University of Glasgow | School of Computing Science

Computational Modelling of Red Blood Cells II

Liam Furphy

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 27, 2015

Abstract

This focus of the project was the creation of a web application that allowed the simulation of experiments on red blood cells through the use of a computational model. The application can be used to quickly and cheaply simulate experiments instead of actually conducting them using physical materials. This can save considerable time and effort and allows scientists to identify which experiments would be most interesting to run in the real world. Additionally, users are able to perform analysis of the results from within the application due to provided graphing functionality.

Acknowledgements

I would like to thank my project supervisor, Simon Rogers, for the excellent guidance he provided throughout the course of this project. Additionally, I would like to thank the client, Virgilio L. Lew, for his patience during the development of the system and for dedicating his time to assist with its progress.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

The chapter will provide an introduction to the problem to be solved and the aims of the project.

1.1 Problem Overview

Problems relating to complex systems cannot always be solved analytically, but could be solved through the use of computational models. A computational model is a mathematical model that allows the behaviour of such systems to be studied through computer simulation. The simulation is performed by adjusting these variables and observing how the changes affect the outcomes predicted by the model.

Researchers can utilise the results of the simulations to make predictions about what will happen in the corresponding real-world system. When combined with extensive computing resources, scientists can conduct thousands of these simulated experiments to identify which experiments are most likely to help find the solution to their problem. [10]

An example of one of these complex systems is the red blood cell, for which a mathematical model was developed by Lew and Bookchin in 1986. The model attempts to describe how the red blood cells of humans behave throughout their lifetime during laboratory experiments when the amounts of different substances are changed within their environment. With the model, the authors additionally developed software capable of running the model and thus allowing the simulation of laboratory experiments.

However this implementation has become increasingly difficult to execute, maintain and extend. This presents an issue for Lew and others, who desire to utilise the software to predict behaviour and results in real-world experiments.

As a result there is an increasing demand for a new implementation of the model.

1.2 Project Aim

The aim of the project is to develop a tool for Prof. Lew, who is acting as the client for this project, that can be used to run computational experiments using the mathematical model for a red blood cell. It should:

- Provide all of the options for running an experiment that the existing implementation has.

- Provide functionality for performing analysis on results from within the application.
- Allow the user to save the results locally.

1.3 Report Outline

The remainder of this report will detail the process carried out during the development of the system.

Chapter 2 will provide a background to the red blood cell computational model and previous software implementations.

Chapter 3 will document the requirements and discuss the process used for capturing them.

Chapter 4 will detail the software development tasks undertaken as part of the project and discuss their successes and failures.

Chapter 5 will provide information on how the system was evaluated and present the results of the evaluation.

Chapter 6 will conclude the report and reflect upon the project.

Chapter 2

Background and Related Work

This chapter will provide a brief background to the red blood cell mathematical model and previous efforts to implement it in software.

2.1 Red Blood Cells

Human red blood cells transport oxygen around the body. In comparison to other cells of the body, the mature form of red blood cells have a comparatively simple structure as they do not contain a nucleus, mitochondria or endoplasmic reticulum. However they still feature a membrane which surrounds the cell and facilitates the transportation of substances between the cell and the extracellular fluid.[6]

Studies into red blood cells are an important area of research for many different fields including haematology, which focuses on the diagnosis, treatment and prevention of blood diseases[11]. For example, malaria is a life-threatening parasitic disease that infects red blood cells. Knowledge gained about how red blood cells are affected by malaria can help the development of treatments and cures. With malaria alone estimated to have caused 584,000 deaths in 2013[12], it is clear that research in this area can benefit society greatly.

Many mathematical models have been developed over time to describe different aspects of the red blood cell, however of particular interest to this project is the one developed by Virgilio L. Lew and Robert M. Bookchin in 1986[9]. This was the first such model that attempted to incorporate all areas relevant to mature red blood cells and was designed to predict the behaviour of all measurable variables within the cell's environment. The model has been further improved over time to widen the scope and allow for the study of alternative cell types.

2.2 PowerBASIC Implementation

To support their model, Lew and Bookchin developed a computer program in the PowerBASIC programming language [4] that was capable of running the model. The program provides a command line interface to the user, which first directs the user through various different screens where they can alter variables in the experiment environment before the beginning of the experiment. This stage is referred to as the 'Reference State' of the experiment as these values affect the creation of the cell, membrane and extracellular fluid. Following this, users are then given the ability to run the experiment and alter another set of values as the program computes through the 'Dynamic State,' which is the state of the cell system as it progresses through the experiment.

Although the original implementation was successful in performing simulated experiments, it has become increasingly difficult to maintain, operate and extend. There were multiple factors that contributed to this problem.

Firstly, the program was only able to execute on machines running now outdated versions of the 32-bit architecture Windows operating system. Therefore when users upgraded their machines they could no longer use the program without emulation tools, which are often tricky to operate.

Furthermore, the quality of the code did not adhere to common programming practices adopted by modern society: design principles and data constructs such as objects were not utilised, and variables were not given descriptive names thus making it difficult to map code to the model. This made it difficult for anyone other than the original authors to maintain the code as they were not able to understand it without considerable effort.

Consequently demand for a new implementation of the mathematical model was created.

2.3 2013-2014 Masters Project Implementation

In 2013, Josef Idris Khan undertook a Masters project at The University of Glasgow which deciphered the original PowerBASIC implementation and provided the model as software written in the Java programming language. [8]

Due to the level of difficulty involved with gaining knowledge of the underlying theory and comprehending the original codebase, the project was not able to produce a product within its assigned time frame. Although most of the implementation was complete, the resultant program had multiple errors and lacked the ability to interact with the model outside the default parameters as methods for updating variables in the system were not included and no interface was provided.

Nevertheless, this work provides an excellent starting point for the implementation to be created as part of this project. His code would remove the need to understand the underlying biology as the work to be completed did not introduce any new variables; all that remained to be done was introduce new functionality that utilised existing constructs within the implementation, without the need to create appropriate names from a prior knowledge of the red blood cell system.

Chapter 3

Requirements

This chapter will provide a specification of the project requirements and discuss the method utilised to capture them.

3.1 Requirements Capture

Before the requirements could be documented they first had to be defined. The project began with a small set of required functionality: complete translation of the model from the original PowerBasic version to a modern programming language, and provide a simple interface to allow for interaction with that model and analysis of its results. This functionality was used to form the initial requirements specification.

However, as development of the project progressed additional features became apparent that would improve the finished product. The decision was made to update the requirements specification to include these features despite the resultant scope change as this guaranteed that focus would be on implementing the most useful functionality regardless of when it was specified. For each new requirement, its priority, difficulty and estimated development time were examined before inclusion into the requirements specification to ensure that they were ranked appropriately.

Consequently, the resultant specification is a comprehensive list of requirements for the product and includes features that were deemed to be outside the scope of this project. The remainder of this chapter will detail this specification as it existed at the end of the project.

3.2 Requirements Specification

The requirements of the system were split into two distinct categories: functional requirements and non-functional requirements. The functional requirements describe particular functions or behaviour to be implemented within the system. In contrast, the non-functional requirements are utilised to judge the operation of the system; they do not specify functionality.

3.2.1 Functional Requirements

The functional requirements have been prioritised using the MoSCoW method[14]. This means that the requirements have been split into a hierarchy of categories, with each element of a category deemed to have the same importance as the rest of its group.

The first of these categories is the 'must have' requirements. Requirements in this category are critical to the success of the project and define the minimum viable product that is to be delivered. For this project the 'must have' requirements are as follows:

- **Completion of the model in a modern programming language.**
  As mentioned previously, one of the motivations behind this project was that the only existing working implementation of the red blood cell model could not run on modern machines. The model forms the core of the project as all other features are useless if there is no underlying working model. Therefore for the project to succeed it is essential that previous work in the this area is built upon to produce a working model in a modern programming language.
- **Development of a basic interface.**
  The model itself is of little use without an interface that allows users to interact with it. Consequently this needs to provide an interface over the model. This interface must adopt a similar design as the original command line interface in the original PowerBasic implementation and offer all of the options that were available within it.
- **Storage of results in a spreadsheet.**
  The existing implementation allows for result data to be downloaded in a variety of different text files. The system developed for this project is to offer similar functionality, however results are to be provided as a spreadsheet as this provides users with the ability to quickly manipulate and graph their data.

After the 'must have' requirements, the group of 'should have' requirements have the next highest priority. Features that fall into this category as not essential for this success of the project, however they should be implemented where possible as they will greatly improve the finished product. Requirements classed as 'should have' for this project are:

- **Line graphs of results.**
  The product should allow the user to create line graphs that display the output of the model over the course of the experiment. Users should be able to select what parameters to graph and they may have multiple graphs created at once. Providing an in line solution for viewing results graphically allows users to quickly see the results of the experiment without having to use alternative tools to graph the data. Thus including simple graphing functionality will improve the user's experience of the product.
- **User registration and experiment history.**
  User's of the system should be allowed to register with the application to receive their own unique login. This login is to be used primarily to connect experiment runs with a particular user. As a result the system can track a user's experiment history and allow them to revisit previous results.
- **View inputs that caused resultant output.**
  When viewing the results of their experiment run, the user may not recall exactly what inputs they passed to the system. To prevent them from having to track these numbers their selves, the system should allow the user to view the inputs used to get the results of the experiment that they are currently viewing. This would also prevent potential human error from manual recording of inputs and as such, this feature is highly desirable.

- **Ability to save created graphs.**
  The results page should allow the user to save the graphs they have created. This functionality would allow users to easily incorporate their graphs into reports, communications, etc. without the need to recreate them in alternative applications.

- **Shareable results page.**
  Users should be able to share the results page for their experiment with others. This feature would reduce the reliance upon other tools for sharing the output of experiments and would further cement the application as a hub for analysis on results.

Following the 'should have' requirements is the 'could have' tier. Requirements that fall into this tier typically describe features or behaviour that would be nice to have, though not necessary for the finished project. Such features are likely to increase customer satisfaction if included. For this project the tier consists of:

- **Alternative graphical outputs.**
  Users may wish to view data in ways other than line graphs. If the system were to provide in-line functionality for producing other types of graphical output, then users would not have to use alternative systems such as Microsoft Excel for their graphing needs. Consequently the product would move further to being the only application users need for to analyse their data.

- **Description of output values.**
  It may be helpful to users who are unfamiliar with the system to provide them with a description of what each output value represents. This would allow them to quickly understand what output is provided by the system.

- **Start experiment from file.**
  Entering inputs using the interface may be laborious for users. To combat this the product could provide the option of starting a new experiment from a file containing the desired inputs. This would allow users to quickly create experiments and still be able to utilise the data analysis functionality.

- **Compare multiple experiments.**
  To analyse the effect that different inputs have on the experiment results, users may wish to compare the results from multiple experiments. Providing this ability from within the application would supply the users with an easy method of performing such analysis.

- **Example experiments.**
  To train users on how to perform experiments using the system, example experiments may be provided. These experiments would guide the user through some example use cases of the product and detail the different ways that the application could be used to analyse the data.

The final tier features the 'would like to have' requirements. Described here are features that would add considerably to the developed product, however are not likely to be developed as part of this project due to the level of work involved. The following requirements fall into this category:

- **Background execution with email notifications.**
  Some experiment runs may take a considerable length of time to complete. To prevent users from having to wait for the experiment to finish, the application could run the experiment in the background and notify the user via email when it is finished. This would allow users to carry out other tasks without having to check on the progress of the experiment.

- **Graphical interface.**
  As an alternative to form based inputs to the model, the product could provide a graphical interface to the user. This interface would allow the user to interact with the model via graphical representations of the components.

3.2.2 Non Functional Requirements

The non functional requirements have not been prioritised as the finished product should aim to satisfy all of them. For this project the product should be:

- **Maintainable**
  The product should be simple to update and maintain as necessary. Maintainers of the codebase should be able to understand the code quickly.
- **Platform independent.**
  The client should be able to easily the system regardless of what operating system he uses. This will allow anyone who wishes to use the product to do so without having to change their platform.
- **Reliable.**
  The system should produce output when expected and the results should be trustworthy. If the results are not reliable then the entire product becomes unusable.
- **Usable.**
  The client should be able to easily navigate the system to perform desired tasks. The interface should be clear and understandable without additional guidance.

Chapter 4

Product Development

This chapter will discuss the implementation of the product, the design decisions that were made and testing that was performed.

4.1 System Design

Before implementation of the system could begin, various design choices had to be finalised.

4.1.1 Application Type

As the requirements were not specific as to the type of application that was to be developed this was the first task to be completed.

One possibility for consideration was a mobile application. Recent years have seen a vast rise in mobile technology within society and they now influence almost every part of our daily lives. Providing the system as a mobile application would allow the system to be available wherever the user was, without the need to carry larger machinery.

An alternative option would be to develop the system as a desktop application in a portable programming language such as Java. This would allow the application to be installed and run seamlessly on almost any machine supported by the language architecture. Installation of the system could be simplified with automated scripts making the application quick to set up and use.

The final option considered was to implement the system as a web application. Web applications are accessible by anyone as long as they have an internet connection. They require no installation and can be used by any device that has a web browser. Consequently the system would be accessible on almost every platform, making it easier for anyone to use the product.

When considering each of the options, it was first noted that it would be difficult to carry out meaningful analysis on data on smaller screens and a standalone mobile application would fail to satisfy the 'Platform independent' non functional requirement. As a result it was decided that the system would not be developed as a mobile application.

From the remaining two options, the decision was made to create the system as a web application. Though

it shared many benefits with a desktop application in a portable language, the ability to access anywhere on any device without installation made this the preferred option.

4.1.2 Language and Frameworks

Having decided upon the type of application to be developed, the next stage was to select the programming languages and frameworks that would be utilised during the project.

Frameworks in software development provide a foundation for the creation of application with the aim of removing the overhead caused by certain common issues such as accessing databases. Many frameworks exist for web applications, and research had to be undertaken to decide which was the most suitable for this project. Three of the frameworks investigated were Ruby on Rails[5], Flask[13] and Django[7].

Ruby on Rails is a framework for the Ruby programming language that places emphasis on programming conventions such as convention over configuration. Despite a large community providing support and tutorials this framework was quickly discounted as it required the mastering of a new language which would consume a considerable amount of time, resulting in it taking less time to develop the project.

In comparison, Flask is a lightweight framework for the Python programming language. It provides a simple core for web development by not offering functionality common to other web frameworks. However, these features can be integrated through the use of extensions.

Django is another Python programming language. It is not lightweight like Flask but can be extended with additional functionality. Many of the functions that come packaged with Django, such as authentication, would allow for quick integration of some requirements. As a result of these desirable features it was chosen as the framework for the project over Flask.

Consequently, the decision was made to develop any additional components for the system in Python. This reduces the list of prerequisites to being able to maintain the system, therefore simplifying the complexity of the system as a whole.

4.1.3 Architecture Design

The final step before undertaking development work for the product was to design the architecture. Three main components were chosen for the system: model, web application and the database.

Creating the model as a standalone component would decouple it from the web interface, meaning that the model could be maintained independently and be incorporated into other interfaces more easily. The model would be run by the web application, and feed its results to the database.

The web application is the core of the user experience. It receives requests from the client's Internet browser, interacts with the model and database where appropriate and returns HTML pages.

The database holds all information regarding experiments, including inputs given and resultant output. When necessary, it could hold user registration information, providing a central hub for data within the application.

This architecture can be viewed diagramatically in Figure 4.1.

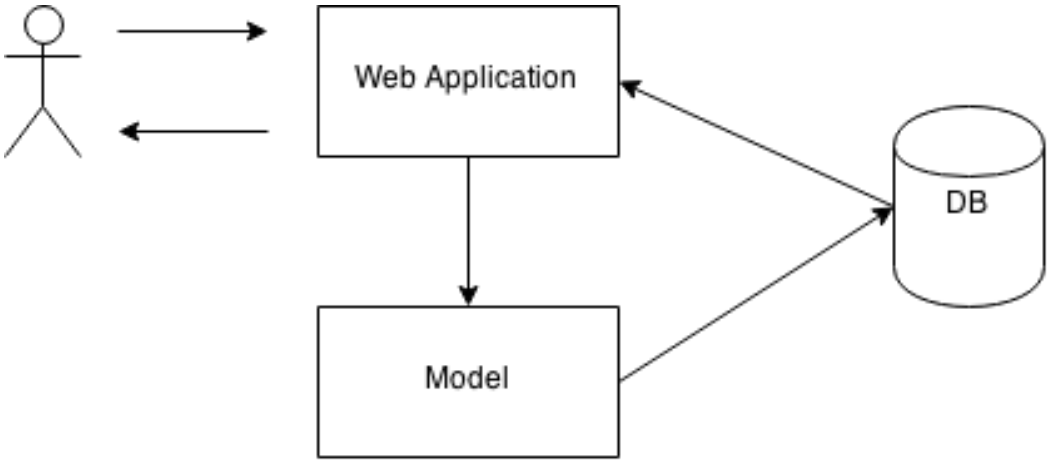

Figure 4.1: System Architecture Diagram.

4.2 Model Development

With the system design finalised, it was time to begin development on the model.

4.2.1 Conversion to Python

As mentioned in Chapter 2, the Java programme developed by Josef Khan provided an excellent starting point for the implementation created for this project. Though his programme had known errors and was incomplete, it had successfully deciphered the core of the original PowerBASIC implementation.

The structure of the Java programme was simple; three classes represented the cell, membrane and extracellular fluid that compose the structure of the red blood cell, whilst an additional three classes were utilised to run the experiment and print the results in various formats. This structure can be viewed in Figure 4.2.

In spite of this simplified structure the code within the classes was often convoluted and difficult to read, regardless of included comments. Part of this problem arose from the use of get and set methods, that added considerably to the codebase size despite providing no meaningful functionality over direct assignment.

As a result development kicked off by converting the Java code to Python and simplifying the code in the process. The action reduced the size of the codebase considerably, making it easier to debug and extend. Nevertheless, as it was mainly a simple translation from Java, the Python code still lacked the same features and had numerous numerical errors for the default experiment that it ran.

4.2.2 Differences in Floating Points

A period of lengthy analysis began to diagnose the cause of the errors in the output of the Python module and resolve them accordingly. Whilst this was successful in resolving some of the differences between the implementations, the majority of errors appeared to be caused by differences in the way that the implementations handled floating point numbers.

In most of these cases, the erroneous number was calculated using others through the use of a mathematical

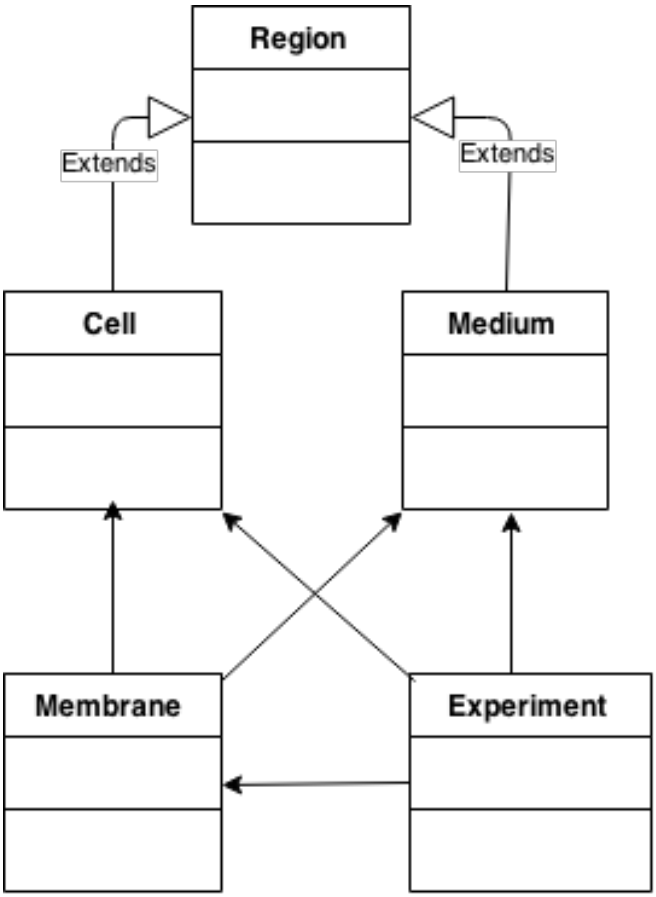

Figure 4.2: Diagram of Josef's system.

formula. However the resultant output varied between Python and PowerBASIC despite all inputs to the formulae and the formulae themselves being correct. In other cases, the simple definition of a floating point number was represented differently within each model.

In general, floating-point numbers are represented in a machine's hardware as a binary fraction. Unfortunately it is impossible to represent the majority of decimal fractions exactly as binary fractions. Consequently floating-point data types in computers only store approximations of the desired floating-point number. This can cause unexpected output from operations. For example in the following Python script[3]:

```
sum = 0.0
for i in range(10):
    sum += 0.1
```

The generated value for the variable sum is 0.9999999999999999 instead of the expected 1. Whilst in this case the difference is minor, in other cases it could cause software to perform incorrectly due to large differences in expected values.

However, the difference in behaviour between the models could not be explained by simply using the general behaviour of floating-point numbers. Further investigation revealed that the PowerBASIC model was using single-precision floating-point numbers for these variables. Single-precision floating-point numbers use 32 bits to store their data and divide it into a 23 bit significand, a single sign bit and eight bits for the exponent. While this allows for a representation of a large range of values, PowerBASIC documentation notes that the data type cannot accurately represent values that contain over six significant digits, anything over this would result in rounding to the closest value in six digits.

In contrast, the Python model uses Python's 'float' data type which is implemented by using the 'double' data type from the C programming language. Double-precision floating-point numbers are used for C's 'double' type and on a typical machine this will use 64 bits to store the number, divided into a 52 bit significand, 11 bit exponent and single bit sign. This allows for a wider range of values to be stored compared to single-precision and with an increased accuracy of 15 to 16 digits.

Clearly the difference in numbers was originating from the differing approaches to floating point numbers in the two different models, though this diagnosis did not provide a solution for validating if the model was performing correctly. Discussions with the client surrounding this issue revealed that even if outputs differed throughout the execution of the experiment but the overall trend followed by the numbers was similar then the model could be deemed to operate correctly. This is because the exact numbers for values were not of importance to analysis as focus was on how different elements of the system changed over time and such trends should remain consistent even if individual values change.

As a result of this information, a Python script was created that could graph the outputs of each value of the models beside each other. The graphs drawn were then manually checked to verify the behaviour of the values. For most cases, the graphs indicated correct behaviour such as that shown in Figure 4.3.

Nevertheless, not all behaviours were correct. For example, Figure 4.4 depicts a value with incorrect behaviour. To rectify these errors analysis had to be done on the codebase of each implementation to discover where the Python implementation differed from the original. Once the causes were identified, fixes were incorporated to the Python module and the graphs redrawn. The process of comparing and fixing was repeated until the behaviour of all values was acceptable.

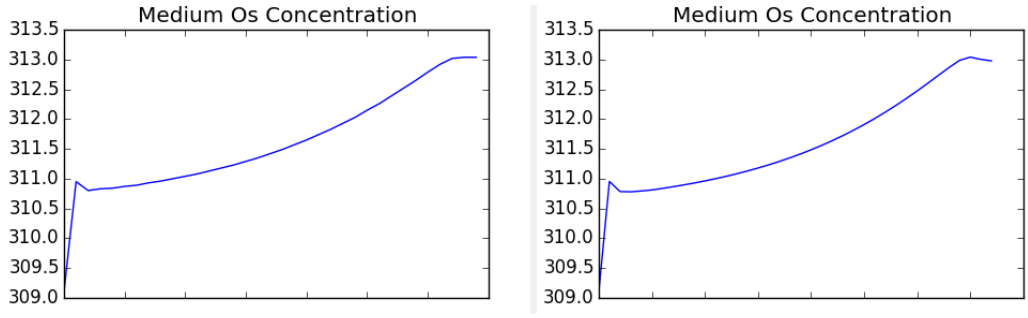

Figure 4.3: Comparison of accurate results.

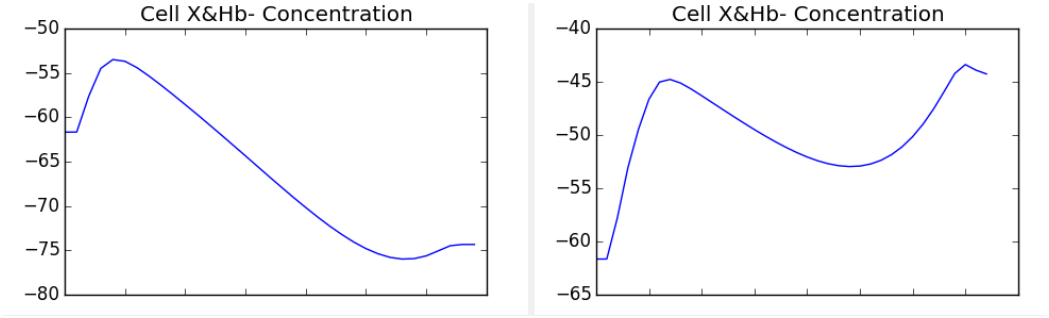

Figure 4.4: Comparison against inaccurate results.

4.2.3 Completion of features

With all values for the default experiment behaving correctly, it was time to expand upon the functionality of the Python model to incorporate all of the customisation options available in the original program.

To accomplish this code relating to the inputs was walked through in the PowerBASIC code to study the value that each input altered and any alternate paths of code that resulted from the changed inputs. Afterwards the ability to set these inputs in the new Python model was added, and additional execution paths incorporated into the model.

However at this time there did not exist any test cases that could be used to verify the functionality of these inputs. Consequently the client was asked to detail a range of meaningful inputs that could be used as test cases for this purpose. Once this information was provided it was used alongside the graphing script discussed in the previous section to manually check that the new features performed as expected. Again, a manual cycle of graphing and fixing ensued until all values were satisfactory.

4.2.4 Automated Regression Testing

As the Python implementation now supported the same functionality as the original, it was deemed appropriate to refactor the code to improve its maintainability and adaptability. But before this refactoring began a suite of automated regression tests were created. The aim of this test suite was to allow for quick checking that the model had not regressed through the refactoring process; meaning that the model continued to behave correctly for each of the defined test cases and no errors were introduced.

Written using the Python unittest library, each test executes the model with a specific set of inputs and compares the resultant values with values captured from a previous successful run of the Python model. An example

of the code used to perform one such test is:

```
def test_pgk_30(self):
        l = Listener()
        e = Experiment()
        e.register(l)
        e.setup()
        e.set_temp_permeability_options({"pgk": 30})
        e.set_screen_time_factor_options({"time": 120})
        e.run()
        test_data = l.vals
        with open("test_1_data.json", "r") as f:
            data = json.load(f)
        for key in data:
            self.assertEqual(data[key], test_data[key])
```

Here we can see how the experiment is created and executed, before each output value is extracted from the text file holding results and compared against the previous value to ensure that no differences exist.

As there are over 100 values to compare, this approach is a vast improvement over the manual alternative and saves minutes each time the tests are run.

4.2.5 Structural Refactoring

Once the automated tests were in place, the refactoring process could begin. The main objective behind refactoring the model was to improve its structure and increase the maintainability and extensibility of the system.

The major focus area to be refactored to achieve these goals was the structure of the system. At this point in the project the codebase followed the same structure as Josef Khan's Java implementation, discussed in Section X. However no investigation had been done into whether this was the most appropriate structure for the model or what the best method for connecting the model to the web interface would be.

Consequently, a series of alternative designs for the model architecture were considered. The first of these can be seen in Figure 4.5. Overall this idea follows a similar structure to the current one, but each object is implemented as a Django model instead of a standard Python object. Using Django functionality for representation of the model would allow for a simple integration between the model, web interface and database where results would be stored as each component would be built using the same framework. The resultant consistency across the system would reduce the effect required to understand the code and therefore increase its ability to be maintained.

Nevertheless, this design limited the model to being used within a Django environment. If the model was to be used with any alternative interface in the future then, unless the new interface could manipulate Django's models, it would have to be redeveloped to be able to integrate effectively. This was undesirable as it could lead to multiple implementations of the model in active existence at once, increasing the level of work needed to maintain the model and extend it's functionality.

As a result it was necessary to consider designs that would separate the model from the interface and provide a standard mechanism for the model to communicate with other components. A standard way of achieving this is to use the Observer pattern. In the observer pattern a single object called the observable is monitored by multiple others, called the observers, who are notified of any changes that occur within the observable.

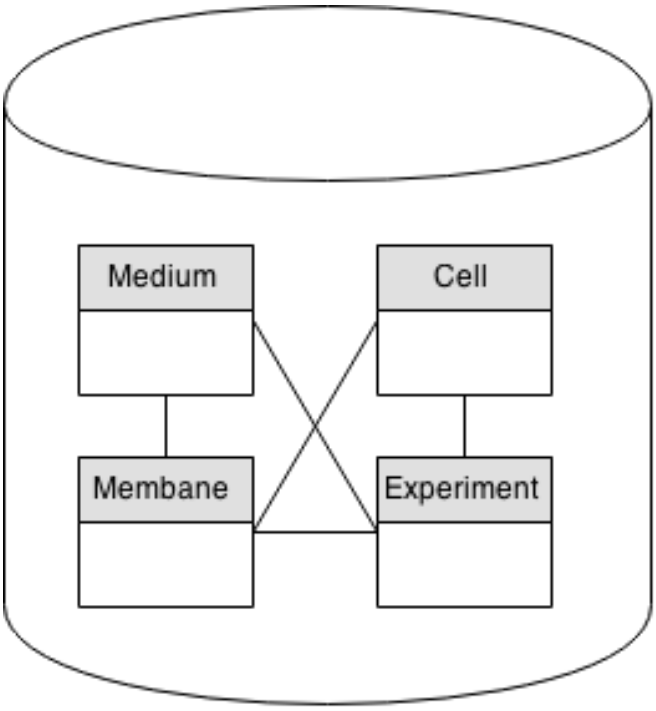

Figure 4.5: Design that utilises django models to represent the system.

An example structure that implements this pattern can be seen in Figure 4.6. Though the class structure is similar, the observable pattern is now utilised to allow for communication of results between the model and any potential listeners. This design does succeed in making the model easier to extend as it provides a simple mechanism for integrating with new interfaces.

However, the class structure representing the cell system itself was still very basic so brainstorming began for more complex structures. The resultant idea can be viewed in Figure 4.7. By following this design the resultant model would reflect the physical structure of the cell at a deeper level than the previous design as the membrane would utilise transporters for moving substances to and from the cell instead of doing it directly. In addition, maintainability would be improved as the complexity of code in the Membrane and Experiment classes would be reduced significantly, thus allowing the system to be understood more easily in the future. Nevertheless, it was difficult to define how to break up the existing code into this structure so it was discarded in favour of the previous design.

4.2.6 Future Improvements

After the completion of the refactoring, the model itself was ready to be integrated with a web interface. However some extra areas of improvement were identified that could not be completed due to time constraints of the project.

Firstly, there were many small improvements that could be made throughout the code that affect how variables are updated during the experiment. For example, during the execution of the experiment an interval is calculated for the next step of the calculation. The code shown in Figure 4.8 was originally used to perform this task, where calculate_integration_interval was the method shown in Figure 4.9.

It is clear to see that the method signature is obtuse and as a result subsequent calls to it become more difficult to read. Analysis of the original and Python code bases showed that the method would never need to be

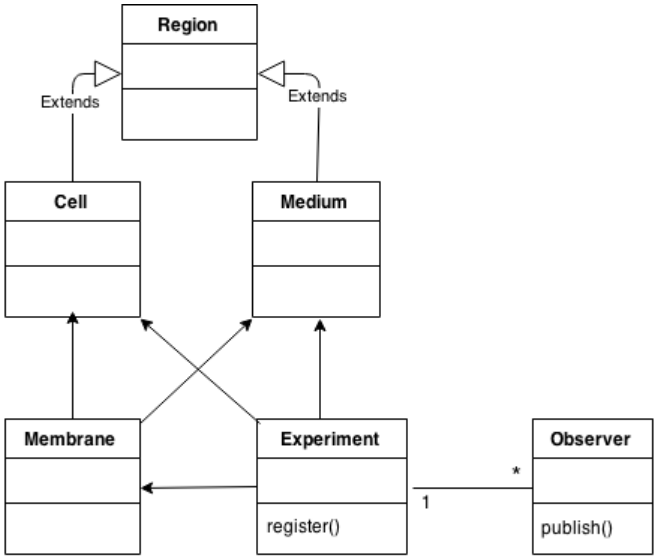

Figure 4.6: Design that utilises the observer pattern.

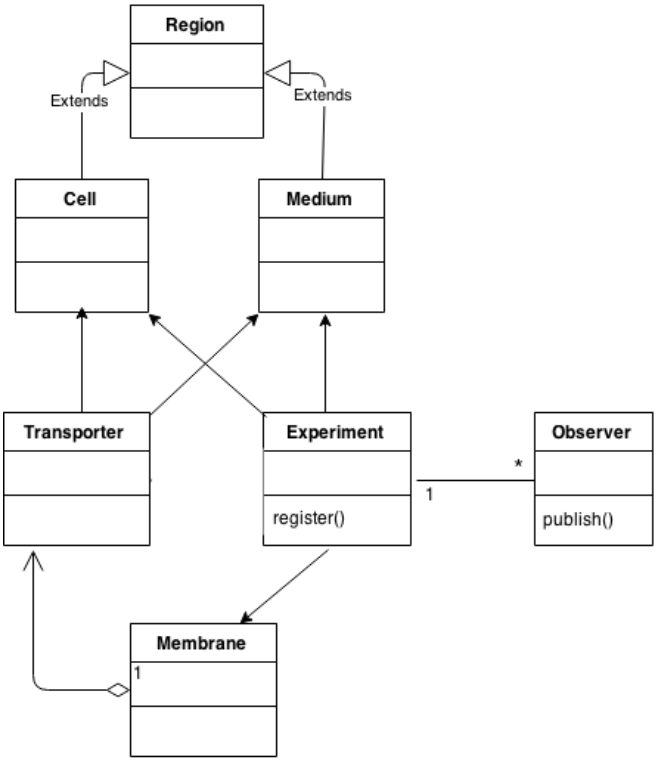

Figure 4.7: Design breaking down the membrane into transporters.

```
self.integration_interval = self.calculate_integration_interval(self.membrane.flux_A23187_Mg2H_Mg,
                                                                self.membrane.flux_Ca,
                                                                self.membrane.flux_Goldman_H,
                                                                self.membrane.dEDGTAH, self.membrane.flux_Na,
                                                                self.membrane.flux_K, self.membrane.flux_A,
                                                                self.membrane.flux_H, self.membrane.flux_water)
```

Figure 4.8: Original calculation of integration interval.

```
def calculate_integration_interval(self, flux_A23187_Mg, flux_Ca, flux_Goldman_H, dEDGTAH0, flux_Na, flux_K, flux_A,
                                   flux_H, flux_W):
    return 10.0 + 10.0 * abs(flux_A23187_Mg + flux_Ca) + abs(flux_Goldman_H) + abs(dEDGTAH0) + abs(flux_Na) + abs(
        flux_K) + abs(flux_A) + abs(flux_H) + abs(flux_W * 100.0)
```

Figure 4.9: Original method for calculation of integration interval.

```
self.calculate_integration_interval()
```

Figure 4.10: New calculation of integration interval.

called with alternative parameters, making the passing of values to the method instead of directly accessing them unnecessary. Subsequently the method was updated to the code shown in Figure 4.11, which is called simply by the line shown in Figure 4.10. Though this is a minor change it improves the readability of the code, making it easier for others to understand what is happening. As there are lots of similar changes that could be made throughout the implementation, the overall effect would be substantial but time consuming to implement.

Furthermore, there are methods within the code that are very long in length. As a result of this they become difficult to understand and harder to debug due to the number of operations within them. Future refactoring work could look to breaking these sections into more manageable pieces of code, which in turn would improve the overall maintainability of the system.

Another area that could be improved upon is the small suite of regression tests. Currently there are only a few test cases in the test suite and expanding upon that to include further automated tests would help ensure the robustness and reliability of the model.

One particular area that could be targeted for testing is the functionality for passing inputs to the system. For each input, at least one unit test could be written to ensure that the input value is able to be modified and that modifications provided the desired result. This would ensure that interfaces that used the model would not have their functionality broken.

In addition to these unit tests, methods within the classes that performed complex computations could unit tests written for them too. These tests would aim to confirm that complicated logic within the model continues to execute correctly and the correct results calculated. As these methods would be more susceptible to error, having tests for them would allow confidence that they will function as expected.

4.2.7 Lessons learned

Throughout the development of the model various events highlighted areas of improvement within the process of the project. Each of these areas can be learned from in order to avoid the same pitfalls in future work.

Firstly, when comparing values that change over time it may often be more useful to analyse the difference in the trend the values follow graphically. Had this happened earlier in this project then a substantial amount

```
def calculate_integration_interval(self):
    self.integration_interval = 10.0 + 10.0 * abs(self.membrane.flux_A23187_Mg2H_Mg + self.membrane.flux_Ca) + abs(self.membrane.flux_Goldman_H) + \
                                abs(self.membrane.dEDGTAH) + abs(self.membrane.flux_Na) + abs(self.membrane.flux_K) + abs(self.membrane.flux_A) + \
                                abs(self.membrane.flux_H) + abs(self.membrane.flux_water * 100.0)
```

Figure 4.11: New method for calculation of integration interval.

of time would have been saved that resulted from investigation into errors that only existing due to difference precisions of floating-point numbers.

Furthermore, it would have been beneficial to introduce automated testing earlier in development. Even if regression and unit tests had to be updated throughout the course of the development, this would approach would likely still have taken less elapsed time that manual testing. As an additional bonus, the automated tests would not be susceptible to human error that may occur during laborious manual testing exercises.

Finally, learning some of the biology behind the computational model may have proven useful during the considering of refactoring options. Knowledge of the subject matter for this system would have aided with the detection of possible improvements to the structure of the implementation. Though the existing design is simple there are areas within it that appear likely to benefit from further refactoring; this was unable to be determined initially as it was difficult to identify the best logical split without knowledge of the underlying biological theory.

4.3 Web Application Development

Development for the web application component of the system also began after decisions were made on the system design, however the majority of the work could not be completed until the model component was ready.

4.3.1 Interface Design

Before any coding could begin, the interface itself had to be designed.

To kick-off this process, the command line interface from the PowerBASIC implementation was analysed to identify strengths and weaknesses. As mentioned in Chapter 2, this interface walked the user through a series of screens where they could set values for different aspects of the system during either the reference or dynamic states. Examples of these screens can be seen in Figures 4.12 and 4.13.

This simple interface is very straightforward and natural to use and subsequently it was decided that the concept of walking the user through the different screens should be translated to the appropriate web page constructs. However there was potential to simplify the process even further by merging several of the smaller screens together and displaying all of the options for a screen at once.

With these changes in mind, paper wireframes for each of the pages were drawn. This was a quick way of visualising different ways to design the web pages and identifying which ideas should be discarded. Figures 4.14 and 4.15 show two of the wireframes that were proposed for the dynamic state screen. In the first all possible input options are provided when the user loads the page, however the second stays closer to the original design by separating inputs into various groups and providing access to them in separate screens. As the first design would result in a lengthy page that required lots of scrolling due to the number of options available, the second design was selected as the most suitable.

This process allowed for wireframes with a more final design to be drawn in order to be presented to the client for validation. However, due to geographical separation it was recognised that the paper prototypes would be insufficient in order to perform evaluation of the design effectively. To combat this, a set of HTML pages were created as a prototype for the final solution and provided to the client. These pages made use of the Bootstrap CSS framework as this simplified the creation and styling of various constructs within the pages. The prototype pages for the reference state and the dynamic state screens can be viewed in Figures 4.16 and 4.17 respectively.

```
**********  Cytoplasmic Mg buffering screen  **********

The program incorporates cytoplasmic Mg buffers to simulate effects
of Mg redistribution induced with the ionophore A23187.
Press ENTER to skip Mg screen and accept defaults or enter any
number to see or change the data.         3

MgoT? Default=0.2 mM. ENTER for default or change to:

MgiT? (in mmole/loc); default=2.5.  This gives Mg2+ of about 0.3
mmole/lcw in oxygenated red cells with the default ATP+P and (2,3-DPG+P)
concentrations of 1.2 and 7.5 mmole/lc offered below.

High-affinity buffer? (in mmole/loc); ENTER for default=0.05:

ATP+P? (in mmole/loc); ENTER for default=1.2:

2,3-DPG and other organic phosphates? (in mEq P/loc); ENTER for default=15:
```

Figure 4.12: Input screen of original program.

```
TO SET UP AND RUN AN EXPERIMENT CHANGE THE VALUES
IN ONE OR MORE OF THE FOLLOWING SCREENS:-

--------------------    ------------------    ----------    --------------
SCREEN 1                SCREEN 2              SCREEN 3      SCREEN 4
Time-factors            Cell fraction         Transport     Temperature
                        Medium composition    inhibition    Permeabilities
--------------------    ------------------    ----------    --------------
Length of experiment    Cell fraction         Na-pump       Temperature
Integration Interval    Na/K/A(Cl)            Na:K:2Cl      PWater
Cycles per printout     Na x glucamine        Jacob-Stewart PGK
                        Cl x gluconate        PL(Na:A)      PGNa
                        sucrose               PL(K:A)       PGA(Cl)
                        medium pH             Ca-pump       PGH
                        Buffer                Ca-leak       PMg/Ca(A23187)
                        MgoT                  Gardos        pI of Hb
                        CaoT
                        EGTA
                        EDTA


Choose a screen (1 to 4), enter 5 to run the experiment
or enter 6 for new (Na)pump-leak steady-state:     _
```

Figure 4.13: Dynamic state screen of original program.

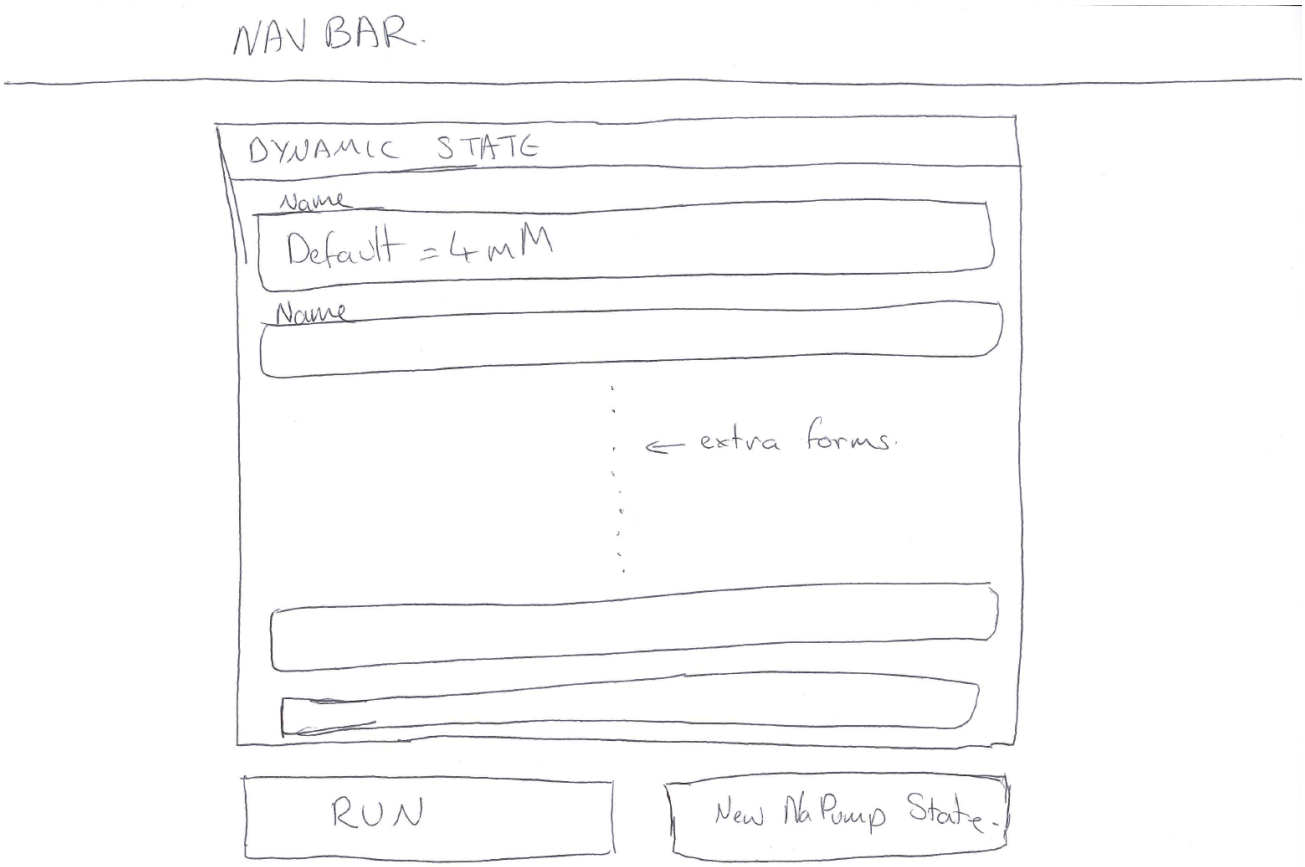

Figure 4.14: Dynamic state screen with all inputs.

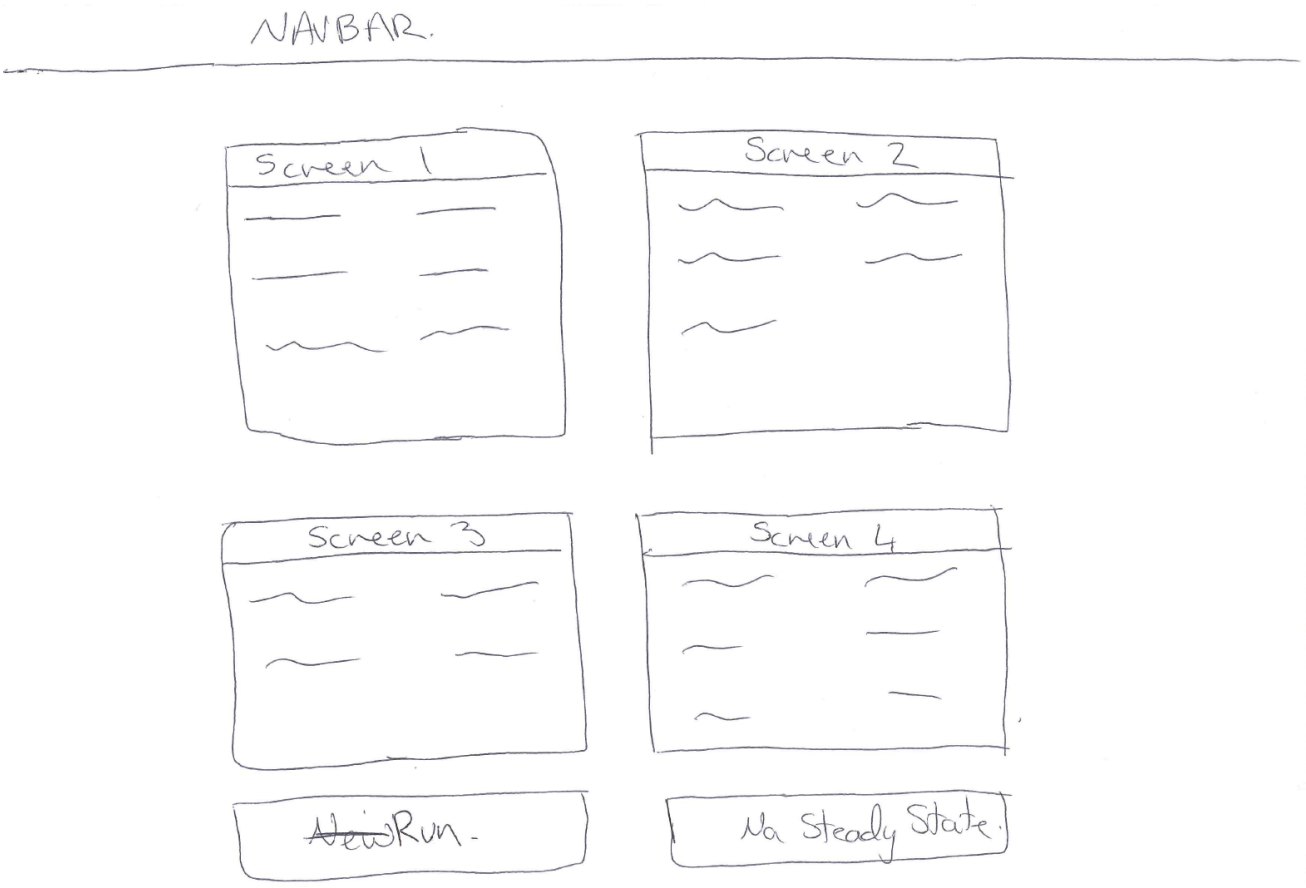

Figure 4.15: Dynamic state screen with separate pages.

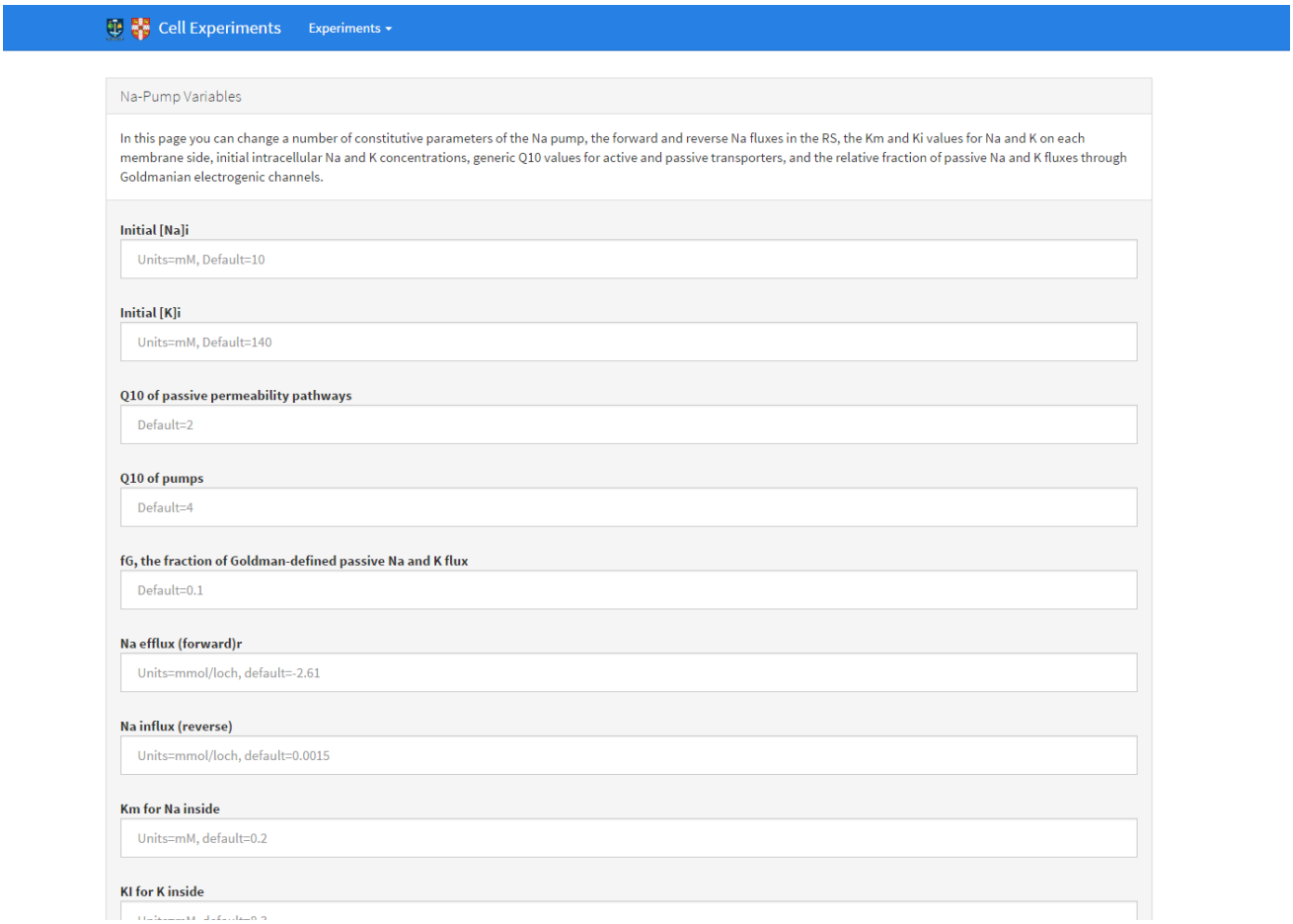

Figure 4.16: Initial prototype for an input screen.

Subsequent feedback gained from the client was extremely helpful in pinpointing areas of the design that could be improved. Although he felt that the design was heading in the right direction there were aspects he wished to be changed for the final implementation. These changes included updating the colour scheme and text used on the page, and splitting the reference state input screen into separate pages that represented the original screens from the corresponding section of the PowerBASIC program.

This feedback was taken on board and used to iterate upon the original prototypes. Once the client was happy with the designs, this process ended. Figure 4.18 shows the finalised design for the dynamic state screen. It is vastly different in comparison to the original prototype from Figure 4.16, the main differences being: overhaul of the entire colour scheme, units for inputs are now clearly indicated at the edge of the form, and inputs for the time factors are included here instead of a separate page.

4.3.2 Developmental Issues

With the design finalised the only barriers left to development of the product were a few decisions that had to be made regarding implementation of various sections of the web application component.

The first of such issues was the data storage system. The web application would require the ability to store results and other data regarding experiments and users. Though there are many database backends that can integrate successfully with Django, Sqlite3 was chosen for this application as it is the simplest to use and was sufficient for this application as high levels of traffic were not expected.

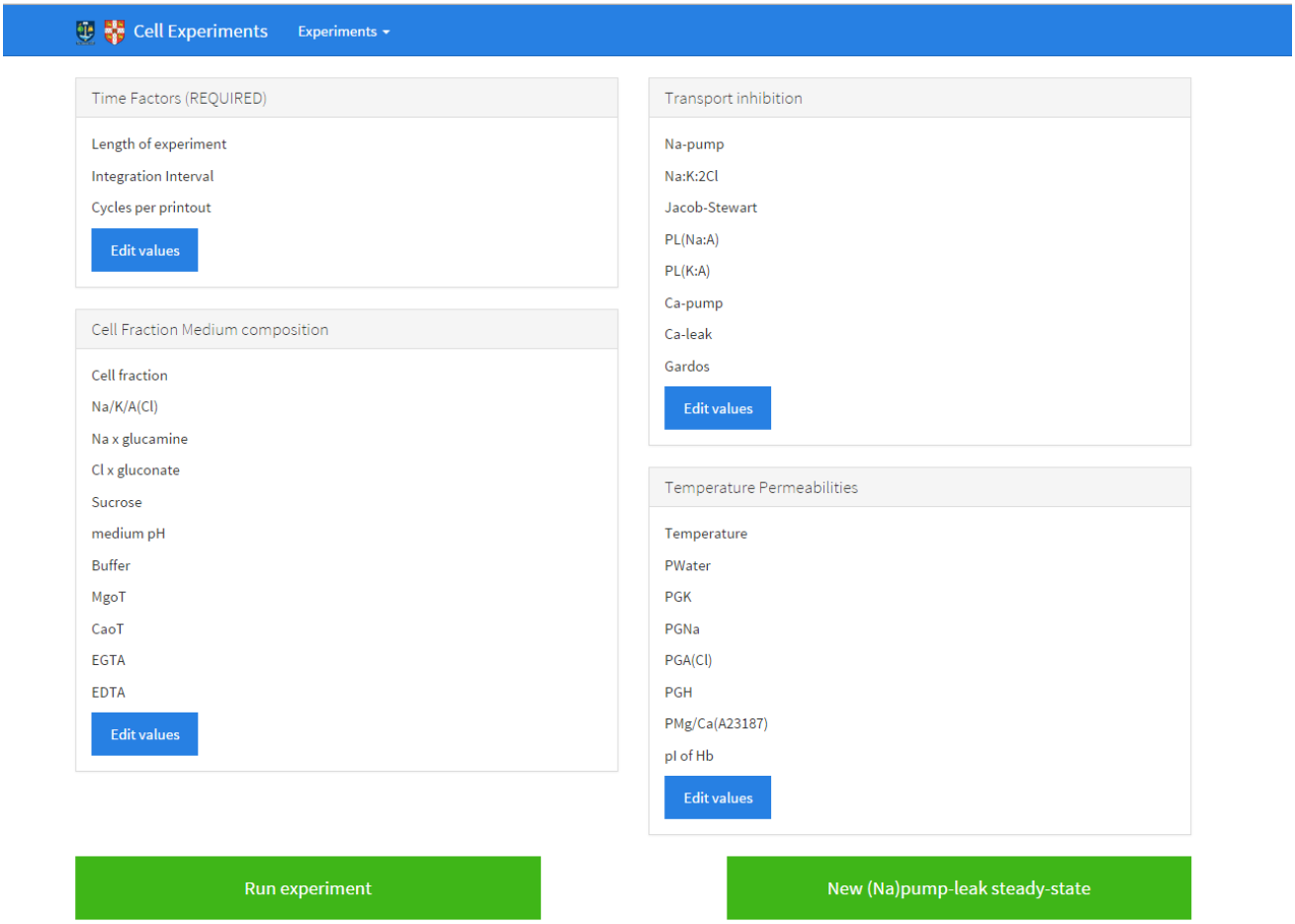

Figure 4.17: Initial dynamic state prototype.

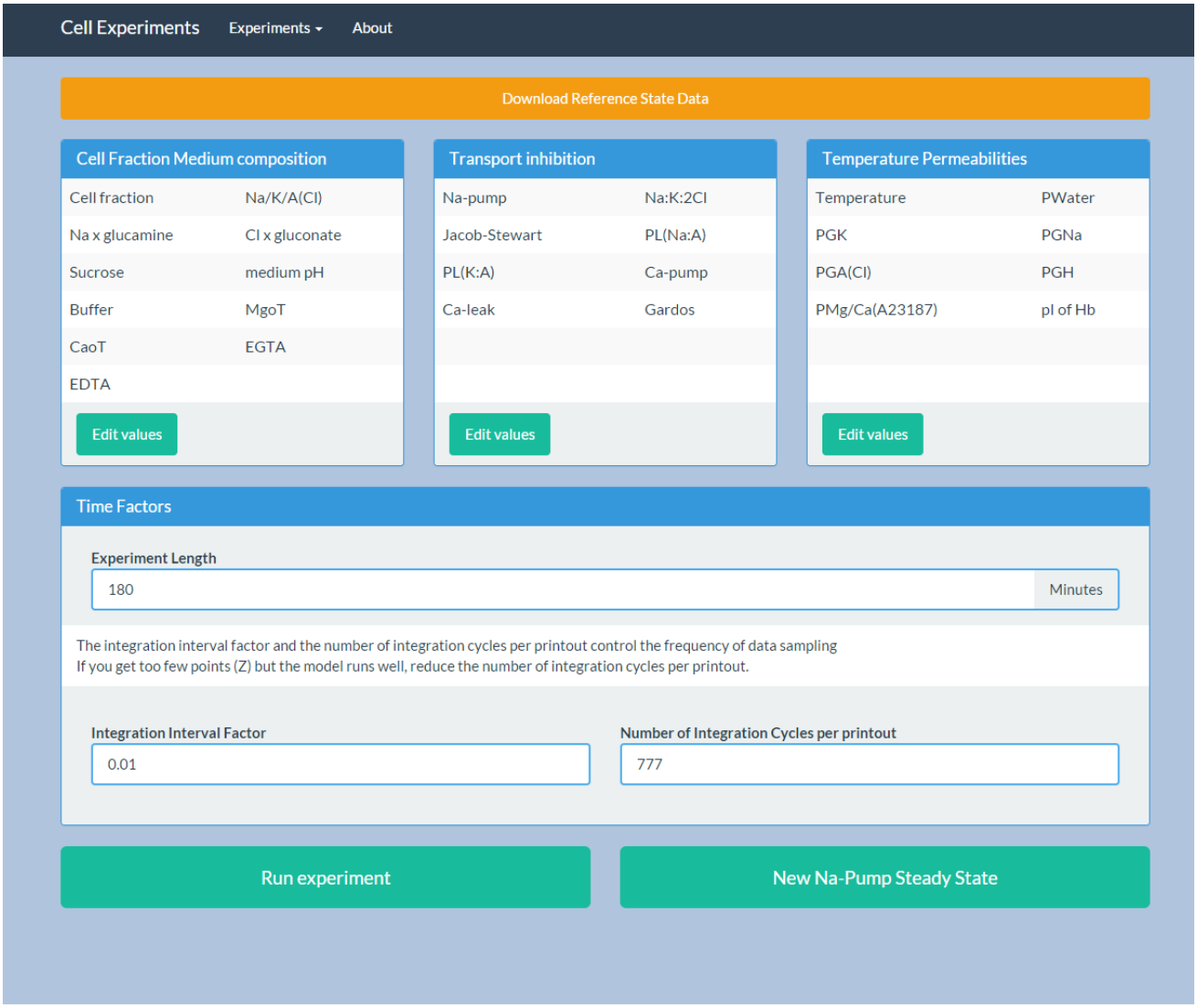

Figure 4.18: Final prototype for dynamic state screen.

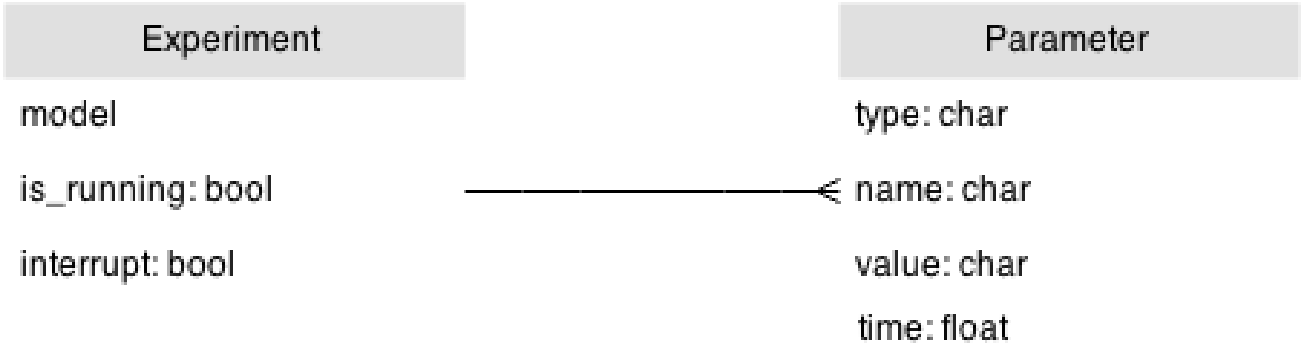

Figure 4.19: ER Diagram.

Having selected Sqlite as the database backend, the next task was to define how data would be represented within it. This was accomplished by creating the ER diagram for the system, which can be seen in Figure 4.19. The ER diagram shows the structure of the database and the relationships between the objects it stores. For this application, the data we want to store concerns users, experiments and results.

Although most of the fields are simple data types, the 'model' column in the 'Experiment' table is the JSON serialised version of the instance of the 'Experiment' class from the model that corresponds to a particular experiment run. Storing this information in the database allows the state of the model to be retained for any further changes that the user wishes to make. Though the model could be recreated by using the inputs that were already provided, this would not always be practical due to the runtime required for the simulation.

The other decision that had to be made regarded the asynchronous execution of experiments. As the user should be able to pause the experiment run if they wish to make changes, the experiment would normally be executed in a separate thread and interrupted appropriately. In Django applications, this functionality is achieved through the use of asynchronous task managers such as Celery or django-async. Although django-async is easier to integrate with a Django application, Celery was chosen for this task as it was still simple to integrate and featured a wider range of supporting documentation that

4.3.3 Developed Product

With all preparation complete, implementation work began on the web application. This was undertaken as an iterative process, where functionality was added and improved upon in small stages until eventually the application had been successfully connected to the model component and implemented all of the functionality specified by the 'must have' requirements.

Development then began to focus on implementing lower priority requirements. As of the end of the project, the 'Line graph of results', 'Ability to save created graphs' and 'Shareable results page' requirements from the 'should have' category have been included in addition to the core functionality. Though most of the pages remained identical in appearance to their final HTML prototype, the results page changed dramatically as a result of this increased functionality. As visible in Figure 4.20, when users arrive at the results page they are presented with a variety of different options. The first options available are to download their results as a csv or make further changes to the experiment. Below these options is the graphing section of the page, which was implemented using the Google Visualisation Toolkit. Users can plot up to 2 different variables on the graph and download it in the PNG image format. Further graphs can be added or deleted as required through the remaining buttons on the page. In addition to the pages required for the basic functionality of the system, two new pages were

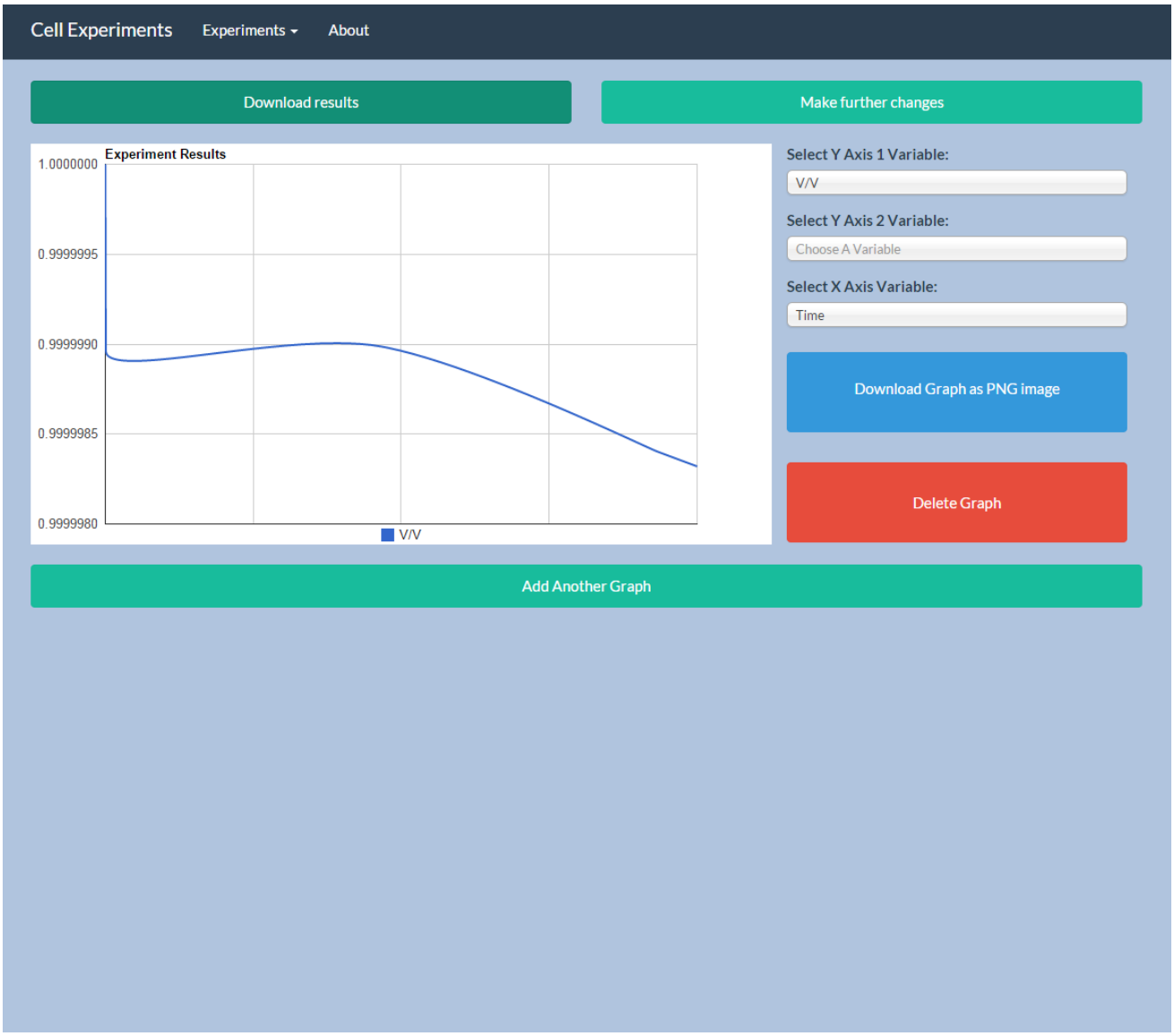

Figure 4.20: Results Page.

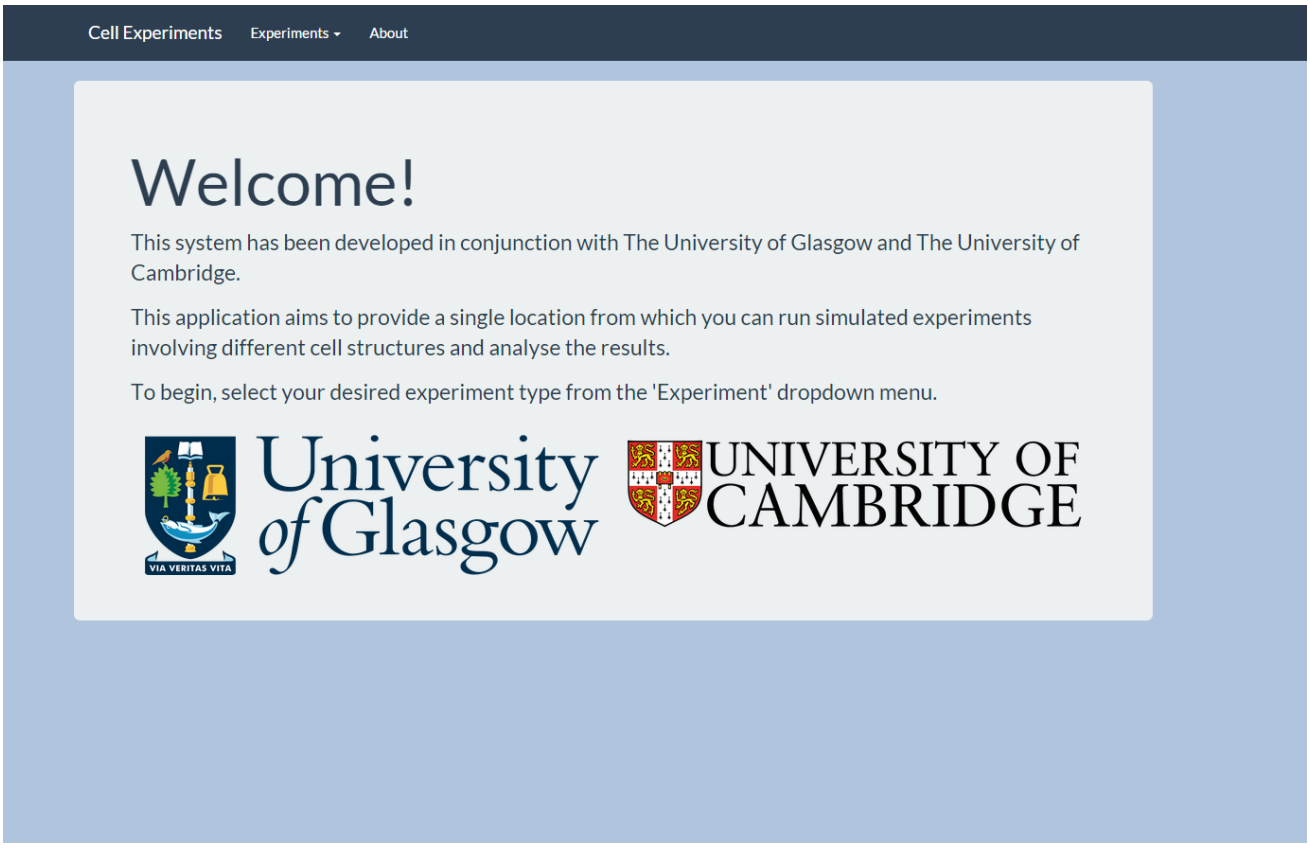

Figure 4.21: Home Page.

introduced. The first of these pages can be seen in Figure 4.21 and acts as the home screen to the application by providing a small introduction to the purpose of the web application and how it can be used. This page was included to provided a generalised landing area for users in case alternative experiment types are included into the application in the future. The other page introduced is the 'About' page, which can be viewed in Figure 4.22. This page aims to provide a short background to the system and give credit to relevant stakeholders.

4.3.4 Key Learning

Though the web application was successful fulfilling its highest priority requirements by the end of the project, as observation has been made that would make future development on this project or others more successful.

After the addition of each new feature the web pages were tested to ensure that no functionality or styling of the pages had been broken unintentionally. This was achieved by manually navigating through each of the pages and testing the available actions that could be undertaken. This process consumed a lot of valuable time that could have been used for the integration of extra functionality.

A better alternative approach to this would have been to develop an automated testing suite that would have allowed for quick verification of the page content and behaviour. Such automated testing would be able to identify failing areas more quickly and not be prone to human error.

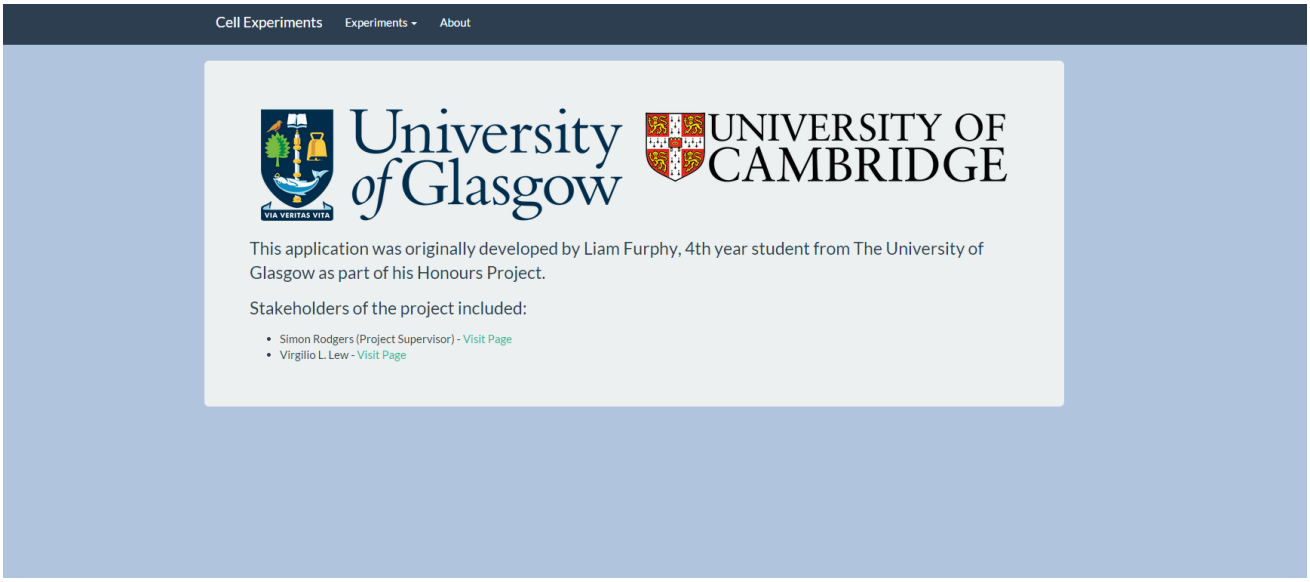

Figure 4.22: About Page.

Chapter 5

Evaluation

This chapter will discuss how the system developed for this project was evaluated. It focuses on evaluation that involved client participation and does not discuss automated testing as this has been covered in Chapter 4.

5.1 Acceptance Testing

The first stage of the evaluation was to perform acceptance testing. The aim of this section was to verify that the system was functioning as expected and implemented all necessary requirements.

During this stage, the client was provided with a live instance of the web application which he used to validate that the system was working correctly. The set of test cases used for this validation consisted of those previously provided for use in the automated regression test suite for the model and a range of previously unseen tests. For each of the tests, the results generated were compared to the expected output to determine if the system was performing reliably.

Although the system performed as expected in most cases, errors were found during some of tests that used unseen inputs. All of the issues discovered were constrained to the implementation of the model and efforts were made to resolve these problems. The status report found in Appendix B contains details of issues that still exist within the system and could not be resolved within the time constraints of the project.

In addition to errors within the model, the tests also identified some failing functionality within the web application. These bugs were caused by various input options having no effect on the underlying model due to the relevant modification options not being called by the web application. All bugs of this nature were quickly fixed and the corresponding inputs now function correctly.

Overall the tests found that despite errors existing within the model implementation, the client was able to successfully use the system to produce results.

5.2 Interview Feedback

Although the acceptance tests were useful for validating the performance of the system, they were not able to provide qualitative data regarding the usefulness of the project for prospective users.

As this project had only a single client, it was decided that this data would be best attained through interview style questioning of the client as this would allow for a much more in depth analysis of the system.

The first set of questions focused on the design of the interface, how easy he found it to use and its suitability for the task. He stated that the interface reflected his needs, and described the design as providing 'an infinitely easier, more intuitive and rewarding experience' than the original command line interface that should be clear to any specialists in this field of study. However, he did note that there were some areas that could benefit from further improvement, such as updating the explanatory text used throughout the system. From these comments we can see that the design of the web application's interface was highly successful, though could be tweaked to provide a richer experience.

Moving onwards, the questions then focused on whether the system could be used to successfully carry out the desired tasks and provide reliable results. He acknowledged that the tests that had executed outside the known issues had returned 'perfectly accurate data' and stated that he believed once the known issues are resolved there would be no reason other use cases would not be the same. This shows that implementation of the model is producing reliable results and that the other reason he is prevented from successfully carrying out some tasks with the applications is due to errors given in the output.

Overall, the client was pleased with the result of the project. Once the known issues are corrected he plans to 'use it extensively' and stated that there is a 'long queue of haematologists and researchers in RBC physiology, pathophysiology, haemolytic anaemias and malaria waiting for this tool, frustrated by the limitations and complexities of the original programme' and would benefit from being able to use the system. These comments highlight that this system has the potential to be of use to a large number of scientists and that he believes the product will be able to fulfil that potential once the known issues are eradicated.

5.3 Requirement Fulfilment

With the capturing of evaluation data complete, all that remained of the project was to determine if it was successful in fulfilling the requirements laid out in Chapter 3.

As functionality for all of the 'must have' and a majority of the 'should have' requirements was included in the system, the critical features required for the project to potentially be considered a success are present. Though errors existed in the implementation of the model, these should not take long to fix and only affect a subset of the potential test cases.

Regarding the non-functional requirements, it is clear from the previous interview with the that the system meets the 'Usable' criteria, and should be able to meet the 'Reliable' requirement following the resolution of errors. Due to the nature of web applications, the product also satisfied the 'Platform independent' point. Although the design of the code base should allow it to satisfy the 'Maintainable' requirement, the success of the project in meeting the point can only be determined by any future maintainers of the system and may also rely on the improvements previously suggested in Chapter 4.

Therefore, after taking these points into consideration, the project can be viewed as being largely successful at achieving its requirements.

Chapter 6

Conclusion

This chapter will provide a summary of the project, potential areas of work for any future developments and reflect upon experience gained throughout.

6.1 Summary

The aim of this project was to develop a tool that Prof Lew could use to run commutated experiments of the red blood cell using his mathematical model. By the close of the project, all of the options for running the experiment that existed in the original implementation were replicated, and functionality was provided to allow the user to perform analysis on results or save the results locally.

However, a number of errors exist within the implementation of the model that cause inaccurate results to be calculated. Due to the reliance of the remainder of the functionality on the model providing correct output, this limits the number of cases where the entire product can be used successfully.

Therefore, it is clear that the project was successful at achieving its defined aims however would need further development to reduce the number of errors before it could be used effectively in a production environment.

6.2 Future Work

If development were to continue on the project, the following tasks have been identified as key areas where efforts could be focused:

- **Increase model code quality.**
  To improve the code quality of the model, the changes suggested in Chapter 4 should be incorporated. Additional comments would also be beneficial to explain what each section of code achieves in terms that are easy to understand for those who do not have knowledge of the underlying theory.
- **Fix known errors.**
  This should be the main priority for any future developments as fixing the known errors in the system would allow for the system to be deemed fully reliable and as such be able to be deployed publicly for a wider audience.

- **Increase model test coverage.**
  As issues discovered during the evaluation of the product where found through the use of previously unseen test cases, it would be valuable to expand the current automated test suite for the model to cover more tests. This would allow the model to be fully verified by the test harness and allow for full confidence that subsequent changes to the code base have not broken functionality.

- **Completion of the remaining functional requirements.**
  By incorporating the functionality and behaviour of the remaining functional requirements, the user experience of the application would be greatly enhanced resulting in a more effective product overall.

- **Inclusion of improvements gained from evaluation.**
  As mentioned previously during evaluation with the client, various different suggestions for improving the interface were raised. Clarifying these changes with the client and incorporating them would improve the interface and increase his satisfaction with the product.

6.3 Reflection

Overall, participation on this project was very satisfying and provided many challenges to overcome including some unique to this particular project.

Having an actual client for the project provided a real purpose behind the work to be carried out and allowed for the gaining of experience of developing a system to meet someone else's needs. Knowing that the final product would actually be useful to a large number of people made the experience very rewarding and provided a great deal of motivation throughout.

Furthermore, the project provided an excellent insight into the level of difficulty involved with updating complicated legacy code. It highlighted how error prone and time consuming the process can be and allowed for an understanding of why so many legacy systems still exist in the world today.

If development of the project could be restarted, focus would be placed on developing comprehensive automated testing suites before the implementation of any features. If this had been done for this project, substantial amounts of time that was spent performing manual testing could have been used elsewhere. Additionally, the likelihood that new errors would be discovered late into the project would be greatly reduced with test cases that cover a wider range of execution paths within the model.

Appendices

Appendix A

Installation and Configuration

A.1 Prerequisites

The installation instructions provided here will assume that you are running a machine that currently has Python and Pip installed upon it. Recommended version is 2.7.5 and you are advised that using other versions is at your own risk.

A.2 Installation

To install the system, unpackage the codebase into a folder of your choosing. Then, from within the 'cell-experiments-app' folder, run the following command in a terminal window to install:

```
pip install -r requirements.txt
```

This will install all of the required libraries for the application. Afterwards, run the following command and follow the given instructions to prepare the database:

```
python manage.py syncdb
```

A.3 Running the System

To run the system, you will require two terminal windows open within the 'cell-experiments-app' folder. In the first, the execute the following command to run the django application server:

```
python manage.py runserver
```

In the second, run the following command to start the celery task process:

```
celery -A cell_experiments_app worker
```

The application should now be accessible be visiting http://localhost:8000/ in your browser.

A.4 Deploying the System

To deploy the system, please refer to the Django[2] and Celery[1] deployment tutorials and adapt for your system set up.

A.5 Executing Tests

To execute the regression test suite, perform the following command:

```
python regression_test.py
```

Appendix B

Status Report

This appendix details known flaws in the system as of the project's close.

B.1 Alternative Screen 3 Missing

In the original program, if the user accepts all default values for the na-pump screen in the reference state, then they are presented with an alternative screen 3 in the dynamic state. This screen offers slightly different functionality, however is missing from the current system.

To rectify this issue, the model has to be updated to allow for the alternative options provided by this new screen, and the web frontend should be updated to utilise this functionality where appropriate.

B.2 Errors in Results

There are errors in the results calculated by the system for various test cases. Investigation has shown that most of these differences appear to originate from a difference in the 'concentration H' instance variable in the 'Medium' class when chelation is enabled. Therefore resolution of these errors should first investigation any variances of logic in this section of the system between the new application and the original PowerBASIC version. The following inputs should be used to verify if this issue has been resolved:

```
Screen 1: set duration of experiment to 120 min
Screen 2: set cell fraction to 0.1;
          at the "Na2H2-EGTA <ENTER 1>....." prompt, enter 1;
          at the "EGTA concentration <mM>" prompt, enter 0.9
Screen 4: set "PMg/Ca (A23187)(high=2e18)" to 2e18
```

Appendix C

Summary Log

The project went through several different stages during the course of development:

- **September-October:** The problem was defined and the initial Python implementation created.
- **November-December:** The web interface design was developed through the use of prototypes, and errors within the model implementation were investigated and eradicated.
- **January-February:** The model was extended to allow user input and refactored to become extensible. Development of the web application was undertaken.
- **March:** The system was finalized and evaluated. The project report was written.

Bibliography

[1] Deploying celery. http://docs.celeryproject.org/en/latest/tutorials/daemonizing.html.

[2] Deploying django. https://docs.djangoproject.com/en/1.5/howto/deployment/.

[3] Floating point arithmetic: Issues and limitations. https://docs.python.org/2/tutorial/floatingpoint.html.

[4] Powerbasic. http://powerbasic.com/.

[5] Ruby on rails. http://rubyonrails.org/.

[6] Dr. Chris. Red blood cells functions, size, structure, life cycle, pictures. http://www.healthhype.com/red-blood-cells-functions-size-structure-life-cycle-pictures.html.

[7] Django Software Foundation. The web framework for perfectionists with deadlines. https://www.djangoproject.com/.

[8] Josef Idris Khan. Computational modelling of red blood cells, 2014.

[9] Virgilio L. Lew and Robert M. Bookchin. Volume, ph, and ion-content regulation in human red cells: Analysis of transient behavior with an integrated model, 1986.

[10] National Institute of Biomedical Imaging and Bioengineering. Computational modeling. http://www.nibib.nih.gov/science-education/science-topics/computational-modeling.

[11] American Society of Hematology. http://www.hematology.org/Patients/Blood-Disorders.aspx.

[12] World Health Organisation. 10 facts on malaria. http://www.who.int/features/factfiles/malaria/en/, 2014.

[13] Armin Ronacher. Web development, one drop at a time. http://flask.pocoo.org/.

[14] Kelly Waters. Prioritization using moscow. http://www.allaboutagile.com/prioritization-using-moscow/, 2009.