A PROGRAMMING LANGUAGE FOR BUILDING CHARTS

STUDENT NAME: Thomas Shaer

STUDENT ID: 1905941

PROGRAMME: BSC COMPUTER SCIENCE FT

PROJECT SUPERVISOR: PAUL BLAIN LEVY

WORD COUNT: 9858

1 Content

2 Abstract

Building powerful and expressive charts can be tedious with existing tools such as Excel. These tools often rely heavily on user input through high-level user interfaces which often restrict the scope of the software. An alternative approach to building charts is with programming languages. Programmatically building charts affords chart builders the advantages of programming such as mathematical and logical expressiveness while reducing the need for the restrictive layer of user interface abstraction.

While some existing programmatic charting software exists, they are usually targeted toward specific areas or industries and come in varying levels of difficulty. This project is designed to solve this problem by providing an alternative to existing programmatic chart building software, following two key themes: simplicity and general usability. The tool is designed not to replace software such as Excel, but to act as an alternative to its charting capabilities with the ability to input and output data back into a format that Excel can understand.

3 Introduction

3.1 Project aims

While Excel is a great tool for visualising and managing data, I believe that its charting capabilities could be improved significantly. It is somewhat tedious to build charts in Excel, involving highlighting and formatting data, navigating the UI to find specific settings, difficulty in expressing complex logic, all to produce charts that lack substantial interactability such as zooming and panning, which allows users to explore and gain better insight into their data**.**

A different approach to building charts has been explored by several different software. The domain of this software can best be described as "charting languages", where a programming language is utilised to build charts as opposed to building charts via a UI. As all charts are concerned with plotting data, and programming languages are a method of manipulating data, combining these two areas removes many of the restrictive qualities that non-programmatic charting software has.

This project aims to create an all-in-one software for programmatically creating charts via a custom-built programming language, designed specifically to provide an alternative to Excels chart building features. The programming language is an array language[1], a paradigm where computations apply to entire arrays at once, often in parallel, which is necessary for charting since chart data is in the form of data series that can be interpreted as an array.

The software is delivered in the form of an IDE (Integrated Development Environment), which allows the user to write and test code. Although the software utilises a programming language, one of the primary aims of the project is to keep the language as beginner-friendly as possible such that it is accessible to non-technical users. Additionally, since the project is aiming to act as a replacement for Excels chart building features, another important aspect is compatibility with Excel. To act as an effective replacement, it is essential that the user can import and export data to and from Excel.

3.2 Literature review

The main technical themes of the software were interpreters and array programming. I planned to implement the language through an interpreter, and array programming was the most suitable paradigm for working with charts.

A useful document for understanding the basic concepts of implementing an interpreter was Ruslan's 19-part blog titled "Let's Build A Simple Interpreter"[2]. It provides in-depth details about how to implement a basic programming language in the form of an interpreter. This was an incredibly useful document for understanding the phases of an interpreter including parsing, semantic analysis, and interpretation.

Calvin Lin[3] provides a detailed description of the benefits of array programming languages and an overview of how they operate regarding concurrent manipulation of several arrays. This is a useful document for understanding how an array language operates from a high-level view and the performance benefits they provide through performing entire calculations at once. Although it is a useful document for understanding array languages, it does not provide low-level technical details on how to implement them.

3.3 Software review

The software review identified key shortcomings of existing programmatic charting software. This information was used to shape the functional and non-functional requirements of this software. The review was concerned only with software rather than libraries/modules such as Python's Matplotlib[4] and Plotly[5].

3.3.1 Excel's VBA

Excel also supports the ability to create charts programmatically through its integrated programming language called VBA[6], which is designed to allow the user to interface with many of its applications. Using VBA, one can programmatically create charts within Excel. However, the syntax of the language is far from beginner-friendly and involves a verbose object system where users must declare chart and series objects with many parameters and options. The complexity is not entirely surprising since VBA was designed for interfacing with all of Excel's features rather than purely charting.

```
Sub create_embedded_ScatterPlot()
Dim oChartObj As ChartObject

Set oChartObj = ActiveSheet.ChartObjects.Add(Top:=10, Left:=100, Width:=250, Height:=250)
With oChartObj.Chart
    .ChartType = xlXYScatter
    .SeriesCollection.NewSeries
    .SeriesCollection(1).Name = "My Data Name"
    .SeriesCollection(1).XValues = ActiveSheet.Range("B1:B10")
    .SeriesCollection(1).Values = ActiveSheet.Range("A1:A10")
End With
End Sub
```

Figure 1 Plotting a basic scatter graph using Excel's VBA. The example is taken from https://wellsr.com/vba/2018/excel/automatically-create-excel-charts-with-vba/

3.3.2 MATLAB

MATLAB[7] is a popular example of a programming language that also supports the vector paradigm[8] with charting support. It is a multi-purpose tool, allowing for all kinds of numeric computations with many extensions to further the base functionality. The language itself is very beginner-friendly, with simple syntax and naming conventions for built-in features.

However, I believe MATLAB is too general-purpose to act as an effective replacement for Excel's charting feature. It is multiparadigm, with vector programming being just one of those paradigms and with the amount of features MATLAB supports, I believe that it's too broad and complicated for a general use charting software.

3.3.3 Tradingview's Pine Script

Tradingview is a forum for trading and investment. One of its features is a tool that allows users to programmatically create indicators and trading strategies via its scripting language called Pine Script[9]. The code is executed on their back-end servers, where the results are returned and plotted on a web browser chart. The language features an extensive build-in function and variable library

and a beginner-friendly python-like syntax. Pine Script is in many ways an ideal charting language for these reasons.

However, it is targeted exclusively toward finance and trading, all data is OHCL[10] (open-high-low-close) price data for various financial instruments and tickers. Tradingview tries to restrict users from obtaining this data in its raw form (as pricing data is expensive) and thus the user cannot import/export data which makes in-depth analysis difficult since the only means of output is plotting data onto the chart. The platform exists as somewhat of an isolated ecosystem and cannot be used as a general-use tool for charting.

4 Design

The analysis of existing programmatic charting software provided an idea of the direction the software should take, including minimum required features that are prerequisites for the software to function, as well as several nice-to-have features that would add to the level of polish for the software.

4.1 Functional Requirements

The requirements mentioned below exist to specify the minimum features necessary to achieve basic functionality for the software to be able to satisfy the project aims.

Language functional requirements

- The language must support number and Boolean types.
- The language must support null values.
- The language must support arithmetic and comparisons.

IDE functional requirements

- The IDE must be stable and effective GUI to use the software.
- The IDE must display language output on its chart.
- The IDE must support inputting and outputting of user data.

4.2 Non-functional Requirements

There are several non-functional requirements that are not required for the base software to function, however, would significantly extend the usability and polish of the software.

- The language should support methods.
- The IDE should allow users to export charts as an image.
- The IDE should highlight errors in the user's code.
- The software should allow the user to plot multiple different charts.
- The software should have cross-platform compatibility between Linux, macOS and Windows.

5 Architecture

The software was written in C++ and designed to run as an offline desktop application. There are two main components to the software, the language itself and the IDE environment which is how the user interacts with the software. Most of the time was spent working on the language, which had significant challenges, largely due to it being implemented as an array language. The lines of code and language features can be found in the appendix, under language features and lines of code breakdown.

5.1 Language

The programming language is implemented as an interpreted language rather than a compiled language[11] which allows for easy integration with the rest of the software. The language is a statically typed language that has a simple syntax like Python[12] but differs from many existing languages due to being a single dimension array language (see Runtime Environment) which had huge implications on the design of the entire language.

The implementation of the interpreter is split into three phases:

- **Lexing[13]/parsing[14].** This is the first stage primarily concerned with translating the input text (i.e. the code) into the AST which is the supporting data structure of the entire language.
- **Semantic Analysis[15].** This is the second stage which is concerned with all semantic verification of the user's program, as well as decorating the AST with information to help with the execution of the code during the runtime.
- **Interpretation (execution).** This is the final stage, utilising information gained from semantic analysis to perform the data computations and desired side effects[16] such as output. This stage is described extensively in the runtime section.

5.1.1 Abstract Syntax Tree

The central data structure that supports languages is an Abstract Syntax Tree[17] (AST), which is implemented as a node tree. The AST is created via semantic actions during the parsing phase when grammar is matched. An AST node can be classified as two distinct nodes, an "Expression" or "Statement". The primary difference is that "Expression" nodes return a type and a value during the semantic analysis and interpretation phase. The full list of AST nodes can be found in the appendix under AST nodes.

5.1.2 Syntax & Grammar

One of the aims of the software was for the language to be as simple as possible such that people with less coding experience can quickly pick up the language. I modelled the syntax of the language of Python since it is often regarded as one of the easiest languages to learn. Examples of the syntax of the program can be found within the language features table.

5.1.2.1 Syntactic sugar

I made heavy use of syntactic sugar[18] in my language. The clearest examples are that all binary, unary and assign operators are simply syntactic sugar, usually translating into a method call AST nodes.

For example, the following input:

```
x = 2 + 2
```

Is interpreted as:

```
x = operator+(2, 2)
```

There are many advantages to using syntactic sugar, mainly the ability to provide a wider range of syntax without having to create entire new implementations (e.g., AST nodes, semantic rules, and execution logic), one can simply reuse existing systems such as the built-in method system, which is extensively used for syntactic sugar because it already contains type validation logic and interpretation support. All operators are implemented as syntactic sugar for built-in methods (Figure 2).

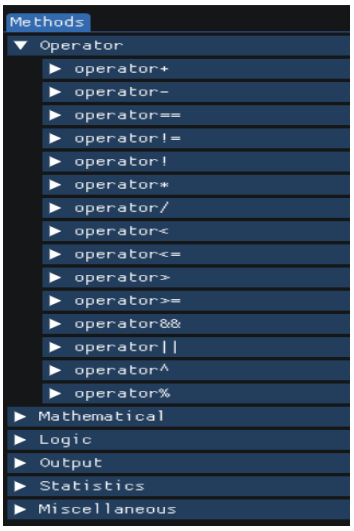

Figure 2 Auto-generated method documentation all binary/unary operators.

This is convenient because we can then overload a method to provide different implementations of these types. For example, the equality operator is overloaded three times, allowing for all three types to be compared for equality as seen in Figure 3.

In addition to this, we can provide syntax for all assign operators while not providing a unique implementation for it. An example of this is the += operator which when parsed generates the equivalent code:

```
x += 2
```

```
x = operator+(x, 2)
```

We can also provide multiple unique tokens that generate the same AST nodes. For example, the AND operator is recognised by both “&&” and “and”.

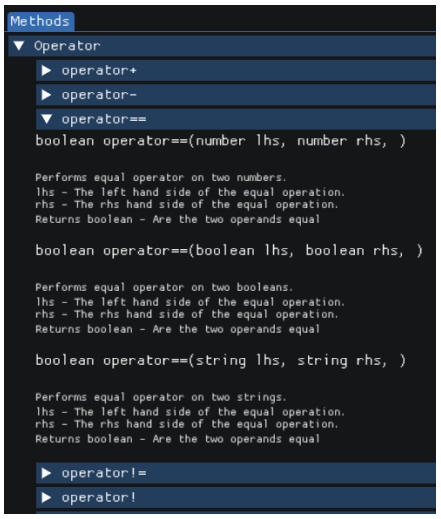

Figure 3 Auto-generated method documentation showing multiple overloads for equality operator for each different type.

5.1.2.2 Operator Precedence

In terms of operator precedence, I modelled the precedence from C++’s operator precedence rules[19]. The table can be found in the appendix under operator precedence.

5.1.3 Typing system

The language is implemented as a statically typed language[20] utilising type inference to assign types to variables. This means that once a variable is assigned a type, it cannot change. The type of the variable is inferred from the first expression that is assigned to it.

There are several reasons why I implemented the language as statically typed.

- **Performance.** Dynamically typed languages require the runtime environment to check what type a value is when an operation occurs. This causes performance problems. Additionally, since our language is an array language the code is executed multiple times (see runtime environments) therefore each execution needs to be quick otherwise it will take a long time to finish.
- **Fewer runtime errors.** Debugging runtime errors in an array language (see runtime environments) is difficult because data is hard to analyse since code is executed in multiple passes. Therefore, where possible, I would like to minimise runtime errors, and one potentially large source of runtime errors is typing related errors. By making the language statically typed, all typing errors can be caught before the code reaches the runtime resulting in fewer possible runtime errors.
- **Data types are rarely mixed in Excel.** Excel users rarely mix different data types in a series. This is mainly for convenience but also so formulas can be designed for a single type for a series of data. Since our program can input and export data to and from Excel, it does not make sense to support the mixing of types.

One of our functional requirements was to support at least a number and Boolean type. This is so that the language can support arithmetic and comparisons, which is another functional requirement. There are three main types of the language.

- **Number type**. This type's value is implemented in C++ as a **float**. We call it a number as opposed to a float to make it more understandable for non-technical persons. The number type is used to represent all numbers within the software.
- **String type**. This type's value is implemented in C++ as a **std::string**[21]**.** The string type allows for text input into the program.
- **Boolean type**. This type's value is implemented in C++ as a **bool.** It is the truthy type of expressions.

In addition to these main types, there is support for **constant** types for **string** and **number**. Constant types are used to provide a semantic guarantee that the runtime value of a type will never change. They use the same C++ implementation as their non-constant counterpart, except that the only way to create a constant type is through literal values. For example, if a **NumberNode** is visited during semantic analysis, it will return the type of **number constant.** The same is true for **StringNode.** However, if a constant type is assigned to a variable, the variable will be assigned the non-constant version of the type. This is because a variable can have its value changed through reassignment, and therefore we cannot provide a semantic guarantee that the value will not change. However, this proves to be inconvenient for reusability (see Variable Descriptors).

Constant types are useful when we want to gain access to a value before executing it at runtime. It is used for identifying the charts to plot data (see charts).

As specified in the functional requirements we must support a **null** value for each type. This is because it is common to have gaps in data, and since it's necessary to maintain the same size arrays

for parallel computations, we need to fill these gaps with a value. Thus, all types within our language have a null version of their value.

In addition, we also have the **void**[22] type. This type is used just to specify that a function has no return type, and therefore if an attempt is made to assign a void returning method call to a variable, an error will be thrown.

5.1.3.1 Method system

A non-functional requirement of the software was for the language to support methods. The language I have written does not support user-defined methods. This is largely due to time constraints but also early uncertainties surrounding the runtime environment (see short term improvements).

Instead, I have written an extensive standard library[23] of methods containing 63 built-in methods. This library contains some essential methods such as output methods, but also many additional methods ranging from logical, mathematical, and statistical methods. Each one of these methods is implemented in C++ via a **MethodSymbol** object that allows for the method to be callable by the users when a **MethodCallNode** is matched with one of these method symbols.

The process of matching a **MethodCallNode** to a built-in method is dependent on several pieces of information.

- **Method name.**
- **The number of parameters.**
- **The types of parameters.**

Built-in methods can be implemented in three forms:

- **Positional Methods[24].** To match this method, you must provide the right name and the right amount/types of parameters.
- **Overloaded Methods[25].** My language supports multiple different positional methods with the same name. When an overloaded method is called, it will check every positional method for a match. This is very convenient for methods such as **isnull()** which are used to check if a value is null or not. Since all 3 types have a null value, we can overload the same **isnull()** method with 3 unique implementations for each type. Using overloaded methods has reduced the standard library from **82** to **63** unique methods.
- **Keyword Methods.** This is a positional method with extra optional parameters that the user can choose to pass in. If the user does not pass in the parameters, a default value will be used. This feature is especially convenient for methods with large signatures that the user may not want to pass in every time e.g., **plot()** which takes two optional parameters, “line name” and “chart id”.

5.1.3.1.1 Adding new methods

The built-in standard library for the language is made up of 63 methods, which when including all overloads amounts to 82 unique implementations of built-in methods. Therefore, it was necessary to establish a convenient system for creating and registering these methods so new methods could

quickly be added to the system. To register a method, all a developer must do is call one method. Consequently, this means someone can implement a new method without understanding the underlying implementation logic supporting the method system. Thus, the standard library can very easily be extended with additional methods.

Below is a complete example of implementing a new method into the standard library. The method in question is an average method, that given two number arguments will return the averaged result of it. There are 5 steps to implementing a method into the standard library. For each step, I mention the file where the code should be placed, and examples of doing so.

1. Creating a new class extending one of the **MethodSymbol** base types: **PositionalMethodSymbol** for standard method matching or **KeywordMethodSymbol** for methods with optional parameters. In this example, **PositionalMethodSymbol** is used. As a matter of consistency, and to help with compile-time - no real logic should be written here but rather declared in the respective files below. **(methodimplementations.h)**

```
11
12  // declare built-in method average
13  class  MethodAverage : public PositionalMethodSymbol {
14  public:
15
16      // method constructor
17      MethodAverage();
18
19      // semantic analysis (abstract method)
20      // performs type checking of arguments and returns a type
21      const TypeSymbol* semanticAnaylsis(MethodCallNode* methodCallNode, std::shared_ptr<SymbolTable> symboltable);
22
23      // interpretation (abstract method)
24      // executes method logic and returns a value
25      ExpressionValue interpret(const unsigned int tick);
26
27      // clone method (abstract method)
28      // returns pointer copy of this method
29      virtual MethodAverage* clone() {
30          return new MethodAverage();
31      }
32
33  private:
34      // pointers to where the runtime argument values will be stored to avoid runtime lookups
35      NullableValueNumber* value1;
36      NullableValueNumber* value2;
37  };
38
```

Figure 4 Registering a new method: Declaring the MethodSymbol class

2. Providing the constructor of the class, which forces the developer to register the semantic signature of the function and descriptions for the auto-generated documentation. This is done by providing the **MethodSymbol**'s constructor which will ask for the name, description, list of parameters (via **ParameterSymbol**) and a return type (**ReturnSymbol**). **(methodimplementations.cpp)**

```
9
10  // constructor of method average is where we specify the "signature" of the method
11  // i.e. method name, types of parameters and return type
12  MethodAverage::MethodAverage() : PositionalMethodSymbol("avg",
13      // method description (for auto-generated documentation)
14      "Returns the average of two numbers.",
15      {
16      // parameter symbols, takes a type, a identifier and a description (for auto-generated documentation)
17      ParameterSymbol(TypeInstances::GetNumberInstance(), "value1", "Argument 1 for average."),
18      ParameterSymbol(TypeInstances::GetNumberInstance(), "value2", "Argument 2 for average.")
19      },
20      // return symbol, takes a type and a description (for auto-generated documentation)
21      ReturnSymbol(TypeInstances::GetNumberInstance(), "The average value")) {}
22
```

Figure 5 Registering a new method: Providing the constructor to specify the signature of the method

3. Filling out the **semanticAnalysis** method, which is executed during the semantic analysis phase of the program, where the **MethodSymbol**'s type checking logic should exist and is also an opportunity to do any other initialisations related to the method. The developer should call the extended **MethodSymbol**'s method for **semanticAnalysis,** which contains the standard logic for matching a method, as well as decorating the **MethodCallNode** AST node, and to return its result which will be the **ReturnSymbol** type specified in the constructor.

(methodimplementationssemantic.cpp)

```
const TypeSymbol* MethodAverage::semanticAnaylsis(MethodCallNode* methodCallNode, std::shared_ptr<SymbolTable> symboltable) {
    // execute generic type checking
    const TypeSymbol* returnType = PositionalMethodSymbol::semanticAnaylsis(methodCallNode, symboltable);
    // save pointers to locations where the runtime argument values will be stored
    value1 = boost::get<NullableValueNumber>(&methodCallNode->argNameToArgumentSymbol["value1"]->expressionValue);
    value2 = boost::get<NullableValueNumber>(&methodCallNode->argNameToArgumentSymbol["value2"]->expressionValue);
    // return the method type of the method
    return returnType;
}
```

Figure 6 Registering a new method: Defining the "semanticAnalysis" method to allow for type checking and saving the location of where the runtime argument values will be stored.

4. Fill out the interpret method, which is executed every tick in the interpretation phase. This is where the runtime logic of the method goes including the value it will return. Care must be taken to account for any arguments with null values, as a matter of convention, it is recommended to return null when an argument is null. The return value will be returned by the method of every execution of the interpreter. **(methodimplementationsinterpreter.cpp)**

```
ExpressionValue MethodAverage::interpret(const unsigned int tick) {

    // if any argument is a null, return null
    if (!value1->value || !value2->value) {
        return NullableValueNumber();
    }
    // return new number value
    return NullableValueNumber((*value1->value + *value2->value) / 2);
}
```

Figure 7 Registering a new method: Defining the "interpret" method to provide the runtime logic that will be executed every at every tick.

5. Finally, the user must register the method in the global symbol table[26] which will make the method callable in the language. The symbol table can be identified by the **SymbolTable** member **SymbolTable::GLOBAL_SYMBOL_TABLE**. This is a simple one-line method called **registerMethod** that should be called within the body of the symbol table, the method takes a method name and **MethodBucket** object as a parameter. A **MethodBucket** is simply a wrapper for the **MethodSymbol** object that mainly exists to group different implementations for a method. **SingleMethodBucket** is used for standard methods, while **OverloadedMethodBucket** is used for overloaded methods, which requires a list of **PositionalMethodSymbol** objects. **(symboltable.cpp)**

```
// global symbol table
std::shared_ptr<SymbolTable> SymbolTable::GLOBAL_SYMBOL_TABLE = std::make_shared<SymbolTable>(

    // method declarations
    std::map<std::string, MethodBucket*>(
    {

        // registers the method on the global symbol table
        registerMethod("average", new SingleMethodBucket(new MethodAverage(), METHOD_CAT::MATHEMATICAL)),


        registerMethod("plot", new SingleMethodBucket(new Plot(), METHOD_CAT::OUTPUT)),
        registerMethod("tick", new SingleMethodBucket(new GetTick(), METHOD_CAT::MISC)),
        registerMethod("mark", new SingleMethodBucket(new Mark(), METHOD_CAT::OUTPUT)),
        registerMethod("text", new SingleMethodBucket(new Text(), METHOD_CAT::OUTPUT)),

```

Figure 8 Registering a new method: Calling the "registerMethod" function to make the new method accessible from the global symbol table (i.e. the standard library)

5.1.4 Runtime environment

5.1.4.1 Vector language

The programming language can be described as a vector/array programming[27] language where operations are performed on entire arrays at once. This is convenient when working with charts because the series data plotted onto them can be treated as arrays. Although my language is designed to operate as a vector language, the code is executed in multiple passes (addressed below) and therefore does not fit the exact definition of a vector language. However, it still adheres to the array paradigm.

5.1.4.1.1 All variables are arrays

All variables are implemented as single dimension arrays (in the coming section I often refer to these arrays as "internal arrays", which are present in variables and built-in-method implementations that require keeping track of history). This is not essential for all charting languages (see evaluation), arrays can be isolated to functions that require history (moving averages, standard deviations etc.), but since a requirement of the language was to export data back into Excel, it is convenient to store all historical values passed through a variable within its own array. This is so we can easily and precisely export data by extracting the array from a variable. However, this decision had implications for memory usage that are explained more in the evaluation.

However, a consequence of all variables being arrays is that to compute array operations, the user's code is executed many times during the interpretation phase unlike in a traditional programming language where the user's code is executed once. Every execution will assign a value to an internal array at index *i*. The number of times to be executed is dependent on the largest size of the input array into the program or user input via max ticks. This design is justified in the section below, but Figure 9 and Figure 10 show how the user's code is interpreted by our language.

```
average = 1.2

rainfall_above_average = rainfall > average
```

Figure 9 Code a user might write to calculate if the rainfall data is above an average

```
for(int i = 0; i < data.size(); i++) {

        average[i] = 1.2

        rainfall_above_average[i] = rainfall[i] > average[i]

}
```

Figure 10 Pseudocode representing how the backend of the language executes that same code

5.1.4.1.2 Why code is executed in multiple passes

Consider that the following variables are loaded into the program:

1. Amount of rainfall (hours) per day **rainfall**
2. Amount of sunshine (hours) per day **sunshine**

The data might look like this:

Index	0	1	2	3	4	5	6	7	8
Rainfall(hours)	4	3	5	2	0	1	3	6	4
Sunshine(hours)	1	0	0	3	4	6	2	0	1

If we wanted to find out on each day if it rained more hours than it was sunny, the input code would look like this:

```
more_rain_than_sun = rainfall > sunshine
```

This would result in a new Boolean array being generated like so:

Index	0	1	2	3	4	5	6	7	8
Rainfall(hours)	4	3	5	2	0	1	3	6	4
Sunshine(hours)	1	0	0	3	4	6	2	0	1
more_rain_than_sun	**True**	**True**	**True**	**False**	**False**	**False**	**True**	**True**	**True**

This works by comparing each value of the rainfall array with the value of the sunshine array at the same index. The pseudocode of the internal implementation of working out **more_rain_that_sun** might look like Figure 11.

```
int[9] greaterComparisonOperator(int[9] lhs, int[9] rhs) {
        int[9] temp;
        for(int  i = 0; i < 9 < i++) {
                temp[i] = rainfall[i] > sunshine[i];
        }
        return temp;
}
int[9] rainfall = load(…);
int[9] sunshine = load(…);
int[9] more_rain_that_sun = greaterComparisonOperator (rainfall, sunshine);
```

Figure 11 Pseudocode Internal implementation of a comparison function based on two arrays

5.1.4.1.2.1 Mixing of scalar values and arrays

This works well for performing an operation on two variables that contain arrays. However, a complication appears when we try to implement the same but this time with a scalar value e.g., a literal[28] such as the number "3".

Suppose we try and implement the following:

```
alot_rain = rainfall > 3
```

Intuitively, the variable **alot_rain** will contain a Boolean array like so:

Index	0	1	2	3	4	5	6	7	8
Rainfall(hours)	4	3	5	2	0	1	3	6	4
alot_rain	**True**	**False**	**True**	**False**	**False**	**False**	**False**	**True**	**True**

The implementation of this in pseudocode might look like Figure 12:

```
int[9] greaterComparisonOperator (int[9] lhs, int rhs) {
        int[9] temp;
        for(int  i = 0; i < 9 < i++) {
                temp[i] = rainfall[i] > rhs;
        }
        return temp;
}
int[9] rainfall = load(…);
int[9] more_rain_that_sun = greaterComparisonOperator (rainfall, 3);
```

Figure 12 Pseudocode internal implementation of comparison function based on an array and a scalar

However, note the signature of the **greaterComparisonOperator** method in Figure 12. The LHS is an array and the RHS is a scalar value. Therefore, 4 overloads will have to be implemented for the **greaterComparisonOperator** to consider the possible different combinations of input types (Figure 13).

```
int[9] greaterComparisonOperator (int[9] lhs, int[9] rhs);
int[9] greaterComparisonOperator (int[9] lhs, int rhs);
int[9] greaterComparisonOperator (int lhs, int[9] rhs);
int[9] greaterComparisonOperator (int lhs, int rhs);
```

Figure 13 All the required internal implementations to support a comparison function. Note how the amount of implementation is exponential to the number of arguments.

This means that for every built-in method that takes a value, each internal implementation will have to be programmed to account for either a scalar or array input value. This results in an exponential number of implementations given the number of parameters. This is a bad design decision.

5.1.4.1.2.2 Implicit conversion of scalar values into arrays

Standardisation of the input parameter's dimensions is necessary to avoid this code duplication.

One way of standardising the input types is to convert all scalars into arrays, by implicitly converting non-arrays into an array by creating a temporary variable for each one.

E.g., for the code below

```
alot_rain = rainfall > 3
```

We can convert 3 into a variable **temp_3** and use that for comparison with the rainfall array: e.g.

Index	0	1	2	3	4	5	6	7	8
Rainfall(hours)	4	3	5	2	0	1	3	6	4
temp_3	3	3	3	3	3	3	3	3	3
alot_rain	**True**	**False**	**True**	**False**	**False**	**False**	**False**	**True**	**True**

This means that we only need one internal implementation for each operation since we standardise the inputs into arrays.

However, an issue with this approach is that it wastes memory. Suppose the **alot_rain** array was of size 100,000. Therefore, for a comparison to the literal 3, we would need to create an equal array of size 100,000 for the temporary variable **temp_3**. Additionally, we might use many literals in one expression e.g.

```
alot_memory = (((((rainfall + 10) + 31) + 41) + 312) + 132)
```

In the expression above we use 5 literal values, therefore we will have to produce 5 extra arrays of size 100,000. So, while this solution allows for standard input for implementations, it wastes a lot of memory.

5.1.4.1.2.3 Treating arrays as scalar values

A solution to this problem is to do the opposite of the above. Instead of applying operations to entire arrays at once, we can do it on their elements. Therefore, we can have just one implementation for each operation, standardising all input values as non-arrays, and just call that code *n* number of times for each index of the array. So, the following input code:

```
alot_rain = rainfall > 3
```

Might be implemented as shown in Figure 14:

```
// now this method is designed purely to be used with non arrays
int greaterComparisonOperator (int lhs, int rhs) {
        return lhs > rhs;
}
int[9] rainfall = load(…);
int[9] alot_rain;
// main for loops that executes 9 times (for each element of rainfall)
for(int i = 0; i < 9; i++) {
        //equivalent to alot_rain = rainfall > 3
        // extract out the value of rainfall at i and store the result in alot_rain at i
        alot_rain[i] = greaterComparisonOperator(rainfall[i], 3);
}
```

Figure 14 Pseudocode for the greaterComparisonOperator function, treating all values as non-arrays.

Therefore, the only time we will need to consider whether a value is an array or not is when we load it into the main execution loop. If it's an array we simply need to extract a value at the current index we're calculating for, and if it's a literal we don't have to do anything.

There are some downsides to this approach. The code will have to be executed *n* times, where *n* is the size of the largest array. This means the entire AST will have to be walked *n* times which can have implications on performance. Additionally, since arrays aren't computed in a single pass, calculations on arrays can't take advantage of cache hits as only one element of each array will be computed per pass.

Decision

It appears that the trade-offs for the three described methods of implementing arrays into the language are performance vs memory vs code quality/maintainability.

The first approach allows for better performance since expressions are being calculated at once which takes advantage of spatial locality[29] for arrays. However, it requires sacrificing code quality since we will have to consider that calculations will either work on arrays or scalar values, therefore for every method that takes a parameter, we will have to provide two internal implementations – one for if the argument is an array or one for if it is a scalar value. The number of implementations we will have to provide will be exponential as each type can have an array or non-array form.

The second approach of converting all scalar values into arrays means that we can still take advantage of spatial locality for array calculations and that we also have better code quality since we only need to provide one implementation (treating everything as arrays) for each function. However, this approach has implications for memory. By converting every scalar value into an array, we use up a lot of memory.

The final approach is to execute the code *n* number of times in an execution loop, where *n* is the value of the largest array inputted into the program. For each execution, we will extract values from any variable's array at index *i* were $0 <= i < n$, performing operations on the values and storing them

into variable arrays at index *i*. This will yield the same results as the other two approaches but has the added benefit of not creating arrays for scalar values (memory-intensive), and only one internal implementation for each function (better code quality). The downsides are a hit to performance as we can no longer take advantage of spatial locality and because the entire AST will be visited multiple times.

I ended up selecting the final approach. Implementing the same code multiple times but for scalar and array values seems like a bad design. A further benefit is that while speed might take a small hit, less memory will be used up than the other approaches, which makes the software more accessible.

5.1.4.2 Static variables

Since all variables are implemented as arrays, it is necessary to maintain a reference to a variable for the entire duration of the program, otherwise, we would be unable to build up data to be associated with a variable since our program is executed in multiple passes. This means that all variables can best be described as static[30] relative to the runtime environment. This has many implications for the runtime, including several advantages and disadvantages.

Since variables are static it is convenient for us to store runtime data within the actual variable symbol associated with each variable - seen in code as **VarSymbol**, which is generated during the semantic analysis phase and stored in a symbol table[31].

5.1.4.2.1 Performance benefits

Since a variable is static, a reference to a variable will always point to the same **VarSymbol** object. This allows us to decorate the AST nodes (e.g., **IdentiferNode** and **AssignNode**) with a pointer to a **VarSymbol** object, meaning no identifier lookups occur in the runtime. This provides huge performance benefits, since the code is executed multiple times, and a lookup for each pass would be highly inefficient.

5.1.4.2.2 No temporary variable generation

Since all variables are static, there is no temporary variable generation; therefore, we do not need runtime memory management constructs such as stack frames[32] or call stacks[33]. This makes the runtime environment significantly simpler in design as opposed to other languages which support stack[34] memory.

However, this is also where our language sees its greatest deviation from typical programming languages. An example of a deviation is that our language cannot support looping, which is often implemented via for loops or while loops, which generally use stack frames to store temporary variables associated with each pass. However, without stack frames, there is no mechanism to support temporary variables creation: the effect of looping without fresh variables would result in reassigning the same variables in that scope, causing confusing behaviour, a reason why FORTRAN II does not support recursion[35].

5.1.4.2.3 Built-in methods are inlined

Built-in methods operate in a similar way to variables. Since they are also executed multiple times, they are subject to the same rules as variables.

Method inlining[36] is the process of replacing invocations of methods with its body i.e. when a **MethodCallNode** is found. We inline methods because they also contain data that is necessary to have for the duration of the execution. An example of this is the **plot()** method which contains an internal array that is populated with values at the current tick index every time execution of the code occurs. Once the program has finished executing, that array is extracted and plotted to the charts. We have to copy (aka inline) the method every time we call it, otherwise, multiple **plot()** calls will be manipulating the same internal array.

Although the underlying implementation (i.e., the body) of built-in methods is C++ code, we can still perform inlining by duplicating the actual **MethodSymbol** object as it encapsulates all data and logic for a method and decorate the **MethodCallNode** with a reference to this duplicate. This duplicate is what will be executed when the code is run.

5.1.4.3 Control flow

Control flow[37] is how the order of statements is executed, in my language, in the form of if statements and ternary statements resulting in conditional execution.

Control flow was a feature that was originally planned for completion towards the end of the project as an additional feature, however, it became clear that due to the unique way the runtime was evaluated, the integration of control flow into the language would take careful planning and modifications of existing systems, mainly the method system and the underlying internal arrays for variables. Significant changes were made to the method implementations to account for this (see lessons learnt).

This main issue with control flow is to do with building up internal arrays at various stages in the code. Without control flow, we could guarantee that every part of code the user wrote would be executed once a tick, meaning that we could build a history of values (i.e., internal arrays) by simply pushing a value to an internal array (implemented in C++ as a std::vector[38] since we can modify its size to reflect the user's largest input array) each pass. Therefore, these internal arrays would all be the same size, meaning that each element will map to the correct corresponding element calculated at the same index. This is especially useful for plotting since we could plot these arrays on a chart without having to worry about them being out of synch.

However, with control flow, we can no longer guarantee that every part of the user's code will be executed at least once a tick – for example, if an if statement returns false its block of code will not be executed. This means that internal arrays may not be of the same size at the end of the program, since they may have been skipped in a tick due to the control flow condition being false. This poses a serious problem because now the array data can be different sizes, meaning they will be out of synch when plotting.

A solution to this is to work out the max size of all the input arrays before we run the code (dictated by the largest-sized input array), create a vector of that size, and fill that vector with null values corresponding to the type determined during semantic analysis. Now, we just need to make sure that we do not push or delete any elements from that vector, and instead, simply update the value in the vector at the index **tick** which is the current tick of the execution. We can now guarantee that all vectors will be of the same size and can be compared correctly without having to worry about them being out of synch. Therefore, even if the code is not executed, there will be no gap since they will use the null value already stored in their array.

5.2 IDE

The Integrated Development Environment[39] (IDE) is the container of the entire software. It is a GUI that allows you to interface with the programming language and charts, along with importing and exporting data.

5.2.1 Language Integration

5.2.1.1 Charts

One of the functional requirements of the project was to display output from the language onto the charts. This can be achieved using the languages 3 output methods:

1. **Plot.** Plots a line on the chart.
2. **Mark.** Plots a marker on the chart when a condition is true, at a specified y level.
3. **Text.** Draws text on the chart when a condition is true, at a specified y level.

All three of these output methods make use of internal arrays which are updated with the latest values whenever they are called. These internal arrays become the chart data that is fed into the charts once the program has finished executing.

In addition, each of these methods allows the user to input (via optional arguments) a key name for the line (apart from the **text()** method), and a chart id to specify which chart to plot to.

The IDE can support multiple charts, meaning that the user can plot multiple charts from the same script. This can be incredibly useful for analysing different plots with substantial axis size differences, or just for spreading out the data to avoid a single chart becoming too complicated. This is also a non-functional requirement.

The creation of multiple charts is handled automatically once a script has finished executing. The program will spawn a chart window for every unique chart id passed into any of the plot functions and draw the output functions' respective data to the chart with that id. Both the name and chart id are of constant string type, forcing the user to use only a string literal. This is because we register these charts during the semantic analysis phase of the program so that they are only generated once, and therefore to extract a value before the execution phase, we need to force the user to use a literal so we can guarantee that the id will not change during runtime, and therefore we have a correct value.

Figure 15 Plotting to multiple charts from the same script

5.2.1.2 Error highlighting feature

A non-functional requirement of the program was that the IDE should highlight errors in the user's code. This is achieved via decorating the AST node with a source code location (i.e., line/column beginning/ending) from the parser. Since the AST is walked in subsequent phases, an error will always be associated with some AST node. We provide **TextEditor's** error marker feature, with the AST's source code information, and part of the code will be highlighted as seen in Figure 16. This feature allows the identification of language errors as the user writes their code.

We further extend the error highlighting feature by performing static error checking on code as it is being written. This is achieved by executing only the parsing and semantic type-checking phases of the language. Using this feature, we can identify and warn users of some errors before they execute the code.

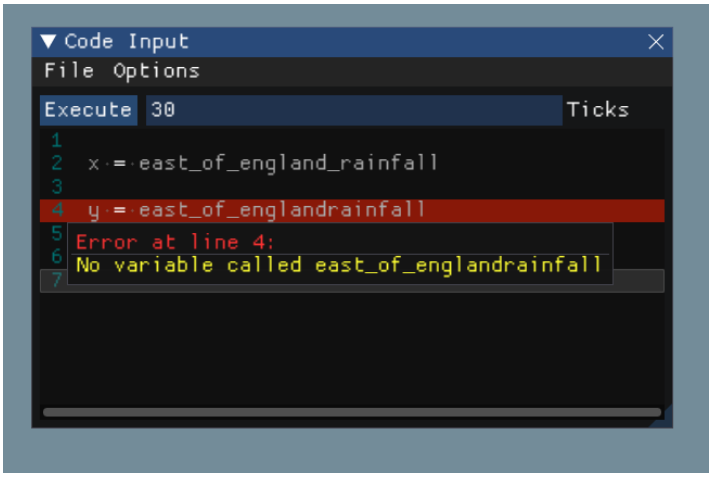

Figure 16 Error highlighting feature. The user is warned of the incorrect spelling of "east_of_england_rainfall" while writing it

However, it is only possible to identify certain errors that are related to the parsing and the semantics of the program. It is not possible to identify runtime errors.

5.2.1.3 Importing/Exporting of data

For the tool to act as an effective replacement for Excel's charting features, it is essential that it can import and export data. This is also a functional requirement of the software. We use the Comma Separated Value (CSV) file format[40] to support reading and writing data, which is a supported file format of Excel. We read and write in series of data which can either be done column-wise or row-wise. Each series maps back to a unique variable.

Inputting data into the language is done via parsing a CSV file, parsing each series of data found in it. The logic for parsing is slightly involved because it requires the data to follow several rules or else the program won't be able to figure out what type of variable it should be. It also requires the parser to determine a type for the series of data.

Once a series is successfully parsed, a **VarSymbol** object is created and loaded with the series data. This **VarSymbol** is loaded onto the global symbol table so that it is accessible to the language part of it. Exporting data is more straightforward as any data stored within a variable is guaranteed to be correct. We can convert the value of any of these types into a string representation that can be written to CSV.

5.2.2 HCI

5.2.2.1 Persistence

For convenience, all settings will be automatically saved to disk every 5 seconds. Nearly all major settings including window positioning, sizing, chart settings, loaded in data and more are automatically saved so that the user does not need to reset them every time they restart the application. These settings are saved to a JSON file[41].


```
{
  "chartAntiAliasing": true,
  "currentCodeFile": "C:\\Users\\shaer\\Desktop\\Uni Work\\Year 3\\Project\\Project\\Example Code\\demo3.al.al",
  "defaultFalseExportLiteral": "FALSE",
  "defaultFalseImportLiteral": "FALSE",
  "defaultNANExportLiteral": "NAN",
  "defaultNANImportLiteral": "NAN",
  "defaultTrueExportLiteral": "TRUE",
  "defaultTrueImportLiteral": "TRUE",
  "intellisense": true,
  "isOpenChartWindow0": true,
  "isOpenChartWindowDefault": true,
  "isOpenChartWindowmain": true,
  "isOpenDataManagerWindow": false,
  "isOpenDocumentationWindow": false,
  "isOpenOutputWindow": true,
  "isOpenSettingsWindow": false,
  "isOpenTextEditorWindow": true,
  "lastCodeOpenDirectory": "C:/Users/shaer/Desktop/Uni Work/Year 3/Project/Project/Example Code",
  "lastCodeSaveDirectory": "C:/Users/shaer/Desktop/Uni Work/Year 3/Project/Project/Example Code",
  "lastDataExportDirectory": "C:/Users/shaer/Desktop/Uni Work/Year 3/Project/Project/Example Exports",
  "lastDataImportDirectory": "C:/Users/shaer/Desktop/Uni Work/Year 3/Project/Project/Example Data",
  "lastScreenshotExportDirectory": "C:/Users/shaer/Desktop/Uni Work/Year 3/Project/Project/Example Screenshots",
  "loadedInData": [
    {
      "NANImportString": "NAN",
      "falseImportString": "FALSE",
      "name": "east_england_rainfall.csv",
      "path": "Data Manager Window",
      "policy": "column-wise",
      "trueImportString": "TRUE"
    }
  ],
  "windowX": 0,
  "windowY": 29,
  "windowheight": 991,
  "windowwidth": 1920,
  "zoom": 2.002000093460083
}
```


Figure 17 Contents of JSON save file

6 Discussion

6.1 Examples

1. Use case: Backtesting an algorithmic trading strategy

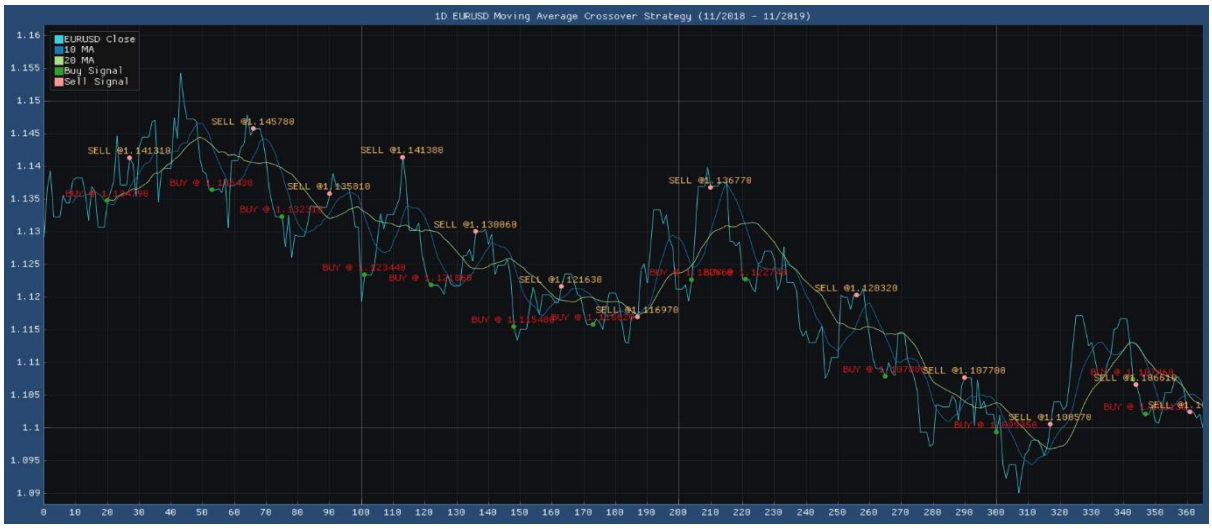

Figure 18 Trading strategy back testing results plotted on a chart

Figure 18 was built using the software. It is a simulation of a very simple trading strategy, which generates buy/sell signals at moving average crossover events based on historical EURUSD 1-day bar close price data[42]. This is also known as backtesting[43] within the trading industry, where one attempts to predict the future performance of a trading strategy based on past simulated performance.

Since the trading strategy is based purely on an algorithm that makes use of simple maths and statistics, we can comfortably model the entire strategy programmatically within the confines of our language. The charts give a visual clue as to how well the trading strategy will perform, but we can also extract the buy/sell prices back into Excel which will let the user do in-depth calculations to determine profit. The code can be found in the appendix.

2. Use case: Predicting if it will flood next year

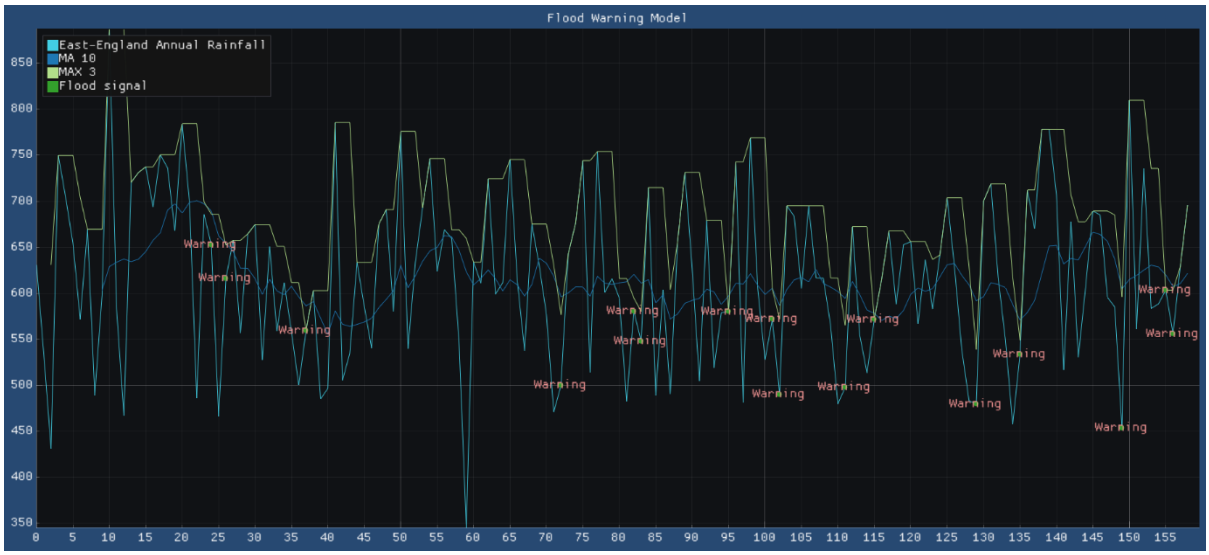

Figure 19 Flood warning model plotted on a chart

Figure 19 is an attempt to predict whether the next year will flood due to high rainfall. The data used is a numeric series of East England annual rainfall between 1867-and 2021[44]. A flood can be identified as a significant increase in rainfall in comparison to the previous year. What this model attempts to do, is output a warning signal to suggest whether it will food due to high rainfall. A signal is generated when the highest rainfall value of the last 3 years, drops below the 10-year moving average, and the text "Warning" is drawn on the chart where this occurs. The chart gives a strong visual clue as to how well the strategy performs and we can perform additional analysis on the data within Excel if we would like to do so. The code can be found in the appendix.

6.2 Use of Libraries

I made extensive use of libraries throughout the project with an attitude of "do not reinvent the wheel" i.e. if a library exists that can achieve a task better, utilise it and instead focus time on more unique parts of the project that required greater attention.

6.2.1 Flex/GNU Bison

I used Flex[45] and GNU Bison[46] which are lexer and parser generators for C/C++ programs. They generate the C/C++ code to lex and parse the input. We integrate this software together, such that the lex output (tokens) is fed directly into the parser (grammar matching of tokens), meaning that the only input is string text, and the output is an AST. Flex/Bison provide significant freedom over how you specify the lexer and parser.

```
"and"                   value->emplace<int>(yy::parser::token::TAND); return yy::parser::token::TAND;
"or"                    value->emplace<int>(yy::parser::token::TOR); return yy::parser::token::TOR;
"not"                   value->emplace<int>(yy::parser::token::TNOT); return yy::parser::token::TNOT;
"if"                    value->emplace<int>(yy::parser::token::TIF); return yy::parser::token::TIF;
"true"                  value->emplace<int>(yy::parser::token::TTRUE); return yy::parser::token::TTRUE;
"false"                 value->emplace<int>(yy::parser::token::TFALSE); return yy::parser::token::TFALSE;
"{"                     value->emplace<int>(yy::parser::token::TOPENBLOCK); return yy::parser::token::TOPENBLOCK;
"}"                     value->emplace<int>(yy::parser::token::TCLOSEBLOCK); return yy::parser::token::TCLOSEBLOCK;
[a-zA-Z_][a-zA-Z0-9_]*  value->emplace<std::string>(yytext); return yy::parser::token::TIDENTIFIER;
[0-9]+                  value->emplace<std::string>(yytext); return yy::parser::token::TNUMBER;
[0-9]+\.[0-9]*          value->emplace<std::string>(yytext); return yy::parser::token::TFLOAT;

"+="                    value->emplace<int>(yy::parser::token::TPLUSASSIGN); return yy::parser::token::TPLUSASSIGN;
"-="                    value->emplace<int>(yy::parser::token::TMINUSASSIGN); return yy::parser::token::TMINUSASSIGN;
"*="                    value->emplace<int>(yy::parser::token::TMULASSIGN); return yy::parser::token::TMULASSIGN;
"/="                    value->emplace<int>(yy::parser::token::TDIVASSIGN); return yy::parser::token::TDIVASSIGN;
"^="                    value->emplace<int>(yy::parser::token::TPOWASSIGN); return yy::parser::token::TPOWASSIGN;
"%="                    value->emplace<int>(yy::parser::token::TMODASSIGN); return yy::parser::token::TMODASSIGN;

"+"                     value->emplace<int>(yy::parser::token::TPLUS); return yy::parser::token::TPLUS;
"-"                     value->emplace<int>(yy::parser::token::TMINUS); return yy::parser::token::TMINUS;
"^"                     value->emplace<int>(yy::parser::token::TPOW); return yy::parser::token::TPOW;
"%"                     value->emplace<int>(yy::parser::token::TMOD); return yy::parser::token::TMOD;
"*"                     value->emplace<int>(yy::parser::token::TMUL); return yy::parser::token::TMUL;
"/"                     value->emplace<int>(yy::parser::token::TDIV); return yy::parser::token::TDIV;
```

Figure 21 Snippet from the language's Flex "lexer" file

However, the process of integrating Flex and Bison, in C++ mode (the default output is C code) was extremely tedious, with many cryptic error messages being outputted that forced analysing of the outputted generators to determine what had gone wrong. Additionally, Bison/Flex use a very specific layout (Figure 21 and Figure 20) for their input files, with numerous options that sometimes conflict with each other.

```
program : stmts {
                    *inputnode = $1;
                }
        ;

stmts : stmt { $$ = new BlockNode(yy::SourceLocation()); $$->statementNodes.push_back($1); }
      | stmts stmt { $1->statementNodes.push_back($2); $$ = $1; }
      | /*blank*/ { $$ = new BlockNode(yy::SourceLocation()); }
      ;

stmt : assign {$$ = $1;}
     | exprstmt {$$ = $1;}
     | ifstmt {$$ = $1;}
     ;

exprstmt : method {$$ = new ExpressionStatementNode($1, $1->sourceLocation);}
         | ternary {$$ = new ExpressionStatementNode($1, $1->sourceLocation);}
         ;

ifstmt : TIF TOPENBRACKET expr TCLOSEBRACKET TOPENBLOCK stmts TCLOSEBLOCK {
                                        $$ = new IfStatementNode($3, $6, @1 + @7);
                                    }
       ;

identifier : TIDENTIFIER {$$ = new IdentifierNode($1, @1);}
           ;

assignoptokens : TPLUSASSIGN | TMINUSASSIGN | TMULASSIGN | TDIVASSIGN | TMODASSIGN | TPOWASSIGN
               ;

assign : TIDENTIFIER TASSIGN expr {$$ = new AssignNode($1, $3, @1 + $3->sourceLocation);}
       | TIDENTIFIER assignoptokens expr {$$ = new AssignNode($1,
                                                new MethodCallNode("operator" + token_name(assign_op_converter.at($2)),
                                                    { new IdentifierNode($1, @1), $3},
                                                    @1 + $3->sourceLocation),
                                            @1 + $3->sourceLocation);
                                        }
```

Figure 20 Snippet from the language's Bison parser file

Despite the difficulties with these two libraries, I believe that it was a better alternative to writing my lexer and parser as it allowed me to spend more time on the rest of the language.

6.2.2 Dear ImGUI[47]

Considerable research went into choosing the GUI library. The following criteria were considered in the final decision:

- Had to work with C++.
- Had to be compatible with a charting library (integrated would be preferred).
- Chart library must have acceptable performance i.e., can plot thousands of data points with ease.
- Chart library must be flexible enough such that shapes can be plotted on the chart too e.g., Booleans.
- Chart library should have some navigation features e.g. zoom, scroll.
- Compatible on Windows/Linux/macOS.
- Must have a sufficient amount of documentation and examples.
- Must not be a "dead" project (i.e., discontinued or not actively used anymore).
- (Extra) Aesthetic, modern-looking GUI (i.e., not designed decades ago for deprecated operating systems).

Dear ImGUI was the library of choice. It is an extremely popular GUI library, primarily designed for game development meaning it is extremely responsive and quick to create new GUI with. In addition to this, it has an eco-system of extensions created by other users that extend the base library functionality. Dear ImGUI requires a graphics API backend, it supports many, but I decided to use OpenGL[48] as it is a cross-platform backend which would help allow for the software to be ported over to Linux/Mac.

I utilise three of these extensions within my code:

1. **ImPlot[49].** Pure Dear ImGUI does not support any charting widgets, but an extension called ImPlot satisfies all the charting requirements needed by my software. The charts they provide are extremely performant, interactable and flexible in what can be plotted to the charts. The API for creating the charts is very convenient to use and is simple to integrate into the language component of the software for output methods such as plot(), mark() and text()
2. **TextEditor[50].** The text editor widget is a very simple code editor input widget where you enter the language code. It comes with several convenient features, such as syntax highlighting, error marking, line numbering, and keyboard shortcuts. I was able to integrate several aspects of the language into the extension, including displaying information from the auto-generated documentation upon hovering, error highlighting and syntax highlighting specifically for my grammar. However, unlike the rest of the GUI extensions, there were several bugs and issues with it requiring me to dive into its code to identify and fix them.
3. **Filebrowser[51].** This provides the supporting widget for browsing your file system, used for importing/exporting data and also saving and loading code files.

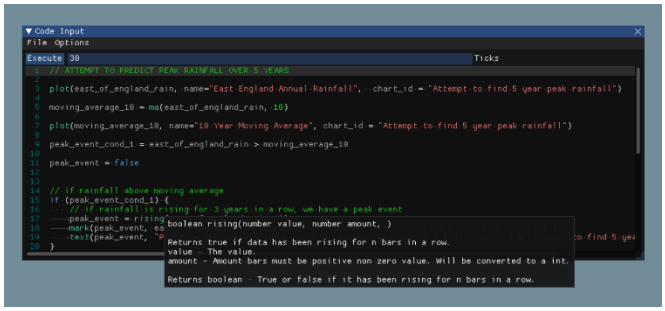

Figure 22 Preview of the text editor widget configured to support our language

6.2.3 Boost

Boost[52] is a collection of libraries built to extend/provide alternatives to C++'s standard library. It is an extremely powerful cross-platform library. My software occasionally uses it in some areas, but the main use I get out of it is for its testing library[53], which is used for all the unit testing needs. It is particularly useful because it's cross-platform – there were several native Window testing kits integrated into Visual Studio 2019 that I purposely didn't use because of the software's non-functional requirement of being cross-platform.

6.2.4 Nlohmann-JSON

This is a JSON library[54] for C++, supporting reading and writing to JSON files. I used this to store settings. I use JSON because it's human-readable and a user can choose to manually edit it if they'd like.

6.3 Challenges

There were significant challenges associated with the project. Much of the challenges related to implementing the language stem from the fact that there is little information on the implementation of array programming languages, and that language design is very subjective.

Regarding the array programming language design, one of the most difficult aspects was designing the runtime environment of the language simply due to the lack of online resources on array programming. As mentioned in the section on runtime environment, there were two ways of executing the code in the runtime, either in one pass or multiple different passes, both systems had huge implications for how the rest of the design of the language occurred. Additionally, this idea that all variables are "static" relative to the runtime environment, was not immediately obvious, nor were the consequences of this property such as a need for method inlining and no stack frames.

Moreover, further challenges were experienced due to the choice of designing the software in C++. While I have experience in C++, it can be difficult to work with - especially quickly testing libraries and dealing with memory leaks[55]. I wanted to test multiple different GUI libraries for the IDE, however, I found that it took me nearly one week to thoroughly test two different GUI libraries. For example, before the actual testing - the QT[56] library (a very popular cross-platform GUI library) took me around 8 hours to set up. This is largely due to external dependencies, mainly linking[57] the correct .dll and .lib files to the compiler, which can sometimes be hard to find.

Additionally, several memory leaks were identified during the testing phase of the project. Memory leaks occur when you fail to free memory on the heap. Allocating memory on the heap is optional depending on the extent to which the programmer wants to work with pointers[58]. However, there are many advantages to using pointers, the main one being performance, but also is especially useful when it comes to ASTs as the Nodes contain pointers to other nodes as their children. All though I took care to make sure that all heap memory was freed, there were still many instances where memory was not being freed and therefore memory leaks were occurring. In some cases, it took a while to find these leaks which involved breaking down and debugging the program, and heavy analysis of control flow (which can be difficult in a recursively visited AST). I made heavy use of Visual Studio 2019's memory profiling tool[59].

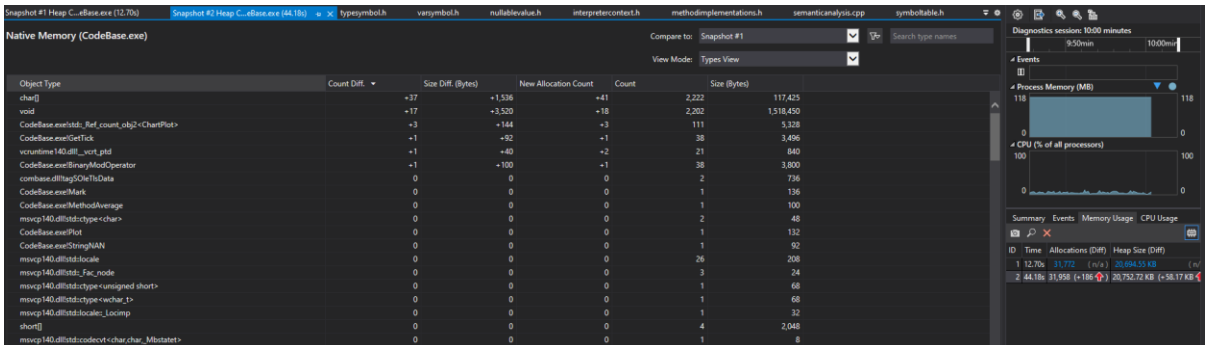

Figure 23 Visual Studio 2019's Memory Profiling tool for detecting memory leaks

6.4 Lessons Learnt

The key lesson I learned from the project was the importance of planning. Although I created a development timeline in the beginning, elements of the architecture of the language, specifically the

runtime environment were not entirely addressed during the planning stage forcing me to change the timeline.

The implementation of the standard library is an example of something that could have benefited from better planning. In my haste to implement all 63 functions, I neglected how other features yet to be finished would impact the standard library. I implemented the first instance of the standard library before many of the essential features in the language were fully understood. The result was several rewrites. Rewrites of the build-in method system occurred at least three times after the addition of control flow, keyword methods and testing. Considering that the standard library makes up nearly half of the total code of the language (~3500 lines) the rewrites were frustratingly long and took away time from the overall polish and extra features of the library.

In hindsight, the standard library should have been one of the final things to implement. Better planning of the order of implementation of features might have resulted in fewer rewrites of the standard library, saving time elsewhere.

6.5 Further improvements

While I'm satisfied with the core functionality of the software, undeniably improvements can be made in both the language and IDE areas, and with a project of this size, there is a plethora of changes and new features that could be made to further the functionality of the software, both in the short and long term.

6.5.1 Short term improvements

6.5.1.1 User Defined Methods

One of the key items that I would like to implement for the language is user-defined methods. Uncertainty with stack frames and the static nature of variables discouraged me from implementing this feature earlier on. However, my implementation of the standard library revealed to me that user-defined methods could be implemented quite intuitively via method inlining. User-defined methods bring with them several advantages such as code reusability and more maintainable code.

6.5.1.2 Variable Descriptors

I would add more features for variables in the form of descriptors. One example of this is a constant identifier which will specify that a variable should have a constant type. Currently, constant types can only be assigned to literal values i.e., a number or a string, which isn't ideal if we want to reuse a constant value e.g., for the id of a chart to plot to. A much more convenient approach is to allow a variable to have a constant type, then simply reference that variable where we needed that value. The only difficulty with this is verifying that an expression is constant which might involve making

many overloads for the method system considering if the parameters are constant and therefore if the return type should be too.

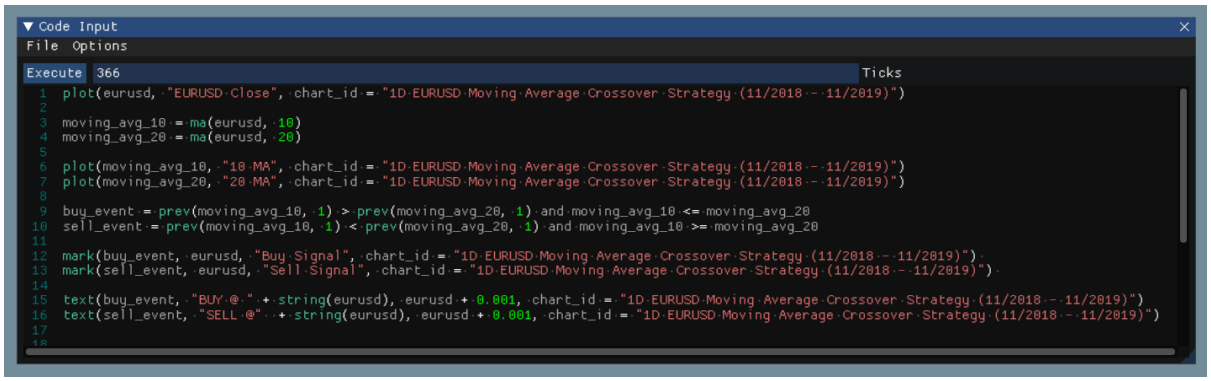

Figure 24 The languages inability to support constant variables means that the literal value must be duplicated for all "chart_id" arguments.

6.5.2 Long term improvements

6.5.2.1 More charting features

Something I would like to extend for the language and IDE is the scope and control of charting features. So far, the two possible types of charts we can build with the tool are standard numeric plots and scatter charts. While these are some of the most common plot types, Excel can support many additional plots such as pie charts, histograms, and bar charts. The plotting library I have chosen can also support many of these plot types, so it would not be difficult to extend the software to use these different chart types. Additionally, greater control of the chart such as colours, font sizes and marker types are also possible with the library and is something that we can interface through built-in methods within the language.

6.5.2.2 DateTime datatype

A more challenging addition that I would implement with more time, is the ability to support a DateTime datatype within the language. This is also a supported value type in Excel. While I'm satisfied with the three data types I've implemented, the DateTime datatype could provide a lot of extra utility in the language. Often, datasets come labelled with a DateTime series as a primary key, used to associate some data with a day and time. Supporting language features of a DateTime type would require the ability to compare different DateTime values and concatenate DateTime values to produce new ones. The primary challenge with DateTime is the parsing of a DateTime series into the correct C++ value. DateTime values can come in many different formats, and our parser would have to interpret them correctly.

7 Evaluation

7.1 Excel compatibility

While Excel does support the CSV file format, a typical spreadsheet is saved[60] as a .xlxs or .xls. This is so that the spreadsheet can contain data related to all aspects of the spreadsheets, such as colours,

macros, templates etc. Meanwhile, a CSV is very simple in comparison to one of these save files, it is simply a file containing values separated by a comma. This makes it very easy to parse and is one of the reasons why it is supported in this project. However, typical spreadsheets (.xlxs or .xls) do not adhere to this format and will lose data if converted directly into a CSV. Therefore, for the user to import data from a spreadsheet, they may first need to extract the relevant data into a new CSV file. This may be inconvenient.

Moreover, while the software can import both number and Boolean series, the parser does not support importing string series. This was largely due to time constraints but also uncertainties around type deduction, i.e., if it is a number series with a single typo in a cell, does the whole series become implemented as a string since it cannot be a number?

7.2 Development Timeline

The planned development of this software was split into three major sections, with each section corresponding to a significant milestone of development and an expected time of completion. Each section had several goals within them, with the completion of all of them marking the achievement of the milestone. Where necessary, I also ordered the goals in a way that chronologically made sense for development, e.g., it does not make sense to work on charts when the supporting GUI framework is not implemented yet.

At the start of the project, my attitude was not to modify the development timeline, as its modification could risk deviation from the original timeline and a failure to complete all goals. While generally I stayed disciplined and stuck to the original plan, new insights came up that forced me to clarify and change the ordering of the goals.

Despite this, I stuck to this milestone system for most of the development of the software. It was a useful tool for development and generally, I managed to stick to it, completing 2/3 of the milestones, with the final milestone having completed 3/4 goals. The final goal I did not complete, “testing the application for cross-compatibility”, was due to time constraints. This was moved from the second milestone because testing if the application worked on different operating systems, seems like something that should be done in the final stages of development. It did not make sense to mix it in with the second milestone which is related to mostly improving the language.

7.3 Internal arrays for variables

In reflection, the decision to have all variables store an internal array could have been avoided resulting in using up less memory in the program. The primary reason why we had internal arrays for each variable, was so the user could export the contents of any variable back to CSV. However, if the

user doesn't want to export the data, then the variable is just wasting memory by holding onto the array.

For example, if the user inserted an array of size 100 000 into the software, every single variable would have to have an internal array matching that size. That means that whenever a user declares a number variable (implemented internally as a float), its internal array would take up 0.4MB of memory (4 bytes * 100 000). This is wasteful and can quickly add up if many variables are declared.

A better solution would have been to create an **export** built-in function that would be used to export data passed into it. Internal arrays can be removed from each variable, and they can simply store a single temporary value that is updated every tick. This would mean the program would not store any additional data that we do not need but still allow the user the choice to export data back into CSV format.

7.4 Testing

Unit testing was essential for verifying that the logic I implemented was correct. Unit testing was concerned largely with testing the language, and some of the parsing capabilities for inputting data into the system. In total 96-unit tests were created. However, I did not write unit tests for the GUI.

Writing the tests for the standard library was extremely useful for making sure the standard library functioned appropriately when changes were made to the runtime environment. However, the project could have benefitted more from writing them earlier on, so that changes causing bugs could be detected straight away.

8 Conclusion

The goal of the project was an "all-in-one" software to allow users to create charts programmatically to provide an alternative solution to Excel's charting features.

While all the functional requirements were met, miscalculations in planning resulted in code rewrites which ultimately led to the failure to finish the final objective laid out in the development timeline. The point in question was testing the software for different OS compatibility. This was disappointing since I deliberately picked libraries that were compatible with other operating systems in preparation for this. However, there were many design decisions involved in the creation of the software, largely to do with language design, that had implications on how the rest of the software's development would progress and therefore it was difficult to create a concrete plan for the development from the beginning.

Despite not completing all milestones initially set out in the development timeline, the project was successful in achieving its primary goals and functional requirements, the programming language is stable, and can create detailed charts and there is compatibility with Excel through inputting and outputting CSV files. It was also the case that additional features that were not planned were implemented such as the string type. Sufficient time was allocated towards fixing bugs and other general polish issues, arguably resulting in software that is functional and reliable.

9 Appendix

9.1 Screenshots

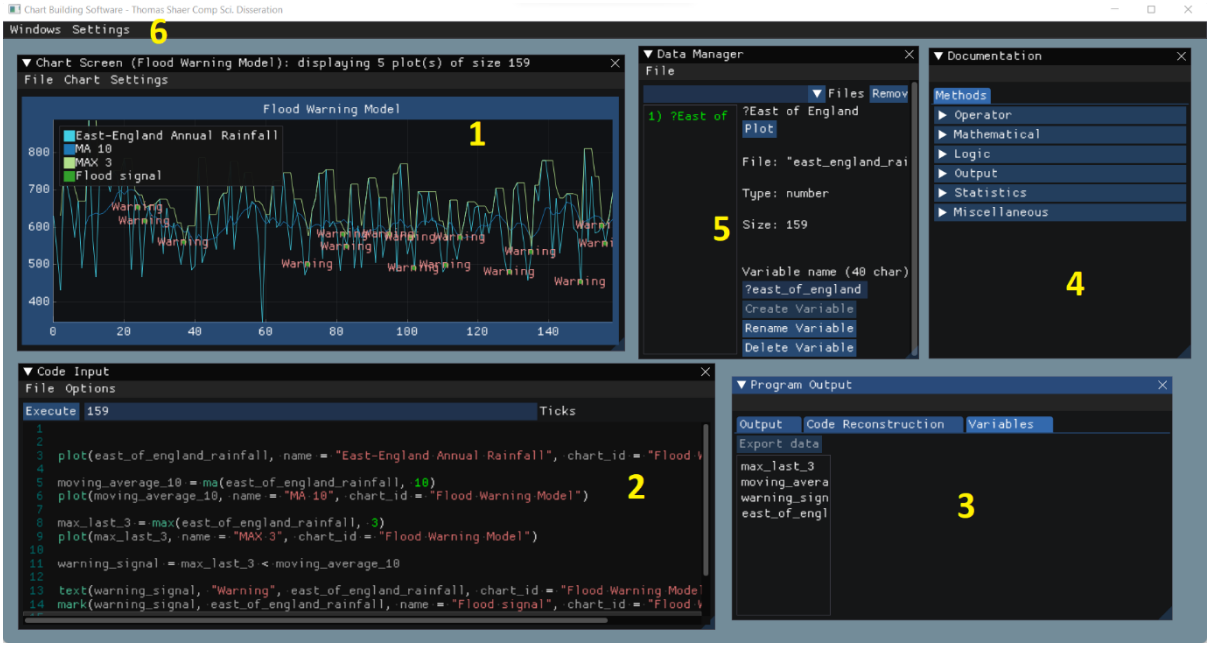

Figure 25 Overview of all windows
1. Chart window – displays chart content, dynamically created depending on what the script plots to.
2. Code editor window – where the user inputs/manage their code files.
3. Program output window – output from the script is displayed here, including error messages, code reconstruction and user-defined variables.
4. Documentation window – details of all registered built-in methods by category, are automatically displayed here.
5. Data manager window – where data is imported into the program and turned into language variables.
6. Main window toolbar – windows can be opened/closed from here and the global text size can be adjusted

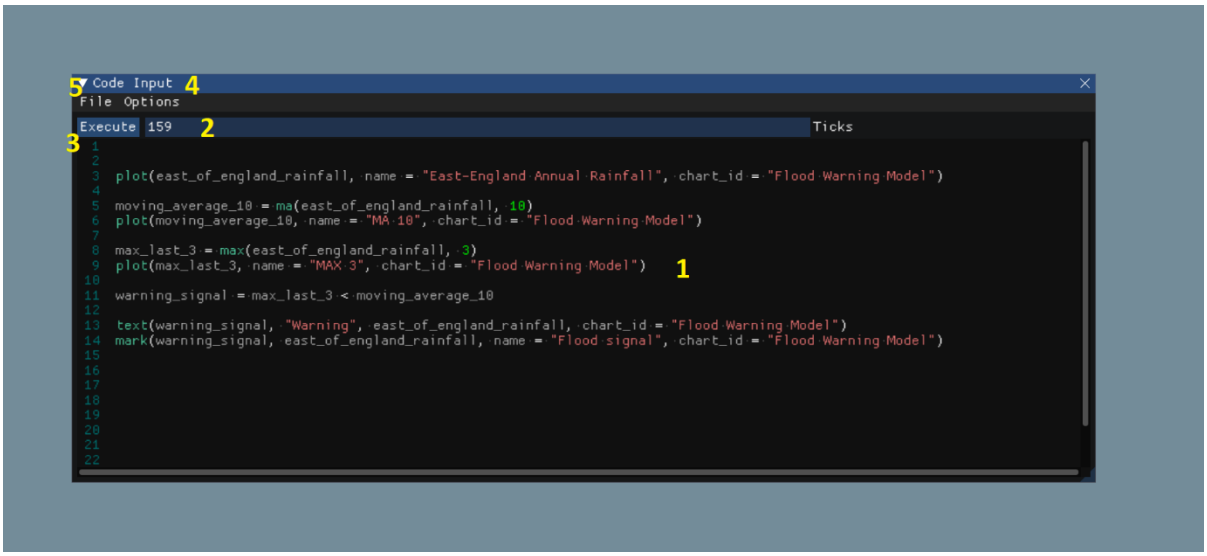

Figure 26 Overview of the code editor window
1. Text input – this is where the user writes the code.
2. Number of ticks input – the user can select how many times they want the code to execute (i.e., the size of internal arrays), by default this will revert to the maximum size of the largest inputted variable.
3. Execute button – runs the code.

4. Options menu – allows the user to toggle code highlighting.
5. File menu – allows the user to create/load/save code files to/from disk.

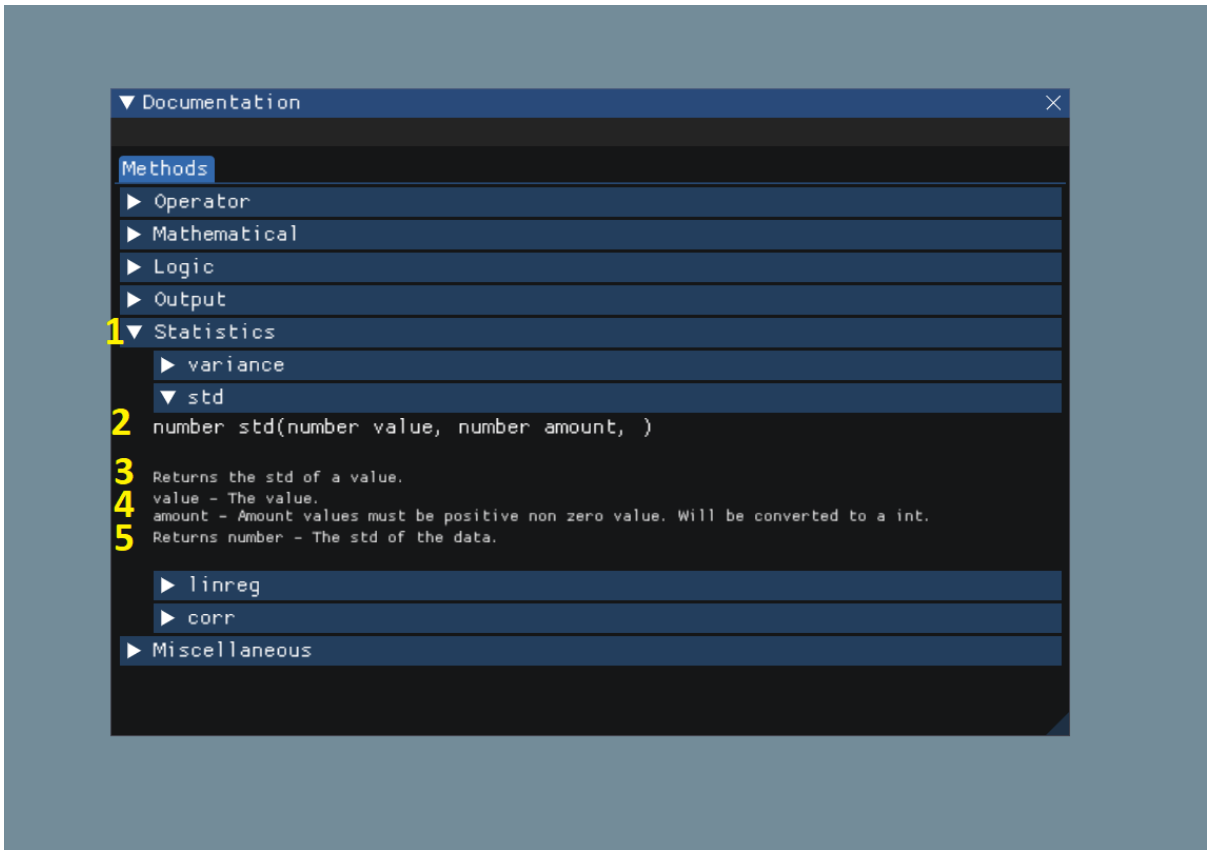

Figure 27 Overview of documentation window
1. Method category tabs – groups all methods via tabs, currently there are operators, mathematical, logic, output, statistics and miscellaneous.
2. Signature of method – shows the signature of the method, including return type, name and parameter types.
3. Method description – displays a description of the method.
4. Method parameters – displays type and description of parameters.
5. Method return type – displays type and description of the return type.

Figure 28 Overview of the output window

1. Text output tab – displays any error messages after running the script.

2. Code reconstruction tab – displays how the code is parsed and interpreted in AST form.

3. Variables tab – displays information about all variables in the script including export settings.

4. Export data – takes the user to export dialogue after selecting some variables to export.

5. Variables – lists all variables in the program.

6. Variable information – shows information about selected variable.

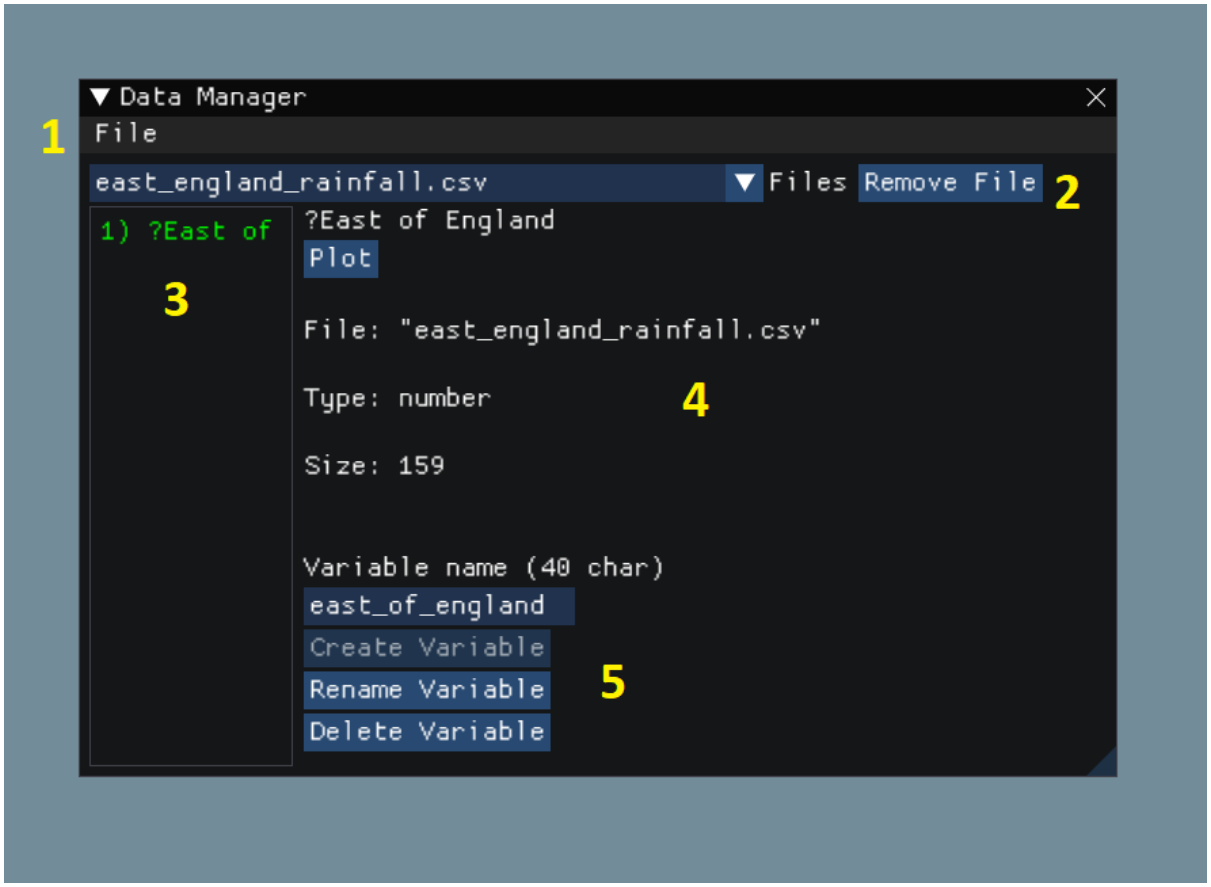

Figure 29 Overview of the data manager window
1. File menu – allows users to import/remove all data files.
2. Remove file – allows the user to select a data file to remove. Will remove all associated series/variables with the file.
3. Series list – displays a list of all parsed series from imported data files.
4. Series information – displays information about selected series.
5. Variable creation settings – allows the user to convert the series into a variable, including other management settings.

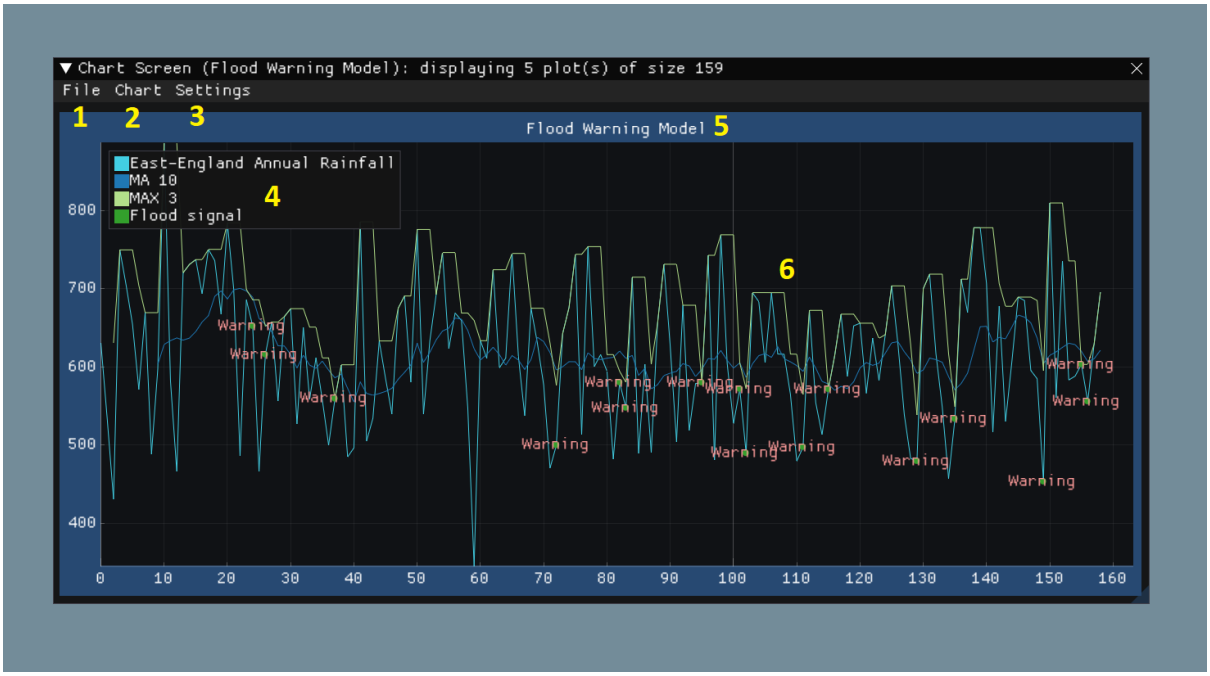

Figure 30 Overview of a chart window
1. File menu – contains settings for exporting chart as a PNG to disk.
2. Chart menu – contains chart control settings, such as clearing the chart, rescaling axis, and ability to disable the key.
3. Settings menu – contains setting for toggling chart anti-aliasing.
4. Chart key – name to colour mapping for each plotted series – the name of the key is specified by the optional "name" parameter of the plot() and mark() functions (default is empty)
5. Chart title – the title of the chart – specified by the optional "chart_id" parameter of the plot(), mark() and text() functions (default is "main")
6. Contents of the chart – interactable chart explorer.

9.2 Language Features

Name	Syntax	Description
Assign/declare	x = 2 y = tick()	Declares or assigns a value to a variable. If it's the first time a variable is assigned, its type will be inferred from the assignable expression. Any other assigns after the initial one must have the assignable expression of the same type as the initial one.

If statement	if (3 > 2) {
	…
	}
		An if statement is formed from a Boolean condition and block of code that will be executed if the condition is true.
		Note: else statements are not supported.

Method Call	plot(2) y = avg(3, 5)	A method call is used to invocate a function's logic from the standard library.

			Method calls can occur either as part of an expression or called as a statement on their own.
Ternary Statement		y = x > 3 ? 6 : 2	A ternary statement is an expression made up of a Boolean condition and two expressions. If the Boolean condition is true, the left-hand expression will be executed and returned, and if it is false, the right-hand expression will be executed and returned. The two return types of expressions must be the same.
Comment		// this line will be ignored	A comment is where a user can leave a non-code annotation in the source code. The entire line will be ignored by the parser. Note: only single-line comments are supported.
Operators	Addition	x = 2 + 2 y = + 2	Adds two numbers. Can also be used in unary form. Implemented as built-in function **operator+**
	Subtract	x = 2 – 2 y = - 4	Subtracts two numbers. Can also be used in the unary form. Implemented as a built-in function **operator-**
	Multiplication	x = 5 * 2	Multiplies two numbers. Implemented as built-in function **operator***
	Division	x = 6 / 2	Divides two numbers. Implemented as built-in function **operator/**
	Modulus	x = 4 % 2	Performs modulus on two numbers. Implemented as built-in function **operator%**
	Exponent	x = 2 ^ 4	Performs exponent on two numbers. Implemented as built-in function **operator^**
	Logical Not	x = ! true y = not false	Performs logical not operation on a Boolean. Tokens: **!** or **not** Implemented as built-in function **operator!**

	Less Than	x = 3 < 2	Performs less than comparison on two numbers. Implemented as built-in function **operator<**
	Less Than or Equal	x = 3 <= 2	Performs less than or equal comparison on two numbers. Implemented as built-in function **operator<=**
	Greater Than	x = 3 > 2	Performs greater than comparison on two numbers. Implemented as built-in function **operator>**
	Greater Than or Equal	x = 3 >= 2	Performs greater than or equal comparison on two numbers. Implemented as built-in function **operator>=**
	Equal	x = 2 == 2 y = true == true z = “Test” == “Test”	Compares two identical types for equality. Implemented as built-in function **operator==**
	Not Equal	x = 2 != 1 y = true != false z = “Test1” != “Test2”	Compares two identical types for inequality. Implemented as built-in function **operator!=**
	Logical And	x = true && true y = true and true	Performs logical and operation on two Booleans. Tokens: **&&** or **and** Implemented as built-in function **operator&&**
	Logical Or	x = true || false y = true or false	Performs logical or operation on two Booleans. Tokens: **||** or **or** Implemented as built-in function **operator||**
Assign Operators	Plus equals	x += 2	Assign operators are syntactic sugar for: **ID** = **ID OP** EXPR ID is the variable subject OP is the operator EXPR is the right-hand expression
	Minus equals	x -= 3	
	Multiply equals	x *= 2	
	Divide equals	x /= 2	
	Modulus equals	x %= 2	
	Exponent equals	x ^= 2	

9.3 Operator Precedence

Table 1 Language operator precedence - descending precedence

Operator	**Name**

()	Parentheses
! not	Not operator
***** **/** **%**	Multiply operator Divide operator Modulus operator
+ **-**	Addition operator Subtraction operator
< **<=** **>** **>=**	Less than operator Less than or equals operator Greater than operator Greater than or equals operator
== **!=**	Equal operator Not equal operator
^	Exponent operator
&& and	And operator
|| or	Or operator
? **+=** **-=** ***=** **/=** **%=** **^=**	Ternary question mark Plus equals operator Minus equals operator Multiply equals operator Divide equals operator Modulus equals operator Exponent equals operator

9.4 Standard library

Complete documentation is available inside the software under the “Documentation” window.

Method Name	Category	Signature(s) & description(s)
abs	Mathematical	number abs(number value,)
		Returns the abs of a value.
acos	Mathematical	number acos(number radians,)
		Returns the arc cosine of x radians. If null supplied, will return null.
asin	Mathematical	number asin(number radians,)
		Returns the arc sine of x radians. If null supplied, will return null.
atan	Mathematical	number atan(number radians,)
		Returns the arc tangent of x radians. If null supplied, will return null.
average	Mathematical	number avg(number value1, number value2,)
		Returns the average of two numbers.
boolean	Miscellaneous	boolean boolean(number value,)
		Casts value to a boolean.
ceil	Mathematical	number ceil(number value,)
		Ceils value
corr	Statistics	number correlation(number data1, number data2, number length,)
		Returns the correlation of two data sources in the last n bars.
cos	Mathematical	number cos(number radians,)
		Returns the cosine of x radians. If null supplied, will return null.count
count	Logic	number count(boolean condition,)
		The amount of times a condition has been true.
e	Mathematical	number e()

		Gets the E constant.
falling	Logic	boolean falling(number value, number amount,)
		Returns true if data has been dropping for n bars in a row.
floor	Mathematical	number floor(number value,)
		Floors value
gcd	Mathematical	number gcd(number value1, number value2,)
		Returns the greatest common divider of two numbers. Note: converts both numbers to ints.
isnull	Logic	boolean isnull(number value,)
		Returns true if number value is null
		boolean isnull(boolean value,)
		Returns true if boolean value is null
		boolean isnull(string value,)
		Returns true if string value is null
isprime	Mathematical	boolean isprime(number data,)
		Returns true or false if it is a prime number. Will be converted to a integer. Ignores null values.
istriangle	Mathematical	boolean istriangle(number data,)
		Returns true or false if it is a triangle number. Will be converted to a integer. Ignores null values.
lcm	Mathematical	number lcm(number value1, number value2,)
		Returns the lowest common multiple of two numbers. Note: converts both numbers to ints.
linreg	Statistics	number linreg(number data, number bars, number x,)
		Returns the rolling linear regression of last n bars.
log	Mathematical	number log(number value,)
		Returns the natural base-e log of a number. Can throw a run time error if <= 0.
		number log(number value, number base,)
		Returns the log of a number given a base. Can throw a run time error if <= 0.
ma	Mathematical	number ma(number value, number amount,)
		Returns the moving average of a value.
mark	Output	void mark(boolean when, number value, string_constant name = "", string_constant chart_id = "main",)
		Conditionally marks points on the chart when a condition is true.
max	Mathematical	number max(number value,)
		Returns the max value that has ever been passed to it.
		number max(number value, number bars_back,)
		Returns the largest value that has ever been passed to it in the last bars_back bars
maxnumber	Mathematical	number maxnumber()
		The max number value
mean	Mathematical	number mean(number value,)
		Returns the mean of values passed to it so far. (Doesn't take na values into account).
median	Mathematical	number median(number data,)
		Returns the median value. Ignores null values.
		number median(number data, number barsback,)
		Returns the rolling median value. Ignores null values.
min	Mathematical	number min(number value,)
		Returns the smallest value that has ever been passed to it.
		number min(number value, number bars_back,)
		Returns the smallest value that has ever been passed to it in the last bars_back bars
minnumber	Mathematical	number minnumber()

		The minimum number value
null_b	Mathematical	boolean null_b()
		Get a boolean null.
null_n	Mathematical	number null_n()
		Get a number null.
null_s	Mathematical	string null_s()
		Get a string null.
number	Miscellaneous	number number(boolean value,)
		Casts value to a number.
operator!	Operator	boolean operator!(boolean expr,)
		Negates a boolean expression.
operator!=	Operator	boolean operator!=(number lhs, number rhs,)
		Performs not equal operator on two numbers.
		boolean operator!=(boolean lhs, boolean rhs,)
		Performs not equal operator on two booleans.
		boolean operator!=(string lhs, string rhs,)
		Performs not equal operator on two strings.
operator%	Operator	number operator%(number lhs, number rhs,)
		Applies modulus operator to number
operator&&	Operator	boolean operator&&(boolean lhs, boolean rhs,)
		Performs and operator on two booleans.
operator*	Operator	number operator*(number lhs, number rhs,)
		Multiplies two numbers.
operator+	Operator	number operator+(number expr,)
		Propagates a expression.
		number operator+(number lhs, number rhs,)
		Adds together two numbers.
		string operator+(string lhs, string rhs,)
		Concatenate two strings.
operator-	Operator	number operator-(number expr,)
		Negates a expression.
		number operator-(number lhs, number rhs,)
		Minuses two numbers.
operator/	Operator	number operator/(number lhs, number rhs,)
		Divides two numbers.
operator<	Operator	boolean operator<(number lhs, number rhs,)
		Performs less than operator on two numbers.
operator<=	Operator	boolean operator<=(number lhs, number rhs,)
		Performs less than or equal operator on two numbers.
operator==	Operator	boolean operator==(number lhs, number rhs,)
		Performs equal operator on two numbers.
		boolean operator==(boolean lhs, boolean rhs,)
		Performs equal operator on two booleans.
		boolean operator==(string lhs, string rhs,)
		Performs equal operator on two strings.
operator>	Operator	boolean operator>(number lhs, number rhs,)

		Performs greater than operator on two numbers.
operator>=	Operator	boolean operator>=(number lhs, number rhs,)
		Performs greater than or equal operator on two numbers.
operator^	Operator	number operator^(number lhs, number rhs,)
		Raises number by power of another.
operator||	Operator	boolean operator||(boolean lhs, boolean rhs,)
		Performs or operator on two booleans.
pi	Mathematical	number pi()
		Gets the PI constant.
plot	Output	void plot(number value, string_constant name = "", string_constant chart_id = "main",)
		Plots a series of values onto the chart
prev	Miscellaneous	number prev(number data, number barsback,)
		Gets the previous value. Lookback will return the last nth value.
		string prev(string data, number barsback,)
		Gets the previous value. Lookback will return the last nth value.
		boolean prev(boolean data, number barsback,)
		Gets the previous value. Lookback will return the last nth value.
random	Mathematical	number random(number minvalue, number maxvalue,)
		Returns random value between the two ranges
rising	Logic	boolean rising(number value, number amount,)
		Returns true if data has been rising for n bars in a row.
round	Mathematical	number round(number value,)
		Rounds to nearest int. Returns NA if NA passed.
sin	Mathematical	number sin(number radians,)
		Returns the sine of x radians. If null supplied, will return null.
sqrt	Mathematical	number sqrt(number value,)
		Returns the square root of a number.
std	Statistics	number std(number value, number amount,)
		Returns the std of a value.
string	Miscellaneous	string string(number value,)
		Casts value to a string.
		string string(boolean value,)
		Casts value to a string.
sum	Mathematical	number sum(number value,)
		Returns the sum of the values passed to it so far
		number sum(number value, number bars_back,)
		Returns the sum of the last bars_back bars
tan	Mathematical	number tan(number radians,)
		Returns the tangent of x radians. If null supplied, will return null.
text	Output	void text(boolean when, string value, number ylevel, boolean vertical = false, string_constant name = "", string_constant chart_id = "main",)
		Conditionally draws text on the chart when a condition is true.
tick	Miscellaneous	number tick()
		Gets the current tick.
valuewhen	Mathematical	boolean valuewhen(boolean when, boolean value,)
		Stores and returns a value that only changes when the condition is true.

		number valuewhen(boolean when, number value,)
		Stores and returns a value that only changes when the condition is true.
		string valuewhen(boolean when, string value,)
		Stores and returns a value that only changes when the condition is true.
variance	Statistics	number variance(number value, number amount,)
		Returns the variance of a value.

9.5 AST Nodes

Node Type	Node Name	Description	Example
Statement	BlockNode	Represents a scope e.g. if statement scope. Contains a vector of statements that are executed when the block node is visited.	**{** **x = 2** **x += 5** **}**
	AssignNode	Represents an assign statement	**x = 2**
	IfStatementNode	Represents an if statement	**if (x > y)** { }
	ExpressionStatement	A wrapper node that allows an expression to be contained as a statement. This allows **TernaryNode** and **MethodCallNode** to be called as a standalone statement, as the **BlockNode** can accept a statement of any kind.	**plot(2)** **x > 3 ? 12 : 17**
Expression	NumberNode	Represents a literal number	x = **2** x = **3.12**
	BooleanNode	Represents a literal Boolean value	x = **true** x = **false**
	StringNode	Represents a literal string value	x = **“Hello World!”**
	IdentifierNode	Represents a variable lookup identifier	x = **foo**
	KeywordNode	Represents a supplied keyword argument	plot(2, **chart_id = ”Chart2”**)
	MethodCallNode	Represents a call to a method	x = **plot(2)**
	TernaryNode	Represents the ternary operator	y = **x > 3 ? 12 : 17**

9.6 Backtesting use case code

```
plot(eurusd, "EURUSD Close")


moving_avg_10 = ma(eurusd, 10)

moving_avg_20 = ma(eurusd, 20)


plot(moving_avg_10, "10 MA")

plot(moving_avg_20, "20 MA")


buy_event = prev(moving_avg_10, 1) > prev(moving_avg_20, 1) and moving_avg_10 <= moving_avg_20

sell_event = prev(moving_avg_10, 1) < prev(moving_avg_20, 1) and moving_avg_10 >= moving_avg_20


mark(buy_event, eurusd, "Buy Signal")

mark(sell_event, eurusd, "Sell Signal")


text(buy_event, "BUY @ " + string(eurusd), eurusd + 0.001)

text(sell_event, "SELL @"  + string(eurusd), eurusd + 0.001)


// EXPORT BELOW

EXPORT_BUYS = buy_event ? "BUY" : null_s()

EXPORT_BUY_PRICE = buy_event ? eurusd : null_n()

EXPORT_SELLS = sell_event ? "SELL" : null_s()

EXPORT_SELL_PRICE = sell_event ? eurusd : null_n()
```

9.7 Flood warning use case code

```
plot(east_of_england_rainfall, name = "East-England Annual Rainfall", chart_id = "Flood Warning Model")


moving_average_10 = ma(east_of_england_rainfall, 10)

plot(moving_average_10, name = "MA 10", chart_id = "Flood Warning Model")


max_last_3 = max(east_of_england_rainfall, 3)

plot(max_last_3, name = "MAX 3", chart_id = "Flood Warning Model")


warning_signal = max_last_3 < moving_average_10


text(warning_signal, "Warning", east_of_england_rainfall, chart_id = "Flood Warning Model")

mark(warning_signal, east_of_england_rainfall, name = "Flood signal", chart_id = "Flood Warning Model")
```

9.8 Development Timeline

Milestone 1 ✓

Finish a working prototype with all basic components implemented.

Planned time of completion: between week 10 to 13 ✓ *week 11*

1. **Basic version of the language with following features implemented** ✓
    - **Float/Boolean/null type** ✓
    - **Built In Methods** ✓
    - **Arithmetic/Comparison** ✓
    - **Ability to output data that can be rendered by a chart** ✓
2. **Get basic GUI framework going for interaction with the program** ✓
    - **Basic integration of language into the GUI** ✓
3. **Basic chart working that can display output information from the language** ✓
4. **Basic input and labelling for user submitted data so that it can be used and referenced in the language** ✓

Milestone 2 ✓

Add extra features that make the software more convenient for the user to use. These items are not required but enhance the product greatly.

Planned time of completion: between week 13-18 *(Christmas holiday)*

- **Interface for saving/loading code to and from the user's file system** ✓
- **Ability for user to export chart data from the program** ✓
- ~~**More language features such as if statements or user defined functions (moved from milestone 3)**~~ **Control flow features if statements and ternary conditions** ✓
- **Additional built in functions supporting a common basic mathematical operations such as regressions, stochastics etc.** ✓
- **If required, more sophisticated methods of loading in data from CSVs, e.g. tables of data**~~**, different file types etc.**~~ ✓

Milestone 3

Add features that add to a sense of "polish" for the software. These features will range in difficulty and its likely that I will not be able to implement all of them. However, implementing them would significantly enhance the program.

Planned time of completion: between week 18-22

- **Intellisense/autocomplete for writing code** ✓
- **Ability to spawn and plot on multiple different charts** ✓
- **Ability to save a chart to disk as a PNG or JPEG** ✓
- Testing the application for system compatibility on different OS's (moved from milestone 2)

9.9 GUI library candidates

Library Name	Link
Dear ImGUI w/ ImPlot	https://github.com/ocornut/imgui https://github.com/epezent/implot#:~:text=ImPlot%20is%20an%20immediate%20mode,requires%20minimal%20code%20to%20integrate.
Juce	https://juce.com/

wxWidgets w/ wxMathPlot/wxFreeChart/wxCharts	https://www.wxwidgets.org/ https://www.wxwidgets.org/blog/2018/08/wxchartdir-using-chartdirector-in-wxwidgets-applications/
Sciter	https://sciter.com/
QT w/ QT Charts	https://www.qt.io/ https://doc.qt.io/qt-5/qtcharts-index.html

9.10 Lines of code breakdown

Table 2 Lines of code breakdown. These are referring to the lines of code that I have written. Please note these numbers are approximates.

Component	Lines of code (approximate)
Programming Language Framework	4189
Programming Language Standard Library	3500
IDE	2615
Unit Tests	1846
Total	**12150**

9.11 Unit tests

Class/Construct	Test	Class/location	Pass/Fail
ArgumentSymbol	argument_symbol_correct_null_value	argumentsymboltest.cpp line 25	Pass
ArgumentSymbol	argument_symbol_literal_value_at_compile_time	argumentsymboltest.cpp line 10	Pass
ExpressionValue	expression_value_load_test	expressionvaluetest.cpp line 8	Pass
InputSeries	input_series_parse_boolean_test	inputseriestest.cpp line 70	Pass
InputSeries	input_series_parse_misc_test	inputseriestest.cpp line 28	Pass
InputSeries	input_series_parse_number_test	inputseriestest.cpp line 43	Pass
KeywordMethodSymbol	keyword_method_symbol_semantic_test	methodsymboltest.cpp line 90	Pass
TypeSymbol	match_type_test	typesymboltest.cpp line 8	Pass
Absolute	method_absolute_test	methodimplementationstest.cpp line 552	Pass
ArcCosine	method_arccosine_test	methodimplementationstest.cpp line 703	Pass
ArcSine	method_arcsin_test	methodimplementationstest.cpp line 712	Pass
ArcTan	method_arctangent_test	methodimplementationstest.cpp line 694	Pass
MethodAverage	method_average_test	methodimplementationstest.cpp line 92	Pass
BinaryAndOperator	method_binop_and_test	methodimplementationstest.cpp line 236	Pass
BinaryDivideOperator	method_binop_divide_test	methodimplementationstest.cpp line 154	Pass
BinaryFloatEqualOperator BinaryBooleanEqualOperator	method_binop_equal_test	methodimplementationstest.cpp line 261	Pass
BinaryStringEqualOperator	method_binop_greater_equal_test	methodimplementationstest.cpp line 224	Pass
BinaryGreaterOperator	method_binop_greater_test	methodimplementationstest.cpp line 212	Pass
BinaryLessEqualOperator	method_binop_less_equal_test	methodimplementationstest.cpp line 201	Pass
BinaryLessOperator	method_binop_less_test	methodimplementationstest.cpp line 189	Pass
BinaryMinusOperator	method_binop_minus_test	methodimplementationstest.cpp line 143	Pass

BinaryMultiplyOperator	method_binop_multiply_test	methodimplementationstest.cpp line 165	Pass
BinaryFloatNotEqualOperator **BinaryBooleanNotEqualOperator** **BinaryStringNotEqualOperator**	method_binop_not_equal_test	methodimplementationstest.cpp line 290	Pass
BinaryOrOperator	method_binop_or_test	methodimplementationstest.cpp line 248	Pass
BinaryPlusOperator	method_binop_plus_test	methodimplementationstest.cpp line 126	Pass
BinaryPowOperator	method_binop_pow_test	methodimplementationstest.cpp line 177	Pass
Float2BooleanCast **Number2BooleanCast**	method_boolean_cast_test	methodimplementationstest.cpp line 530	Pass
Ceil	method_ceil_test	methodimplementationstest.cpp line 472	Pass
Correlation	method_corr_test	methodimplementationstest.cpp line 990	Pass
Cosine	method_cosine_test	methodimplementationstest.cpp line 664	Pass
Count	method_count_test	methodimplementationstest.cpp line 493	Pass
Falling	method_falling_test	methodimplementationstest.cpp line 867	Pass
Boolean2NumberCast	method_float_cast_test	methodimplementationstest.cpp line 522	Pass
MaxNumber	method_float_max_test	methodimplementationstest.cpp line 483	Pass
MinNumber	method_float_min_test	methodimplementationstest.cpp line 488	Pass
Floor	method_floor_test	methodimplementationstest.cpp line 462	Pass
GCD	method_gcd_test	methodimplementationstest.cpp line 610	Pass
GetE	method_get_e_test	methodimplementationstest.cpp line 447	Pass
GetPi	method_get_pi_test	methodimplementationstest.cpp line 442	Pass
GetTick	method_get_tick_test	methodimplementationstest.cpp line 318	Pass
IsNullN **IsNullB** **IsNullS**	method_is_null_test	methodimplementationstest.cpp line 630	Pass
IsPrime	method_is_prime_test	methodimplementationstest.cpp line 778	Pass
IsTriangle	method_is_triangle_test	methodimplementationstest.cpp line 790	Pass
LCM	method_lcm_test	methodimplementationstest.cpp line 592	Pass
LinearRegressionAtTick	method_linreg_test	methodimplementationstest.cpp line 970	Pass
LogBase	method_log_base_test	methodimplementationstest.cpp line 569	Pass
LogE	method_log_e_test	methodimplementationstest.cpp line 561	Pass
MA	method_ma_test	methodimplementationstest.cpp line 801	Pass
MaximumBars	method_max_bars_test	methodimplementationstest.cpp line 818	Pass
Maximum	method_max_test	methodimplementationstest.cpp line 389	Pass
Mean	method_mean_test	methodimplementationstest.cpp line 423	Pass
MedianBars	method_median_bars_test	methodimplementationstest.cpp line 915	Pass
Median	method_median_test	methodimplementationstest.cpp line 759	Pass
MinimumBars	method_min_bars_test	methodimplementationstest.cpp line 834	Pass
Minimum	method_min_test	methodimplementationstest.cpp line 371	Pass
BooleanNull	method_null_boolean_test	methodimplementationstest.cpp line 364	Pass

NumberNull	method_null_number_test	methodimplementationstest.cpp line 352	Pass
StringNull	method_null_string_test	methodimplementationstest.cpp line 358	Pass
PreviousNumberValue PreviousStringValue PreviousBooleanValue	method_prev_test	methodimplementationstest.cpp line 722	Pass
Random	method_random_test	methodimplementationstest.cpp line 648	Pass
Rising	method_rising_test	methodimplementationstest.cpp line 891	Pass
Round	method_round_test	methodimplementationstest.cpp line 452	Pass
Sine	method_sine_test	methodimplementationstest.cpp line 682	Pass
SquareRoot	method_sqrt_test	methodimplementationstest.cpp line 582	Pass
STD	method_std_test	methodimplementationstest.cpp line 951	Pass
Float2StringCast	method_string_cast_test	methodimplementationstest.cpp line 538	Pass
SumBars	method_sum_bars_test	methodimplementationstest.cpp line 850	Pass
Sum	method_sum_test	methodimplementationstest.cpp line 406	Pass
Tangent	method_tangent_test	methodimplementationstest.cpp line 674	Pass
UnaryMinusOperator	method_unop_minus_test	methodimplementationstest.cpp line 110	Pass
UnaryNotOperator	method_unop_negate_test	methodimplementationstest.cpp line 118	Pass
UnaryPlusOperator	method_unop_plus_test	methodimplementationstest.cpp line 102	Pass
ValueWhenBoolean ValueWhenNumber ValueWhenString	method_value_when_test	methodimplementationstest.cpp line 328	Pass
Variance	method_variance_test	methodimplementationstest.cpp line 932	Pass
NullableValueBoolean	nullable_value_boolean_to_string_test	nullablevaluebooleantest.cpp line 8	Pass
NullableValueBoolean	nullable_value_boolean_value	nullablevaluebooleantest.cpp line 16	Pass
NullableValueNumber	nullable_value_number_to_string_test	nullablevaluenumbertest.cpp line 8	Pass
NullableValueNumber	nullable_value_number_value	nullablevaluenumbertest.cpp line 16	Pass
NullableValueString	nullable_value_string_to_string_test	nullablevaluestringtest.cpp line 8	Pass
NullableValueString	nullable_value_string_value	nullablevaluestringtest.cpp line 16	Pass
OptionalParameterSymbol	optional_parametersymbol_to_string_test	parametersymboltest.cpp line 21	Pass
OverloadedMethodBucket	overloaded_method_bucket_test	methodbuckettest.cpp line 45	Pass
ParameterSymbol	parametersymbol_to_string_test	parametersymboltest.cpp line 9	Pass
PositionalMethodSymbol	positonal_method_symbol_semantic_test	methodsymboltest.cpp line 18	Pass
ReturnSymbol	return_symbol_construct_test	returnsymboltest.cpp line 9	Pass
SingleMethodBucket	single_method_bucket_test	methodbuckettest.cpp line 18	Pass
SourceLocation	source_location_concat_test	sourcelocationtest.cpp line 8	Pass
SourceLocation	source_location_highlight_test	sourcelocationtest.cpp line 21	Pass
SymbolTable	symbol_table_get_variable_test	symboltabletest.cpp line 29	Pass
SymbolTable	symbol_table_variable_declared_test	symboltabletest.cpp line 10	Pass
SymbolTable	symbol_table_variables_to_vector_test	symboltabletest.cpp line 38	Pass
VarSymbol	varsymbol_creation_test	varsymboltest.cpp line 10	Pass
VarSymbol	varsymbol_get_value_test	varsymboltest.cpp line 26	Pass
VarSymbol	varsymbol_set_array_size	varsymboltest.cpp line 46	Pass
VarSymbol	varsymbol_set_value_test	varsymboltest.cpp line 35	Pass
VarSymbol	varsymbol_valid_name_test	varsymboltest.cpp line 20	Pass

9.12 Additional resources used during development

Item	Description
Tradingview's Pine Script method list[61]	Used to gain ideas about what methods to include in the standard library.
MATLAB's method list[62]	Used to gain ideas about what methods to include in the standard library.
Windows Flex/Bison Visual Studio Setup Tutorial[63]	Tutorial on configuring Visual Studio 2019 to use Flex/Bison.
Regex visualiser[64]	Used to help come up with regular expressions for tokenisation in the lexer.
Philippe M. Groarke's C++ UI library list[65]	Used to help inform my decision about the UI library to use.
C++ operator precedence table[66]	Modelled the language's operator precedence off this table.
Flex recognising single line comments[67]	Used this tutorial to create a regex to parse single-line comments.
Flex recognising string literals[68]	Used an answer from this stack overflow post for regex to parse string literals without including the quotation marks.
Code for exporting ImGUI to PNG[69] [70]	Used a heavily modified version of Dear ImGUI's sample screenshotting tool to support exporting charts to PNG.
Variance & STD logic for built-in method[71]	Used logic from this stack overflow post while writing the variance and STD built-in functions.
Correlation logic for built-in method[72]	Used logic from this stack overflow post while writing the correlation built-in function.
Insertion into sorted vector[73]	Used a code snipped from this stack overflow post to support sorted insertion into a built-in function for median functions.
Is prime logic for built-in method[74]	Used logic from this blog post to help check if a number is prime.
Is triangle logic for built-in method[75]	Used an answer from this stack overflow post for supporting the logic of the istriangle function.
Number parsing in series[76]	Used regex from an answer in this stack overflow post to parse a string into a number.

9.13 Main Codebase folder breakdown

9.13.1 Folders

- data:
  - contains files related to Bison/Flex GNU library.
- include:

    - contains source files for the libraries: boost, GLFW and nlohmann-json
- lib:
    - contains lib files for static linking for the libraries: boost, GLFW and nlohmann-json
- misc:
    - contains files related to the graphics library, Dear ImGUI
- Release:
    - contains .obj files for compiled source files
- src:
    - extra source and header files

9.13.2 Files

9.13.2.1 Executables (.exe)

- win_bison.exe
    - A compiled version of parser generator GNU Bison
- win_flex.exe
    - A compiled version of lexer generator Flex

9.13.2.2 Program save files

- imgui.ini
    - GUI layout save file for Dear ImGUI library
- settings.json
    - All over saveable settings related to the functioning of the software

9.13.2.3 Flex/Bison input files

- bison.y
    - GNU Bison input file. Contains instructions on how to generate the parser
- flex.l
    - Flex input file. Contains instructions on how to generate a lexer

9.13.2.3.1 Generated Flex/Bison files

- bison.tab.cpp
- bison.tab.h
- flex.flex.cpp
- flex.flex.h

9.13.2.4 Visual Studio save files

- CodeBase.aps
- CodeBase.rc
- CodeBase.vcxproj
- CodeBase.vcxproj.filters
- CodeBase.vcxproj.user

9.13.2.5 ImGUI library files

- imconfig.h
- imfilebrowser.h
- imgui.cpp
- imgui.h
- imgui_demo.cpp
- imgui_draw.cpp

- imgui_impl_glfw.cpp
- imgui_impl_glfw.h
- imgui_impl_opengl3.cpp
- imgui_impl_opengl3.h
- imgui_impl_opengl3_loader.h
- imgui_internal.h
- imgui_tables.cpp
- imgui_widgets.cpp
- implot.cpp
- implot.h
- implot_internal.h
- implot_items.cpp
- imstb_rectpack.h
- imstb_textedit.h
- imstb_truetype.h
- stb_image_write.h
- TextEditor.cpp
- TextEditor.h
- resource.h

9.13.2.6 Project files

- argumentsymbol.cpp
- argumentsymbol.h
- chartplot.cpp
- chartplot.h
- chartwindow.cpp
- chartwindow.h
- datamanagerwindow.cpp
- datamanagerwindow.h
- dataparseexception.h
- documentationwindow.cpp
- documentationwindow.h
- expressionvalue.h
- inputseries.cpp
- inputseries.h
- interpreter.cpp
- interpretercontext.cpp
- interpretercontext.h
- jsonsettings.cpp
- jsonsettings.h
- languageexception.cpp
- langaugeexception.h
- main.cpp
- maingui.cpp
- maingui.h
- menubar.cpp

- menubar.h
- methodbucket.cpp
- methodbucket.h
- methodimplementations.cpp
- methodimplementations.h
- methodimplementationsinterpreter.cpp
- methodimplementationssemantic.cpp
- methodsymbol.cpp
- methodsymbol.h
- node.cpp
- node.h
- nullablevalue.h
- nullablevalueboolean.cpp
- nullablevalueboolean.h
- nullablevaluestring.cpp
- nullablevaluestring.h
- outputwindow.cpp
- outputwindow.h
- parametersymbol.cpp
- parametersymbol.h
- returnsymbol.cpp
- returnsymbol.h
- screenshot.cpp
- screenshot.h
- semanticanalysis.cpp
- sourcelocation.cpp
- sourcelocation.h
- symboltable.cpp
- symboltable.h
- texteditorwindow.cpp
- texteditorwindow.h
- typesymbol.cpp
- typesymbol.h
- varsymbol.cpp
- varsymbol.h
- window.cpp
- window.h

9.14 Main Tests folder breakdown

9.14.1 Folders

- include:
  - contains source files for the libraries: boost, GLFW and nlohmann-json
- lib:
  - contains lib files for static linking for the libraries: boost, GLFW and nlohmann-json
- Release & Testing:
  - contains .obj files for compiled source files

9.14.2 Files

9.14.2.1 Visual Studio save files

- Tests.vcxproj
- Tests.vcxproj.filters
- Tests.vcxproj.user

9.14.2.2 Project files

- argumentsymboltest.cpp
- expressionvaluetest.cpp
- inputseriestest.cpp
- main.cpp
- methodbuckettest.cpp
- methodimplementationstest.cpp
- methodsymboltest.cpp
- nullablevaluebooleantest.cpp
- nullablevaluenumbertest.cpp
- nullablevaluestringtest.cpp
- parametersymboltest.cpp
- returnsymboltest.cpp
- sourcelocationtest.cpp
- symboltabletest.cpp
- typesymboltest.cpp
- valuetesthelper.h
- varsymboltest.cpp

9.15 References

[1] https://en.wikipedia.org/wiki/Array_programming
[2] https://ruslanspivak.com/lsbasi-part1/
[3] *https://link.springer.com/referenceworkentry/10.1007/978-0-387-09766-4_25*

[4] https://en.wikipedia.org/wiki/Matplotlib
[5] https://en.wikipedia.org/wiki/Plotly
[6] https://docs.microsoft.com/en-us/office/vba/library-reference/concepts/getting-started-with-vba-in-office
[7] https://uk.mathworks.com/products/matlab.html
[8] http://progopedia.com/paradigm/array/
[9] https://www.tradingview.com/pine-script-docs/en/v4/Introduction.html
[10] https://en.wikipedia.org/wiki/Open-high-low-close_chart
[11] https://kb.iu.edu/d/agsz#:~:text=The%20difference%20between%20an%20interpreted,program%20written%20in%20assembly%20language.
[12] https://docs.python.org/3/reference/grammar.html
[13] https://en.wikipedia.org/wiki/Lexical_analysis
[14] https://en.wikipedia.org/wiki/Parsing

[15] https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/
[16] https://en.wikipedia.org/wiki/Side_effect_(computer_science)
[17] https://en.wikipedia.org/wiki/Abstract_syntax_tree
[18] https://en.wikipedia.org/wiki/Syntactic_sugar#:~:text=In%20computer%20science%2C%20syntactic%20sugar,style%20that%20some%20may%20prefer.
[19] https://en.cppreference.com/w/cpp/language/operator_precedence
[20] https://www.techopedia.com/definition/22321/statically-typed#:~:text=Statically%20typed%20is%20a%20programming,with%20variables%2C%20not%20with%20values.
[21] https://www.cplusplus.com/reference/string/string/
[22] https://en.wikipedia.org/wiki/Void_type#:~:text=The%20void%20type%2C%20in%20several,writing%20to%20their%20output%20parameters.
[23] https://en.wikipedia.org/wiki/Standard_library
[24] https://clouds.eos.ubc.ca/~phil/docs/problem_solving/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments.html#:~:text=Positional%20arguments%20are%20arguments%20that,positional%20argument%20listed%20third%2C%20etc.
[25] https://en.wikipedia.org/wiki/Function_overloading
[26] https://en.wikipedia.org/wiki/Symbol_table
[27] https://en.wikipedia.org/wiki/Array_programming
[28] https://en.wikipedia.org/wiki/Literal_(computer_programming)
[29] https://en.wikipedia.org/wiki/Locality_of_reference#:~:text=Spatial%20locality%20(also%20termed%20data,in%20a%20one%2Ddimensional%20array.
[30] https://en.wikipedia.org/wiki/Static_variable#:~:text=In%20computer%20programming%2C%20a%20static,entire%20run%20of%20the%20program.
[31] https://en.wikipedia.org/wiki/Symbol_table#:~:text=In%20computer%20science%2C%20a%20symbol,or%20appearance%20in%20the%20source.
[32] https://www.techopedia.com/definition/22304/stack-frame#:~:text=A%20stack%20frame%20is%20a,existent%20during%20the%20runtime%20process.
[33] https://en.wikipedia.org/wiki/Call_stack
[34] https://en.wikipedia.org/wiki/Stack-based_memory_allocation
[35] https://en.wikipedia.org/wiki/Fortran
[36] https://en.wikipedia.org/wiki/Inline_expansion
[37] https://en.wikipedia.org/wiki/Control_flow
[38] https://en.cppreference.com/w/cpp/container/vector
[39] https://en.wikipedia.org/wiki/Integrated_development_environment
[40] https://en.wikipedia.org/wiki/Comma-separated_values
[41] https://www.json.org/json-en.html
[42] EURUSD data sourced from: https://www.myfxbook.com/forex-market/currencies/EURUSD-historical-data
[43] https://www.investopedia.com/terms/b/backtesting.asp#:~:text=Backtesting%20is%20the%20general%20method,to%20employ%20it%20going%20forward.
[44] Rainfall data sourced from: https://www.metoffice.gov.uk/research/climate/maps-and-data/uk-and-regional-series
[45] https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)
[46] https://www.gnu.org/software/bison/
[47] https://github.com/ocornut/imgui
[48] https://www.opengl.org/
[49] https://github.com/epezent/implot#:~:text=ImPlot%20is%20an%20immediate%20mode,requires%20minimal%20code%20to%20integrate.
[50] https://github.com/BalazsJako/ImGuiColorTextEdit

[51] https://github.com/AirGuanZ/imgui-filebrowser
[52] https://www.boost.org/
[53] https://www.boost.org/doc/libs/1_75_0/libs/test/doc/html/index.html
[54] https://github.com/nlohmann/json
[55] https://en.wikipedia.org/wiki/Memory_leak
[56] https://www.qt.io/
[57] https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/
[58] https://en.wikipedia.org/wiki/Pointer_(computer_programming)
[59] https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage?view=vs-2022
[60] https://support.microsoft.com/en-us/office/file-formats-that-are-supported-in-excel-0943ff2c-6014-4e8d-aaea-b83d51d46247
[61] https://www.tradingview.com/pine-script-reference/#fun_abs

[62]

https://uk.mathworks.com/help/matlab/referencelist.html?type=function&category=index&s_tid=CRUX_lftnav_function_index

[63] https://sourceforge.net/p/winflexbison/wiki/Visual%20Studio%20custom%20build%20rules/
[64] https://regex101.com/
[65] https://philippegroarke.com/posts/2018/c++_ui_solutions/
[66] https://en.cppreference.com/w/cpp/language/operator_precedence
[67] https://www.cs.virginia.edu/~cr4bd/flex-manual/Start-Conditions.htm
[68] https://stackoverflow.com/questions/2039795/regular-expression-for-a-string-literal-in-flex-lex
[69] https://github.com/nothings/stb/blob/master/stb_image_write.h
[70] https://github.com/ocornut/imgui/wiki/screenshot_tool
[71] https://stackoverflow.com/questions/33268513/calculating-standard-deviation-variance-in-c
[72] https://tutorialspoint.dev/algorithm/mathematical-algorithms/program-find-correlation-coefficient
[73] https://stackoverflow.com/questions/15843525/how-do-you-insert-the-value-in-a-sorted-vector
[74] https://www.geeksforgeeks.org/c-program-to-check-prime-number/
[75] https://stackoverflow.com/questions/2913215/fastest-method-to-define-whether-a-number-is-a-triangular-number
[76] https://stackoverflow.com/questions/4654636/how-to-determine-if-a-string-is-a-number-with-c