Object-oriented programming in C++. Final project: Data management system

Thomas Kite

9746983

School of Physics and Astronomy

University of Manchester

April 2018

Abstract

A C++ program for loading, storing and analysing experimental data was created. The set of priorities for this project were the following: Firstly the user experience, which has been enhanced by creating a non-linear user interface. This allows for many actions to be performed in any order possible. Secondly the generality of the code, allowing for almost any kind of experiment to be stored and analysed in the program. Thirdly the quality of the analysis, giving as much information as possible for any experiment that the user inputs.

1 Introduction

Experimental research requires managing large amounts of data efficiently. Gaining familiarity with the data gathered is essential, since this can help to guide future measurements, and eventually aid in making discoveries. It is therefore immensely helpful to have a system which assists the researcher in this goal: Efficient storage, easy manipulation and helpful visualisation of the data. This system has the potential to significantly increase efficiency when working with most types of data, since many processes can be automated.

With this goal a C++ based program has been developed. This program can read, store and analyse experimental data of various kinds. Data input can be done both by user input and by reading correctly formatted files. The system will then store this data, and allow the user to manipulate it in various ways through a helpful user interface. Upon request the program will return a full analysis and report either to the screen or to a file. This analysis includes taking averages and standard deviations, looking for correlations between certain words and numeric values, dividing data into bins like a histogram, and more.

2 Code design and implementation

2.1 Basic structure

The structure of the code comprises a hierarchy of classes. Not in the sense of inheritance, but rather classes including instances of other classes. The three classes that have been defined are *timestamp*, *measurement* and *experiment*. These classes with their member data are shown in table 1.

timestamp	**template <class V>** *measurement*
vector<int> *date*	**V** *value*
static size_t *timestampCounter*	**V** *error*
	V *systError*
	string *unit*
	timestamp *time*

template <class V> *experiment*
size_t *measurementCounter*
measurement<V>* *measurements*
size_t *errorCounter*
measurement<string>* *measurementErrors*
string *title*
vector<pair<string, size_t>> *orderVector*
vector<experiment<double>*> *numericCorrelatedVector*
vector<experiment<string>*> *stringCorrelatedVector*

Table 1: A table outlining the class structure of the program. The three classes are shown, with their respective member data. The variable type is shown in bold font, while the variable names are in italics. The asterisk indicates the object a pointer. Note that *timestamp* is used in *measurement*, which in turn is used in *experiment*.

The *timestamp* class contains the date and time of a measurement. It contains methods for checking if a timestamp is of valid format and other useful functions. The *measurement* class is a template which contains a value, an error and a systematic error of a general type V. It also contains a unit and a timestamp. The *experiment* class then has containers of these measurements, the memory for which is allocated dynamically. An experiment contains measurements of a general type V, and a set of string type measurements for erroneous results in the data. These will be detected when the value is simply "error". Counters indicating the number of measurements and errors are included in the class, as well as a title and a vector of pairs that records the order in which measurements were input. Finally there are vectors pointing to other correlated experiments. The meaning of these correlated experiments will be discussed in section 2.2.

Throughout the code there is a difficult balance between generality and functionality. An attempt has been made to allow for the most general types of experiments possible, while still giving ample functions able to work with them. To do this the main classes have been made templates, and the principle measurement types were chosen to be the following:

- Numeric measurements: Any measurement that contains a numeric value. For example an experiment measuring the charge of the electron. For this type of measurement the general type V was set to be type double.
- String measurements: Any measurement that records an outcome specified by a word. For example an experiment looking at the types of particles passing through a detector. As the name suggests these objects get created with type string.
- Correlated measurements: Correlations of any combination of the previous two types. For example in an experiment measuring pressure and temperature. This isn't actually a separate object in the program, but rather occurs in the correlation of experiments.

2.2 User Interface

One major design feature was the user interface being non-linear. Menus can be entered and exited with ease, and most options can be undone. Upon starting the program a main menu will open, which presents a series of actions accessible through special command keywords. Similarly the user can request to see the full list of current experiments, and perform some actions on them individually. This is shown in figure 1, with the full set of options in each menu. A brief description of the most important functions follows.

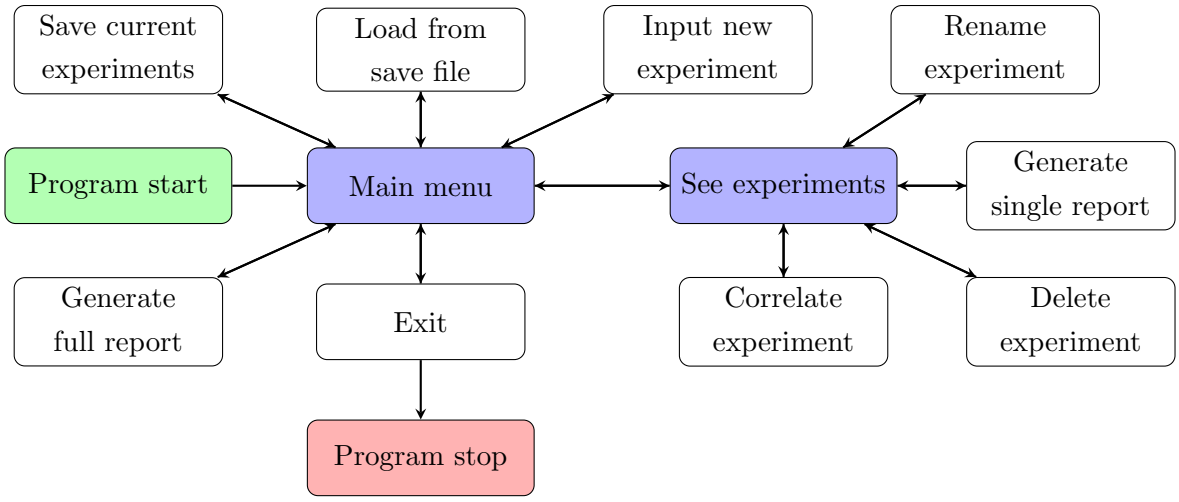

Figure 1: A flow chart indicating the basic structure of the code. Green and red nodes indicate the start and end of the program respectively. The blue nodes indicate an option that opens up a menu with various options attached.

New experiment: This function allows the user to input new data into the program. This can be done from a file or from the keyboard. In the former case the program can automatically determine what type of experiment it is. Whichever type of input is chosen the program will use the same code to create the experiment using iterators and a lambda function.

Make report: This function allows the user to create a report of either an individual experiment or all experiments at once. The functions that generate the report return stringstreams, which can easily be output to screen or keyboard depending on the user's choice. These reports will differ depending on the nature of the experiment, but can give information about averages, standard deviations, linear fits, frequency of occurrence of a specific value etc.

Correlate experiments: This function allows the user to correlate experiments, meaning the measurements of one correspond in some way to the measurements of the other. This is done by first checking they have a 1:1 correspondence of measurements. The program will then link said measurements using the order in which they were input. i.e. the nth measurement in one experiment, corresponds to the nth measurement in the other. This, perhaps slightly convoluted, way of achieving related measurements was chosen for a few reasons. Firstly, it allows for the correlation of any number of experiments with no extra code, since you can simply run the correlate function multiple times. Secondly, this avoids the need to implement a new file input system. This would require new formats for the file, and would be complicated for an arbitrary number of correlated experiments, meaning a lot of extra code. Note that when correlating numeric experiments the first experiment chosen is taken to be the independent variable.

Save & load: These functions allow the user to save and load the current state of a program to a specially formatted .txt file. This is faster than inputting all the data again for two reasons. Firstly, many experiments can be in the save file, and will all be loaded at once. Secondly, the program doesn't need to check the measurements for validity, given that they were checked before being saved. The assumption has been made that the user would not tamper with the file.

2.3 Reading data

The format the data can take depends on the type of measurements being input:

Numeric: *value, error, systematic error, unit, timestamp*
String: *value, timestamp*

The timestamp was chosen to follow a decreasing format:

Timestamp: *year/month/day hour:minute:second*

Note that the values not specified by the string input get set to "N/A". If any part of a measurement contains the string "error" then it will be stored as a measurement of value "error" and all other properties "N/A" except timestamp.

Both keyboard and file inputs demand that all parts of the input be of the correct type, and the timestamp valid. No attempt was made to correct mistakes, as this could cause problems with misrepresenting the data later. This includes finding a rogue string amongst numeric data, or vice versa.

2.4 Analysing data

The analysis of the data happens upon the user requesting a report. This section of the code presented some difficulties since, as the programmer, it is impossible to know exactly what the user expects from their data. Due to this an attempt was made to be as general as possible, giving any information that could perhaps be relevant.

Numeric report: A clear starting point for these reports is giving the average, standard deviation and error on mean. This is complemented by giving the distribution of values by dividing the data into bins.

String report: For these reports it is not possible to give averages and standard deviations. Instead the most useful information is the frequency with which a certain outcome occurs.

Correlated report: When various experiments are correlated the situation becomes quite complicated. In principle there are infinitely many combinations of numeric and string experiments. In the code presented there was a focus on correlations of two experiments, specifically numeric-numeric and numeric-string.

3 Results

The program successfully reads in data from both file and keyboard. It will detect errors and inconsistencies in them, and warn the user. Individual experiments can then be renamed, deleted and correlated with other experiments by use of simple commands. These can be saved to a file, and loaded on the next use of the program, which is faster and more convenient than adding experiments individually. All of these actions are handled smoothly by the user interface, no matter which order they are performed in.

A series of example data files are provided with the program. These attempt to illustrate most of the program's functionality.

- *normDist.dat* and *coinFlip.dat* show basic numeric and string experiment types respectively. The first is a simple normal distribution generated to have an average of 50 km and a standard deviation of 5 km. The second is the results from a series of coin flips to see if there is some bias in the results.
- *pressure.dat* and *temperature.dat* are two numeric experiments which are designed to be correlated, with temperature being the independent variable. This will give a good linear fit upon generation of the report.
- *energy.dat* and *particle.dat* are numeric and string types respectively and are, once again, made to be correlated. They contain a series of particle detections with their associated energies.

All of these experiments are also contained in an example save file accessible through the title *saveFile*. Examples of the generated reports are shown below.

Numeric:

Experiment: NormDist	The distribution of the values is the following:
Type: numeric	(36.2612-39.0528): 6 entries (= 2.06186%)
Contains 291 valid measurements and 9 errors.	(39.0528-41.8444): 10 entries (= 3.43643%)
Average: 49.5297±0.282898 km	(41.8444-44.636): 27 entries (= 9.27835%)
Standard deviation: 4.82588 km	(44.636-47.4276): 51 entries (= 17.5258%)
	(47.4276-50.2191): 77 entries (= 26.4605%)
	(50.2191-53.0107): 49 entries (= 16.8385%)
	(53.0107-55.8023): 38 entries (= 13.0584%)
	(55.8023-58.5939): 22 entries (= 7.56014%)
	(58.5939-61.3855): 9 entries (= 3.09278%)
	(61.3855-64.1771): 2 entries (= 0.687285%)

String:

Experiment: CoinFlip	The most commmon entries were the following:
Type: string	Value tails found 104 times (= 53.0612%)
Contains 196 valid measurements and 4 errors.	Value heads found 92 times (= 46.9388%)

In the reports for correlated experiments all of the same information is given as for individual ones. Some additional information is then given regarding the connection between each experiment, as shown below.

Numeric-Numeric:

A linear fit on these two experiments gives the following:
Pressure = m × Temperature + c
With m = 4.99902 +/- 0.00244952
and c = 1.99098 +/- 0.281785.
This fit gives a chi squared value of 208.269 (when reduced 1.05186).

Numeric-String:

The average numeric value corresponding to each string outcome is the following:
W: 80.2393 GeV
Z: 91.3956 GeV

4 Discussion and conclusion

The goal of this project is in essence very general, and hence the scope can be as large as the programmers imagination. Due to time constraints the scope of the code presented had to be somewhat limited, and many ideas were simply too broad to implement. Priority has been given to producing a program that is user friendly, allowing for quick access to the data and easy manipulation of it. Another focus of the code is generating complete reports, giving as much information as possible. Perhaps not all the information given will be relevant to each experiment due to the general approach of the project. Because of this the best use of this program is as a first look at data, giving some quick information and an idea of what properties it has. A more thorough analysis of the data would then be undertaken by more specialised tools.

One major feature that could be implemented is the the creation of basic graphs and histograms. This was attempted by creating and running python scripts from the console. But this could not be done in a way that any user's computer could handle, since .py files may not be recognised by the system. Another feature that could be implemented is the ability to add various correlated experiments at once with a defined file format. This option was rejected for the code presented as it would require a completely new file input system, and an amount of additional code not justified by the increase in functionality.

Word count: 2160.