1 Design

1.1 Methodology

When Designing software there are a number of approaches that can be taken. Most of these approaches break development down into a number of different phases and describe the order in which they should be completed.

Most methodologies were designed for teams of engineers, as this project was completed by one individual many components of the methodologies are redundant. However there are some principles and techniques which can be utilised by an individual developer and aid in organisation of a project.

1.1.1 Waterfall

The waterfall methodology is a linear approach to development. Each stage of development is completed all at once for the whole piece of software. This means that once a stage is completed there is no need to return to it and the development process flows in one direction like a a waterfall.

The waterfall approach is good because the whole project is rigorously planned and documented before coding begins. This allows major difficulties in projects to be dealt with before significant time has been invested. It also allows projects to be easily passed between developers as a new developer can look at the plan of the project. In less linear approaches there may be plans for future parts of software which are never documented.

The downfall of the waterfall approach comes when a project is not fixed or there is a a change in requirements, there is no stage in which plans can be modified as that would involve going backwards in development.

1.1.2 Agile

The agile approach is a methodology which focuses on iterative development. Deliverables are identified and designed individually, there is also a much greater focus on executable code than documentation than in waterfall

Many parts of the agile methodology are not needed in this project such the parts that deal with the customer and cooperation between developers as this is a solo project without a customer.

The most important part of agile is its ability to deal with changing scope through process. Also since testing is done at each stage it will allow the project to be easily shortened or extended based on time constraints, as after every sprint a working piece of software has been developed.

1.1.3 Spiral

The spiral model for software development is a risk driven process. The spiral method can contain aspects from the other processes.

The actions taken and the amount of work done on areas of the project are determined by what would minimize risk.

Risk can be a number of things. In an industrial setting risk could be how how spending more time on a product and delaying it's release would affect it's sales figures. In a team project, risks could consist of members of the team being unable to work on software unexpectedly because of illness. In essence, risk is anything that would be detrimental to the stakeholders in the project.

Spiral development also requires risks to be assessed at every stage of development. This would increase the work load specifically in agile as risks would need to be identified at every iteration. This would lead to a lot of repeated work that could be better spent on development or testing.

1.1.4 Chosen Approach

The chosen approach for this project was agile. The reduced amount of documentation will speed up development. Agile development also allows the scope of the project to change over time. This allowed the scope of the project to start off small and increase if time allowed it.

For example, once the PC algorithm had been implemented, if time allowed, the project could easily be extended to include the FCI algorithm.

The spiral approach will not be used for this project as it will lead to an unnecessary increase in work for a relatively low risk project. The project also has a fairly logical order of implementation so using risk minimization to determine what to implement will not be necessary.

Within the Agile methodology there are a number of popular frameworks. However, these teams are designed for teams of software engineers with clients. Since this is a solo project without a client many parts of these frameworks will not be applicable. Therefore, the applicable parts of various frameworks were chosen and compiled into a framework that will work for this project.

1.2 Requirements

Requirements were elicited at the beginning of a sprint. A deliverable was chosen and the requirements were found for that deliverable. Requirements were found by considering how the software would be used.

1.2.1 Epics

Epics describe a very high level interaction between a user and the software. They describe the general functionality of a deliverable without discussing it's implementation. Epics were the first stage of requirement elicitation as they they capture the functionality that the requirements must capture.

1.2.2 User Stories

Each epic is then broken down into user stories. The stories describe what a user must be able to do with the software during the epic. They are fairly low level and form a set of required functionality of the software.

Users stories can be used as requirements as software that would allow user stories to occur would be functional.

1.3 Sprints

Sprints are sections of development defined by deliverables. They outline the tasks required to be completed to develop a deliverable and the time frame needed.

When a sprint is completed, the scope of the project and the requirements can be updated and modified if needed.

for this project there were four main deliverables implemented and four sprints to implement them.

The first sprint consisted of implementing the χ^2 conditional independence test. This test was needed before anything else could be properly implemented.

Next, the skeleton learning software was written. The skeleton is needed for all algorithms in this project so is the next logical step from the independence test.

Then the pc algorithm was fully implemented, this algorithm was the simplest of the 3 so made sense to start with. Starting with the easiest algorithm gave me experience using networkx and the other libraries being used that would be very useful in the final sprint.

Finally, the FCI algorithm was implemented. This was more complex than the pc algorithm, however the experience with the library in previous sprints greatly helped with the more complex orientation.

1.3.1 Tasks

From the stories can be formed. Theses tasks describe the software that must be implemented to fulfil the requirements.

Each sprint had a number of associated tasks. The tasks were small and each task could be verified through testing.

Each task has associated stories so that all software can be traced back to an epic. Tracing back to an epic shows that each task will in some way contribute to functional software.

1.4 Architecture

Only the architecture for a particular deliverable was designed in each sprint. This meant that the architecture grew over time. Considerations for general extensibility of the software needed to be made in the design phase to ease design and implementation during later sprints.

A class diagram was produced, this specified the relationships between classes and the functionality of each class. This diagram evolved throughout the project as various sprints were completed.

The class diagram for this software was fairly straightforward. This is because the software did not implement a complex system. Only a few classes were needed as most of the complexity is contained in the methods of the classes.

1.4.1 Unit Testing

Test driven development (TDD) was one of the main features of the development framework. In this development, tests were created before the actual software was written. This ensured that all software that was written would be functional. It also ensures that any refactoring performed would not compromise the functionality of software.

All software was tested using the python unittest framework. In this framework, list of tests can be compiled into test cases and cases compiled into a test suite. Test cases test an individual feature of software, so can be run individually when a feature is being developed or maintained. Test Suites can then test the software as a whole.

1.5 Documentation

All software written was thoroughly documented. Every class, method and function was given an attached doc string detailing it's functionality and use.

NumPy style documentation was used. An example is shown in —. This style was chosen because it gives a useful summary of the functionality of a piece of code. It also allows other developers to easily use the code by showing exactly what arguments are taken and exactly what will be returned by a function.

Inline comments were also used to clarify areas of code within methods or functions. These comments can allow a developer to quickly and easily understand the code for the purposes of extension or maintenance.

1.6 Tools

A number of existing tools were used throughout the project to aid in development.

1.6.1 Version Control

An important part of software development is Version Control (VC). VC allows changes to be made to software without risk of losing previous iterations. At any point software can be reverted to a previous version if needed. VC also helps with agile development as deliverables can be easily identified in a VC system.

Git is a standard VC system used in development. Git was chosen as github provides free repositories which can be used to store code. This makes code accessible for anywhere and greatly reduces the risk of it being lost. Github makes git repositories incredibly easy to set up and maintain.

1.6.2 Python Libraries

Various external python libraries were used to facilitate development.

Pandas is a library containing DataFrames which allow easy manipulation of data. This includes labelling variables and calculating contingency tables which is important in the χ^2 test of independence.

NetworkX was used for graphical models, the Graph and Digraph were used and extended to store various graphs used in the algorithms. The built in functionality of greatly helped with the implementation of the PDAG and PAG classes. NetworkX also has functions for finding paths in graphs, this was especially useful for the FCI algorithm.

Finally, Scipy was used to calculate the χ^2 statistic and p-value.

1.7 Refactoring

Refactoring is a very important part of the agile framework and allows a developer to apply newly gained knowledge to old code. Refactoring consisits of rewriting functioning code to improve it.

This improvement could be one of many things. Obvious things like speed and memeory consumption may be optimised. However, refactoring can also aid in making code more readable or using a new library that is better suited to the task.

Refactoring may however fail and render previously functional code nonfunctional. However, this is easily mitigated by string unittesting and version control. Unittesting allows a developer to see whether code that has been refactored still functions in the same way as old code, this will show if refactoring has been unsuccessful. In the case where refactoring fails a developer can either attempt to identify the non-functional area of code and repair/replace it or use the version control system to easily roll back to an existing functional version of the software. These two systems hugely mitigate the risks of refactoring.