that the process was sufficiently robust to consistently result in the same options. Overall, this formal benchmarking process helped to ensure that the development software chosen was adequate to work efficiently on the project within the given time and avoid any major issues in the future of the product, such as the development framework becoming unsupported or outclassed.

3.2.3 Database System Selection

The decision of which database backend system to use was heavily influenced by the selected framework, Django. Similar to the framework selection, it was difficult to find any strong evidence for why one system was vastly superior to the others, however with this decision the choices were much more limited.

Django primarily recommends four supported database systems, which include PostgreSQL, MySQL, Oracle, and SQLite [68]. Although other less popular systems are also supported, these will not be considered due to lower probability of being well documented both in general and in their interaction with Django.

Oracle Database was eliminated from the pool because of the author's lack of experience with it. Some limited experience working with SQL-based languages, combined with their competitive quality and feature set [69], made them a superior choice for prototyping. Additionally, Oracle Database is not strictly free software, and the alternatives are open source so this has potential to make them more accessible.

SQLite was eliminated because it is generally considered to lack the power and robustness of the alternative SQL-based systems. It is well-suited for creating small systems or simple prototypes [70], however a prototype built using SQLite must undergo a non-trivial process if it is to be transferred to a more powerful database engine. For a system making extensive use of databases for user control and data storage, the alternative SQL-based systems have stronger advantages [71].

This left two of the most powerful open source database systems in MySQL and PostgreSQL. These two systems appeared to be quite even in terms of performance and functionality, however MySQL was chosen due to its emphasis on speed and its larger popularity [72]. It is also cited to function well with Django [73], and is used in many examples.

3.2.4 Dynamic Content Systems

The dynamic content of the web service was primarily the system for attaining lists of algorithms or datasets meeting certain compatibility requirements such that they can be compared to an input. Additionally, he leaderboards should be robust to sorting by multiple result quality measures attached to them, such as error and smoothness. The second task is quite a bit simpler, achievable with a simple script which doesn't require any new data from the database. Generally operations like this are completed with JavaScript, which has become a staple of creating simple dynamic functions of web pages [15]. As such little consideration was put into the system for this goal, however the former case of a system which must interact with the database is a more complex problem.

In order to retrieve data from the database without loading a new web page every time, an asynchronous database interaction is required, where the database is queried without any effect on the currently displayed webpage. Technologies that are capable of this generally fall under the category of Ajax (Asynchronous JavaScript and XML) systems [74].

For implementing a system capable of the required asynchronous operations, perhaps the most well documented and established technique is with the use of jQuery's Ajax methods. jQuery is a library for advanced JavaScript functionality which is widely used in complex web applications. It focuses on HTML manipulation and Ajax operations amongst other things [75], which makes it well suited to the particular problem. However, another system implementing Ajax operations specifically made for Django exists [76]. This system promised simplified Ajax which was designed to fit perfectly with Django, however it appeared somewhat limited in its quality of documentation, and due to its niche market lacked the numerous and diverse examples present on internet forums and blogs for jQuery's Ajax.

Due to time constraints, the system implementation was attempted in Dajax, however this ended up failing to provide results in time due to the lacking documentation as discussed below in Section 4.5. It is possible that in this case simplicity is not a sufficient determinant to justify selection on its own, as the lack of widely varied examples and general community understanding hindered the use of the apparently simpler Dajax system. Although jQuery would have required some knowledge of an additional language and library in JavaScript and jQuery, the time taken to implement a system in it could have been less due to the presence of higher quality and more diverse examples and guides. Being able to easily view cases where other programmers have faced similar issues is an invaluable tool for troubleshooting complex systems, and the lack of this held back the asynchronous compatibility checking system's implementation.

3.3 DETAILED DESIGN PROCESS

With resources chosen and after some preliminary high level design as detailed in Section 4.1, the project's next phase began. The design was not complete, but because of detailed resource selection the project was at a stage where software development could begin without being likely to cause any major unforseen issues such as designed components being incompatible with the limitations of their chosen systems. However, the software development systems are not completely unconstrained, and have specific ways in which products must be implemented in them. These factors can play a heavy role in the design process of products, and in order to avoid dead ends in certain development paths while maintaining progress, they can be considered in parallel with design.

As such, the methodology chosen for detailed design was an iterative design and build process wherein the specifics of the implementation in a certain language and system could affect the design of a subsystem. This simultaneous evolving design and implementation

aimed to make it so that as specific features of the software come under use they could affect design, and at the same time high level design would continue to become more specific.

This methodology proved effective, as the specifics of how designs must be implemented in software play a large role in their development. For example, the database system is constrained by exactly what types of data one can put in a field and the interactions between fields. Without learning the functionality of this by attempting implementation some of the design work may have become invalid. As such it is likely that without considering detailed design and implementation as one task that each would have taken more time.

Chapter 4 Design and Implementation

In this chapter the initial high level system design and detailed design of subsystems is considered, along with their implementation where applicable. Section 4.1 considers the function of the system as a whole, the necessary subsystems, and how they interact with each other. Section 4.2 considers the design and implementation of the database backend, which the product is built around. Section 4.3 covers the web frontend which is what users will interact with. Section 4.4 considers the attitude estimation package designed by Conor Horgan, the relevant functions to be used, and its place in the system. Section 4.5 considers asynchronous database communication reliant on scripts within the web frontend. Section 4.6 covers the design of user-group based user rights control. Section 4.7 considers necessary tests for subsystems and high level tests for the system.

4.1 System Design

The overall web service system considered in Section 3.1 is made up of several sub systems. After selecting a basis of underlying software, a high level design of the system became the focus of the project.

The system can be broken down in several ways. As addressed more generally in Section 2.2, the system will have a web frontend which the user interacts with directly, displaying information and passing on requested functions; a database backend, handling data storage and user rights; and supporting code which defines how the web front end is structured, and how the database is interacted with, binding both of these elements together.

The competition service will have several primary use cases. The user can upload datasets and algorithms, download a set of them, run them against each other on their computer to produce results, and upload these results to the leaderboard. It is important to have algorithms stored on the web service as this enables users to verify previous results, and discourages attempts at falsifying results to cheat the leaderboard. For such a complex system where compatibility is unlikely to be at all consistent, it is impossible to merge together results from various datasets, so each dataset must have its own attached leaderboard with results extracted from a generic results table within the database. Users will view the results they require by selecting a dataset which they have run their algorithm on, or which the performance of other algorithms on is of interest to them. Another important function of the central competition system is the ability to find algorithms and datasets comparable with other objects. This is considered as implemented via a dynamic compatibility retrieval system.

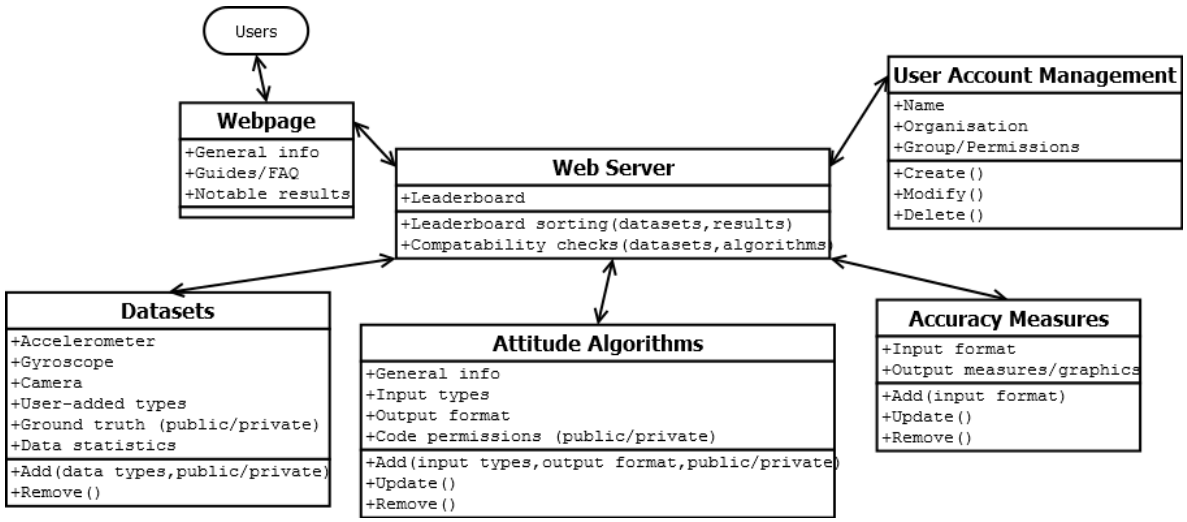

Figure 3: Preliminary high level system block diagram for the competition web service system. Arrows represent strong interaction between components.

The preliminary high level design block diagram in Figure 3 shows the fundamental components of the system, which are elaborated in the remaining sections of Chapter 4. The user interacts with the web service through the webpage displayed in their browser. This is generated by the web service in a combination of HTML, CSS, and JavaScript. This page would be composed of many subpages showing both general information and specific information requested from the database. The webpage should retrieve the information the user requests via links and should also be capable of receiving database information asynchronously, but for the most part is either static or synchronous in its communication as the majority of its content is generated using only the supplied URL. It interacts with the core of the web service, written with Django, to pass requests to and receive data from the database backend. As such, the core web service block facilitates such communication between the webpage and the database backend, including leaderboard information retrieval and asynchronous communication.

The database backend is composed of tables for datasets, algorithms, and also accuracy measures to be used in generating ranks for leaderboards. These tables are not considered in detail on the high level, but would contain generic information about the data as well as actual data such as algorithm code or dataset values. Section 4.2 provides more detail.

Finally, user account management is also a part of the database backend, but is complex enough to be considered separately. It adds user identity and permission data to requests in order to provide security and personalisation.

This system breakdown is centred more about the technology used to create the web service than the function of the web service itself, and as such a slightly modified breakdown is considered in the following sections with complex and relatively independent features having sections on their own.

4.2 DATABASE BACKEND

Certain subsystems were well suited to being designed at early stages, such as the database backend which would store all algorithm, user, and dataset information. The database backend is the most fundamental element of the system, as every other element must interact with it and thus depend on its structure. It was the first subsystem to be designed for this reason.

The design was done by considering data to be stored and use-cases which define how users would need to interact with it. The overall aim of the design was to select a structure that stored information to minimise the complexity of each use-case operation where appropriate. The database structure also aimed to avoid duplicating information, and on making sure each element was independent in that no other element directly determined its value. These considerations aimed to keep the database simple, and thus make implementing functions around it easier.

Table 4: Use-cases for the database.

#	Function	Inputs	Database Actions
1	Create new user	User name, name, affiliation	- Create new user element - Associate with default user group
2	Deactivate user	User name	- Make user unviewable by all - Change ownership of objects if required
3	Change user permissions	User name	- Add user to new group if default - Modify user's group
4	Modify user group	User group name	- Change permissions of group - Ensure all users in this group have permissions updated correctly
5	Upload an object	Object type, object name, in/out format, object data, privacy settings	- Create new object in appropriate table - Give uploading user all permissions for object - Assign uploader as creator of object - If file is a result, run measures on it and send data to leaderboard
6	Deactivate an object	Object name	- Make object unviewable by all - Make results associated with object unviewable by all
7	Update an object	Object name, new object name, in/out format	- Edit parameters of updated object
8	Upload results	Object names, publish to leaderboard boolean, run inputs	- Check compatibility of objects - Check named objects are correct - Publish results to appropriate leaderboard with run-specific inputs attached - Increment usage statistics table

The use-cases shown Table 4 helped to structure the database design, and included some important design decisions. Users would have control over their uploads such as being able to change their properties, but in early versions of the competition system would have to upload new versions as separate files to avoid overcomplicating the prototype. If version information became a desirable feature it would be would be quite trivial to add. Users would also control who could see their data, which is discussed in further detail in Section 4.6. Additionally, this

design emphasises not deleting data. By deactivating algorithms, datasets, results, and users instead of simply deleting them, the integrity of the database is always maintained as interlinking objects are kept. This was decided because making data unviewable by anyone has the same impact as deleting it, and this also prevents accidental removal of information by making everything retrievable by a system administrator if required. It should be noted that functionality for the system administrator to edit database entries like this has been included in the Django web service by use of a generic admin module. Permanent deletion may be a future possibility, but is unnecessary for a first version and has certain issues of potential data loss associated with it.

With these use-cases in mind along with the data that would be required to be stored, a block diagram for the database system was constructed.

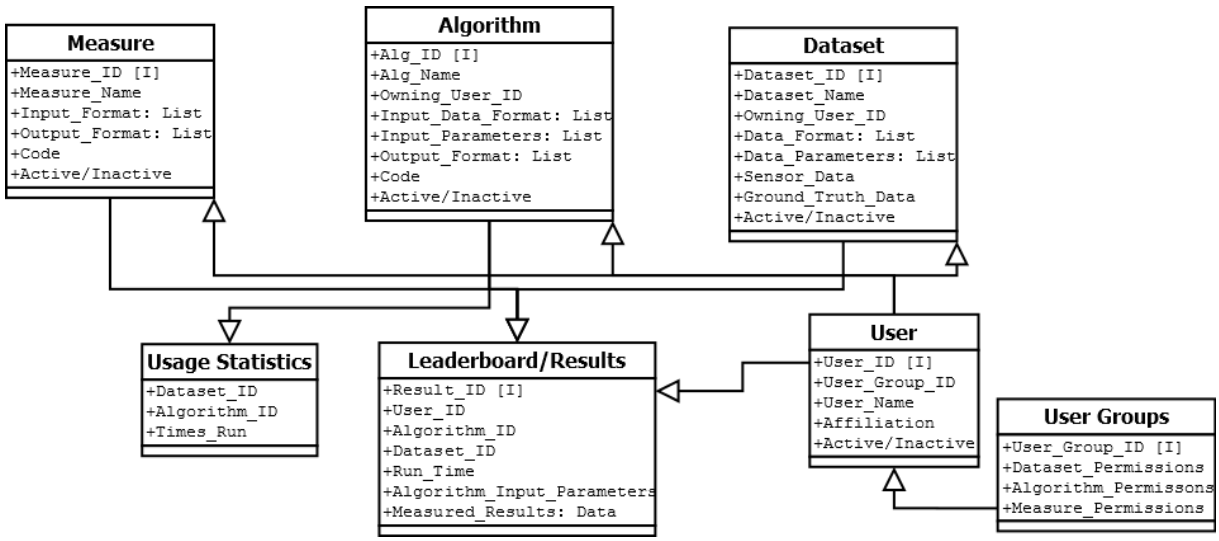

Figure 4: A database design block diagram. The arrows show the modules at the tail of the arrow providing data for the module at their head. The [I] symbol represents the primary keys of data where relevant.

Figure 4 shows the conceptual design used for the database. The leaderboard data is all stored in one table, and will be sorted into individual leaderboards of comparable results by the web service requesting all entries for a specific dataset and ranking them appropriately. Users and user groups are stored in separate tables which will be used to implement a robust user control system as described in Section 4.6.

The user-uploaded data will be primarily stored both as results in the leaderboard, and in the algorithm and datasets tables. This design also provides a table for measures, which are run on result files to produce leaderboard entries. Although room is allowed for a table of measures in the database design, this is probably not something that should be uploadable by users. It is unlikely that having this as a user-uploaded data object would ever be feasible because every time a user added a measure it would be necessary for the web service to run it on every result and update ever leaderboard entry, thus making this a very computationally expensive operation which in practise would likely only be used a few times. Doing so would also

expose a variety of security holes, such as users adding useless or excessive columns to leaderboards, or simply overloading the server. However, if it proves desirable, some possible future version may include this functionality, allowing users to have enormous control over how they rank the results.

The final block is usage statistics, which simply provides information on how many times certain algorithms and datasets have been run. This is primarily interesting to the web service providers, and may assist tracing trends in user action and providing information on web service usage. It could also be extended to provide a measure for the reliability of results for a certain combination, as if many users have uploaded results independently the chances of a result being falsified or tampered with is decreased.

Django uses a special kind of function group to automatically generate database tables which it refers to as models. As these are defined within Django's code, they reduce repetition of database structure, as one only needs to define it within the models file and this implicitly adds a table to the database backend when running the web service [77]. The implementation of the database would mainly involve translating this design to MySQL variables and defining them within a Django model, and this was done for the dataset table as a proof of concept as show in Figure 5.

```
mysql> select * from datasets_dataset;
+----+----------------------+--------------------+---------------------+---------------------+
| id | name                 | comments           | date_add            | date_upd            |
+----+----------------------+--------------------+---------------------+---------------------+
| 15 | Sensor Measurements 1 | An example dataset | 2013-10-24 22:39:38 | 2013-10-24 22:39:38 |
| 16 | Sensor Measurements 2 | Another example    | 2013-10-24 22:43:33 | 2013-10-24 22:43:33 |
+----+----------------------+--------------------+---------------------+---------------------+
2 rows in set (0.00 sec)
```

Figure 5: A basic table generated in MySQL via a Django model, with two entries manually added. Not all fields are shown.

Most of the fields used in this proof of concept are quite simple, such as text and dates, however the decision to use file fields is noteworthy. As all of the files to be stored within the database are plain text and either datasets made of short-term physical measurements, code for filters, or results from datasets; and thus unlikely to contain the order of magnitude of data required to be even megabytes; there is valid reason to store them within the database as actual files. It is possible that in certain circumstances this approach may be less efficient than storing files as text, however it is unlikely to be an issue for small files.

For a complete implementation of the system, the file size of uploads should be limited to some reasonable value. This would prevent malicious users from attempting to overload the server's storage. Additionally, precautions should be taken about users uploading code in their files, such as MySQL or Python scripts, as the files are read by the system. However, relatively simple databases without custom queries such as this implementation are resistant to this in Django [78], and it is unknown any exploitation around this would be possible in a simple system.

4.3 WEB FRONTEND

The web frontend of the system contains all of the information the user sees. Some of this is generated via functions which call on the database backend, however a lot of the information is static; unaffected by the database. There are three important elements of the web frontend: the static text of the webpage, which give users context for all of its functions; viewing stored data, such as the details of algorithms and datasets, as well as leaderboards; and finally forms which allow the user to upload and modify their own data. For the purpose of this design section, asynchronous compatibility checking is considered as a separate subsystem, as it depends not only on the web frontend, but also on server-side scripting. This system is detailed in Section 4.5.

The specifics of the static information is not part of the scope of the prototype, however it would include information such as guides on important topics like how to submit datasets and algorithms, how information privacy is handled, how to interact with the leaderboard, and how rankings are determined. Additional supporting information and frequently asked questions could be added as the need arises.

Additional information would include actual leaderboards generated from the database, as well as information about datasets, algorithms, and their results which are available to the user. In the prototype the ability to retrieve ordered database entries via the webpage was achieved, as shown in Figure 6 below.

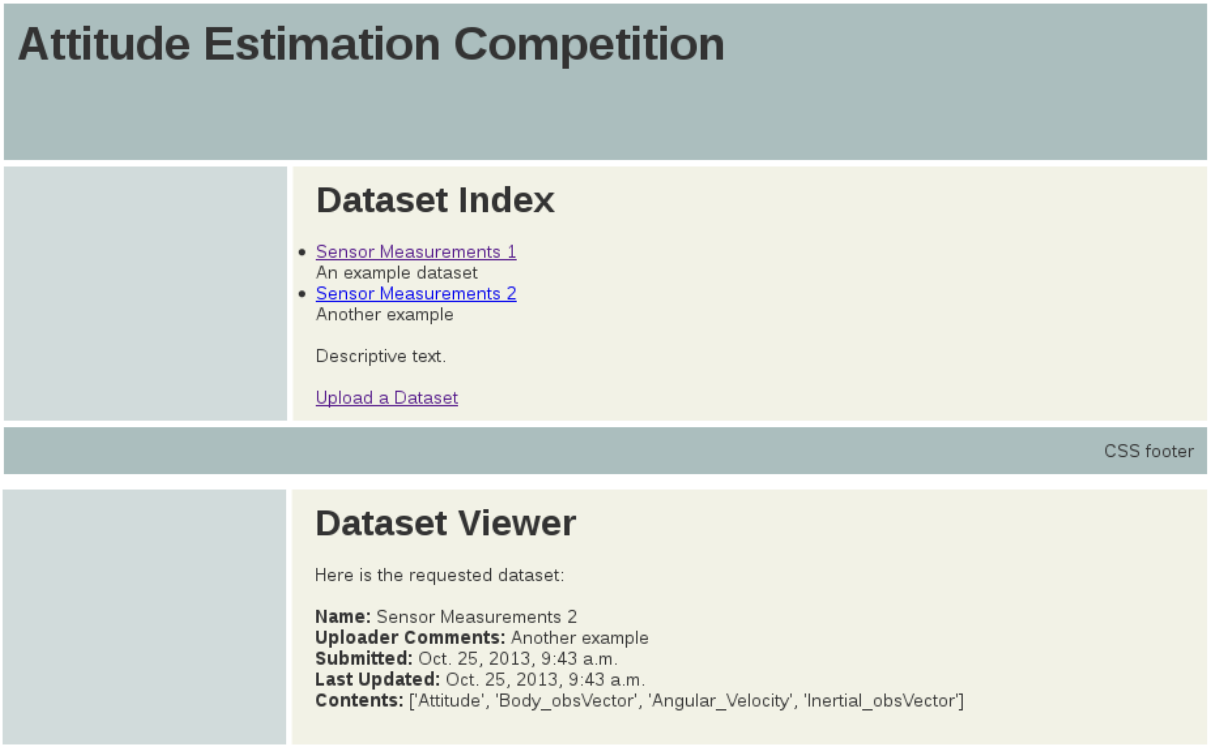

Figure 6: Proof of concept for viewing ordered lists of database entries on the webpage. The top image shows a list of two items, the bottom shows the details of one item viewable by clicking on it.

These entries are generated in HTML by Django. The system is supplied with non-standard HTML files and generic CSS files. The CSS specifies how the page should be styled and gives it graphic elements as well as text properties. The non-standard HTML contains generic HTML with static elements that define text and structure of text, as well as Django-specific tags which associate with dynamic elements. These tags are replaced when Django is called on to retrieve the HTML. Calling the URL associated with these HTML pages will cause the server to execute functions from a set of URL-associated functions which are called *views*. In this specific case, the views associated with the URL return database elements which are then inserted to replace their nametags within the HTML. For example if one navigates to the URL of a specific dataset, the system executes a view functions which returns the dataset's properties, reads in the HTML file of the URL, and then inserts these into the tagged location of the HTML, returning a valid HTML file with dataset properties shown.

This is essentially all of the functionality required for implementing the viewing of database items, perhaps with the addition of download links to retrieve files from the system, which is a trivial problem. However additional functionality could be added to leaderboards, where reordering items is an important feature. This would take the form of a JavaScript file to reorder a table based on the values stored in its columns. The web service can already publish data, it would simply have to format it into a table and call a generic JavaScript script which could likely be downloaded and edited rather than manually programmed, as sorting tables by clicking their headings is a widely used function of web services. Due to time constraints, JavaScript was not added to the prototype, however with some basic knowledge of JavaScript it is unlikely that this particular addition would be difficult since similar functionality is widely implemented on many web sites, becoming a generic problem.

Security is also a concern for this part of the subsystem, as a malicious user could attempt to gain access to private files by impersonating another user, such as by falsifying their login data. In general, Django is robust to such methods [79], and due to the relative simplicity of the base web service and database interaction of this subsystem, it is unlikely to be a major flaw [78]. However, additional testing should be done as advanced web security is a matter with complexities beyond the scope of this project.

The final important feature of the web front end is forms. These are HTML elements which can take in user inputs, and when executed they attempt to send these inputs to the server as requests. Due to the design methodology chosen in Section 3.3, it was possible to exploit a feature of Django in order to make the design simpler. Django is capable of generating forms from database models, which it calls *modelforms*. These can be modified to subtract or add elements to the form, but generally assume the same structure as the database table which they associate with.

By associating a modelform with the table it adds to, the process is simplified greatly. As such, the actual structure of the forms is implied, with some necessary modifications. For example, the forms are not required to have a logged-in user manually add their uploader

name, as this would instead be implied from their identity within the system. This overriding behaviour was added for upload dates as part of a proof of concept.

Additionally, some error handling is required. While Django's modelforms already supply basic error handling to prevent input data from defying database table requirements, such as entering a string or an excessively large number in a field that can only be a small number, additional error handling would be required for uploaded files. The most significant case where Django's implicit error handling for forms is insufficient in the competition web service project would be for uploaded datasets and algorithms, which must conform to a specific format. Just as it is possible to override form elements of individual fields within the modelforms structure, it is possible to override validation procedures, so this would be a required part of the system to verify dataset files, result files, and algorithm files.

The implementation of forms was achieved, with file uploads possible, however the validation of uploaded files was not attempted due to time constraints. This functionality would be handled by calling on validation functions of the attitude estimation package described in Section 4.4, and considering the returned value to validate uploaded files before sending any data to the database. The prototype includes a system for uploading dataset files and associated parameters for the database, which could easily be extended to any other upload to the database. The forms system is shown in Figure 7 below.

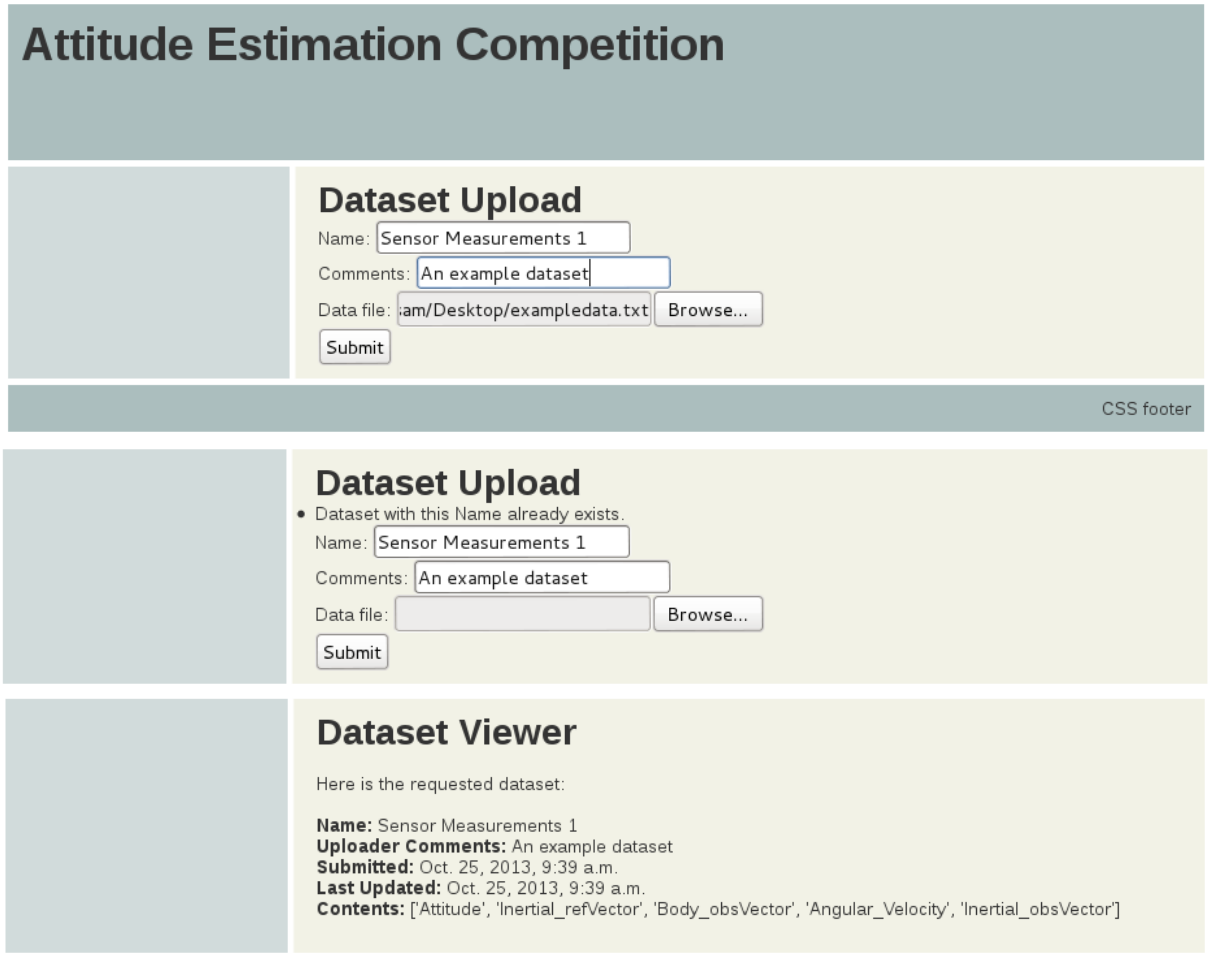

Figure 7: Modelform implementation of HTML forms. The top shows the structure of the form. The middle shows the response to entering an already taken name. Below is the dataset resulting from submitting the form, with overridden fields added by the system.

As shown, a system to extract information from uploaded files was also implemented. It is detailed in Section 4.4.

The only possible complication would be when uploading algorithms. These take the form of actual code, and as such it is possible that these should not be parsed directly by the web service in order to avoid security issues. One solution to this would be to require running a supplied python script locally before uploading an algorithm, which would validate it and extract its compatibility parameters to produce an information file. The information file could then be uploaded as part of the form, along with the algorithm file, thus enabling the system to read compatibility data for algorithms and verify their format without ever directly parsing code.

Security is also a concern for user uploads in general, as a user could attempt brute force attacks such as repeatedly uploading results to be processed by the server, or falsify their identity to upload when they should be unable to. Inbuilt Django security is likely to handle the latter case, however for the former some upper limit to user submissions per unit time should be applied to reduce risk of allowing overloading of the server or adding pointless files to the database.

4.4 ATTITUDE ESTIMATION PACKAGE INTERACTION

The attitude estimation package written by a previous student, Conor Horgan, was used to support out-of-scope functions relating to the specifics of processing and defining attitude estimation algorithms, datasets, and result measures. Although incomplete in some areas at the time of writing, the functionality would eventually exist within this package to perform critical tasks like reading uploaded files, checking files followed correct formatting, extracting data type information, and comparing data type information to determine compatibility. It is written in Python, and thus due to a success of the benchmarking process in Section 3.2.2 easily called from within Django.

The module is useful beyond its functions, as it also provides a strict formatting basis for the data handled by the competition web service system. This formatting information would be supplied to the user, and required in files they uploaded. To enforce this, upload verification as an extension to modelforms would be used, as discussed in Section 4.3. The relevant functions for this verification within the package return errors if given invalid data files, such as *get_DataInfo()* which checks that datasets begin with the correct file tag and returns an error if it is not. These errors could be detected and incorporated into web forms for robustness.

It should also be noted that a central function of this package is to actually run algorithms on datasets to produce results, which although not strictly part of the web service system, means that this package should be supplied to users by the web service in order to allow them to produce results of the correct format and verify their files.

Additional functions that the web service would make use of include functions for comparing the compatibility of datasets, algorithms, and measures; and extracting these compatibility values. If needed, additional extracted data could also be stored in the database to provide further context to uploaded files, however the focus of the web service's use of this package is on compatibility and error handling.

As a proof of concept for this cross-package interaction, the extraction of datasets' compatibility values was implemented. This involved calling the *get_DataInfo()* from *Data_Extractor.py* within Django's view for processing uploaded datasets. The view is run when a user posts a form, and it processes their uploaded file before it is added to the database, parsing it to find the data types within. It returns them as a string which would then have to be parsed again to compare compatibility, as relational databases typically store information as a single field. There may be a better alternative to this, however for a prototype it is sufficient. From this proof of concept comparing compatibility logically follows, as it would simply require making another external function call. However, as some of the associated systems were not implemented this was left out.

This part of the system is as secure as the package it calls. As long as the attitude estimation package is not vulnerable to malicious code insertion or other methods, it should be secure and robust.

4.5 Asynchronous Compatibility Checking

The system for dynamically checking compatibility exists to make comparing algorithms easier for the user. It should be possible for the user to select from a list of database entries some datasets, such as datasets they wish to check the performance of available solutions on, and/or some algorithms, such as algorithms they wish to directly compare, and extract a list of algorithms and/or datasets which are compatible with their selection. The system should also allow easy downloading of these returned lists, as this would be efficient for the user to compare them. Such a system would speed up comparisons, which is the central focus of the system. As such it is an important part of the web service.

Such a system could potentially be implemented without asynchronous communication, however this would likely require a new page to be loaded after each selection is made, thus slowing the process and leading to complicated URLs. As such, asynchronous database communication was chosen to be a central part of this system. By implementing it with this type of communication, no page reloading would be required, as the database would be queried and the visible information updated via a script on the current page.

The system's function is essentially supplying a list of all available algorithms and datasets, taking in a set of selections which could be either datasets or algorithms, finding the common data types between these files, and then processing all available algorithms and datasets to determine which are compatible with the selections. This could be achieved by making use of a function to find common data types and then using the attitude estimation package to compare every other relevant object. However, this has potential to be a computationally expensive operation with sufficient objects within the database, so it is likely that the attitude estimation package may need to be examined to see if any vestigial lines of code could be removed from the relevant functions, thus creating a separate Python script to ensure maximum efficiency and minimise redundant code.

As described in Section 3.2.4, there are two clear methods to implement the required asynchronous communication with the database: using jQuery, a JavaScript package; or using Dajax, a Django add-on offering minimal JavaScript requirements for asynchronous operations. Implementation was attempted in Dajax, however this proved difficult due to the complexity of the system and the lack of highly accessible examples. Documentation was also a problem for the Dajax system, as the official wiki seemed incomplete and poorly maintained. Due to these factors, the attempted implementation failed to produce results in time, and had to cancelled. For a future implementation it is likely that sticking to well-established methods applicable to all web services such as the jQuery implementation would produce better results for a system of this complexity.

Such a system is quite complex in terms of web development, as such security is a difficult issue. Some possible weaknesses include editing the script being run locally on one's browser to extract restricted content, or exploiting the system to repeatedly query the database. As such, security measures such as limiting requests and ensuring authentication is legitimate before allowing file downloads would need to be investigated.

4.6 USER CONTROL

Enforced user signup within the web service allows preventing just anyone from accessing the more vulnerable elements of the system by enforcing manual account creation, and also can be extended to allow for user-group-based permission handling.

The most basic purpose of this system is to provide an extra layer of security for the service, allowing actions to be traceable to specific users or sources, and making accessing the service require some manual data entry to limit the ability of automated systems from interacting with the web service. This can be achieved simply by adding signup and login which are required by the service before a user can access any sensitive systems. For the competition web service, it makes sense to allow anyone to view the leaderboard and general website information, but more sensitive areas such as object downloads, upload, and comparison functions should not be viewable or usable by just anyone.

User-groups are more advanced and typically used in systems requiring strict access rights and containing private data, as is common in complex software systems such as Linux systems [80]. To make the competition system more flexible, a user-group-based system is considered.

The fundamental rights can assume a standard definition as read, allowing a user to simply view information including downloading files; write, allowing them to update object information or modify uploads; and admin, allowing them to deactivate objects, give other users rights, and change user-group permissions. In addition to these rights there is also the lack of rights. These fundamental rights will be assigned to each user group for each file. One possible method to efficiently store user group rights would be to have an entry in a database table for each group for each right equal to or greater than reading associated with the group. A user could then be a part of multiple groups, including some default group which has read access to all public algorithms and datasets, and their rights determined by finding their maximum right for a database object. The user groups would be created and facilitated by the original publishers of datasets and algorithms, as well as other promoted administrators for the groups. This would allow for example a researcher to create a group that has read access to a private algorithm they uploaded, and then add all of their students to it.

Implementation of user groups was not attempted as it is a complicated task with less relevance to the core aims of the project compared to other tasks. Its implementation would

not be simple, but due to the large number of web services and systems making use of this model, examples and documentation would likely be easy to find.

In order to keep this method secure, it would be necessary to make it clear to group administrators exactly who was in a group, as well as to make it clear exactly which rights the group had. This would help remedy cases of users accidentally passing out incorrect rights.

4.7 SYSTEM TESTING

The design and implementation of the software system also needs to consider testing on some level, as this is an important process to prevent errors early on while the system is in development rather than later in the process when they become much harder to detect.

In order to verify the components of the system, an additional consideration would be that of unit testing and more general testing. Unit testing aims to ensure that each function of a module works independently by making attempts to break them with input often composed of boundary cases, and due to the narrow focus of such tests they are quite effective at finding issues in subsystems [81]. Because of their specific nature, there are also many of them for complex systems. More general testing is less effective at finding flaws in subsystems, but provides necessary insight into how the system works as a whole.

Fully implementing a prototype system would reveal many more possible unit tests, however some basic cases including cases for the already implemented systems are shown in Table 5 below.

Table 5: Some examples of unit tests for system components.

Subsystem	Test Case	Expected Result
HTML forms	Upload incorrectly formatted dataset	Fail, file not of required format
	Upload no file	Fail, no file selected
	Upload a valid file with valid data	Accepted, added to database
	Upload form with invalid field...	Fail, informative error message
HTML handling	Enter URL with out of range dataset	Fail, no such object
	Enter URLs of sub pages defined in the Django scripts and some public database objects	All load correctly, display properly formatted information
User control	Enter URL of private dataset while logged in	Fail, access restricted
	Attempt to upload a file while not logged in	Fail, login required
	Modify an object while possessing write privileges	Object modified
CSS formatting	Resize web browser	Web page not overly distorted
Leaderboard	Click a numerical column's heading	Sort table in ascending/descending order according to heading
	Click a result	Information associated with result is displayed
Asynchronous compatibility information retrieval	Select a dataset and search	All compatible, visible algorithms returned
	Select many datasets such that no compatibility can be found	No items found error
Attitude package interaction	Upload a dataset	Compatibility extracted and matches contents of file
	Upload empty file	Fail, file contains no measurements

Due to the complexity of the project and the specificity of unit tests, there would be many more possibilities, however these give a basis the type of possible errors to be investigated and some possible failures of certain subsystems. Many more tests would be required to approach any sort of verification of the product, and testing alone is not sufficient to ensure a lack of errors. The aim of implementing unit tests such as these is simply to attempt to detect some more likely errors prior to late design stages where the system as a whole is more complicated and problems introduced early on become difficult to identify.

Testing of the system as a whole could be done with more general tests such as attempting to use every function normally, attempting to access or use data that one should not be able to, and uploading marginally incorrect datasets, algorithms, or invalid results. Security would be a major concern of this holistic system testing.

Chapter 5 Conclusions and Further Work

This chapter summarises results of the thesis project and considers future work that could be undertaken in order to construct a fully functional system based on the given design specifications.

This thesis covered the design and partial implementation of a competition web service for the comparison of attitude estimation algorithms. The project provides a detailed design as well as an implemented proof of concept for a system which could facilitate increased information sharing and efficient contrasting of algorithms for applications in aerial robotics and other areas. The web service was designed in detail, however additional work remains before a fully functional prototype can be implemented.

Over the course of the project it became clear that web development of the scale of competition web services is a complex and multifaceted process requiring the understanding of many large systems, their interactions, and the application of several widely varied programming languages. With the added complexity of catering to attitude estimation algorithms, the problem was of a scale requiring a well considered approach. Although time constraints restricted the level to which a system could be developed, the systems engineering approach applied well to the design process. Breaking down the problem into work packages and interacting subsystems is a strong approach to any problem of this scale, and was critical to work towards developing a system depending on previously unexplored fields such as attitude estimation and web development.

Overall, the designed system formed a solid basis for future implementation work, complete with a proof of concept that could with several additions be taken towards a prototype. The methodologies of web development in Django were investigated and used to create the skeleton for a web service. Basic concepts of web design such as CSS and HTML were explored and used to create web page templates with minimal repetition which could easily be added to by Django methods. The fundamentals of MySQL database generation and interaction were established using a table for datasets, and file uploads along with basic parsing of uploaded files were implemented. A web frontend complete with methods to inspect, upload, and view a complete list of datasets was implemented by making use of Django's views and modelforms. The attitude estimation package written by Conor Horgan was called via the web interface to extract data from uploaded datasets for use in compatibility comparison, and the functionality of this basic implementation was investigated on Django's local development server.

Although the proof of concept covered many of the fundamentals of the web service system, some subsystem designs could not be implemented due to time constraints and difficulties in

the chosen approach for asynchronous communication. It is possible that with stricter planning and more detailed consideration of asynchronous systems that more time could have been spent on the addition of other systems to the prototype. Rather than sinking time into a this system which proved to be a dead end, it would have been better to leave it and move on to other tasks more readily.

The next major step recommended for the implementation part of the project would be the addition of users and user-groups as described in Section 4.6 to existing code, and the subsequent addition of similar code to what is already implemented to represent the remaining database tables of Section 4.2. The latter would not require much work as the functionality is already present, however user-groups may be somewhat difficult to implement well. At this point with the complete database in place the next steps would be to add supporting code written in JavaScript to allow for sorting operations on leaderboards as described in Section 4.3, and asynchronous comparison of the compatibility of input groups of algorithms and datasets could be attempted again with more knowledge of JavaScript and use of a more standard library in jQuery, as described in Section 4.5. After this, final quality assurance steps like unit testing and overall system testing as considered in Section 4.7 would be necessary to ensure the system works as desired, and to check for possible weaknesses in security.

Given additional work on the practical implementation, the system designed in this project could contribute to the area of attitude estimation by facilitating information sharing and promoting efficient comparison of algorithms. This would aid both the development of attitude estimation algorithms and any engineering projects that make use of them.

Appendix - Django Modules

It should be noted that the CSS style sheet used was based on the work of Keith Donegan. The specific style was taken from *www.code-sucks.com/css layouts/fixed-width-css-layouts/*. This module is included in the appendix because it was edited to add text to HTML pages.

Django also depends on many generic settings files which were not modified. Only files that were modified from a default project are included in the appendix.

The highest level file is *settings.py* which defines which other files are included as well as some decisions about the function of the underlying system. It also contains information used to access the database. The password used in this file is not used anywhere else.

urls.py specifies the URLs that the web service considers when attempting to navigate to pages within it. The symbols specify which general expressions they can take as input.

models.py defines how Django constructs the database in MySQL.

views.py determines the function calls associated with URLs and forms. It replaces tags within Django's HTML files with actual data, and handles the POSTs of forms.

index.html, *detail.html*, and *upload.html* define the three main pages used for the proof of concept. The index lists all datasets, detail shows the specific information of one dataset element, and upload presents a form to add an additional dataset.

style.css defines some constants within the HTML such as how it is rendered and title/footer text.

Inserting all of these files into a generically created Django project on a compatible system with all supporting software installed should be sufficient to recreate the prototype.

SETTINGS.PY: CONFIGURATION FILE

```
## @package server_settings
#  Global settings for the web service.
#
#  Defines where other elements are located and the way in which they are
accessed.

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
```

```
    # ('Your Name', 'your_email@example.com'),
)

MANAGERS = ADMINS

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Using MYSQL
        'NAME': 'sdb',                        # DB name
        'USER': 'sdbuser',
        'PASSWORD': 'alphabet1',
        'HOST': '',                       # Set to empty string for
localhost.
        'PORT': '',                       # Set to empty string for default.
    }
}

# Hosts/domain names that are valid for this site; required if DEBUG is
False
# See https://docs.djangoproject.com/en/1.4/ref/settings/#allowed-hosts
ALLOWED_HOSTS = []

# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# In a Windows environment this must be set to your system time zone.
TIME_ZONE = 'Australia/ACT'

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en-us'

SITE_ID = 1

# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True

# If you set this to False, Django will not format dates, numbers and
# calendars according to the current locale.
USE_L10N = True

# If you set this to False, Django will not use timezone-aware datetimes.
USE_TZ = True

# Absolute filesystem path to the directory that will hold user-uploaded
files.
# Example: "/home/media/media.lawrence.com/media/"
MEDIA_ROOT = ''

# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash.
# Examples: "http://media.lawrence.com/media/", "http://example.com/media/"
MEDIA_URL = ''

# Absolute path to the directory static files should be collected to.
# Don't put anything in this directory yourself; store your static files
# in apps' "static/" subdirectories and in STATICFILES_DIRS.
# Example: "/home/media/media.lawrence.com/static/"
STATIC_ROOT = ''

## URL prefix for static files.
STATIC_URL = '/static/'
```

```
## Additional locations of static files
STATICFILES_DIRS = (
    # Put strings here, like "/home/html/static" or "C:/www/django/static".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    '/home/sam/django/static',
)

# List of finder classes that know how to find static files in
# various locations.
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
#    'django.contrib.staticfiles.finders.DefaultStorageFinder',
)

# Make this unique, and don't share it with anybody.
SECRET_KEY = 'o6yt6rrta-1w^^@@_$+hhwxf4%)o5uf5op_bjp578*9@*f45@)'

# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
#     'django.template.loaders.eggs.Loader',
)

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    # Uncomment the next line for simple clickjacking protection:
    # 'django.middleware.clickjacking.XFrameOptionsMiddleware',
)

ROOT_URLCONF = 'server.urls'

# Python dotted path to the WSGI application used by Django's runserver.
WSGI_APPLICATION = 'server.wsgi.application'

TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or
"C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    '/home/sam/django/templates',
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    'django.contrib.admindocs',

    ## Custom apps to load
    #
    #  See /server/* for apps
```

```
    'datasets'
)

# A sample logging configuration. The only tangible logging
# performed by this configuration is to send an email to
# the site admins on every HTTP 500 error when DEBUG=False.
# See http://docs.djangoproject.com/en/dev/topics/logging for
# more details on how to customize your logging configuration.
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse'
        }
    },
    'handlers': {
        'mail_admins': {
            'level': 'ERROR',
            'filters': ['require_debug_false'],
            'class': 'django.utils.log.AdminEmailHandler'
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
    }
}
```

URLS.PY: URL DEFINITIONS AND INPUTS

```
## @package datasets_urls
#  URL locations and properties for datasets table.
#
#  Defines regular expressions for URLs to access data.

from django.conf.urls import patterns, include, url
# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

#from views import *

urlpatterns = patterns('',
    # Uncomment the admin/doc line below to enable admin documentation:
    url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)

# Added sites, first argument defines URL structure, second is view name
urlpatterns +=  patterns('datasets.views',
    url(r'^datasets/$', 'index'),
    url(r'^datasets/(?P<dataset_id>\d+)/$', 'detail'),
    url(r'^datasets/upload/$', 'upload'),
    url(r'^datasets/upload/edit/(?P<id>\d+)/$', 'edit'),
    url(r'^datasets/uploaded/$', 'uploaded'),
)
```

MODELS.PY: DATABASE STRUCTURE DEFINITION

```
## @package dataset_models
#  The database model used for dataset representation.
#
#  Defines the structure of the dataset table, and includes several
functions for accessing and manipulating this data.

from django.db import models
from django.forms import ModelForm

## The definition for dataset table structure.
#
#  Defines data fields and functions.
class Dataset(models.Model):
    # ID handled externally
    name = models.CharField(max_length=200, unique=True)
    comments = models.CharField(max_length=20000)
    #version = models.IntegerField(max_length=200)
    # Owner ID references a table outside of this app: check
    #owner_id = models.ForeignKey(django.contrib.auth)
    #owner_id = models.ForeignKey(auth_user)
    date_add = models.DateTimeField('Date Added')
    date_upd = models.DateTimeField('Date Updated')
    data_format = models.CharField(max_length=500)
    # No data parameters yet, could be read from file alone
    #sensor_path = models.CharField(max_length=200)
    #truth_path = models.CharField(max_length=200)
    data_file = models.FileField(upload_to='stored_files/', max_length=100)

    ## Unicode printing method.
    def __unicode__(self):
        return self.name

    ## Test method.
    #def test(self):
        #return self.data + ' Printer test'

## Generate input form based on Dataset definition
#
#  Uses ModelForm to generate form basis
class DatasetForm(ModelForm):
    class Meta:
        model = Dataset
        fields = ['name', 'comments','data_file']
```

VIEWS.PY: INPUTS TO REPLACE HTML TAGS IN RENDERED PAGES

```
## @package datasets_views
#  Defines elements to insert into HTML and HTML locations for requested
data.
#
#  Refers to HTML files and STATIC files for structure.

# For basics
from django.http import HttpResponse # May not need this, only for
returning basic html
from django.shortcuts import render_to_response, get_object_or_404
from datasets.models import Dataset
# For forms
from datasets.models import DatasetForm # Possibly move to new file?
from django.shortcuts import render
```

```
from django.http import HttpResponseRedirect
from django.utils import timezone
# For file processing
from conor.Data_Extractor import *

## Index page for datasets
#
#  Lists all datasets
def index(request):
    set_list = Dataset.objects.all().order_by('id')[:5]
    return render_to_response('datasets/index.html', {'set_list':
set_list})

## Details for a requested dataset
#
#  Shows defined info
def detail(request, dataset_id):
    d = get_object_or_404(Dataset, pk=dataset_id)
    return render_to_response('datasets/detail.html', {'dataset': d})

## Upload form for datasets
#
#  For adding a new dataset to the database
def upload(request):
    if request.method == 'POST': # If the form has been submitted...
        form = DatasetForm(request.POST, request.FILES) # A form bound to
the POST data
        if form.is_valid(): # All validation rules pass
            # Process the data in form.cleaned_data
            nameF = form.cleaned_data['name']
            commentsF = form.cleaned_data['comments']

            date_addF = timezone.now()
            date_updF = timezone.now()

            # File processing
            data_fileF = request.FILES['data_file']

            d = Dataset(name=nameF, comments=commentsF, date_add=date_addF,
                        date_upd=date_updF, data_format='temp value',
                        data_file=data_fileF)
            d.save()
            data_properties =
Data_Extractor().get_DataInfo(d.data_file.path)
            d.data_format = data_properties[2]
            d.save()
            return HttpResponseRedirect('/datasets/uploaded/') # Redirect
after POST
    else:
        form = DatasetForm() # An unbound form

    return render(request, 'datasets/upload.html', { 'form': form,
    })

## Edit form for datasets
#
#  For editing an existing dataset
def edit(request):
    if request.method == 'POST': # If the form has been submitted...
        form = DatasetForm(request.POST, request.FILES) # A form bound to
the POST data
        if form.is_valid(): # All validation rules pass
            # Process the data in form.cleaned_data
            nameF = form.cleaned_data['name']
```

```
            commentsF = form.cleaned_data['comments']

            date_addF = timezone.now()
            date_updF = timezone.now()

            # File processing
            data_fileF = request.FILES['data_file']
            d = Dataset(name=nameF, comments=commentsF, date_add=date_addF,
                        date_upd=date_updF, data_format='temp value',
                        data_file=data_fileF)
            d.save()
            data_properties =
Data_Extractor().get_DataInfo(d.data_file.path)
            d.data_format = data_properties[2]
            d.save()
            return HttpResponseRedirect('/datasets/uploaded/') # Redirect
after POST
    else:
        form = DatasetForm() # An unbound form

    return render(request, 'datasets/upload.html', { 'form': form,
    })

## Redirect page for dataset submission
#
#  Processes input and saves dataset
def uploaded(request):
    html = "<html><body>Uploaded.</body></html>"
    return HttpResponse(html)
```

INDEX.HTML: DJANGO-TAGGED HTML FOR DISPLAYING A LIST OF DATASETS

```
{% load staticfiles %}

<!DOCTYPE html>
<head>
<meta charset="UTF-8">
<!--meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"
/-->
<title>Dataset Index - Attitude Estimation Competition</title>
<link rel="stylesheet" type="text/css" href="{% static "datasets/style.css"
%}" />
</head>

<body>

   <!-- Begin Wrapper -->
   <div id="wrapper">

         <!-- Begin Header -->
         <!-- Empty tag to have text added in CSS-->
         <div id="header"></div>
         <!-- End Header -->

         <!-- Begin Faux Columns -->
         <div id="faux">

               <!-- Begin Left Column -->
               <div id="leftcolumn">
```

```
                </div>
                <!-- End Left Column -->

                <!-- Begin Right Column -->
                <div id="rightcolumn">

                <!-- Django Content -->
                 <h1>Dataset Index</h1>
                 <br />
                                        {% if set_list %}
                     <ul>
                     {% for dataset in set_list %}
                         <li><a href="/datasets/{{ dataset.id }}/">{{
dataset.name }}</a></li>{{ dataset.comments }}
                     {% endfor %}
                     </ul>
                 {% else %}
                          <p>No polls are available.</p>
                 {% endif %}

                 <br />

                <p>Descriptive text. </p>

                 <br />
                 <a href="/datasets/upload/">Upload a Dataset</a>
                <!-- End Django Content -->

                <div class="clear"></div>

                </div>
                <!-- End Right Column -->

                <div class="clear"></div>

          </div>
          <!-- End Faux Columns -->

          <!-- Begin Footer -->
          <div id="footer"></div>
          <!-- End Footer -->

    </div>
    <!-- End Wrapper -->
</body>
</html>
```

DETAIL.HTML: DJANGO-TAGGED HTML FOR DATASET DETAILS

```
{% load staticfiles %}

<head>
<!meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dataset Details - Attitude Estimation Competition</title>
<link rel="stylesheet" type="text/css" href="{% static "datasets/style.css"
%}" />
</head>

<body>

    <!-- Begin Wrapper -->
```

```
    <div id="wrapper">

          <!-- Begin Header -->
          <!-- Empty tag to have text added in CSS-->
          <div id="header"></div>
          <!-- End Header -->
          <!-- End Header -->

          <!-- Begin Faux Columns -->
          <div id="faux">

                <!-- Begin Left Column -->
                <div id="leftcolumn">



                </div>
                <!-- End Left Column -->

                <!-- Begin Right Column -->
                <div id="rightcolumn">

                <!-- Django Content -->
                      <h1>Dataset Viewer</h1>
                     <br />
                      <p>Here is the requested dataset: </p>
                     <br />
                     <b>Name:</b> {{ dataset.name }}<br />
                     <b>Uploader Comments:</b> {{ dataset.comments }}<br />
                     <b>Submitted:</b> {{ dataset.date_add }}<br />
                     <b>Last Updated:</b> {{ dataset.date_upd }}<br />
                     <b>Contents:</b> {{ dataset.data_format }}

                      <br /><br />
                <!-- End Django Content -->

                <div class="clear"></div>

                </div>
                <!-- End Right Column -->

                <div class="clear"></div>

          </div>
          <!-- End Faux Columns -->

          <!-- Begin Footer -->
          <div id="footer"></div>
          <!-- End Footer -->

    </div>
    <!-- End Wrapper -->
</body>
</html>
```

UPLOAD.HTML: DJANGO-TAGGED HTML FOR UPLOADING DATASETS

```
{% load staticfiles %}

<head>
<!meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dataset Upload - Attitude Estimation Competition</title>
```

```
<link rel="stylesheet" type="text/css" href="{% static "datasets/style.css"
%}" />
</head>

<body>

   <!-- Begin Wrapper -->
   <div id="wrapper">

         <!-- Begin Header -->
         <!-- Empty tag to have text added in CSS-->
         <div id="header"></div>
         <!-- End Header -->

         <!-- Begin Faux Columns -->
         <div id="faux">

               <!-- Begin Left Column -->
               <div id="leftcolumn">

               </div>
               <!-- End Left Column -->

               <!-- Begin Right Column -->
               <div id="rightcolumn">


               <!-- Django Content -->
                <h1>Dataset Upload</h1>

                {% if error_message %}<p><strong>{{ error_message
}}</strong></p>{% endif %}
                <!--form action="/datasets/upload/" method="post"-->
                <form enctype="multipart/form-data" method="post"
action="/datasets/upload/">
                {% csrf_token %}

                <!-- FORM -->
                {{ form.as_p }}
                <input type="submit" value="Submit" />
                </form>

               <!-- End Django Content -->

               <div class="clear"></div>

               </div>
               <!-- End Right Column -->

               <div class="clear"></div>

         </div>
         <!-- End Faux Columns -->

         <!-- Begin Footer -->
         <div id="footer"></div>
         <!-- End Footer -->

   </div>
   <!-- End Wrapper -->
</body>
</html>
```

STYLE.CSS: PROVIDES STRUCTURE AND FORMATTING TO HTML PAGES

```
/*
      Based on Style Created by Keith Donegan of Code-Sucks.com

      E-Mail: Keithdonegan@gmail.com

      You can do whatever you want with these layouts,
      but it would be greatly appreciated if you gave a link
      back to http://www.code-sucks.com

*/

* { padding: 0; margin: 0; }

body {
 font-family: Arial, Helvetica, sans-serif;
 font-size: 14px;
}
#wrapper {
 margin: 0 auto;
 width: 922px;
}
#faux {
 background: url(faux-1-2-col.gif);
 margin-bottom: 5px;
 overflow: auto; /* Paul O Brien Fix for IE www.pmob.co.uk */
 width: 100%
}
#header {
 color: #333;
 width: 902px;
 padding: 10px;
 height: 100px;
 margin: 10px 0px 5px 0px;
 background: #ABBEBE;
 font-size: 36px;
 font-weight: bold;
}
#leftcolumn {
 display: inline;
 color: #333;
 margin: 10px;
 padding: 0px;
 width: 195px;
 float: left;
}
#rightcolumn {
 float: right;
 color: #333;
 margin: 10px;
 padding: 0px;
 width: 673px;
 display: inline;
 position: relative;
}
#footer {
 width: 902px;
 clear: both;
 color: #333;
 background: #ABBEBE;
 margin: 0px 0px 10px 0px;
 padding: 10px;
 /*font-weight:bold;*/
```

```
 text-align:right;
}
.clear { clear: both; background: none; }

#header:before
{
content:"Attitude Estimation Competition";
}

#footer:before
{
content:"CSS footer";
}
```

References

[1] J. L. Crassidis, F. L. Markley and Y. Cheng, "Survey of Nonlinear Attitude Estimation Methods," *Journal of Guidance, Control, and Dynamics*, pp. 13-25, 2007.

[2] A. Carmi, "Sequential Monte Carlo Methods for Spacecraft Attitude and Angular Rate Estimation from Vector Observations," [Online]. Available: http://ae-www.technion.ac.il/~avishy/00paper.pdf. [Accessed October 2013].

[3] A. Morawiec, Orientations and Rotations: Computations in Crystallographic Textures, Springer, 2004.

[4] J. Diebel, "Representing Attitude: Euler Angles, Unit Quaternions, and Rotational Vectors," 2006.

[5] P. E.-N. Franklin, Feedback Control of Dynamic Systems (6th Edition), Prentice Hall, 2009.

[6] "Gyroscope Primer - Phidgets Support," [Online]. Available: http://www.phidgets.com/docs/Gyroscope_Primer#Drift. [Accessed October 2013].

[7] C. Hall, "AOE4140 Spacecraft Dynamics and Control," 2003. [Online]. Available: http://www.dept.aoe.vt.edu/~cdhall/courses/aoe4140/attde.pdf. [Accessed October 2013].

[8] "EularQuats," [Online]. Available: http://www.anticz.com/eularqua.htm. [Accessed October 2013].

[9] "Kalman and Extended Kalman Filters: Concept, Derivation and Properties," [Online]. Available: http://users.isr.ist.utl.pt/~mir/pub/kalman.pdf. [Accessed September 2013].

[10] "An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors," [Online]. Available: http://microsat.sm.bmstu.ru/e-library/Algorithms/AttDetermination/kalman/ex.pdf. [Accessed September 2013].

[11] "EXTENDED QUEST ATTITUDE DETERMINATION FILTERING," [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=35BB64C602757FE61EC4BEE080988CFB?doi=10.1.1.151.7265&rep=rep1&type=pdf. [Accessed September 2013].

[12] E. Orhan, "Particle Filtering," [Online]. Available: http://www.cns.nyu.edu/~eorhan/notes/particle-filtering.pdf. [Accessed October 2013].

[13] "History of the Web - World Wide Web Foundation," [Online]. Available: http://www.webfoundation.org/vision/history-of-the-web/. [Accessed October 2013].

[14] "HTML & CSS - W3C," [Online]. Available: http://www.w3.org/standards/webdesign/htmlcss. [Accessed October 2013].

[15] "JavaScript Tutorial," [Online]. Available: http://www.w3schools.com/js/. [Accessed August 2013].

[16] “web services - Difference between frontend, backend, and middleware in web development - Stack Overflow,” [Online]. Available: http://stackoverflow.com/questions/636689/difference-between-frontend-backend-and-middleware-in-web-development. [Accessed October 2013].

[17] “Hot Scripts - The net's largest PHP, CGI, Perl, JavaScript and ASP script collection and resource web portal.,” [Online]. Available: http://www.hotscripts.com/. [Accessed July 2013].

[18] “WordPress Business Directory Plugin | HotScripts Link Indexing,” [Online]. Available: http://www.hotscripts.com/listing/wp-business-directory-plugin/. [Accessed July 2013].

[19] “Free source code hosting for Git and Mercurial by Bitbucket,” [Online]. Available: https://bitbucket.org/. [Accessed September 2013].

[20] “Manage groups - Bitbucket - Atlassian Documentation,” [Online]. Available: https://confluence.atlassian.com/display/BITBUCKET/Manage+groups. [Accessed October 2013].

[21] “DjangoSuccessStoryBitbucket – Django,” [Online]. Available: https://code.djangoproject.com/wiki/DjangoSuccessStoryBitbucket. [Accessed September 2013].

[22] “Dell - The Official Site | Laptops, Ultrabooks™, Desktops, Servers, Storage and More | Dell Australia,” [Online]. Available: http://www.dell.com.au/. [Accessed October 2013].

[23] “Compare Products | Dell,” [Online]. Available: http://accessories.us.dell.com/sna/productcompare.aspx?c=us&l=en&s=bsd&cs=04&category_id=5188&prods=224-0939,224-7028,224-7172,224-7696,224-9995,224-8104,224-9997,224-5630,224-8089,224-5768,224-5873,224-1655. [Accessed October 2013].

[24] “2011/2012 One-shot-learning Challenge - ChaLearn Gesture Challenges,” [Online]. Available: http://gesture.chalearn.org/2011-one-shot-learning. [Accessed May 2013].

[25] “CGD 2011 Data - ChaLearn Gesture Challenges,” [Online]. Available: http://gesture.chalearn.org/data. [Accessed May 2013].

[26] “Leaderboard - CHALEARN Gesture Challenge | Kaggle,” [Online]. Available: http://www.kaggle.com/c/GestureChallenge/leaderboard. [Accessed May 2013].

[27] NIPS, “Feature Selection Challenge,” 2003. [Online]. Available: http://www.nipsfsc.ecs.soton.ac.uk/. [Accessed 19 April 2013].

[28] “NIPS 2003 workshop on feature extraction and feature selection challenge,” [Online]. Available: http://clopinet.com/isabelle/Projects/NIPS2003/. [Accessed April 2013].

[29] “An Introduction to Variable and Feature Selection,” [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/GuyonE03.pdf. [Accessed September 2013].

[30] “nips2003 challenge on feature extraction instructions,” [Online]. Available: http://clopinet.com/isabelle/Projects/NIPS2003/FAQ.html. [Accessed April 2013].

[31] “Feature Selection Challenge - Results,” [Online]. Available: http://www.nipsfsc.ecs.soton.ac.uk/results/?ds=overall. [Accessed April 2013].

[32] “LAMP Web development | Red Hat,” [Online]. Available: http://www.redhat.com/magazine/003jan05/features/lamp/. [Accessed October 2013].

[33] “How LAMP Is Used Today - Webopedia.com,” [Online]. Available: http://www.webopedia.com/DidYouKnow/Computer_Science/2007/LAMP.asp. [Accessed October 2013].

[34] “Good news: Debian 7 is rock solid • The Register,” [Online]. Available: http://www.theregister.co.uk/2013/05/08/debian_seven_review/. [Accessed October 2013].

[35] “Debian - Wikipedia, the free encyclopedia,” [Online]. Available: http://en.wikipedia.org/wiki/Debian. [Accessed October 2013].

[36] “Apache HTTP Server - Wikipedia, the free encyclopedia,” [Online]. Available: http://en.wikipedia.org/wiki/Apache_HTTP_Server. [Accessed October 2013].

[37] Wikipedia, “Comparison of web application frameworks - Wikipedia, the free encyclopedia,” [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks. [Accessed 18 April 2013].

[38] “Frameworks Round-Up: When To Use, How To Choose? | Smashing Coding,” [Online]. Available: http://coding.smashingmagazine.com/2008/01/04/frameworks-round-up-when-to-use-how-to-choose/. [Accessed March 2013].

[39] “Most Popular Web Application Frameworks,” [Online]. Available: http://www.hurricanesoftwares.com/most-popular-web-application-frameworks/. [Accessed March 2013].

[40] “Comparing Server-Side Web Languages - l2.pdf,” [Online]. Available: http://www.upriss.org.uk/perlcgi/l2.pdf. [Accessed March 2013].

[41] “Summary Of Server Side Languages - Web development,” [Online]. Available: http://forums.whirlpool.net.au/archive/90884. [Accessed March 2013].

[42] “Picking a web server language | Platformability,” [Online]. Available: http://blog.caplin.com/2011/06/08/picking-a-web-server-language/. [Accessed March 2013].

[43] “Compare web frameworks » Best Web-Frameworks,” [Online]. Available: http://www.bestwebframeworks.com/compare-web-frameworks/. [Accessed March 2013].

[44] “Compare PHP, Java, JavaScript, Python & CSS Frameworks » Best Web-Frameworks,” [Online]. Available: http://www.bestwebframeworks.com/. [Accessed March 2013].

[45] “How to choose a web framework and be surprised,” [Online]. Available: http://www.slideshare.net/jmarranz/how-to-choose-a-web-framework-and-be-surprised. [Accessed March 2013].

[46] “Terse Words: Top Python Web Frameworks - Today,” [Online]. Available:

http://terse-words.blogspot.com.au/2012/11/python-web-frameworks-revisited.html. [Accessed March 2013].

[47] "Framework Benchmarks: Performance Comparison of Web frameworks | IT News Today," [Online]. Available: http://itnews2day.com/2013/03/29/framework-benchmarks-performance-comparison-of-web-frameworks/. [Accessed March 2013].

[48] "The Great Web Framework Shootout | Curia," [Online]. Available: http://blog.curiasolutions.com/the-great-web-framework-shootout/. [Accessed March 2013].

[49] "Choose the right PHP framework | Web design | Creative Bloq," [Online]. Available: http://www.creativebloq.com/design/choose-right-php-framework-12122774. [Accessed March 2013].

[50] "PHP: Symfony vs. Zend | Karl Katzke," [Online]. Available: http://www.karlkatzke.com/php-symfony-vs-zend/. [Accessed March 2013].

[51] "Pillars of Python: Six Python Web frameworks compared | Application Development - InfoWorld," [Online]. Available: http://www.infoworld.com/d/application-development/pillars-python-six-python-web-frameworks-compared-169442?page=0,1. [Accessed March 2013].

[52] "Benchmarking Go and Python Web servers," [Online]. Available: http://ziutek.github.io/web_bench/. [Accessed March 2013].

[53] "Mind Reference: Python Fastest Web Framework," [Online]. Available: http://mindref.blogspot.com.au/2012/09/python-fastest-web-framework.html. [Accessed March 2013].

[54] "4 Python Web Frameworks Compared — Six Feet Up, Inc.," [Online]. Available: http://www.sixfeetup.com/blog/4-python-web-frameworks-compared. [Accessed March 2013].

[55] "WebFrameworks - Python Wiki," [Online]. Available: https://wiki.python.org/moin/WebFrameworks. [Accessed March 2013].

[56] "python - Django, Turbo Gears, Web2Py, which is better for what? - Stack Overflow," [Online]. Available: http://stackoverflow.com/questions/3646002/django-turbo-gears-web2py-which-is-better-for-what. [Accessed March 2013].

[57] "Why isn't Java used for modern web application development? - Programmers Stack Exchange," [Online]. Available: http://programmers.stackexchange.com/questions/102090/why-isnt-java-used-for-modern-web-application-development. [Accessed April 2013].

[58] "The Top 10 Javascript MVC Frameworks Reviewed," [Online]. Available: http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/. [Accessed March 2013].

[59] "a short view on Django compared to CakePHPphp | Ttwhy's Weblog," [Online]. Available: http://ttwhy.wordpress.com/2008/08/31/a-short-view-on-django-compared-to-cakephp/. [Accessed March 2013].

[60] "CakePHP vs Django | Open Source Missions," [Online]. Available:

http://opensourcemissions.wordpress.com/2010/06/21/cakephp-vs-django/. [Accessed March 2013].

[61] “Multiple databases | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/topics/db/multi-db/. [Accessed March 2013].

[62] “CakePHP vs Django vs Ruby on Rails | shaikhsohail30,” [Online]. Available: http://shaikhsohail30.wordpress.com/cakephp-vs-django-vs-ruby-on-rails/. [Accessed March 2013].

[63] “From Developers Eye “Django vs. CakePHP vs. CodeIgniter” – Which Framework to Use | Web Builder Zone,” [Online]. Available: http://css.dzone.com/articles/developers-eye-%E2%80%9Cdjango-vs. [Accessed March 2013].

[64] “CakePHP Cookbook v2.x documentation,” [Online]. Available: http://book.cakephp.org/2.0/en/index.html. [Accessed March 2013].

[65] “Django vs. web2py,” [Online]. Available: http://www.mengu.net/post/django-vs-web2py. [Accessed March 2013].

[66] “Django documentation | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/1.5/. [Accessed March 2013].

[67] “web2py Web Framework,” [Online]. Available: http://www.web2py.com/init/default/documentation. [Accessed March 2013].

[68] “How to install Django | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/topics/install. [Accessed May 2013].

[69] “Oracle 10g vs PostgreSQL 8 vs MySQL 5,” [Online]. Available: http://it.toolbox.com/blogs/oracle-guide/oracle-10g-vs-postgresql-8-vs-mysql-5-5452. [Accessed June 2013].

[70] “database - SQLite vs MySQL - Stack Overflow,” [Online]. Available: http://stackoverflow.com/questions/3630/sqlite-vs-mysql/41990#41990. [Accessed May 2013].

[71] “MySQL vs SQLite - Making the Right Choice,” [Online]. Available: http://www.sobbayi.com/blog/software-development/decide-sqlite-mysql/. [Accessed April 2013].

[72] “MySQL vs PostgreSQL - WikiVS,” [Online]. Available: http://www.wikivs.com/wiki/MySQL_vs_PostgreSQL. [Accessed May 2013].

[73] “Databases | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/ref/databases/. [Accessed May 2013].

[74] “Ajax (programming) - Wikipedia, the free encyclopedia,” [Online]. Available: http://en.wikipedia.org/wiki/Ajax_%28programming%29. [Accessed October 2013].

[75] “jQuery,” [Online]. Available: http://jquery.com/. [Accessed September 2013].

[76] “dajaxproject.com - easy to use ajax libraries for django,” [Online]. Available: http://www.dajaxproject.com/. [Accessed September 2013].

[77] “Models | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/topics/db/models/. [Accessed August 2013].

[78] “Security in Django | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/topics/security/. [Accessed October 2013].

[79] “How to use sessions | Django documentation | Django,” [Online]. Available: https://docs.djangoproject.com/en/dev/topics/http/sessions/#topics-session-security. [Accessed October 2013].

[80] Linode, “Linode Library,” 2011. [Online]. Available: https://library.linode.com/using-linux/users-and-groups. [Accessed 20 July 2013].

[81] “The Beginner's Guide to Unit Testing: What Is Unit Testing? | Wptuts+,” [Online]. Available: http://wp.tutsplus.com/tutorials/creative-coding/the-beginners-guide-to-unit-testing-what-is-unit-testing/. [Accessed October 2013].