Chapter 1: The Project

1.1 Introduction

The point to it all...

The aim of this project is to design and implement a tool for the investigation and visualisation of filters. The particular example of the surface acoustic wave (SAW) filter is used to provide context for the project. It must be noted that due to the scale of this system it is not within my remit to carry out such experimentation using the tool. It's capabilities are demonstrated, in particular for the cornu spiral and in other calculations such as Discrete Time Fourier Transform. The tool is fit for such experimentation and is considered to be complete. The project is, first and foremost, an object-oriented software engineering endeavour. The mathematics and concepts concerning SAW filter design are included in detail and provide a solid starting point for another party to conduct such an investigation using the tool.

Due to its highly flexible design, the system can be used for numerical and graphical analysis in any area of science or engineering.

Algebraic Parsing and Evaluation... much more than just a desk calculator...

Using a true-algebraic, multi-precedence, parsing and evaluation engine (the 'calculator'), the user provides constants, variables and equations which model the engineering problem being investigated. The greatest feature is that all calculations are performed using complex numbers. Another cornerstone is the ability to apply summation and prodation (the result of multiplying a series of terms) to an expression. Familiar algebraic syntax is used, so equations are input just as they appear on paper.

Unlimited equation and variable storage...

Equations and variables cross-reference each other providing a system capable of storing heavily decomposed algebra. This is a welcome feature when dealing with the rigours of advanced mathematics. Equations local to a particular calcuator are even checked to ensure they do not contain references to themselves, preventing paradoxical equations being stored and more importantly their evaluation attempted - for this is an impossible task!

Multiple, independent calculators are maintained by a Calculator Manager. The user chooses the number of calculators required - each identified by a unique, user-defined (alphabetic) name. Each possesses a local, independent set of user-defined variables and equations. Although only a single calculator is used when investigating an engineering problem, the multi-calculator environment allows unlimited scope for context switching between many disjoint problems stored simulataneously in the computer's memory.

Data Management facilities...

So far, so good - except evaluating an equation for a particular set of variable values is only the first step in engineering analysis. A method for storing sets of input data, giving the data to the calculator, getting the calculator to evaluate equations based upon the data and storing the evaluated results is required. The system gives scope for storing independent tables of unlimited numbers of input values and output results, and of course, automated batch processing of these tables.

A Data Manager is provided to maintain sets of input data, sets of tables (each column being

A Data Manager is provided to maintain sets of input data, sets of tables (each column being identified by a user-defined alphabetic name) and sets of maps which describe which table columns are to be given to the calculator as input data, and which are to be used to store output results gained from the calculator.

Data Interpretation and Presentation facilities...

Pages of raw tabular data strike fear into the heart of many an undergraduate. Being the vehicle for pure, unadulterated evidence qualifies it for a 'First' in arrogance and a 'Third' in accessibility and meaning.

However, transform the columns of digits into a graph and unforeseen patterns, trends, characteristics and meaning appear providing direction, interpretation, insight and understanding of the underlying mathematical model. The way is now clear for cause-effect analysis. How do the model's variables influence overall behaviour? What are the model's limits? How can the model be improved? What are appropriate applications of the model? These, and many more questions, are what we are asking every day of the increasingly modelled world around us.

Communication...

Hang on - there's much more to this than the development of academia. Graphs are, if their information content is restrained, a good vehicle for communicating trends and ideas to the non-scientific community.

Graphing and Graph Management facilities...

The system allows X-Y rectangular graphing and vector graphing of data present in the tables of the Data Manager. Sets of graph specifications, holding axis range and scale information, are maintained by the Graph Manager.

Integrating the Whole...

Calculator, Data and Graph managers are made accessible to the user by the System Manager. Whilst providing access to the databases contained within each of these, the System Manager provides a method to drive and integrate the entire system. The user indicates (by name) which calculator, which data table, which inputs, which table columns are to be loaded with these inputs, and which input/output maps are to be used for processing (ie to evaluate a set of output results).

The data is then graphed using one of the two methods available, using particular columns of the data table as graphing co-ordinates. X-Y graphing requires two columns of (real number) data, vector plotting requires one column of (complex number) data. The user chooses the number and range of points (ie rows in the table) to plot.

The System Manager then does all the following to achieve a final graph:
* Loads data table with input.
* Reads input data from each table row.
* Stores input data in the appropriate calculator for each row.
* Extracts output data from the calculator for each row.
* Plots selected table data on a graph, using a simple linear interpolation to join the points.

A successful and practical piece of Software Engineering with wide-ranging application...

A successful and practical piece of Software Engineering with wide-ranging application...

Developed using C++ and strongly object-oriented, the system is particularly modular and thus there is substantial code re-use (both at data structure level and functional level). Even so, the implementation is over 8000 lines of code and has been my greatest, most challenging and arguably my most successful, undertaking ever.

The calculating engine has the capacity for being able to parse in excess of 180 functions and operators. Of course this scale of functionality is beyond the scope of this project, and has been restricted to around fifty.

There is plenty of scope for further development within the current design. Optimisation would be the first area to look at. Looking further ahead, extending the calculator to cope with matrix operations would open up endless opportunities for increased functionality. Tightening integration between the Calculator Manager and Data Manager allowing the calculators to operate simultaneously upon whole blocks of granular input data would allow for statistical analysis.

A Graphical User Interface would be the most welcome extension. The work involved would be sufficient to warrant an entire project in itself.

1.2 A Road-Map for Completion

This section discusses how the project was approached, the different areas of investigation and how the design and implementation was tackled. It serves as a guide to how the work was partitioned and timetabled. Notice how design and implementation for each block is grouped together. The object oriented approach to software development allows for clean, independent artefacts to be designed, implemented and tested with little or no dependency on other (as yet maybe incomplete) unrelated parts of the system. The managers require the things they are managing to be designed and coded, as these are providing a logical extension to what is already produced. The list is chronological and there is bound to be much cross-development once the system starts to take shape.

It is difficult to estimate the time requirements for each section, but as a guide the first five sections are carried through and documented during the first term, and the bulk of the system is developed during the second term. The Easter vacation is used for documenting the second term's work.

1. *Background investigation into **Surface Acoustic Wave filtering**.*
2. *Background investigation into **Object Oriented Design**.*
3. *Background investigation into using the **drawing and windowing** capabilities of the Macintosh Operating System.*
4. ***Analysis of the system** - decompositon into manageable parts. (see below)*
5. *Design, integration and test of a **data structure for data management**.*
   A templated (ie generic), singly-linked, self-ordering list using one of two discernable data items in each node to determine ordering.

The following involves the design, implementation and test of each block. Please reference the Top Level Design to see how they are related.

5. *Development of the **calculator engine**.*
   Parses and evaluates expressions, stores variables and equations.

6. *Development of the **preprocessor**.*
   Allows math function names provided by the user in expressions to be customized (by the programmer, not the user) - a front end for the calculator when evaluating expressions, and a back end when displaying equations stored by calculators.

7. *Development of the **equation validator**.*
   Checks equations stored in calculators for self-references.

6. *Development of the **Calculator Manager**.*
   CLI access to a list of independent calculators.

7. *Development of the **data structures** managed by the Data Manager.*
   Structures required for holding data arrays, holding column names, holding descriptions of how to load columns with input data, and holding map descriptions of which columns are used for input and which for output when evaluating calculator equations.

8. *Development of the **Data Manager**.*
   CLI access to separate lists of each of the four data structures in section 7.

9. *Development of the **data structures** manager by the **Graph Manager**.*
   Structure required to hold x/y axis scale and range graphing information (a graph specification).

10. *Development of the **Graph Manager**.*

10. *Development of the **Graph Manager**.*
CLI access to a list of graph specifications.

11. *Development of the device which creates **graph windows** and their artefacts:*
Draws and scales axes pertaining to a graph specification. Also translates graphing co-ordinates to actual screen co-ordinates for the same graph specification.

12. *Development of **System Manager 1**.*
First stage in total integration. Allows elementary graphing of equations stored in calculators of the Calculator Manager. Data and Graph managers are complete, and their CLIs are operational, but their data held within is not used for graphing. Graph scale/range parameters are stored as variables in a 'graph' calculator present in the Calculator Manager. These values are extracted by the System Manager in order to set up graph axes. There is no facility for storing evaluated results.

13. *Development of **System Manager 2**.*
Second stage in total integration. Final stage in this system's development. Unifies all managers, enabling all features mentioned in the Introduction.

14. *Production of a **user manual** for the system*

15. *Examples of **applying the system** and generating data/graph results.*

1.3 A Road-Map for the Report

1.3.1 Background Material

These two sections are not related and may be read separately.

Surface Acoustic Wave Background

This section examines the motivation for the project.

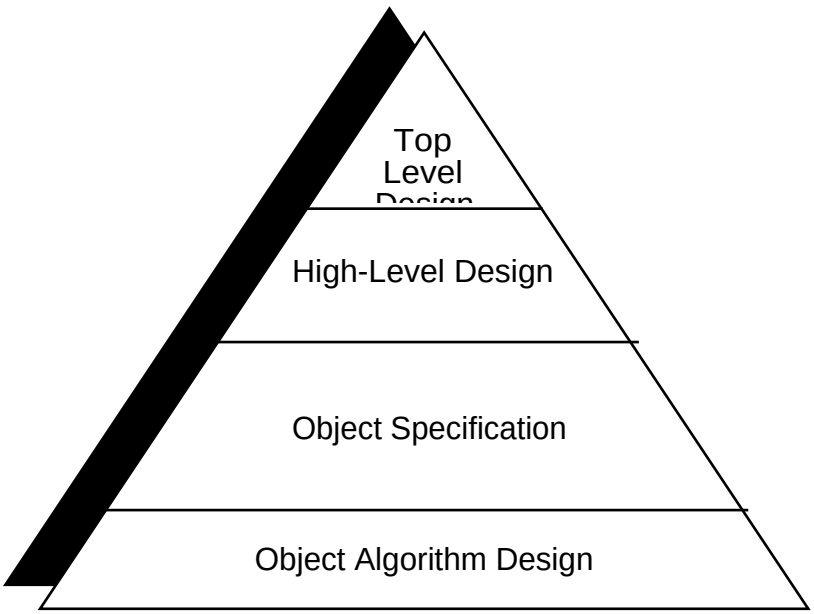
- Introduction to SAW filter technology
- Qualitative Study
- Quantitative Study
- Primary Aim to link in with the developed system

Object-oriented Design - Booch Method Background

1.3.2 System Development - a guide to documentation

The **requirements analysis** and **user manual** sections describe the two ends of the project development cycle. They cover the expectations of the system and how they are accessed by the user. They are good for gaining familiarity with the system.

The design of the system is organised thus:



The area of a section in the pyramid above gives a guide to the volume of documentation associated with that section. The width can be considered to be a guide to the amount of detail that a section has interest in. Design gives way to implementation in the direction from apex to base.

The pyramid could quite easily be partitioned vertically, showing granuality of system components increasing towards the base.

Each sub-section in the **object specification** corresponds to the contents of a single C++ header file. Here, class, data member and function member declarations and roles are defined, along with pre- and post-conditions for each function, and where appropriate, return data for those functions. File dependencies are also included. This section describes the behaviour of the entire system.

Each sub-section in the **object algorithm design** corresponds to the contents of a single C++ code file. Here all static data members have values associated with them (this is data common to all objects of a particular class) and pseudo code may be found for every class function (and related functions such as overloaded output operators). This section describes the implementation of the entire system.

There is a one-to-one relationship between sub-sections in both of these sections. There is one exception to this rule (which is highlighted) which has no pseudo code as the file contains only enumeration declarations.

The **high-level design** groups together modules, presenting their key features and design techniques used. Utility classes such as complex number and string classes are treated singularly.

The **top-level design** abstracts further, representing the object structure required for each Manager, and their dependent objects. Other interesting object diagrams are included to provide a feel for how the system modules fit together.

Volume of documentation increases as granularity increases. It is advisable to approach the more implementation-oriented sections with reference to the top-level diagrams, which will highlight the more interesting objects.

1.3.3 System Documentation Organisation

Requirements Analysis

Broad expectations of each major system component, organised thus:

System Manager 1
System Manager 2
Data Manager
Graph Manager
Calculator Manager
Calculator

Specification - reference the User Manual

Top-Level Design: Diagramatic Documentation, organised thus:

System Manager 1&2
Graph Manager
Data Manager
Calculator Manager

High-Level Design: Modules grouped - detailed specifications, organised thus:

Linked List Class
List Node Classes
System Manager 1
System Manager 2
Calculator Manager Class
Calculator Class
Preprocessor Class
Data Manager Class
Graph Manager Class
Graph Device Class
Validator Class
Complex Number Class
String Class

Low-Level Module Design : Detailed module specifications, including data and function member specifications.

Please see contents page for organisation of this section.

Algorithmic Design: Pseudo code for all module functions.

Please see contents page for organisation of this section.

User Manual (serves as a specification for the system).

Using System Manager 1
Using System Manager 2
Using Calculator Manager
Using Data Manager
Using Graph Manager

1.3.4 Remainder of the Project Report

Using the System.

Applying the System to SAW filter analysis.

Chapter 2: SAW Filters

2.1 Introduction to Surface Acoustic Wave Transducers

2.1.1 The Problem

There are many potential applications for a filter design methology which accepts a set of filter characteristics (eg band-pass limits, skirt steepness, ripple percentage) and transforms this to a set of physical filter parameters (eg spectral weightings). This section examines the design of apodized, unapodized and hybrid (involving the use of both these techniques) Surface-Wave transducers. Implementation of 'match' filter designs, which maximise the signal-to-noise ratio of a specific signal, are of particular interest. Match filters are heavily employed in radar systems due to this property.

2.1.2 The Aims

In order to simulate such a system the following components are considered: an input signal, a pair of Surface-Wave Transducers mounted upon a crystalline, piezo-electric material (eg quartz), and methods for analysing output characteristics. Possible tracks for investigation using the system include: ripple reduction by modifying input signal envelope and how transducer parameters (such as inter-digit spacing and digit-overlap) affect the impulse response of the filter. A significant goal would be the design of filter impulse response, using a frequency response specification.

2.1.3 The Methods

Input parameter controllers may include: modulation waveform, modulation envelope, amplitude envelope and phase envelope. Filter parameters may include: total length of transducer (measured in digits), inter-digit spacing envelope and inter-digit overlap envelope (separate parameters for both transmitting and receiving transducers). Output analyzers may include: gain - time, phase - time, gain - frequency, phase shift -frequency and a 'Cornu Spiral' representation (decomposing single-frequency response into individual finger gain and phase data).

2.2 SAW Filter Qualitative Background

2.2.1 Introduction

This chapter is intended to provide an introduction to the ideas and theories investigated by this project. The principal source for the material in this section is Reference 3. A more analytical treatment is provided in the following chapter.

2.2.2 Motivation

A major requirement of a radar system is to overcome receiver noise and related 'clutter'. By improving the signal-to-noise ratio at the receiver, smaller signals (reflected from radar target echoes) may be detected from what appears to be a noisy channel containing negligible discernable information. The optimum filter for achieving this is the 'matched filter', which provides the largest possible peak output for a particular signal and a specified noise level.

The filter need not preserve wave-shape, as the next stage in the signal processing (the threshold detector) operates given maximum signal amplitude alone. The only requirement is that timing of the signal is preserved (by either a fixed delay, or some predeterminable relationship), allowing range-finding later in the processing system. The order of the delay should be of equal magnitude to the wavelength of the detected signal. This permits detection within a reasonable time-frame, relative to the characteristics of the signal being processed.

2.2.3 The Match Filter

The filter's impulse response takes the form of a time-reversed copy of the particular signal that is to be optimally detected. The implication is that unless adaptive filtering techniques are used, the system being discussed must operate upon one particular signal. The convolution of such a signal and it's time-reversed impulse response yields the signal's 'auto-correlation' function. Whatever the form of the signal, the autocorrelation function is symmetrical with a peak value at its centre. This peak is used for threshold detection.

Although match filters provide the best method of signal-to-noise enhancement, they are not perfect and 'false' echoes will trigger the threshold detector - albeit with frequency statistically minimized.

Match Filter Operation

Reference figure 1. Part (a) shows two rectangular echo pulses, and part (b) shows the received signal prior to filtering. If this signal were to be processed directly by the threshold detector, some of the noisy peaks would be incorrectly identified as echoes. Part (c) illustrates the action of the match filter on this signal. Compared with the noise level, signal peak is considerably greater, permitting the use of a threshold level, V_T.

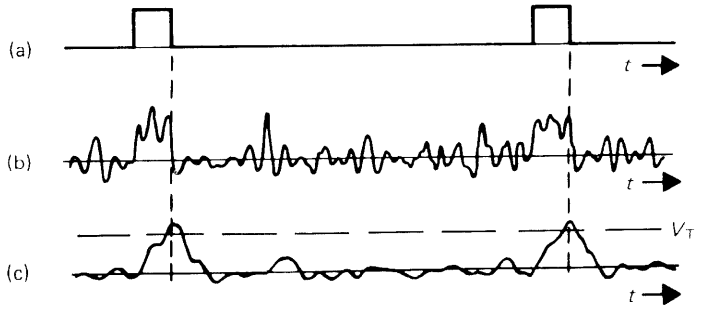


Figure 1: Improving signal-to-noise ratio by matched filtering.

The match filter outputs a triangular wave (given a noisy rectangular wave) which has a peak signal synchronised with the trailing edge of the input pulse. The detection method therefore involves a time delay equal to the period of the input signal.

Time-Domain Analysis of Match Filter

The impulse response of the filter, h(t), is defined as a time-reversed version of x(t) - the time-limited signal pulse input whose waveform is known in advance. As this is a linear filter operation, convolving the input pulse with the impulse response yields the output of the system:

$$y(t) = x(t)*h(t) = \int_{-\infty}^{\infty} x(\tau')h(t-\tau')d\tau \qquad \text{Eqn 1}$$

Where τ' is an auxiliary time variable

In the particular case of the match filter:

$$h(t) = x(t_0-t), \text{ giving } h(t-\tau') = x(\tau'+\{t_0-t\}) \qquad \text{Eqn 2}$$

Hence:

$$y(t) = \int_{-\infty}^{\infty} x(\tau')x(\tau'+\{t_0-t\})d\tau' \qquad \text{Eqn 3}$$

This equation is identical to the 'autocorrelation' operation, hence the autocorrelation references above. The peak value occurs at its centre and this occurs after a delay equal to the duration of x(t). The maximum output value may be found by substituting $t=t_0$ in equation 3:

$$y(t_0) = \int_{-\infty}^{\infty} x(\tau')x(\tau')\,d\tau' = \int_{-\infty}^{\infty} x^2(\tau')d\tau' \qquad \text{Eqn 4}$$

The is a measure of the total energy of x(t). The benefits of match filtering depend only upon signal energy, rather than detailed signal shape. Regardless of the transmitter waveform, target detectability depends upon average power.

Frequency-Domain Analysis of Match Filter

Time reversal does not affect the magnitude of the spectral function corresponding to the time-reversed x(t) impulse response, h(t) - only phase is affected. Therefore, strong frequencies in the original input will be strongly transmitter from the filter. Weaker components will be transmitted less so. Noise will therefore be eliminated outside the signal's bandwidth, optimising the signal-to-noise ratio.

The response of the match filter is the complex conjugate of the signal's phase spectrum. This realignment of phases of various components of x(t) results in all sinusoidal components of y(t) reinforcing each other, forming a peak at $t=t_0$. See Figure 2, illustrating the frequency characteristics of the match filter.

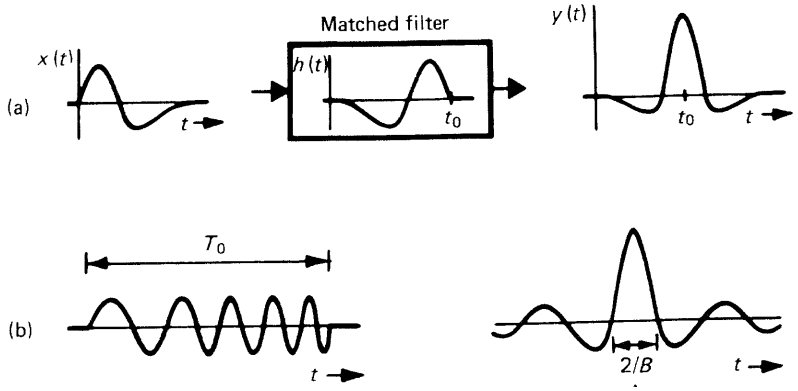


Figure 2: (a) The matched filter. (b) Pulse compression of a 'chirp' waveform.

2.2.4 Signal Input

Pulse Compression

Pulse compression involves sending 'long' coded transmission pulses in time. These are 'sharpened up' or compressed in the receiver using the match filter. The advantage of spreading out the energy in the time domain, is that peak power output from the transmitting device is greatly reduced, whilst maintaining the mean peak power. In order to achieve optimal signal detection, the filter must be realised with considerable accuracy. In radar applications, particularly long-range, high-power systems, reducing power is important as peak electrical stresses are lowered improving device reliability. Bandwidth is proportional to the reciprocal of pulse period.

Linear Frequency Modulation

The main criterion is that the signal should have a narrow, well-defined autocorrelation peak allowing the signal to be effectively sharpened through the Match Filter. This implies a wide bandwidth in the frequency domain. In order to spread the energy throughout the duration of the signal, the amplitude should be without large fluctuations. Practical compression systems use frequency-coded or phase-coded waveforms. The former method will now be examined in detail. Phase-coded systems are not considered here.

Linear FM pulse compression involves the transmitter frequency changing linearly with time over the duration, T_0 of the pulse. Reference figure 2. The pulse envelope will contain hundreds or thousands of RF cycles, in practice. This type of signal is known as a ***chirp*** waveform. Assuming there is a frequency change of B Hz, then its autocorrelation function will have a sin(x) / x form with a central peak of width 2/B.

Pulse Compression Ratio

This is the ratio between the duration of the transmitted pulse and the received pulse after match filtering. It is also a measure of the increase in effective peak power of the transmitter, due to the compression. In Linear FM, the compression ratio is BT_0.

Limitations of Pulse Compression

There are problems with the *time sidelobes* occuring in the autocorrelation function. These occur either side of the main peak. There is a risk that these may be mistaken for separate target echoes, or may mask echoes from other smaller targets. The time sidelobes may be reduced by weighting of the received pulses in the time or frequency domains.

It is also important to note that the system's minimum range is limited by the transmitter pulse length. This is because a pulse radar cannot receive echo signals until transmission of the pulse is completed.

2.2.5 The Surface Acoustic Wave (SAW) delay line

This is one method for making a matched filter for pulse compression, operating at an intermediate frequency (IF) from between 30 and 70 Mhz. It has been widely adopted by radar designers.

The SAW device consists of input and output transducers mounted on a piezoelectric substance such as quartz. An electrical signal applied to the input transducer couples to the crystal's surface, inducing an equivalent mechanical deformation. This travels as an ultrasonic wave along the surface. As this wave passes the *interdigital electrodes* of the output transducer, an electrical output is generated which is characterised by the electrode geometry. Figure 3 illustrates the transducer configuration.

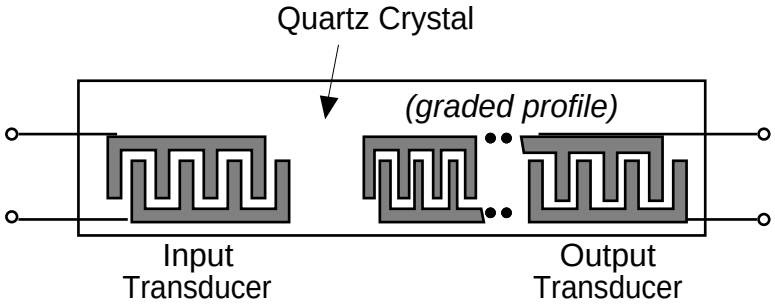


Figure 3: SAW transducer configuration

The operation of the SAW device may be described in two ways :

1. Frequency-Response

Consider the chirp signal, having a low-frequency leading edge and a high frequency trailing edge. By delaying the leading edge relative to the trailing edge, it is possible to achieve pulse compression using the frequency-dependent, or *dispersive*, delay line.The output transducer provides efficient acoustic-to-electrical coupling only when the electrodes are spaced apart by one half-wavelength of the propagating signal. By applying a continuous gradation to the electrode spacing, lower-frequency components in the chirp waveform travel further along the surface before contributing to the output signal.

2. Impulse-Response

In order to be a time-reversed version of the signal to be matched, h(t) must itself be a chirp waveform, but with a frequency which decreases towards the trailing edge. Consider a brief impulsive signal transmitted by the input transducer. The acoustic impulse passes each pair of output transducer interdigital electrodes in turn, inducing an output in the electrical connections of the output transducer at each electrode. This output has a frequency which is inversely related to the electrode spacing at each pair of electrodes. The outputs sum to provide the required impulse response.

3. Elementary Characteristics

Consider a unit impulse input to a single element transmitter. The element will resonate at a frequency proportional to the inverse of twice the distance between the element's two fingers. The frequency response is described by a (sin x / x) function. Now consider two elements, with different inter-finger distances. A second sin x / x response (with phase shift relative to the original response, due to a different position on the crystal medium) is summed with the original response. As more elements are added, the sin x / x components sum to form an approximate band-pass response, as long as the inter-finger distances shorten quadratically across the transducer.

The signal generated on the crystal medium is Linear FM in nature due to the quadratic change in finger-positions corresponding to a resonance frequency sweep across the transducer. The change is quadratic as finger position is decided by phase. Being the integral of frequency, phase will be described by a quadratic function, assuming a linear frequency function of time.

Given that the inter-finger distance in the first element (with respect to the electrical supply) of the transducer resonates at a lower frequency than that of the final element, and that mirrored transmitter and receiver transducers are being used, low frequencies propagate further through the crystal medium than high frequencies before being detected. Phase shift due to traversal through the medium decreases linearly with frequency. Put another way, the complex-response rotates anti-clockwise with frequency. This is called dispersive delay.

Non-dispersive delay, determined by the phase characteristics (ie construction) of the transducer array, is unimportant in applications where identical transducers are used for both input and output. Due to the mirrored configurations non-dispersive delay is cancelled out.

The output of a surface-wave acoustic filter having identical transducers, in response to an input signal $V_i(t)$ can be written as :

$V_0(t+r) = V_i(t)*h(t)*h(-t)$

where h(t) is the unit impulse response of the input transducer, τ is the non-dispersive delay and * indicates convolution. Transforming to the frequency domain we obtain :

$$V_0(f).\exp[j\varphi_0(f)] = V_i(f)\exp[j\varphi_i(f)].H(f)\exp[j\varphi_h(f)].H(f)\exp[-j\varphi_h(f)].\exp[-j2\pi f\tau]$$
$$= V_i(f)H(f)^2.\exp[j\varphi_i(f)].\exp[-j2\pi f\tau]$$

where the notation that $V_0(t)$ transforms to $V_0(f).\exp[j\varphi_0(f)]$, etc., is used. The output signal $V_0(f).\exp[j\varphi_0(f)]$ is independent of the phase of the transducers $\varphi_h(f)$.

4. Design Parameters

There are three design parameters: the number of digits used in the transducer, the interdigit spacing function and the apodizing function across the transducer. The first of these is determined by the bandwidth of the filter, the second by the characteristics of the signal being matched (ie LFM) and the third by the weightings applied by the filter across the frequency spectrum.

The interdigit spacing function is determined by examining the phase-change, with time, of the transmitted signal (LFM). For maximal coupling, there needs to be one finger positioned at every π phase change along the medium.

A wider transmitter bandwidth requires a longer frequency slope (in time) (given the same rate of change in frequency) to be transmitted. Phase is the integral of frequency, so there is also a correspondingly longer phase slope (in time) with increased bandwidth. Thus, a wider bandwidth results in a greater number of π changes in phase, requiring a larger number of fingers to transmit the signal.

Apodization is the term given to the overlap between two successive fingers in the transducer. Overlap is proportional to weighting. The weighting is maximised around the frequency $[2(t_{n+1} - t_n)]^{-1}$, where t_n is the position of finger 'n'. The summation of all weightings across the transducer gives an overall apodization function. The product of the frequency response of the unapodized transducer (where overlap is constant for all fingers) and an apodizing function gives the frequency response of an apodized filter.

Apodization is one method for obtaining a nominally flat pass-band frequence response. The unapodized filter possesses a gain envelope that is proportional to the cube of frequency. See Figure 4. This occurs due to the higher energies associated with higher frequency signals. As the π phase-change positions get closer together, the spectral peaks (sinx/x) get closer together and so overlap increases, enhancing constructive interference. This increases the overall response at higher frequencies. To cancel this cubic growth, the apodizing function contains a term proportional to the inverse cube of frequency.

A disadvantage of using apodization is mentioned in Reference 1 :

"In some applications the apodization presents complications in that the beam generated has nonuniform width, and diffraction and phase-front problems can result."

An alternative method is presented in this reference, where the number of effective transducer elements is varied as a function of frequency to provide the conversion-loss variation. Fingers are not placed at every π phase change, but are positioned according to desired amplitude response.

5. The Cornu Spiral

Figure 5 shows an example of a cornu spiral vectorial representation for 120-finger unapodized array. This vector polar-plot represents system response to a single frequency sinusoidal input. Each line segment of the spiral corresponds to the gain and phase contribution from a single finger (on the receiver), when excited by a particular frequency. Length represents gain and angle represents phase. The overall response for the system is measured between the two end-points.

The set of cornu spirals describing the transducer array across the frequency domain form a continuous series. The elements forming the two coils are responsible for ripple in the frequency-domain. The elements making up the main sweep of the spiral characterise the system. Cosine-type apodization at the ends of the array can reduce stop-band ripple to any desired limit. An acceptable approximation, under this condition, is to consider only those elements in the range N_1 to N_2 where the phase rotation between the ends of the synchronous band is π.

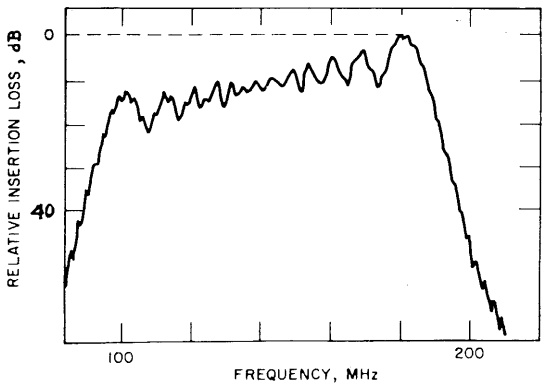


Figure 4: Frequency response of a 120 finger SAW filter

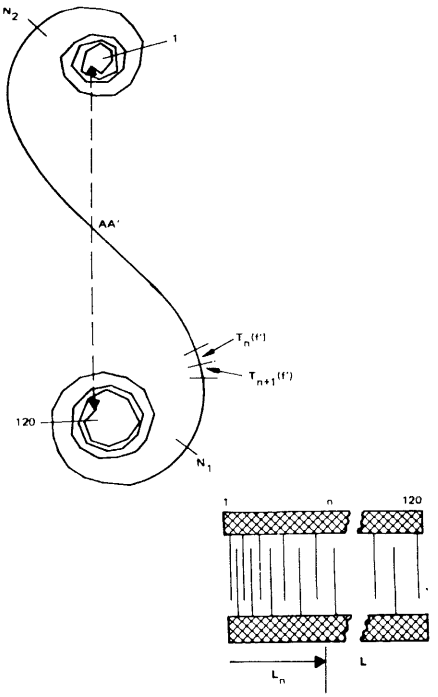


Figure 5: Cornu Spiral showing a vector plot of how the individual digits of the output transducer respond to a particular input frequency.The dashed line is the vector sum of all digit responses, and represents the total response for this frequency.

2.3 SAW Filter Quantitative Background

2.3.1 Introduction

This chapter collects the mathematical background required for the analysis of SAW filters. Included here are key derivations and expressions which would be used to perform a simulation. I have described in the previous chapter how non-dispersive delay between the transmitter and receiver transducers is cancelled. I continue by covering the electro-acoustic coupling model used to derive the transfer function for the system. This serves as a reference, as it is not strictly required within the simulation. The principal source for the material in this section is Reference 1.

2.3.2 The Mason Equivalent Circuit

This models a pair of fingers in terms of input and output acoustic ports and a transformer coupled electrical port. Impedances of the crystal medium between each finger and capacitances associated with the structure are taken into consideration. See figures 3 and 5 for transducer electrode geometry and figure 6, below, for the Mason equivalent ciruit for a single element of the array.

The transformer coupling ratio, r_n, is given by :

$$r_n = (-1)^n.\sqrt{(2f_nC_nk^2Z_0)}.K(q_n)/K(q_n')$$

where

k	: electromechanical coupling constant
C_n	: static electrode capacitance of the section
Z_0	: acoustic-wave impedance
f_n	: synchronous frequency of the element
L_g	: transducer gap dimension
L_s	: transducer electrode width
q_n	: = sin ($\pi L_s/2(L_s+L_g)$)
q_n'	: = cos ($\pi L_s/2(L_s+L_g)$)

and K is the Jacobian complete elliptic integral of the first kind.

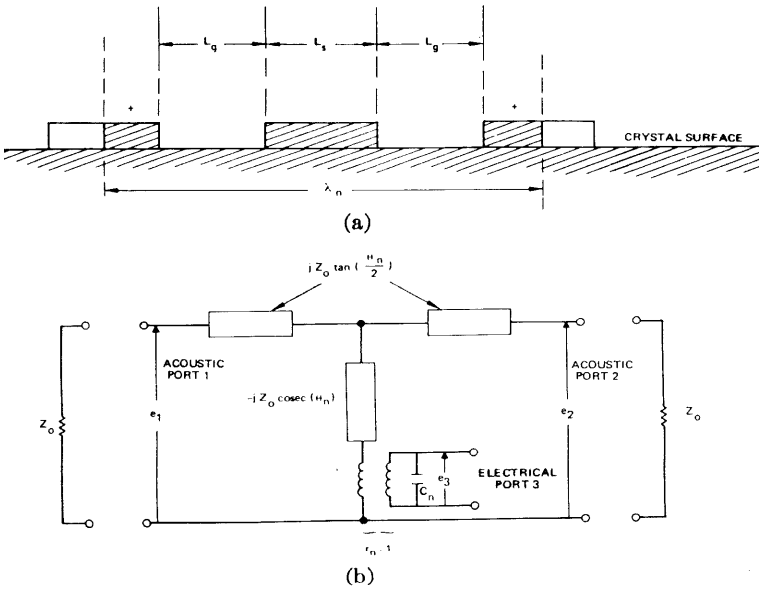


Figure 6: (a) Single element cell considered by...
(b) Mason Equivalent electro-acoustic circuit model.

Transit angle, which is a measure of how well tuned/matched the received frequency is to the resonance frequency of the inter-digit pair, is given by $\theta_n = \pi f / 2f_n$.

The applied voltage e_3 at port 3 and the equivalent voltage e_1 of the acoustic signal are related by :

$$e_1/e_3 = jr_n.\sin(\theta_n).\exp(-j\theta_n)$$

The *j* term in the above equation is due to the effect of the inductors in the mason equivalent circuit. The *sin(θ_n)* term arises due to the spacing between two successive fingers and the *exp(-jθ_n)* is due to the delay exhibited at different positions along the crystal.

The transfer function from port 3 to port 1 is defined as :

$$\begin{aligned} T_{13}(f) &= 2\sqrt{(G_1/G_3)} \,.\, e_1/e_3 \\ &= j2\sqrt{(G_1/G_3)} \,.\, r_n \,.\, \sin(\theta_n).\exp(-j\theta_n) \end{aligned}$$

This may be defined in terms of conversion loss, L_{13} from electrical port 3 to acoustic port 1 as :

$$L_{13} = \left| T_{13}(f) \right|^{-2}$$

The insertion loss (IL) between an external electrical generator e_g and the acoustic ports (referring to figure 6) can be written as :

$$IL = \left| \frac{2}{\sqrt{(Z_0 G_L)}} \,.\, \frac{e_1}{e_g} \right|^{-2}$$

where $e_0 = e_3(G_L + Y_{in})/G_L$:

$$IL = \left| \frac{2}{\sqrt{(Z_0 G_L)(1+Y_{in}/G_L)}} \; r_n \,.\, \sin \frac{\pi f}{2f_n} \right|^{-2}$$

A response for a multi-element array may be found by cascading the equivalent circuits for the individual segments. Assume that the electrical ports are connected in parallel and the equivalent acoustic terminals are summed with the appropriate phase shifts $\exp(-j\omega t_n)$, where t_n is the delay to the centre of the nth section. This gives Insertion Loss and Transfer Function for the entire array as:

$$IL = \left| \frac{2}{\sqrt{(Z_0 G_L)}} \sum_{n=1}^{N} \frac{r_n}{1+Y_{in}R_L} \,.\, \sin \frac{\pi f}{2f_n} \;.\; \exp(-j\omega t_n) \right|^{-2}$$

$$T_{1g}(f) = \frac{2}{\sqrt{(Z_0 G_L)}} \sum_{n=1}^{N} \frac{r_n}{1+Y_{in}R_L} \,.\, \sin \frac{\pi f}{2f_n} \;.\; \exp(-j\omega t_n)$$

where $R_L = G_L^{-1}$ and $\omega = 2\pi f$.

From figure 6, Y_{in} is :

$$Y_{in} \approx G_a \sin^2(\theta_n/2) + jB_a \sin(\theta_n) + j\omega C_T$$

where:
$G_a = 4k^2 f_n C_n$: equivalent acoustic conductance at synchronsim
$B_a = 2k^2 f_n C_n$: equivalent acoustic susceptance at synchronism
C_T : static capacitance of the full array.

Assuming the transducers have constant electrode width-to-gap ratios, $q_n = q_n' = (2)^{-0.5}$, making $K(q_n)/K(q_n') = 1$.

The capacitance of each section C_n is a constant (C_0). For low-coupling materials the approximation $Y_{in} \approx j\omega C_T$ can be made.

2.3.3 Transfer Function based on Phase Solution

$$T_{1g}(f) = \sqrt{(8k^2/G_L)} \cdot \frac{\sqrt{C_0}}{\sqrt{[1+ (\omega C_T R_L)^2]}} \cdot \sum_{n=1}^{N} \sqrt{f_n} \cdot \sin \frac{\pi f}{2f_n} \cdot \exp(-j\varphi_n)$$

where:

$$\varphi_n = 2\pi f t_n + \tan^{-1} (\omega C_T R_L)$$

These two equations are accurate to within a few percent for most transducers. These are used in the design phase to define the finger locations and apertures.

2.3.4 Finger Position Determination

For a two-terminal transducer, finger locations correspond to phases of $n\pi$. The phase at each finger location t_n is expressed as :

$$n\pi = \varphi(t_n) + \tan^{-1} (\omega C_T R_L)$$

For the Linear FM case, h(t) = 1 for $0<t<T$ and zero elsewhere. Phase is described by :

$$\varphi(t) = 2\pi[f_0 - B/2)t + Bt^2/2T]$$
where B is filter bandwidth

Substituting this into the above equation for $n\pi$ gives :

$$n\pi = 2\pi(f_L + Bt_n/2T)t_n + \tan^{-1}[2\pi \cdot (f_L + (B/2T).t_n).C_T R_L]$$
where: $f_L = f_0 - B/2$

Using an approximation for the $\tan^{-1}$ term, the equation determining position for the nth finger is :

$$t_n^2 + \frac{2T}{B}.\left[F_L + \frac{C_T R_L B}{2T[1+ (2\pi f_L C_T R_L)^2]}\right].t_n - \frac{2T}{B}.\left[\frac{n}{2} - \frac{\tan^{-1}(2\pi f_L C_T R_L)}{2\pi}\right] = 0$$

2.3.5 Approximation Transfer Function for Constant Aperture Array

With finger locations determined by the above equation :

$$T_{1g}(f) \approx 4f^{3/2} \cdot \frac{\sqrt{(k^2 C_T)}}{\sqrt{[Bf_0 G_L (1 + (\omega C_T R_L)^2)]}}$$

where: $C_T = f_0 C_0 DT$ is total capacitance of the array.

2.3.6 Apodization to Flatten Frequency Response

Apodization is derived to be expressed by :

$$W_n \approx \text{const.}(f_0/f_n)^3 \cdot [1 + (2\pi f_n C_T R_L)^2]$$

Where W_n is aperture of the nth section.

Cosine weighting (reference the const) may be used with this apodization to reduce Fresnel passband ripple.

Chapter 3: Top Level Design

3.1 Object-Oriented Design and Booch

This section is based upon my knowledge of Object-Oriented design methodology and extracts from notes I made on the book written by Grady Booch, reference 4 in the bibliography.

3.1.1 The Structural Approach To Programming

In traditional, structural design a large system is split up into separate modules, each holding a set of logically related functions. Although there is a theoretical function hierarchy, there is nothing stopping a programmer from accessing any of these functions from any other function. All functions have the same scope. As the number of functions in the system grows the program becomes progressively more difficult to maintain, extend and test.

3.1.2 The Object-Oriented Approach to Programming

In contrast, object oriented decomposes the system design by data. Each module describes an object specification, or class. A class may be viewed as a data structure (as in 'C') which also encapsulates local functions that operate upon that data. A class is a set of objects that share a common structure and a common behaviour. Data and functions residing in a class are called members of that class.

This leads to the feature of data hiding, where a class may contain data and functions which can be privately or publicly accessible. Private members are accessible only by member functions local to that class, although 'friend' classes and functions can be granted access which overrides the private directive. Public members are additionally available from outside the class.

The class is the data type, and an 'object' is an instance of a class. Objects are declared in the same way that simple 'C' data types are declared, eg int a.

3.1.3 Features of C++ which allow for Object-Oriented Design

To access public members of an object, an object's variable name is followed by a period and the name of the member. By making instances of classes data members of other classes, a function hierarchy is developed.

Class overloading operators define the methods for applying standard operators, eg +,-,*,/ to objects of that class. For example, defining these overloaded operators for a complex number class enables complex number objects to be added, subtracted, multiplied and divided using the same syntax as for floating point numbers. The result is simpler, more reliable and more elegant code. Overloaded operators are used extensively in my complex class and my string class.

Function overloading involves defining the same function name within a common scope more than once. Two definitions must not have the same function parameters, otherwise the compiler cannot distinguish between them. Function overloading has been used in my validator class.

Templating allows a generic class to be defined. My linked list class is templated, meaning that as long as the node class of the list conforms to my minimum node specification, any type of list can be instantiated. I use the generic list to build 8 different lists in this project.

Inheritance is the process of defining a sub-type of a class. Consider a class describing all plants. Plants have common features that distinguish themselves from animals. These characteristics form the base class. A more specific class may build on this information by inheriting the members of the base class and specifying a set of its own specific data. Inheritance has not been used in this project. I found that I could design the project without resorting to this method. The calculator class could benefit from being sub-typed as there are really two types of calculator, called SUPER and SUB. However, considering this is the most

really two types of calculator, called SUPER and SUB. However, considering this is the most complex class in the project I didn't think the it would be a worthwhile use of time. Inheritance is not suitable where member functions reference each other. The calculator class uses deep recursive calls between many member functions.

3.1.4 Benefits of the Object Model (Booch)

1) Use of object models assists in exploiting the expressive power of object-oriented languages.
2) Encourages reuse not only of software, but entire designs leading to the creation of reusable application frameworks. (cf linked list design)
3) It produces systems built upon a series of stable intermeditate forms, thus making the design more resiliant to change, permitting systems to evolve over time instead of being redesigned.
4) Integration is spread over the life-cycle.
5) Appeals to human cognition - the benefits are most evident in large systems.

3.1.5 Links, Aggregation and Relationships (Booch)

Links provide a physical or conceptual connection between objects. They denote the specific association through which one object (the client) applies the services of another object (the supplier), whilst objects to naviagate to one another. Linking permits looser coupling among objects. Links are used between calculator class and validator class to allow the calculator to validate its equations.

Aggregation is the formation of a whole/part hierarchy, eg a linked list maintaining a number of objects forming a data structure. The ability to navigate from the whole to its constituent parts is characteristic. This may be directly through data members, or via public member functions of the whole. Aggregation encapsulates parts as 'secrets' of the whole - a black box in effect. Each of the managers in my design uses aggregation. The relationship between a manager and the lists contained within it is a 'using' relationship. The manager uses the lists to perform its function.

3.2 Requirements Analysis

Here follows a broad selection of requirements for several areas of the system.

3.2.1 System Manager 1

Version 1 of the System Manager is the first stage of fully integrating the system. It allows calculations to be performed by the Calculator Manager, displaying the results using a rectangular x-y graph. Parametric graphing is to be allowed. The calculator is provided with the value for a single variable, and evaluates two equations based upon this value - one provides an x co-ordinate, the other a y co-ordinate. This is repeated according to the number of points required on the graph. Point data is not stored after the graph has been drawn.

The graph specifications, ie axes range and scale data, is to be stored as separate variables in a calculator called ‘graph’ in the Calculator Manager.

The calculations are to take place using the caluculator set up as ‘current calculator’ in the Calculator Manager.

3.2.2 System Manager 2

Version 2 of the System Manager is the second and final stage of fully integrating the system. It allows calculations to be performed based upon sets of input data stored in data arrays accessible by the user. A data array is defined by the user, the names and ranges of values for input variables are indicated. The data array has its columns labelled using strings - thus this is an associative array. The user chooses which columns of data to provide the chosen calculator (in the Calculator Manager) with data for each evaluation. The names of the equation to look up after an evaluation has taken place are also provided. These outputs from the calculator are stored in columns of the data array indicated by the user.

A single column labelling method, a single data array, multiple input sets, and multiple input/output column determination are to be used in the evaluation of a set of results.

These items are to be chosen by name by the user. All objects referenced by the user must be present and appropritately initialised in the respective Manager below.

Version 2 provides two graphing facilities, x-y rectangular graphing and polar vector graphing of selected data in a data set. The range of data required for graphing from a particular data set is to be provided by the user.

Evaluated data is stored until the user requests its deletion. Data may be used for graphing and further evaluation for as long as the user keeps the data array in memory.

3.2.3 Graph Manager

The Graph Manager is responsible for holding a collection of graph specifications. These are each identified by a string name unique to the set they are contained within. A graph specification specifies how the scales and ranges of graph axes are to set up. Independent x,y information is available.

3.2.4 Data Manager

The Data Manager is responsible for holding separate collections of column labels, data arrays, input sets and input/output mappings which describe which data is supplied to the calculator before an evaluation takes place, and which data is extracted from equation names in the calculator. Names of items in each collection must be unique to that collection.

3.2.5 Calculator Manager

The Calculator Manager is responsible for allowing user access to a collection of calculators. Each calculator must be an independent entity. Data should not be exchangable between calculators. The user should be able to be able to switch context to any calculator in the Calculator Manager. Once this is done, the user should be able to :

1) Use the calculator as a 'desk calculator' - evaluating expressions.
2) Store variable assignments in the calculator.
3) Store equation definitions in the calculator
4) Select whether automatic equation verification is active.
5) Perform a manual verfication of equations in the current calculator.
6) View and clear the error report inside each calculator.
7) View and clear individual or all variables and equations.

3.2.6 Calculator

Each calculator holds a local collection of variables (each storing a single complex number), a local collection of equations (each storing an expression) and have access to a collection of constants such as pi,e and j common to all calculators.

All calculations are to be performed using complex numbers. All algebraic parsing is to be checked and errors flagged in the error report when there are problems. A method for verifying that equations within a calculator do not contain circular references (resulting in an unsolvable equation) to themselves or other equations in the calculator is required. Equations must be evaluatable by the system if they reference other equations, variables and constants, but not if evaluation is impossible (see verification). If checks weren't made the computer could enter an infinite loop, crashing the machine.

Checks for invalid operands such as divide by 0 are to be made whenever an operator/function is parsed, before it's evaluated.

3.3 Top-Level System Design

3.3.1 Diagrams Introduction

I have used a modified version of Booch's Class diagrams to illustrate the top level system design. The methodology adopted is described on this page.

The system comprises six major entities, and these are headed with a black bar at the top of each entity. Each box shows the name and the data members for a single class in the system. The system design has had to be split over four pages due to its size. Figure 8 collapses each major entity's associated classes, and shows how each major entity is linked with the System Manager. Figure 9 expands the design hierarchy below the Graph Manager. Figure 10 expands the design hierarchy below the Data Manager, and Figure 11 expands the design hierarchy below the Calculator Manager (the real power-house of the system).

To visualise the design on one page, consider the black bar entities to be common over all 4 figures, bearing in mind information has been hidden in the Top Level Design figure.

Broadly speaking, the page represents the hierarchy of classes. From top to bottom, the classes become less abstracted, until you reach string and complex classes at the bottom. A class is linked to a class beneath it if the subordinate class appears in the data members listed in the class box above it. Simple data 'C' data types are not expanded in this way (eg char, int, double, etc.)

Assume links to have one-to-one cardinality, unless otherwise stated. Roles of classes for particular associations are labelled next to the class carrying out the role.

3.3.2 Data Members

Private/Public access modifiers are not included as there is already enough information in the diagrams.

3.3.3 Linked Lists

These are labelled as Ulist<name>. This reflects the generic, templated structure of the ulist class. (Called ulist as the list stores elements only if their index or data fields are unique to the list - the actual dependency is dictated by the object using the ulist. List and Ulist are used interchangably throughout this report).

The angle bracketed name is the object used as the node class in the ulist. Instead of showing the ulist class box and the ulist node class box, I have combined the two as the ulist class has no public data members. The Ulist<name> box represents a linked list comprising <name> nodes, and features the data members for a single node in that list. There is of course a one to many relationship between the nodes and the ulist class, but this isn't shown. The node instantiations are stored using aggregate reference containment within the ulist.

3.3.4 Adornments

If an association has a filled circle at an end, this signifies an aggregation. The class at the circle end can be seen as possessing or containing either an instance pointer (unfilled black square at part end of association), or an actual instance (filled black square at part end of association) of the class at the opposing (or part) end of the association. These adornments highlight hierarchy.

An unfilled circle at the end of an association indicates that the class at this end 'uses' the object at the opposing end of the association. A client/server relationship exists between the objects, with the 'user' being the client.

An arrow pointing to the class name of a ulist means that the object at the non-arrowed end of the association points to a single node of the ulist.

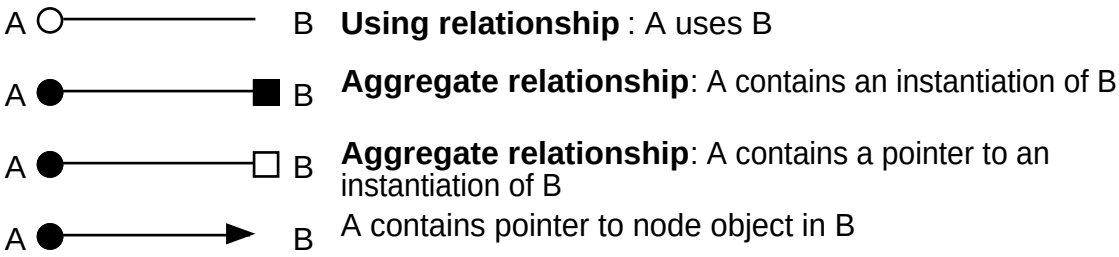


Figure 7: Diagram Relationship Adornments

The ulist<name_object> class featured in the Calculator Manager design is adorned with (2+1 static). This shows that a calculator object has access to two ulist instantiations local to the object (the variable and equation lists), and one which is generic to all calculator objects (the constant list). The double link between Calculator class and ulist<name_object> shows how their are different links according to whether the Calculator class has SUPER or SUB rank. SUPER rank calcuators contain the variable and equation lists by reference forming an agregate. SUB rank calculators use variable and equation lists residing in a SUPER calculator present in the system. They can read the referenced lists, but may not change them, for they are officially owned by the SUPER calculator.

SUPER calculators are used to evaluate a root equation. SUB calculators are instantiated by SUPER calculators to calculate derivative equation expressions referenced by name in the root equation. SUB calculators are automatically constructed and destructed as and when a derivative calculation needs to be performed. This relationship is not indicated in the diagram due to its complexity.

Booch Top Level Diagram Here

Graph Manager Booch Diagram here

Data Manager Booch Here

Calculator Manager Here

Chapter 4: High-Level Design

4.1 System Manager 1 Design

- **Key Features common to System Manager 1 and 2**
  - Instantiates a preprocessor object, sets the complex container class static data member 'postprocessing' to use this preprocessor to perform all postprocessing operations, instantiates a calculator manager using the preprocessor object to perform all preprocessing operations, instantiates a data manager object and a graph manager object, and instantiates a graph_device object.
  - Defines the user labels array (required by the preprocessor object) which maps user function names to operation names.
  - Defines the constant list available to all calculator objects - stores values for pi, exp and j (real=0, imag=1).
  - Sets up the SIOUX (Simple Input/Output User eXchange) terminal window to an appropriate size.
  - Initialises the Mac OS GUI managers permitting window creation and drawing to window graphics ports.

- **Key Features of System Manager 1:**
  - Contains the main() function for version 1 of the program.
  - Demonstrates data manager and graph manager interfaces and maintainance of lists contained therein.
  - Allows access to the calculator manager interface.
  - Uses the method described in the general description to achieve plotting in a graphing window of parametric data evaluated using the current and 'graph' calculators set up in the calculator manager.

- **General Description:**

The program begins by calling the Data Manager interface. After exiting this, the Graph Manager interface is called. Note that access to these two managers is purely for demonstration purposes. No information contained within these managers is used for graphing.

The program then calls the Calculator Manager interface. Upon exiting this, the following variable values are extracted from the 'graph' calculator which must be present in the calculator manager:

xmin : Minimum x-axis range
xmax : Maximum x-axis range
xscale : X-axis numbering scale, determines placing of major divisions
xdiv : Number of minor divisions per major division on x-axis
ymin : Minimum y-axis range
ymax : Maximum y-axis range
yscale : Y-axis numbering scale, determines placing of major divisions
ydiv : Number of minor divisions per major division on y-axis
varmin : First 'var' setting to be sent to current calculator - first point to be plotted.
varmax : Last 'var' setting to be sent to current calculator - last point to be plotted.
res : Value 'var' setting in current calculator is increased by between plotting points.

The x/y data is used to initialise a graph_device object which draws a graph using appropriate axes and scales in a graphing window.
The current calculator targetted by the calculator manager is expected to contain the

The current calculator targetted by the calculator manager is expected to contain the following equations:

xgraph : Parametric x data value dependent upon ‘var’ variable
ygraph : Parametric y data value dependent upon ‘var’ variable

All complex number values extracted/evaluated using the calculator are stripped of their imaginary component, and only their real double value is stored.

◇ Carrying out an evaluation

For each (x,y) plot point required, ‘var’ variable in target calculator is set using an assignment calculator order. The values of ‘xgraph’ and ‘ygraph’ are then extracted from the targetted calculator by sending a reference order for each - the calculator works out the evaluation of each equation returning the resulting complex number.

xgraph and ygraph values are then translated to graphing window co-ordinates using the graph_device object. A 4 pixel wide cross is placed at each plot point on the graph. These are joined using straight line interpolation.

Points are plotted for each sample value of ‘var’ lying between varmin and varmax with sample period ‘res’.

Once the graph plotting is complete, the calculator manager interface is called once again and all the above is repeated allowing the user to change graph parameters and calculator equations in order to draw different plots. Further access to Data and Graph Managers is not available.

◇ Constraints:

Graphing is restricted to rectangular plots. Parametric plots must have ‘var’ variable increased linearly across a specified range. X and Y evaluation must be stored in xgraph and ygraph equations in the targetted calculator. Point data is not stored, its just plotted.

◇ Efficiency:

Redrawing the same graph requires complete re-evaluation of all data by the targetted calculator.

4.2 System Manager 2 Design

- **Key Features common to System Manager 1 and 2**
  - ◇ Please refer to System Manager 1

- **Key Features of System Manager 2:**
  - ◇ Contains the main() function for version 2 of the program.
  - ◇ Allows access to Calculator, Data and Graph Managers.
  - ◇ Allows user to set up process parameters.
  - ◇ Allows user to view current process parameters.
  - ◇ Allows user to execute the process.
  - ◇ Allows user to graph the process using rectangular axes or a polar vector plot (eg for plotting the cornu spiral).
  - ◇ The system manager may be reset by the user, clearing all lists in all managers and setting process parameters to blank values.

- **Command Line Interface:**
  - ◇ A CLI is provided for accessing the features above.
  - ◇ The user may quit the program, by issuing the 'quit' command.
  - ◇ The user may obtain command help by issuing the 'help' command.
  - ◇ Full CLI details may be found in the user manual.

◇ **General Description:**

Integrates all Managers under a unifying interface enabling complete system interaction. Before a process can be executed the following must be done:

The user must set up equations in a target calculator using the calculator manager. The data set used to store input data and results of equation evaluation must be defined in the data manager by the user (using the data set list). The columns/fields labels for the data set must be defined in the data manager (record list). The input data that is to be placed into the data set before evaluation takes place must be defined using the data manager (using the set input list). The input/output mapping(s) required to carry out the evaluation must be defined using the data manager (using the map list). If the user wishes to graph the process, a graph specification must be defined using the graph manager (using the spec list).

In order to execute a process, the user must set the process parameters in accordance to the description in the Glossary. If the names provided here do not reference valid objects present in the relevant managers, an error is reported and execution is aborted.

◇ **Process Execution:**

The System Manager uses the process parameters to direct process execution. The named calculator is used for all calculations. The named data set is used to store all input data and calculation results. Each 'set input name' in the process object is applied to the column/field of the data set array labelled by the corresponding 'set input field' in the process object.

The set input named first in the process's 'Set input names' list is applied to the data set column labelled with the first string in the process's 'set input fields' list. This is repeated for all set input names referenced in the process. There must be equal numbers of strings in both the 'set input names' list and the 'set input fields' list. A set input may be applied to more than one column/field and multiple set inputs may be applied to the same column/field.

Once the data set is loaded with input data, the System Manager begins the task of loading the target calculator with input data from specified columns (input fields) in each row, obtaining results from the calculator and storing this data in specified columns (output fields) in the data set array.

A single Input/Output map describes the input fields and output fields for a single transaction with the calculator. Each input field's name is assumed to be a variable in the calculator, which is set to the value stored in the input field of the row currently being evaluated. All the input fields pertaining to the current IO map are loaded into the calculator in this way. The output fields names are assumed to be equation names in the calculator and their values are then extracted by sending a single calculator reference order for each output field. The results are stored in the respective columns/fields of the data set.

The System Manager process allows multiple IO maps to operate on each row of the data set. IO maps are applied to a single row in the order found in the map names list of the process. The IO maps are applied to every row of the data set using this ordering.

◆ Process Graphing:

The System Manager uses the process parameters to direct process graphing. The named data set contains the data to be plotted. The named graph spec is used to set up graph axes' ranges and scales.

The System Manager is capable of producing rectangular (x,y) graphs and polar vector graphs (eg cornu spiral). The type of graph required is requested from the user.

◇ Rectangular Graphing:

The user is asked for the name of the column used for x-axis data, and the name of the column used for y-axis data. The rows range that is to be plotted is then requested. Data is read from the appropriate columns and rows in the data set. Only the real component of the complex number stored in each element is used. The graph_device object is used to translate each (x,y) graph co-ordinate to graph window drawing co-ordinates. A 4 pixel wide cross is placed at each plot point on the graph. These are joined using straight line interpolation. One point is plotted for each row in the row range.

◇ Polar Vector Graphing:

The user is asked for the name of the column containing vector data. The rows range that is to be plotted is then requested.

Axes and scale numbering are not displayed. The graph spec parameters are interpreted differently for polar vector graphs. An origin where the plot is to start must be specified and a scale-factor in both horizontal and vertical directions is required.

The scale information in each direction is read from the Major Scale Division data in the horiz_scale and vert_scale members in the graph spec. It is interpreted as the number of pixels to be moved horizontally per unit real component of the vector data, and the number of pixels to be moved vertically per unit imaginary component of the vector data.

The origin co-ordinate is measured from the bottom left corner of the graphing area. It is obtained from the Number of Minor Ticks data field in the horiz_scale and vert_scale members in the graph spec. It is interpreted in terms of unit real and unit imaginary vector components. Thus the origin is dependent on scale-factor values.

Complex numbers are read from the vector's column in the specified rows of the data set. Each complex number read in is split into its real and imaginary components. Both components are multiplied by their respective scale-factors. The scaled real component moves the graphic pen position in a right direction (for positive real) and the scaled imaginary component moves the pen position in an upwards direction (for positive imaginary). A line is drawn from old pen position to new pen position. A 4 pixel wide cross is placed at the end of each vector.

Once the first vector has been drawn, the second vector is drawn relative to the new pen position. This is repeated for each vector contained in the rows range.

◇ Glossary:

Process: System_process object containing a specification for using a calculator to plot a set of data, storing evaluated results and graphing these results. Object contains:

Calculator name: used for all equation evaluation.
Data set name: used to store all input/output numeric data.
Graph spec name: used to set up graph axes and scaling.
Set input names: input specifications to be loaded into data set.
Set input fields: column names of data initialised by set inputs.
Map names: Input output maps describing how data is channelled from the data set to the specified calculator, and how calculator results are channelled back into the data set.

transaction: The set of operations required to apply a single Input/Output map to one row in a data set. Input fields named in the map have their values read from the row and are stored as variables in the calculator. Output fields named in the map have their values evaluated by sending a reference order for each output field name to the calculator. It is assumed that output field names correspond to equation names in the calculator. These values are stored in the named output fields of the row.

4.3 Calculator Manager Design

- **Key Features:**
  - ◇ Maintains a list of calculators (calc_object).
  - ◇ User may specify the parameters for a new calc_object:
    name: unique to the calculator list.
    and add this calc_object to the list.
  - ◇ User may remove a named calc_object from the list.
  - ◇ Current calculator can be set to the calculator object stored in any calc_object in the list.
  - ◇ The variables and equations stored in the current calculator may be displayed.
  - ◇ Individually named variables/equations may be cleared from the current calculator.
  - ◇ All variables and equations may be cleared from the current calcuator using a single command.
  - ◇ The error report stored in the current calculator may be displayed.
  - ◇ The error report stored in the current calculator may be cleared.
  - ◇ Auto-verifying of equations for circular references in the current calculator may be activated or deactivated.
  - ◇ A manual verify operation may be carried out upon the equations stored in the current calculator.
  - ◇ All calculator names, variable and equation lists may be displayed using a single command.
  - ◇ Allows calculator orders to be sent to the 'current calculator', displaying the complex number result of the evaluation.
  - ◇ The manager may be reset by the user, clearing the calculator list.
  - ◇ All these facilities are available as public member function calls and as options selectable using an interactive command line interface.

- **Command Line Interface:**
  - ◇ A single call to the interface function allows the user to make unlimited changes to the calc_object list by standard input.
  - ◇ The user may exit the interface function and thus the calculator manager, by issuing the 'return' command.
  - ◇ The user may obtain command help by issuing the 'help' command.
  - ◇ Full CLI details may be found in the user manual.

◇ **Calc_object Design** : please refer to the List Nodes Design (indexed by name - data field is a calculator object.)

◇ **Glossary:**
calculator order: A string containing user function names, operators and user names. This may describe assignment of a variable, a definition of an equation or an expression.
current calculator: Calculator currently being targetted by commands issued through the CLI of the calculator manager, and by their equivalent public member functions.

4.4 Calculator Design

- **Key Features:**
  - ◇ Performs algebraic parsing and complex number evaluation of calculator orders.
  - ◇ Conforms to conventional algebraic precedence rules.
  - ◇ Is capable of storing unlimited number of complex number constants which are available to all instantiated calculator objects.
  - ◇ Is capable of storing unlimited number of complex number variables.
  - ◇ Is capable of storing unlimited number of equations.
  - ◇ Constants, variables and equations may use any alphanumeric identifier. Names must be unique to a calculator object.
  - ◇ Maintains a verbose error report holding all parsing, invalid operand and invalid user name errors.
  - ◇ Error report may be cleared or viewed by the user.
  - ◇ Allows auto-verification of equations against all other equations stored in the calculator (as they are defined by the user) for circular references.
  - ◇ Allows output of variable and equation lists.
  - ◇ Allows clearing of entire variable and equation lists.
  - ◇ Allows clearing of individual variables and equations.
  - ◇ Please reference the user manual for details concerning the mathematical functions available to the user.

◇ **Definition of a calculator order:**

A calculator order is a string holding an expression, an assignment to a variable name or a definition of an equation. Note that calculator token enumeration values have been used in the grammar section below. The order string would need to contain single token characters representing these enumeration values.

◇ **General Description**

The syntax analysis is performed using the method of 'recursive descent'. The calculator class was inspired by the simple desk calculator featured in Chapter 3 of Reference 7.

Stroustrup describes the method as efficient for use in C++ due to the small overheads associated with function calls.

String streams are used to extract characters from the order string, and to store the error report in a character array. The advantage of using streams is that reading and writing is as simple as using standard input. They also prevent overflowing of output buffers.

If an error occurs in the calculator order, a NULLcomplex number is returned by the function detecting the error. This is equal to the complex number 0+0j.

If a variable name is referenced in an expression, its complex value is looked up in the variable list. If an equation name is referenced in an expression, a new calculator is instantiated to evaluate the expression string associated with the equation name.

◇ **Variable and equation lists**.

The Node Class used for both of these lists is the name_object class. A name_object object contains a name for the variable/equation (Index field) and a complex_container (Data field) object which stores a complex number (when name_object object is in variable list) or an equation string (when name_object object is in equation list). It also contains an indicator which states whether the complex number is storing valid data (indicator=CONSTANT), or the string is storing valid data (indicator=EQUATION).

◆ Super and Sub Calculators

Super calculators are created when the program instantiates a calculator object from outside the calculator class. The Super calculator is characterised by the following features:

- ◇ Variable and equation lists are stored within the calculator object.
- ◇ Error stream, error character buffer and total number of errors count are stored within the calculator object.

◆ If a Super Calculator is given an order to evaluate the result of an equation, the associated expression is retrieved from the equation list, and a new Sub Calculator is instantiated from within the Super Calculator object which then evaluates the equation. (This is sent as a calculator order referencing the required equation to the Sub Calculator). If the order contains further references to other equations, this first generation Sub Calculator in turn generates a second generation Sub Calculator object to evaluate each equation name found in the first generation Sub Calculator's order. This is repeated until all the terms in the equation are expanded to their complex numbers. It is important to note that it is imperative that there are no circular references present in the equations. Once a sub calculator has evaluated its expression, the result is returned to the previous generation calculator and the Sub Calculator is destroyed.

All errors trapped by any Sub Calculator generation are stored back in the Super Calculator parent.

Sub Calculators are characterised by the following features:

- ◇ Variable and equation lists are referenced from the parent Super Calculator. Thus the Sub Calculators do not instantiate lists of their own.
- ◇ Error stream, error character buffer and total number of errors count are also referenced from the parent Super Calculator. Thus errors which occur when evaluating a sub-expression are referenced back to the original parent and are not lost when the Sub Calculator is destroyed.

◇ Sub Calculators thus have the responsibility of completely unravelling equation expressions (evidently another example of a recursive process), returning the final result to its parent Super Calculator.

◇ Parsing an order:

The calculator reads the order by character using the get_token function. This stores the token enumeration value of the current character into the token field of 'current_symbol' data member in calculator class. If the character is not recognised an error is flagged. If the token enumeration is 'NUMBER' the number is read in as a floating point double and stored in the complex number field of 'current_symbol'.

◇ Summation and Prodation (multiplication of results obtained from repeated evaluation of an equation whilst changing a control variable between limits) is achieved by reading in a variable, and the lower and upper bounds for that variable. If a lower bound is not specified, it is defaulted to 0 for Summation and 1 for Prodation. The level1 expression following the SUM/PROD parameters is repeatedly evaluated, whilst incrementing the control variable between the lower and upper bounds. Incrementation value is 1 and lower and upper bounds are truncated to integers.

◇ The Window function requires a control variable, a floating point lower bound and a floating point upper bound. The evaluation of the level1 expression following the window function parameters is multiplied by 1+0j if the control variable value lies between the bounds, but is otherwise multiplied by 0+0j.

◇ Grammar accepted by the calculator:

order:
    expression-level1 END
    assignment END
    definition END

assignment:
    NAME = expression-level1

definition:
    NAME : expression-level1

expression-level1:
    expression-level1 + expression-level2
    expression-level1 - expression-level2
    expression-level2

expression-level2:
    expression-level2 / expression-level3
    expression-level2 * expression-level3
    expression-level3

expression-level3:
    expression-level3 POW expression-level4
    expression-level3 ROOTX expression-level4
    expression-level3 LOGX expression-level4
    expression-level4

expression-level4:
    expression-level4 FACTORIAL
    expression-level5

expression-level5:
    expression-level5 MILLI
    expression-level5 MICRO
    expression-level5 NANO
    expression-level5 PICO
    expression-level5 FEMTO
    expression-level5 KILO
    expression-level5 MEGA
    expression-level5 GIGA
    expression-level5 TERA
    expression-level5 PETA
    expression-level5 EXA
    primary

primary:
    NUMBER
    NAME
    - primary
    (expression-level1)
    [expression-level1]
    SQRT expression-level 3
    CBRT expression-level 3
    SIN / COS / TAN / ASIN / ATAN / SINH / COSH / TANH/ASINH /
        ACOSH / ATANH / LN / LOG10 / LOG2 expression-level3
    WINDOW (variable NAME, NUMBER, NUMBER) expression-level1

SUMMATION (variable NAME, NUMBER) expression-level1
SUMMATION (variable NAME, NUMBER, NUMBER) expression-level1
PRODATION (variable NAME, NUMBER) expression-level1
PRODATION (variable NAME, NUMBER, NUMBER) expression-level1

◇ Calculator tokens

The following characters are interpreted as calculator tokens:

char	Operation Name	Token Enumeration Value
+	PLUS	PLUS
-	MINUS	MINUS
*	MULTIPLY	MUL
/	DIVIDE	DIV
^	POWER	POW
!	FACTORIAL	FACTORIAL
@	SQUARE_ROOT	SQRT
£	CUBE_ROOT	CBRT
$	ROOTX	ROOTX
¡	SINE	SIN
™	COSINE	COS
#	TANGENT	TAN
¢	ARCSINE	ASIN
∞	ARCCOSINE	ACOS
§	ARCTANGENT	ATAN
~	SINE-H	SINH
Ω	COSINE-H	COSH
≈	TANGENT-H	TANH
ç	ARCSINE-H	ASINH
√	ARCCOSINE-H	ACOSH
∫	ARCTANGET-H	ATANH
¶	NATURAL_LOG	LN
•	LOG10	LOG10
ª	LOG2	LOG2
º	LOGX	LOGX
;	PRINT	PRINT
=	ASSIGN_CONTANT	ASSIGN_CONSTANT
(	LEFT_BRACKET	LP
)	RIGHT_BRACKET	RP
[	LEFT_MODULUS	LM
]	RIGHT_MODULUS	RM
≠	ARGUMENT	ARGUMENT
œ	MILLI	MILLI
∑	MICRO	MICRO
®	NANO	NANO
†	PICO	PICO
¥	FEMTO	FEMTO
å	KILO	KILO
ß	MEGA	MEGA
∂	GIGA	GIGA
ƒ	TERA	TERA
©	PETA	PETA
Δ	EXA	EXA
:	DEFINE_EQUATION	DEFINE_EQUATION
¬	SUMMATION	SUMMATION
∏	PRODATION	PRODATION
,	COMMA	COMMA
≤	REAL	REAL

≥	IMAGINARY	IMAGINARY
÷	WINDOW	WINDOW

In addition, a NUMBER is considered to be a floating point number (using fixed notation or scientific notation), a NAME is considered as an alphanumeric beginning with an alphabetic character and END is considered as a newline character.

◇ **Extendability**:
Over 180 tokens (all characters not regarded as white-space, and are non-alphanumeric) can be defined using this design. The programmer must decide the precedence level of the function being added and add appropriate code into that levels function call, add the new token to the token_value enumeration definition, and add the new token to the list of token characters recognised by the get_token function.

◇ **Constraints:**
Precision is limited to floating point double.

◇ **Efficiency:**
Using a binary tree or a hash table to hold the variables and equations would be more efficient than a linked list.

The class could be optimized by storing equation expressions in reverse polish notation and eliminating the need for repeated recursive parsing whenever an equation name was referenced. However, the complexity of this system is great enough already without implementing such techniques.

4.5 Data Manager Design

- **Key Features**
  - ◇ Maintains lists of records (record_object), data sets (data_set_obj), set_inputs (set_input_object), input/output maps (IO_map_object).

  - ◇ User may specify the parameters of a record_object…
    index: name unique to the records list
    List of column names each separated by a single space.
    and add this record_object to the records list.

  - ◇ User may specify the parameters of a data_set_obj…
    index: name unique to the data sets list
    record name: must be present in record list - its column names are used to label the data set columns.
    length of data set: number of rows in the data set
    and add this data_set_obj to the data sets list.

  - ◇ User may specify the parameters of a set_input_object…
    index: name unique to the set inputs list
    lower index: first row to be affected by this set_input
    upper index: last row to be affected by this set_input
    start value: value to be loaded into first affected row
    incrementer: increase the start value by this much for each row down the column.
    and add this set_input_object to the set inputs list

  - ◇ User may specify the parameters of an IO_map_object…
    index: name unique to the map list
    list of input fields: column names to have values read from row and sent to a calculator.
    list of output fields:column names to have values read from calculator and stored in respective columns.
    and add this IO_map_object to the input/output map list

  - ◇ User may remove a named record, named data set, named set input or named input/output map from the corresponding list.

  - ◇ All lists may be outputted to a stream or standard output for user perusal.

  - ◇ The manager may be reset by the user, clearing all lists.

  - ◇ All above facilities are available as public member function calls and as options selectable using an interactive command line interface.

  - ◇ In addition, the manager provides two methods for checking data integrity used by the System Manager. The first checks the data list for a data_set_obj with a particular name, checks a list of field names for being valid field names for the forementationed data set, checks a list of map names as being names of IO_map_objects in the map list and checks that all input/output field names in these map objects are valid fields in the forementioned data set.

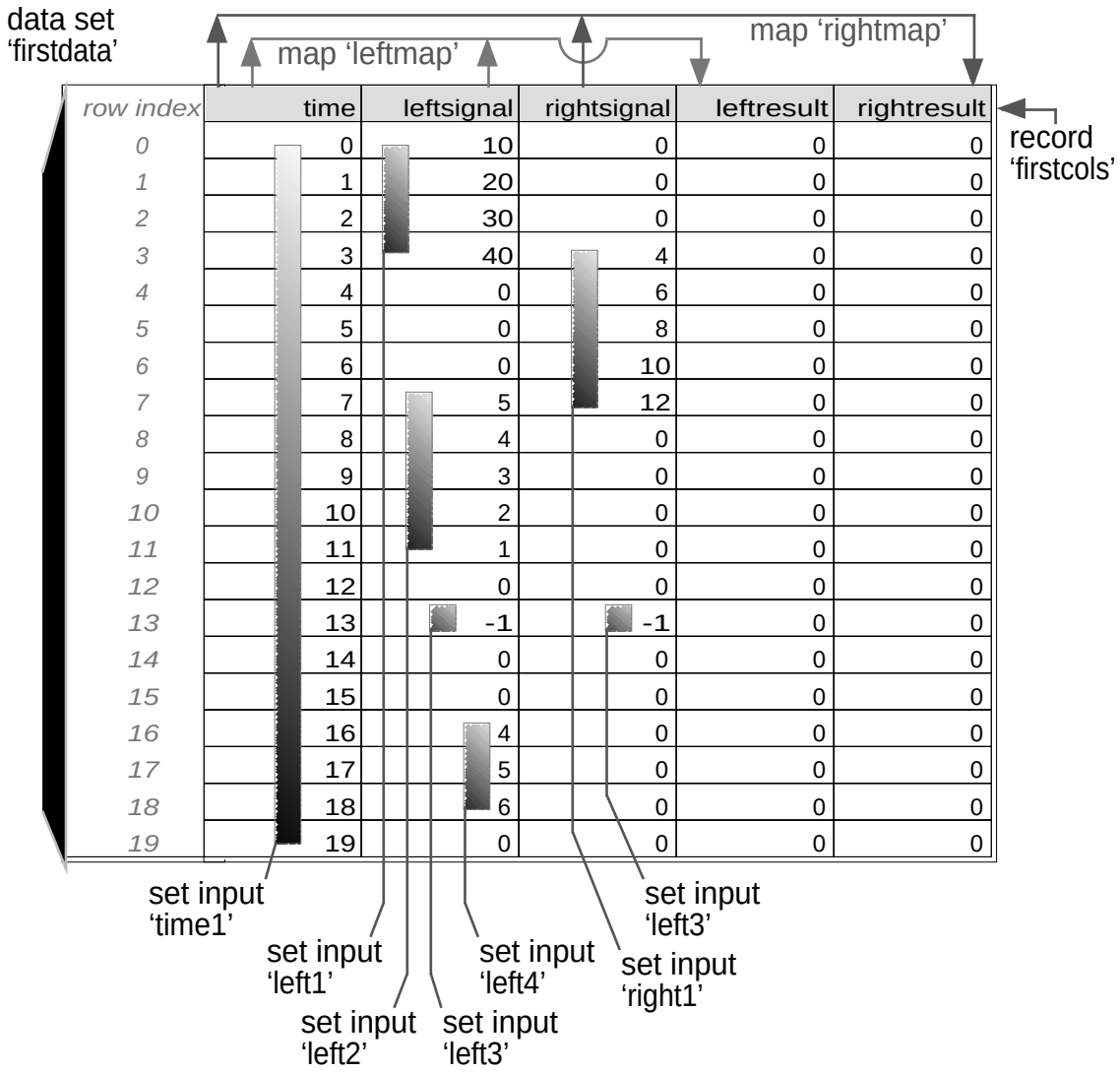
    The second method checks that a list of names reference actual set_input_obj objects in the set input list.

  - ◇ All lists are public data members and so may be manipulated externally from the data_manager class. This is to allow iterative access to the lists.

◆ Command Line Interface:

- ◇ A single call to the interface function allows the user to make unlimited changes to all lists by standard input.
- ◇ The user may exit the interface function and thus the data manager, by issuing the ‘return’ command.
- ◇ The user may obtain command help by issuing the ‘help’ command.
- ◇ Full CLI details may be found in the user manual.

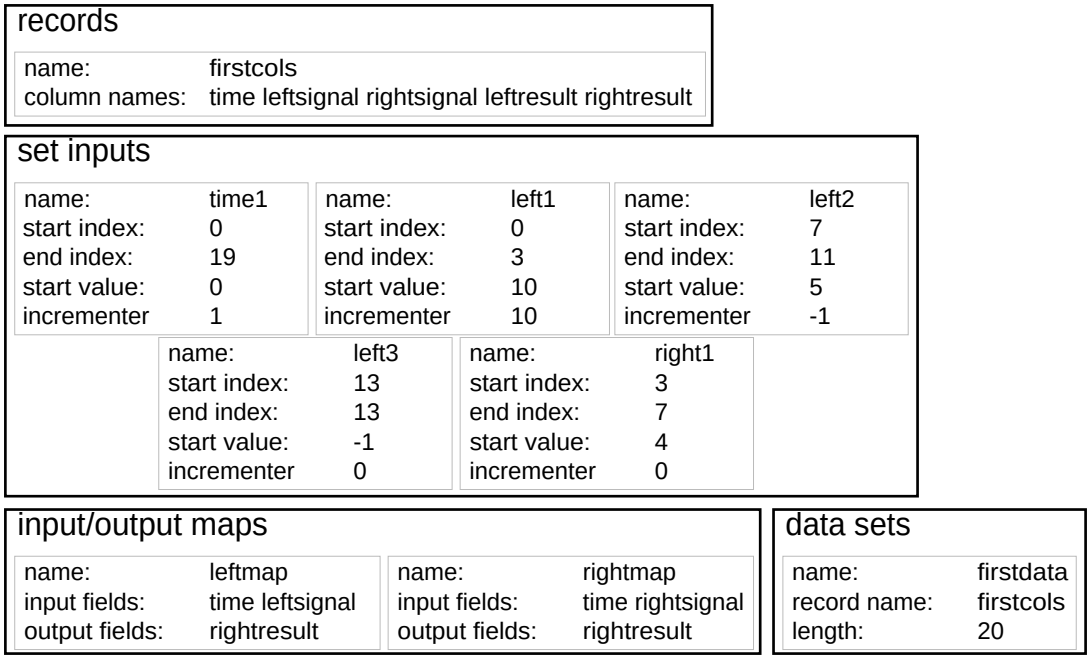
◇ Figure 12 shows how records, data sets, set inputs and IO maps work together:



◇

◇ **Figure 12: Visualisation of data array**

◇ Figure 13 shows the objects required to conceptualise Figure 12 with data set, record, set input and map objects: (note that the following is not the responsibility of the data manager: (1) applying a set input to a particular column/field; (2) applying particular maps to the data set - the System Manager carries out these operations using its system process object.)



◇

◇ **Figure 13 : Node objects used in the data array of Figure 12**

◆ List Node Class data fields:

◇ record_object

Contains a list of strings.

◇ set_input_object

Contains a set input object which holds lower index, upper index, start value and incrementer values.

◇ IO_map_object

Contains an input/output map (IO_map) object which holds an array of input field names and an array of output field names.

◇ data_set_object

Contains a data set object which holds an array for holding complex numbers (number of columns and rows specified by user). The data set object allows access to array elements in two ways: associative and non-associative. Associative access requires a column/field name to be provided, and a row number ranged from 0 upto one less than the length of the array - out of range element access is trapped. Non-associative access requires numerical column and row indices. Individual elements may be read or written by these methods. A function is provided to reset all array elements to 0+0j.

The column names are provided by a record object referenced when the data set is created by the user (see add data set above).

4.6 Graph Manager Design

- **Key Features:**
  - ◇ Maintains a list of graph specification objects (graph_spec_obj).
  - ◇ User may specify the parameters of a graph_spec_obj:
    - index: name unique to the graph specification list
    - horizontal scale: (x-axis scaling)
      - Scale division
      - Number of minor ticks between each division
    - horizontal range: (x-axis range)
      - Minimum visible axis value
      - Maximum visible axis value
    - vertical scale: (y-axis scaling)
      - Scale division
      - Number of minor ticks between each division
    - vertical range: (y-axis range)
      - Minimum visible axis value
      - Maximum visible axis value

    and add this graph_spec_obj to the graph specification list.
  - ◇ User may remove a named graph_spec_obj from the graph specification list.
  - ◇ Graph specification list may be outputted to a stream or standard output for user perusal.
  - ◇ The manager may be reset by the user, clearing the spec list.
  - ◇ All these facilities are available as public member function calls and as options selectable using an interactive command line interface.
  - ◇ The graph specification list is a public data member and so may be manipulated externally from the graph_manager class. This is to allow iterative access to the graph specification list.

- **Command Line Interface:**
  - ◇ A single call to the interface function allows the user to make unlimited changes to the graph specification list by standard input.
  - ◇ The user may exit the interface function and thus the graph manager, by issuing the ‘return’ command.
  - ◇ The user may obtain command help by issuing the ‘help’ command.
  - ◇ Full CLI details may be found in the user manual.

◇ **graph_spec_obj Design**: Please refer to the List Nodes Design.

◇ **graph_spec Design**: Used within graph_spec_obj, this class contains all parameters required for sizing and positioning a graphing window on the screen - although the graph manager doesn’t provide a method for the user to change these parameters, its future inclusion alongside the current design would be elementary. This choice was made as it would require considerable GUI knowledge to afford windows which were movable and re-sizable using the mouse. It was decided that textual entry for pixel dimensions for a window would be cumbersome, meaning a default size would have to be used. graph_spec also contains the scale and range information for both axes. A data structure (tick_info) to hold the size of the tick marks along each axis and the size of the text used for numbering the axes is also included, but again these are set to default values for now.

4.7 Graph Device Design

- **Key Features:**
  - Prepares a graphing window with user defined horizontal and vertical scales and ranges. See figure below showing the default axes displayed once the System Manager starts up.
  - Provides translation service converting (x,y) graph co-ordinates to actual pixel co-ordinates lying within the graphing area.
  - Provides move of origin service: used when displaying polar vector plots.
  - Provides service for clearing the graph window.
  - Provides service for clearing the graph window and drawing axes and scale numbering for current scale/range parameters.
  - Reference 8 was used to obtain information concerning Macintosh Operating System calls.

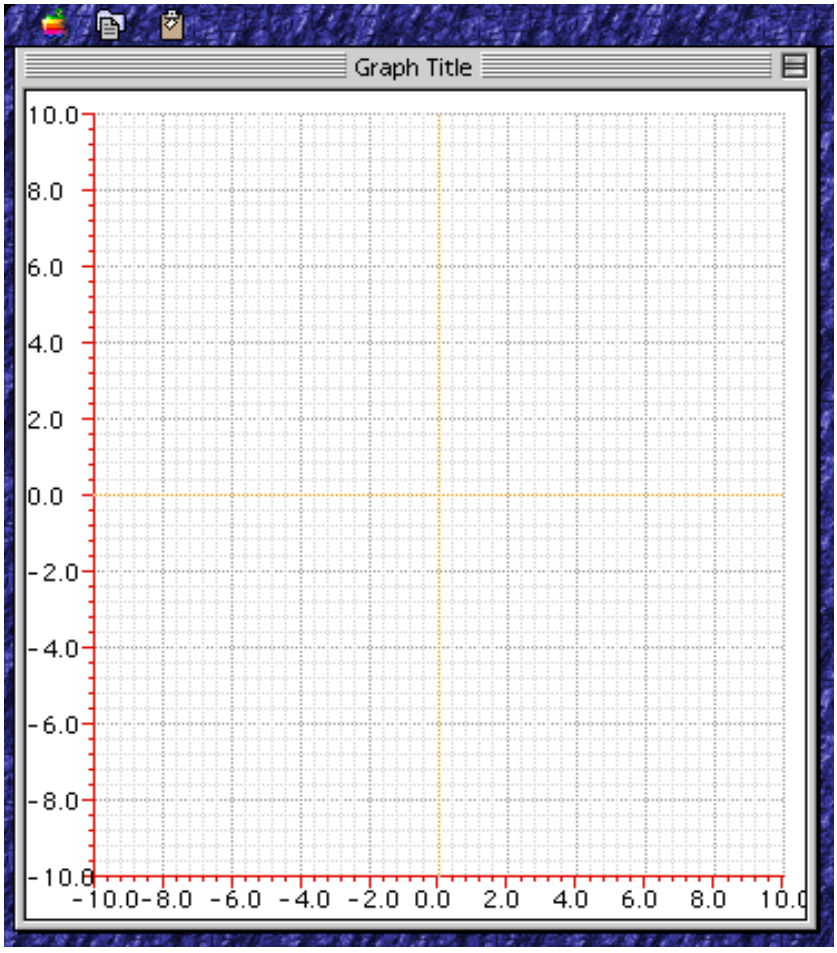


Figure 14 : Default Graph Setup

4.8 Validator Design

◇ General Description:

The validator class is responsible for flagging self-referencing and cross-referencing circular references found in the equations stored in a calculator's equation list.

The validator function iterates through the entire equation list, recursively expanding all equation names referenced in each equation expression appearing in the equation list. The validation is prematurely terminated if a circular reference is found, returning an error to the calling function.

Expressions contain calculator tokens and user names. A preprocessor object is required to strip out all user names from the expression. The preprocessor returns a list of any alphanumeric string which isn't a user function name: these correspond to the user names in the expression. The validator must look up each of these in the constant, variable and equation lists of the connected calculator. Only names found in the equation list are capable of generating circular references. These names have to be expanded and checked themselves.

The check is performed by keeping a record of a root equation's dependents. Before a call to expand an equation name is made, a backup of the current dependents is made, then the name of the current equation name being expanded is added to the dependents. These dependents are passed down to the next expansion level. (A repeated dependent signifies a circular reference.)

Once the list of equation names present in the expression becomes zero, the function can be exited, returning success to the previous level.

After carrying out a successful expansion, the backup dependents are restored and the next equation name may be checked.

◆ Key Features:

- ◇ Upon instantiation, must be provided with a pointer to a preprocessor object.
- ◇ A single call to the 'validate' function passing a pointer to a calculator object checks all equations in that calculator for circular references.

◇ Glossary:

equation: An equation has a name and an associated expression.
expression: A calculator order that does not involve assignment (=) to a variable or definition of an equation (:).
expansion: The process of substituting a variable name in an expression for its tied complex number, or where an equation name is substituted by the evalution of that equation's expression string.
reference: User name present in an expression.
circular reference/definition: Where the expansion of an equation involves the expansion of itself. If such an evaluation is attempted, the resulting infinite loop will overflow the program's function call stack crashing the program and possibly the operating system. There are two types of circular reference:

Self-referencing : equation's expression string references its own equation name, eg "a:a+b"

Cross-referencing: requires one or more further equations to be expanded before circular reference is detected,eg "a:b", "b:c","c:a"

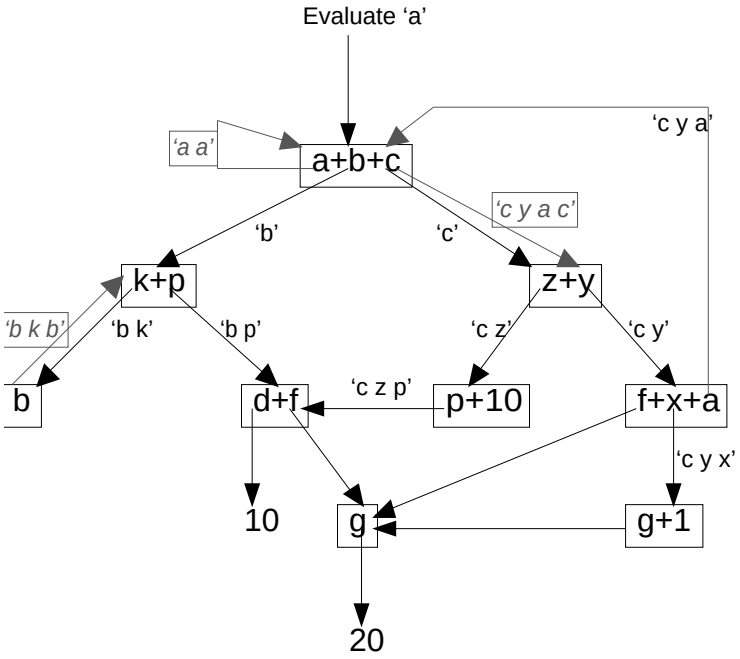
dependent: The dependents associated with an equation name are those equation names which have already been expanded prior to the expansion of the current equation name. If throughout the course of expanding a root equation two instances of the same

name. If throughout the course of expanding a root equation two instances of the same depenent are found, then a circular reference is flagged and an error returned.
root equation: The first equation to be expanded in the recursive function call chain.
user name: A constant, variable or equation name.
user function name: Function names converted by the preprocessor to calculator token characters.

◇ **Example** :

Calculator equations:
a:a+b+c
b:k+p
k:b
p:d+f
f:g
x:g+1

Calculator variables:
d=10
g=20



◇ **Figure 15 : Validation Tree for a system of equations**

◇ **Notes for Example**:
The tree above shows the expansions required to evaluate ‘a’. The equations and variables on the left have been loaded into the calculator. An edge leads from a user name to its expansion. Dependents are labelled on each edge leading to an equation name’s expansion.

The grey arrows show the circular references present in this equation system. The grey dependents signify the point where the validator class can tell that a circular reference is present.

Note that the validator would terminator the traversal of this tree once the first circular definition was found. Edges which lead eventually to a variable name (and then to a lone number) mark the end of that particular recursion chain.

Dependents ‘a a’ marks a self-referencing circular reference.
Dependents ‘b k b’ and ‘c y a c’ mark cross-referencing circular references.

4.9 Preprocessor Design

◇ **General Description:**

The preprocessor class preprocesses a calculator order entered by the user by contracting all user function names to their equivalent calculator token characters. Three advantages of this are:
1) the calculator only needs to read a single character to parse any function.
2) user function names are decoupled from the character tokens.
3) user names remain the only alphanumeric characters remaining in the expression after preprocessing simplifying the parsing process in the calculator.

Single character operators such as (+-*/^!, parentheses and modulus) are left unchanged.

The preprocessor also performs a postprocessing operation involving the expansion of calculator tokens to their user function names for displaying an equation stored in a calulator's equation list to the user.

User names are identified by the preprocessor as they do not match any of the user function names known by the preprocesor. A list of all user names in a preprocessed expression may be obtained from the preprocessor.

◆ **Key Features:**

- ◇ Upon instantiation the preprocessor is given an array of user_label objects each mapping a user function name to an operation name, and an array of token_name objects each mapping an operation name to a calculator token character. The preprocessor uses these to build two look-up tables. The 'correlator' table (order n, no. of entries determined by number of user function names, token_name elements) maps a user function name to a calculator token character. The 'inverse correlator' table (order 0, 256 entries, string_class elements) table maps a calculator token character to a user function name.

- ◇ A preprocessing operation is offered where an input string containing user function names, operators and user names is preprocessed and returned containing calculator tokens and user names. (This operation is used when preparing calculator orders for parsing by a calculator object).

- ◇ An equivalent preprocessing operation is offered which also returns a list of string_objects holding the user names found in the input string. (This operation is used by the validator class.)

- ◇ A postprocessing operation is offered where the input string is parsed for calculator tokens, which are expanded to their respective user function names and returned. (This operation is used by the overloaded output operator for the complex_container class. This class (which is the Data field for the name_object Node Class used to build calculator variable and equation lists) stores equation expressions provided by the calculator - but by this point the expressions contain only user names and calculator tokens. If these were output, the expression shown on screen would be incomprehensible - hence the need to expand calculator tokens to their respective user function names.

◇ Glossary:

User function name: names provided by the user in expressions to represent mathematical functions, eg sin, tan, SUM, re, etc.
Operator: single character operator, eg +-*/!()[]
Calculator token character: Single character recognised by the calculator as a token, eg +,-,*,/ - also includes symbols associated with trigonometric, logarithmic and compound (eg summation) functions. The ASCII characters used are not alphanumeric, and so can be easily distinguished from user names.
Operation name: Each user function name has an associated 'operation' name, eg sin->SINE, tan->TANGENT, and each operation name is associated with a calculator token character.
user_label class: Contains two strings, 'input_string' (holding the user function name) and 'calc_string' (holding the operation name).
token_name class: Contains one string: 'name' and a single character, 'token' (holding the calculator token character).

◆ Arrays supplied to preprocessor:

◇ user_labels array

This array is stored in the main program (System Manager).
Maps user function names to operation names.

◇ token_names array:

This array is stored as static data in the calculator class.
Maps operation names to calculator tokens.

Operation names are common to both preprocessor and calculator classes.

◆ Look-up Tables generated by the preprocessor:

◇ correlator array (token_name elements), user function name->token.
◇ inverse correlator array (string_class elements), token-> user fn name
This is a sparse array indexed by calculator token ASCII code (ie unsigned integer from 0->255).

◇ Scalability:

The maximum number of tokens that may be pre/postprocessed is 256.

◇ Maintainability:

User function names are decoupled from their respective calculator tokens so using different user function names is simply a case of supplying the appropriate arrays upon initialisation. Indeed it is possible to instantiate many preprocessor objects, each using different user function name->token characters mapping rules.

◇ Constraints:

Operation names must be common between both arrays supplied to the preprocessor for building look-up tables. Both arrays must be also of equal element length.

◇ Efficiency:

Searching through the correlator array is an order N process, as linear searching is used. This could be improved by building the correlator array using lexical ordering on user function name and having a binary search.

The inverse correlator cannot be made more efficient as it is effectively an order 0 hash table. The element corresponding to the ASCII code of the token character being expanded contains the user function name for this token.

4.10 Linked List Design

- **Key Features:**
  - Templated (generic) singularly linked List - may be used to store any information.
  - List nodes automatically ordered using an order method, with an Ascending or Descending order property.
  - List nodes are unique in either Index or Data field (called the 'identifier' or 'name' field) - determined by list ordering method chosen (but not when arbitrary ordering is used).
  - Option to append node to list allowing arbitrary ordering.
  - List node may be removed from the list by specifying a reference node and a reference method.
  - List node may be searched for by specifying a reference node and a reference method. If the node is found the reference node's non-reference field is updated accordingly.
  - List node may be updated with new Index or Data values. The affected node is moved to preserve the ordered property of the list if the order field is updated. If an update request breaks the uniqueness criterion the update is refused.
  - List node may be 'maintained' based upon information in a reference node. List will add a new node to the list if order field value of reference node is not found in any of the list nodes. Otherwise, the targetted list node will have its update field set to the value stored in the update field of the reference node.
  - List nodes may be accessed iteratively by using the transverse functions. Extraction of list node information and removal of list nodes is available using these methods.
  - All list nodes in a list may be cleared with one function call.
  - Entire list contents may be outputted to a stream, or to standard output.

- **Glossary of terms:**
  - **Index field**: Node data member used primarily for node identification, may be any object.
  - **Data field**: Node data member holding any object (which is identified by the Index field).
  - **order method/field**: List is ordered on either Index or Data field values. The ordering field must have a value unique to all nodes in a particular instantiation of a list.
  - **reference node**: A node loaded with data in either Index or Data field (the **reference method/field**) which is used as the search criterion when retrieving a corresponding list node. The non-reference field value is specified when maintaining/updating a list node. A successful search will load the non-reference field with the result of the search.
  - **reference method**: Determines which field is to be used as a reference when searching, removing, updating or maintaining a list node.
  - **order property**: Describes whether the list ordering is based upon ascending or descending values in the 'order method' field.
  - **update field**: Field which is to be updated (Index/Data).
  - **node**: A single element in the linked list. Must conform to minimum node specification.
  - **list node:** Node within a linked list.
  - **transverse node**: Node currently pointed to by transverse pointer.
  - **transverse pointer**: Pointer to a list node which may be moved for iterative access to each list node in the list.
  - **transverse functions:** Set of functions that manipulate the transverse pointer: resetting to head of list, get copy of transverse node, removal of transverse node, progress transverse pointer to next list node in list and return transverse pointer value.

◇ **minimum node specification**: The public member functions with particular behaviour expected by the list. Without these the particular type of list will not compiles

◆ Minimum Node Specification

◇ The node object must have the following public member functions, but may include any additional functions as the user sees fit:

default constructor
copy constructor
overloaded = operator
destructor (if dynamic data is stored in index/data fields)
get_index
set_index
set_to_first_index
set_to_next_index
compare_index
get_data
set_data
compare_data
set_pointer_to
get_pointer
print_node
+ friend function: overloaded output operator

The node class must also have the following static data member:

<index_type> undefined_index

Where <index_type> is the name of the class used in the Index field, and undefined_index is set to the value that index field should be set to when a node object is instantiated using the default constructor (where an initial index value has not been specified).

The behaviour of each of these functions is typified by the public member functions of 'string_object' class - SpecRef:, AlgRef:. For the 'compare' functions above, an arbitrary comparison method is to be adopted by the user. In this example, lexical comparison is used between data values. To create a new node these behaviours must be provided; albeit with the node class name substituted for the 'string_object' class name in function prototypes and return parameters.

◆ Making the class available to the main program and instantiating specific lists:

◇ Templated class instantiation is not covered by the ANSI C++ standard at present, thus the method described may not be valid for development environments other than CodeWarrior 9 under Mac OS.

◇ The code file containing the member function definitions for ulist must be #included exactly once into the executable code for the program. The most convenient place for this is in the file holding **main()** function.

◇ For each list type, the header file containing the definition of the node used in this list must be #included at the beginning of the ulist code-file.

◇ For each list type, the following code-line must be present at the end of the ulist code file:

template class ulist<'node_name'>;

where 'node_name' is the class name of the node being used to produce the list. This instantiates a version of the code applicable to the new list type.

◇ For each list type, a prototype of the corresponding ulist overloaded output operator function must appear at the end of the ulist code file in order to instantiate the code for each output operator function.

- **Using the class public member functions:**
  - For the specific behaviour for each member function please reference the Specification for ulist class.
  - The transverse functions: reset_transverse, progress_transverse and remove_transverse, add an element of ‘state’ to the class - with exception to the list of nodes itself. Therefore the user is advised that the function call chain includes only one function call level that calls these functions (for a particular list object). For example, it would be inadvisable to write a ‘do’ loop which progressed the transverse pointer through the list, whilst calling a function within this loop which called remove_transverse. An algorithm which successfully circumvents this argument would be inconspicuous to another user.

- **Extendability:**
  Additional functionality may be built into the class as long as the minimum node specification requires no change. Under these circumstances the entire class would need to be reviewed, along with all node classes. It is also unadvisable to manipulate or otherwise access the transverse pointer without explicit user instruction.

- **Scalability:**
  The size of the linked list is limited by the memory available in the Application Heap. Due to the one-way, linear nature of the list, large lists will be slow.

- **Efficiency:**
  The ‘order n’ search metric of a singly linked list limits this class’s applications to those requiring relatively short length. The most suitable abstract data type for larger structures is the binary tree, where search paths are ‘order $\log_2 n$’. The complexity required in implementing the requirements which this class meets in such a structure would be too great for this project.

4.11 List Nodes Required by the Design

◇ Each node class below is used to generate a particular instantiation of ulist class. Each class meets the minimum node specification described in the list design. In general, the node class does not add to the specification. Public member functions of the Data Class are accessible through equivalent function calls in the Node Class.

◇ Data Manager and Graph Manager node classes contain Data Classes which are described in the respective Manager design documentation.

◇ calc_object class is described in Calculator Manager documentation.

◇ name_object class is described in Calculator Class documentation

◇ Except for the string_object class, all node classes below have an identifier or name string used for indexing.

Node Class	Index Class	Data Class
string_object	integer	string_class
record_object	string_class	ulist<string_object>
data_set_obj	string_class	data_set
set_input_object	string_class	set_input
IO_map_object	string_class	IO_map
graph_spec_obj	string_class	graph_spec
name_object	string_class	complex_container
calc_object	string_class	calculator

◆ Data Manager Node Classes
- ◇ record_object class: A depository for a list of logically related strings.
- ◇ data_set_obj class: Stores Data object with a name reference. Contains a function allowing a set input to be applied to a single column of the data set contained within.
- ◇ set_input_object class: c.f. data_set_obj
- ◇ IO_map_object class: c.f. data_set_obj

◆ Graph Manager Node Classes
- ◇ graph_spec_obj class: c.f. data_set_obj

◆ Caculator Node Classes
- ◇ name_object class: c.f. data_set_obj
- ◇ calc_object class: c.f. data_set_obj

◆ Other Classes
- ◇ string_object class: A depository for a single string.

4.12 Complex (number) Class Design

- **Key Features:**
  - Provides data members to store real and imaginary components of a complex number at floating point double precision.
  - Provides complex addition, subtraction, multiplication and division between complex objects.
  - Provides complex number power operations.
  - Provides input and output operators.
  - Additional functions external to the class provide rectangular to polar form and polar to rectangular form operations.
  - Prevents divide by 0 and 0^0 evaluation.
  - Power operation recognises the special case of exp raised to the power (a+bj) and returns the result in rectangular form.
  - Power operation refuses to evaluate the result of raising any number other than exp to the power of a complex number containing a non-zero imaginary component.

4.13 String Class Design

- **Key Features:**
  - Contains a character array which is dynamically resizable.
  - All memory allocation/deallocation is internally managed.
  - Defines append, sub-string search, assignment and relational comparison operations between string_class objects.

- **General Description:**

  This class implements a BASIC-like string class, hiding all memory manipulations from the class user. The alternative to this would be using a bare ‘C’ character array which is has very poor data integrity, especially where strings have to be shuttled backwards and forwards through function calls. One doubly deallocated pointer would generate a possibly undetectable memory corruption. It is thus necessary to encapsulate the char array in a class.

- **Efficiency:**

  Char arrays stored inside the string class are allocated in byte size blocks and so memory fragmentation could become an issue. An enhanced version of this class would allocate memory in larger blocks to try to minimise this. There is no evidence that byte-size allocation detrimentally affects the operation of this system.

Chapter 5: Low-Level Design

5.1 Guide to Low-Level Design

The specifications for each header file in the system are grouped accordingly:

System Manager Grouping

1. System Manager 1 spec
2. System Manager 2 spec
3. System Process spec

Calculator Manager Grouping

4. Calculator Manager spec
5. rCalculatorUserTypes spec
6. extraclasses spec
7. rCalculatorClass spec

Data Manager

8. Data Manager spec
9. data set spec
10. set input spec
11. IO Map spec

Graph Manager

12. Graph Manager spec
13. graph_spec spec
14. extragraphclasses spec
15. graph_device spec

Validator & Preprocessor

16. Validator spec
17. preprocessorTypes spec
18. calc_preprocessor spec

List and List Nodes

19. ulist spec
20. record_object spec
21. data_set_obj spec
22. set_input_object spec
23. IO_map_object spec
24. name_object spec
25. graph_spec_obj spec
26. calc_object spec
27. stringobject spec

Complex Number

28. iadditional math spec
29. complex functions spec
30. complex (number) spec
31. newstring spec