INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

2-D GRAPH PLOTTER: A TOOL FOR PLOTTING FUNCTIONS

Anh Phong Tran

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For The Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

May 2000

ABSTRACT

2-D GRAPH PLOTTER: A TOOL FOR PLOTTING FUNCTIONS

Anh Phong Tran

Sometimes a picture means a thousand words. It is hard to visualize how a graph of a function looks like if there is no way of plotting and drawing that function. Paper graphs are simple and reusable but there is a possibility of error. Hence, plotting and drawing a graph manually has been a nightmare for people, especially for students who have just started learning mathematics.

What was needed is an easy way to combine the dependability of paper graphs and the accuracy and quickness of computer graphs. That is just what this project is about.

2-D Graph Plotter has been designed to meet not only the requirements of a plotting device (such as the speed, reliability), but it is also very easy to use and very cost efficient. The purpose of this report is to describe the design and the implementation process of this tool, as well as to show how its friendly user interface allows interactive data manipulation and graph creation. I will then point out its advantages (such as why it works more quickly and efficiently than other off-the-shelf tools), and how the tool should be improved and extended to maximize its benefit.

Acknowledgments

I would like to thank my supervisor, Dr. Peter Grogono, for all his help, advice, and ideas during the project. In particular, Dr. Peter Grogono helped me with the parsing and evaluating components of this project. I also would like to thank my family and my fiancée for their love, their continuing encouragement and their support.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, I will first explain the needs to have a simple plotting device to graph functions quickly and efficiently. I will also state the goals of this project and then I will summarize the main features of this tool.

1.1 Why is a tool needed?

Computers nowadays are used everywhere in our daily life. Modern computers can provide both a very fast CPU, and a high-resolution color monitor with rapid refresh rates. A standard computer system right now has a CPU running at a speed of at least 200 MHz and a color monitor with rate of over 75 MHz. Hence, they can support fast, interactive plotting programs without any other special devices.

Obviously, sometimes it is easy to visualize a function's graph without having to draw it out. However, for complicated functions, it is tedious to draw their graphs manually on paper, let alone to visualize them. Such cases require a special plotting tool. The tool will

draw the graphs automatically, making these complicated functions more clearly and much easier to understand.

Even though there are many programs out in the market (see Chapter 2) that provide plotting features, they are either too complicated to use, or too expensive or too large (with respect to memory and disk space), or too professional. A small, easy-to-use tool for plotting functions quickly and efficiently is suitable for people who do not have much computer experience; or for those who just need a tool for plotting functions only; or for those who need a tool to work on something related to mathematics but do not want to spend much time to learn the tool.

1.2 Goals

There are there goals in this project:

Goal of the design: the design is object-oriented. This is mainly because object-oriented design aims for more robust software that can be easily reused, modified, maintained and extended. The greatest strength of this approach is that it offers a mechanism that captures a model of real world, which leads to greatly improved understandability and maintainability of the application.

There are several languages that could be used to implement object oriented design. In this project, C++ was chosen because it is a language that provides the maximum benefits to programmers. It has all the features of object oriented design: identity, classification, polymorphism, and inheritance.

Goal of the interface: very user-friendly, easy to use, needs almost no time to learn.

The tool for graphical interface development is Visual C++ because Visual C++ provides very good interface and C++ is a language that supports object-oriented design.

Goal of the tool: a tool to plot simple functions quickly and efficiently.

1.3 Expression format and assumptions

An expression consists of one independent variable, one parameter, and several constants. An example of an expression is:

$Y = k*sin(A*x) + Pi,$

where x is the independent variable, k is a parameter and Pi and A are constants.

The independent variable, the parameter and the constants are assumed to have value of 0 if they are not set when evaluating the expression.

Multiple graphs could be plotted on the same View, depending on values of the parameter.

1.4 The structure of the major report

The rest of the major report is presented in four chapters. Chapter 2 briefly introduces four off-the-shelf software packages that also provide plotting features and discusses their disadvantages. The goal and design of the project is presented in Chapter 3. Chapter 4 is to explain the implementation of the program and to discuss briefly the problems encounters when developing the tool and their solutions. In Chapter 5, the project is discussed in length. It explains how the program meets the design goals, what technical difficulties were encountered and solved, and how the project can be improved or extended in the future.

Chapter 2

Related Work

In the market nowadays, there are several programs that provide plotting features. Why not use them?

Here are some of the reasons:

- They are expensive (Macsyma, Mathematica, Matlab).
- Their notation is hard to learn (Macsyma, Mathematica, Maple).
- They are too specialized (various programs for design of circuits, motors, antennas, etc).
- They are too large (most require lots of memory and disk space).

In this chapter, I will present four software packages: *MacSyma*[1], *Mathematica*[2], *Maple*[3] *and MatLab*[4] that support graph-plotting capability.

As mentioned above, the disadvantages of these programs can not be ignored. I will discuss the system requirements, functionality, installation, specialization, and prices of these software programs to illustrate these disadvantages.

I will also provide some syntax examples for plotting functions of these programs to show their complexity and the level of difficulty to graph a simple function.

2.1 System requirements

- *MacSyma*: under Windows operating system for Intel based Pentium 90 (or better). It requires a minimum of 12MB of main memory, with at least 16MB of hard disk [5].
- *Mathematica*: under Windows operating system at least 90 MHz. It requires a minimum of 16MB of main memory, with at least 15MB of hard disk [6].
- *Maple*: under Windows operating system for Intel base Pentium 90 (or better). It requires a minimum of 32MB of main memory, with at least 60MB of hard disk [7].
- *Matlab*: under Windows operating system with Intel based Pentium. It requires a minimum of 16MB of main memory, with at least 145MB of hard disk.

2.2 Functionality

- *MacSyma*: provides basic mathematical capabilities; predefined functions, data analysis; algebraic operations; calculus; linear algebra; vector and tensor analysis.
- *Mathematica*: this is an all-purpose computer algebra software that uses a high-level programming languages such as numerical analysis, algebra, trigonometry, calculus, equations, matrices, and special functions.
- *Maple*: provides basic mathematical capabilities: this is an all-purpose computer algebra software that uses a high-level programming languages.
- *Matlab*: supports computation, algorithm development, modeling, simulation, signal processing and engineering graphics.

2.3 Installation

- *MacSyma*: requires an installation program to install MacSyma under Windows operating system.
- *Mathematica*: requires an installation program. A full installation takes up over 150MB of hard disk.
- *Maple*: requires an installation program to install Maple under Windows operating system.
- *Matlab*: requires an installation program to install Matlab under Windows operating system.

2.4. Specialization

- *MacSyma*: specializes in mathematics, engineering, and statistics.
- *Mathematica*: specializes in mathematics, physical science, biology, and engineering.
- *Maple*: specializes in optimal design of circuit, modeling a thermal system, and modeling of an electrical system.
- *Matlab*: specializes in mathematics, engineering and science.

2.5 Price

- *MacSyma*: $300 USD for version 2.4 for window 95 /98/ NT.
- *Mathematica*: $139.95 USD for a student version.
- *Maple*: $129.00 USD.

- *Matlab*: $1,900 USD for a basic version.

2.6 Syntax examples for plotting functions

- *MacSyma*: example for plotting 2-D:

  plot(curve, x, %pi, "X", "Y", "Sine wave with Gaussian Envelope,low resolution")$

- *Mathematica*: example of plotting 2-D.

  plot [(x-2)^2 , {x, 0 ,1}] x in a range from 0 to 1 for equation (x-2)^2

- *Maple*: example for plotting 2d with gridlines:

  with(plots):

  mygrid := coordplot(cartersian , labelling = middle, view= [-6..6, -6 ..6]):

  p := plot(4*sin(x) , x=-6 ..6):

  display ([p, mygrid]);

- *Matlab*: use matlab language. Example for plotting 2-D:

  t = 0:pi/100:2*pi;

  y = sin(t);

  plot(t,y);

2.7 Conclusion

Through the facts presented above, it is fairly obvious to see why these software packages are not a preferred choice for plotting and graphing simple functions. Compared to my plotting tool, they are too expensive, too professional and especially, too hard to use. My tool is simple, requires very few memory space and easy to use. It does not

require much computer experience from users to use this program and it provides all the necessary functionality of a plotting tool.

Chapter 3

System Design and User Interface

3.1 System Design

The main feature of this 2-D Graph Plotter is to plot simple algebraic expression quickly and efficiently, and to evaluate the value of an expression given a valid value of independent variable to which the expression is plotted. The sequence of operations is as follow:

- An expression is entered and parsed.
- If the expression is syntactically correct, then the variable name, the value of the parameter and constants are entered by the user. A number of points are generated and plotted into the View. Otherwise an error is reported.
- A Calculation Dialog is provided so that the user can compute the value of an expression plotted in the View when a value of variable is given.

Based on this, we can divide the functionality of the tool into four service modules:

- A parser/evaluator to parse an expression, set the value of the constants, and the parameter, and give back a value of an expression.
- An Input Dialog to enter the expression, variable name, value of constants parameter.
- A View to display the corresponding graph of an expression.
- A Calculation Dialog to compute a value of an expression with corresponding value of a variable.

3.1.1 Parser

3.1.1.1 The grammar

An expression is parsed correctly if it follows the grammar in figure 1.

Expr -> ['-'] Term { ('+' | '-') Term }.

Term -> Factor { ('' | '/') Factor }.*

Factor -> Primary ['^' Primary].

Primary -> NUM | VAR ['(' Expr ')'] | '(' Expr ')'.

Figure 1: Grammar of an expression.

Things of the form [X] means that X may appear or not. Things of the form (X1 | X2) indicates that one of the Xi must appear once.

The parser was a starting point of this project. Because the parser is the core of the whole system, its design must be conducted carefully. Besides the basic functionality of a parser that parses something into a defined structure based on a grammar, the parser in this project, is designed to have two more operations: assigning a value to a node in the parsing tree and evaluating the whole parsing tree. There are a lot of methods to design a

parser such as recursive descent, LR-Parsing, and simple precedence [8]. Each has its own advantages and disadvantages. However in this project, recursive descent method is used because of the following reasons:

- The grammar in Figure 1 is unambiguous.
- There is no left recursion in the grammar.
- It is easily implemented using object-oriented languages such as C++ that supports recursion.

3.1.1.2 Recursive descent

The recursive descent method is often used for grammars that are unambiguous and not left-recursive. Starting at the root, the parsing tree is built from the sequence of tokens in input (from left to right) according to the rules in the grammar (see page 167 to 174 of [9]).

For each non-terminal in the grammar, there exists a corresponding procedure.

These procedures call on each other when appropriate (see page 36 of [10]). This method needs to know which right hand side to choose. If all the right hand side begins with terminal symbols, the choice is straightforward. If some right hand sides begin with a non-terminal, the parser must know what token can begin the sequence generated by this non-terminal.

The parser was designed to do the following tasks:

- Build a parsing tree when a given expression is syntactically correct.
- Report an error if expression is not correct.
- Assign values to nodes in the parsing tree.

- Return the value of the whole parsing tree.

3.1.2 Input Dialog

Input Dialog is designed as a Windows modeless dialog [11]. Modeless mode allows the user to switch back and forth between the main window and the Input Dialog.

The purpose of the *Input Dialog* is to get the input expression, the independent variable name, and the values of parameter and constants, to validate the inputs and to generate the points in order for the *View* to plot.

Generally, *Input Dialog* is designed to connect the parser and the *View*. It gets the inputs, passes the inputs to parser and then gets the value from the parser to pass to the *View*.

3.1.3 View

The main role of the *View* is to plot data that was passed from the *Input Dialog*. Furthermore, it handles the functionality of the main menu and responds to any events that affect the main window. An example of something that affects the main window is the event when main window is resized or overlapped.

Since the *View* is interacting with other modules, The *View* object is passed into other modules so that other modules can use the *View*'s services.

3.1.4 Calculation Dialog

Calculation Dialog is designed as a Windows modal dialog. This means that the user cannot switch to another Window until this dialog is closed. The purpose of designing this dialog as modal is because the *Calculation Dialog* needs to know in advance how

many graphs are already plotted so that it can display a list of graphs for the users to choose.

The *Calculation Dialog* is simply a tool for users to compute the value of a plotted expression in the *View* when a value of independent variable is given.

3.2 User Interface

The goal of a user interface design is to give users a simple, easy to use and user- friendly interface. The interface designed is based on today standard guidelines for software development [12].

3.2.1 Getting Started

To start the 2-D Graph Plotter, the user types in *2DPlotter* at the DOS prompt or click on *2-D Graph Plotter* shortcut under window environment.

The main 2-D Graph Plotter interface then appears on the screen as shown in Figure 2. The interface is quite simple. It consists of a main window, a menu bar at the top of the interface, and a toolbar that contains shortcuts to some menu bar items.

The 2-D Graph Plotter interface starts with a white background and red XY axes. However, the color background and axes could be changed. This will be discussed later on.

3.2.2 The Menu Structure

3.3.2.1 File

Clicking on FILE with the left mouse button displays a pull down menu (Figure 3).

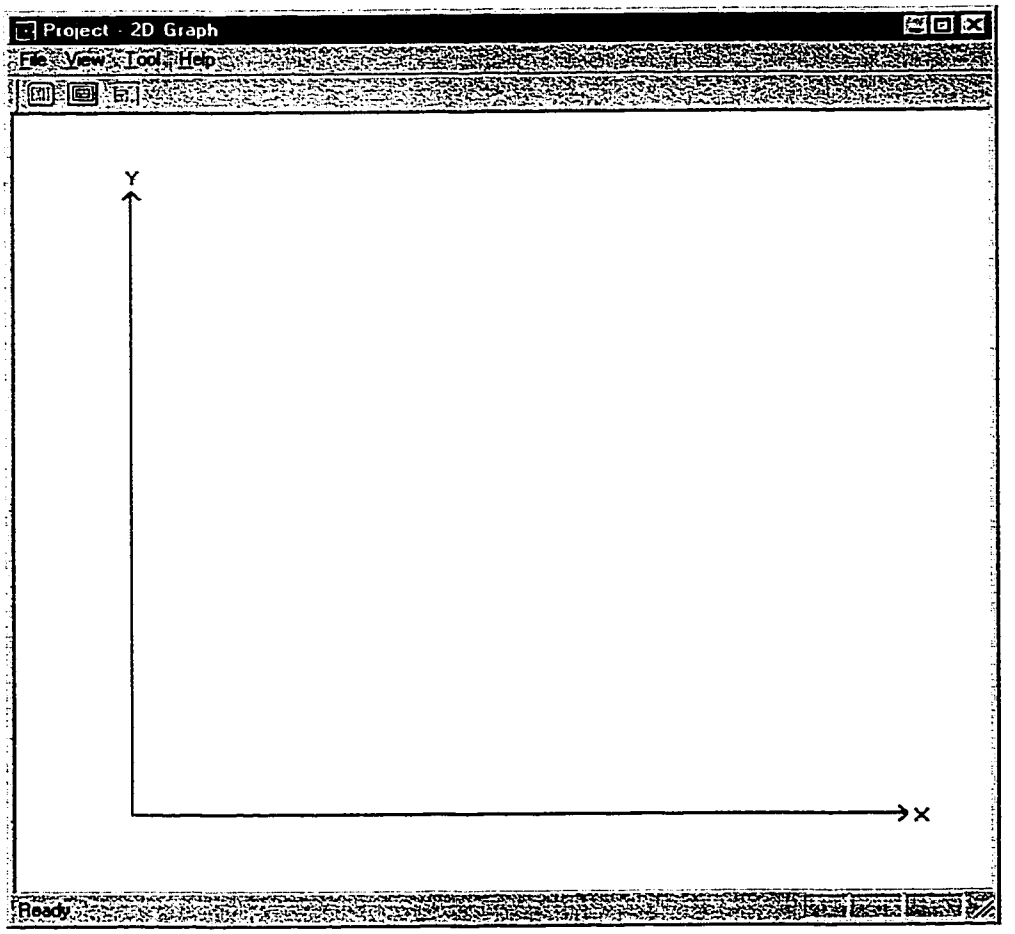

Figure 2: The Main Window.

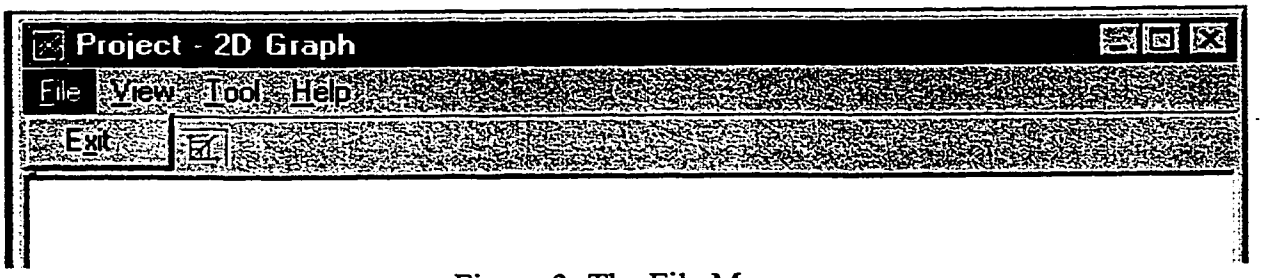

Figure 3: The File Menu.

Exit

If this submenu is clicked, the 2D Graph Plotter will be closed. All opened dialogs will also be closed.

3.3.2.2 View

Clicking on View with the left mouse button displays the View's pull down menu (Figure 4).

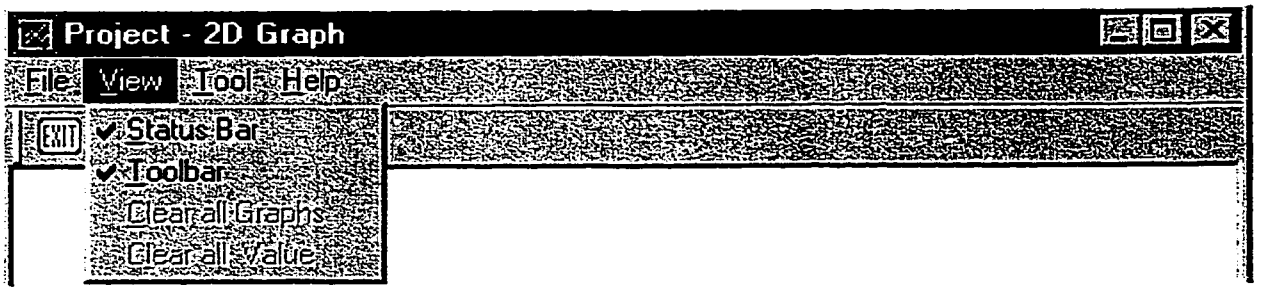

Figure 4: The View Menu

Status Bar

Clicking on Status Bar will switch the status bar's mode on or off. If the Status Bar is on, a check appears next to it.

There are two parts in the Status bar:

Tool Tip part: Display the tool tip when mouse cursor is on the shortcut menu or on the submenu.

Status Part: Display the status of NumLock, CapsLock and ScrollLock key.

Toolbar

Similarly, clicking on Toolbar as in figure 5 will switch the Toolbar's mode on or off. If the Toolbar is on, the checkbox next to it will be checked.

The Toolbar consists of three icons that relate to three shortcut menus:

Exit icon is a shortcut to the Exit submenu under the File Menu.

Input Dialog icon is a shortcut to the Input Dialog submenu under the Tool Menu.

Calculation icon is a shortcut to the Calculation Dialog submenu under the Tool Menu.

Figure 5: The Tool Bar

Clear all Graphs

By default, the *Clear all Graphs* submenu is disabled. It is only enabled if there is any graph plotted in the *View*. By clicking on this submenu, all the graphs plotted in the *View* will be erased.

Clear all Values

By default, the *Clear All Values* item is disabled. It is only enabled if there is any coordinate drawn in the *View*. By clicking on this submenu, all the coordinates drawn in the *View* will be erased.

3.3.2.3 Tool

Clicking on *Tool* with the left mouse button displays the *Tool*'s pull down menu (Figure 6).

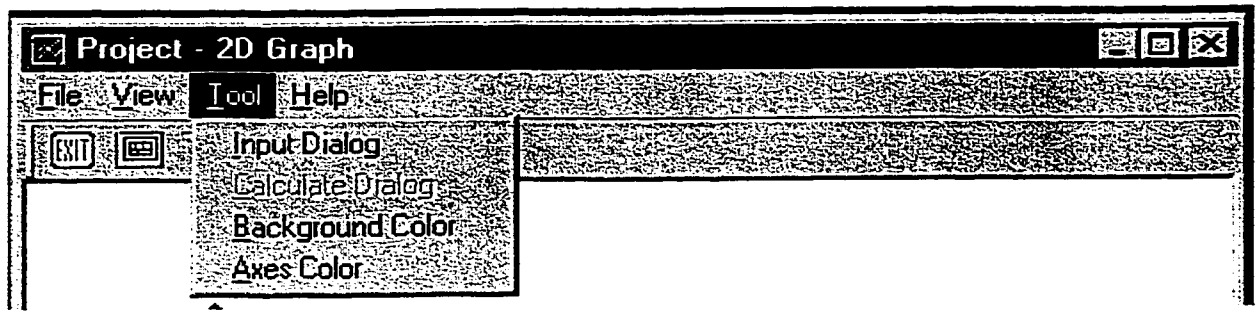

Figure 6: The Tool Menu

Input Dialog

Clicking on the *Input Dialog* displays the dialog as shown in Figure 7.

The *Input Dialog* consists of four sections:

Expression Section: This section includes an Edit box for users to enter the expression and the *Checking Expression Syntax* button for users to check if the expression entered is syntactically correct or not.

Setting Section: This section let the users enter a parameter name if there is any in the expression entered, the start value and the finished value and the step by which that parameter value can increase. For example, if the equation is $y = ax$ where a is a parameter, starting value of a is 1, the finishing value of a is 3 and step value is 1, then, we will have three different equations. They are:

$Y = x$ where $a = 1$

$Y = 2x$ where $a = 2$ and

$Y = 3x$ where $a = 3$

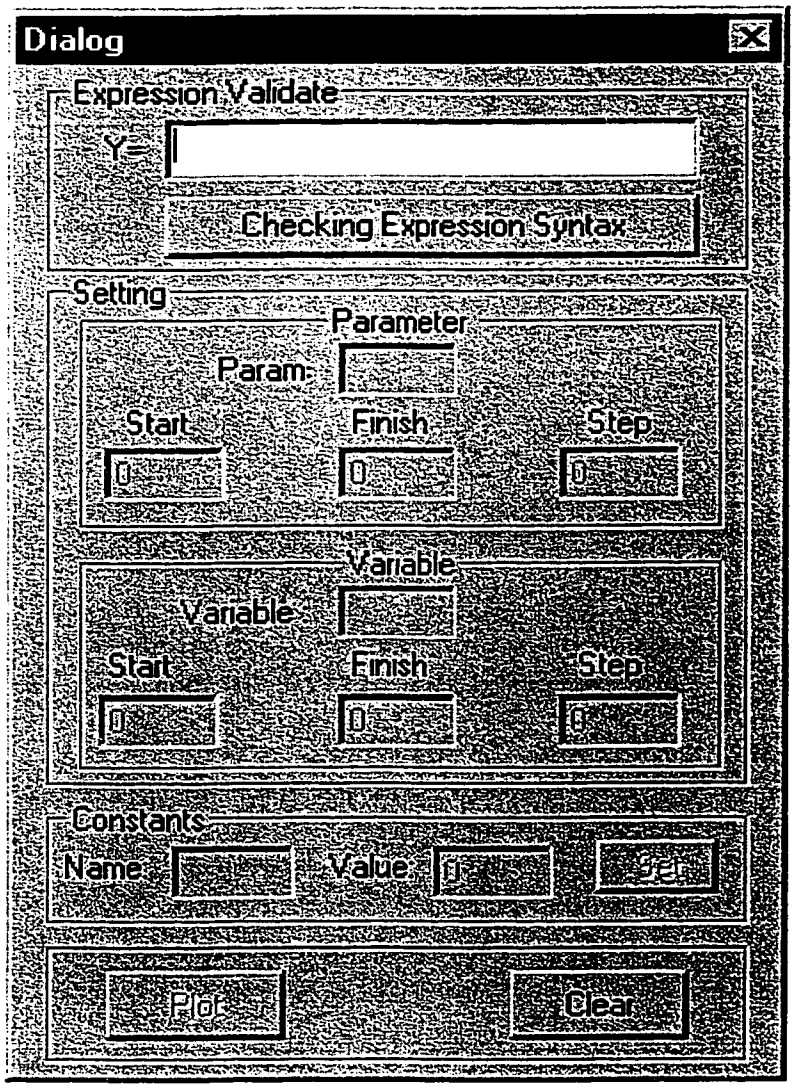

Figure 7: The Input Dialog.

Similarly, this section let users enter the variable name, the start value and the finishing value. The starting value and the finishing value defines the input range. It is required to enter the variable name before plotting any function.

Constant Section: This section let users enter constant names and their values. As mention in Chapter 1, there could be more than 1 constant in the expression. Hence, the *Set* button must be clicked to set the constant value for each constant name.

Action Section: This section includes two push buttons.

- The *Plot* push button is to plot the current expression into the *View*. By default, this button is disabled. It is enabled only if the expression entered is syntactically correct.
- The *Clear* push button is to clear all the plots in the *View*.

Calculate Dialog

This submenu item is grayed out if there is no graph plotted in the *View*. Otherwise, by clicking on this submenu, a *Calculation Dialog* will appear (Figure 8).

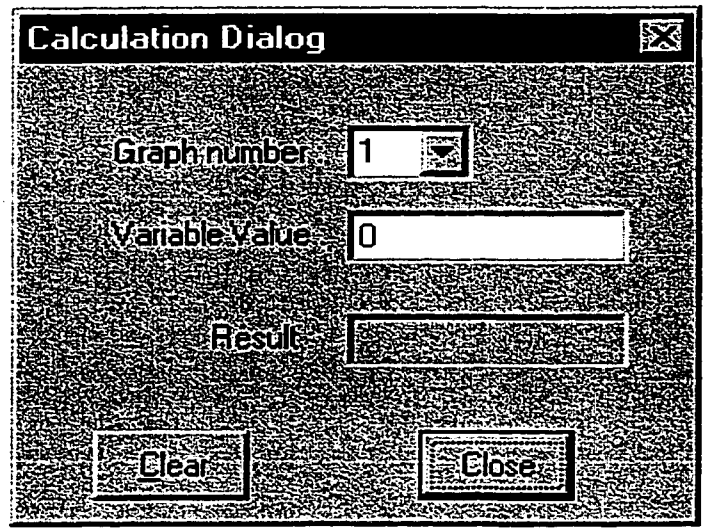

Figure 8: The Calculation Dialog

Graph number: depending on how many expressions are plotted in the *View*, the users can select one of them in the list to view. Each graph corresponds to an expression.

Variable Value: This edit box allows user to enter the value of the variable corresponding to the expression that was chosen in *Graph Number* box. If the value is in the input range, the coordinate is drawn in the *View*. Otherwise, an error message is displayed (Figure 9).

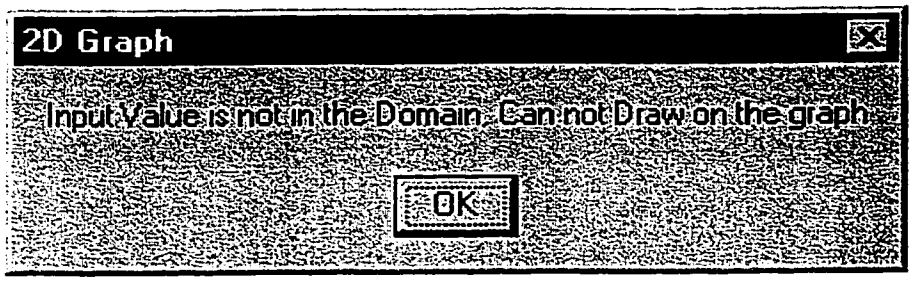

Figure 9: The Message Box

Result: This box displays the value of the expression corresponding to the variable value entered by the user in the *Variable Value* box.

There are two push buttons in this dialog. The *Clear* push button clears all coordinates in the *View* and the *Close* push button closes this dialog.

Background Color

This option is to change the background color of the *View*.

User clicks on *Background Color* submenu to display the color dialog in Figure 10.

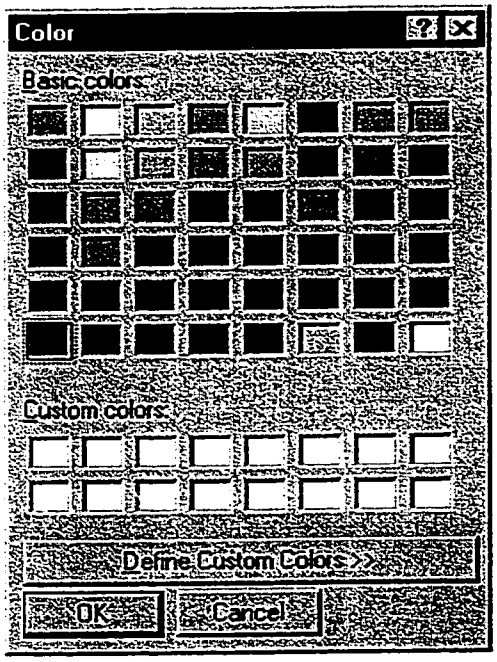

Figure 10: The Color Dialog

Select one of the colors in the box and then press the *OK* button, the background color will be changed to the color that the user has chosen.

If *Cancel* button is pressed, the background color stays unchanged.

Axes Color

This option is to change the axes color of the View.

Clicking on *Axes Color* submenu will display the color dialog in Figure 10.

Select one of the colors and press the *OK* button, the axes color will be changed to the color that the user has chosen.

If *Cancel* button is pressed, the axes' color stays unchanged.

3.3.2.4 Help

Since this tool is so simple to use, a help file is not provided. However, in the future, if help is necessary, it can be placed under this menu.

For now, clicking on the *Help* menu with the left mouse button will display Help's pull down menu (Figure 11).

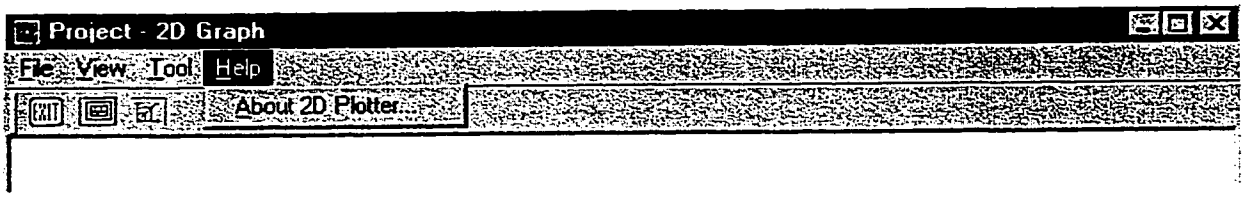

Figure11: The Help Menu

Clicking on the *2D-Plotter* will pop up the *About Dialog* (Figure 12).

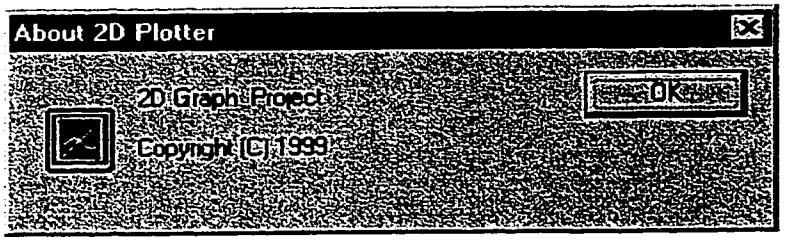

Figure 12: The About Dialog

Chapter 4

The implementation

The parser was written in C++. C++, MFC (Microsoft foundation class) and Visual C++ of Microsoft were chosen to implement the *Viewer*, the *Calculation Dialog*, *the Input Dialog* and the user interface for the window operating system. The tool has been designed, implemented and tested with several test cases. It is, nevertheless, necessary to be tested more by the end-users and further work will be required to obtain a production quality version so that this plotting tool can be widely used (see Chapter 5). The full declaration of each class can be found in Appendix A.

4.1 Implementation of Parser

The parser module consists of three parts:

- A scanner to scan the input expression into a token and return token type.

- A parser which uses the token to check the syntax of the input expression and report errors if there are any.
- A constructor which works inside the parser to build up expression tree.

The main classes that build up a Parser includes:

Class *Parseval*: gets the input expression and builds up the starting point of parse tree. Since *recursive descent* parsing method was used, there is one method corresponding to each non-terminal in the grammar in the figure 1. They are:

- parseExpr() is for a non-terminal Expr.
- parseTerm() is for a non-terminal Term.
- parseFactor() is for a non-terminal Factor.
- parsePrimary() is for a non-terminal Primary.

Class *Scanner*: scan the input expression into a token and return the token type. It uses a *look ahead* method to figure out the next token.

Class *Entry*: is a node in the expression tree with right and left pointer that points to another entry.

Class *Error:* report the error.

The main techniques used to implement the parser

Inheritance: Inheritance imposes a hierarchical relationship among classes in which a child class inherits data and behavior from its parent (see page 27 of [13]).

When building classes for the parser, we recognize that those function classes such as the class *Sin*, and the class *Cos* need the same method *store()* to store the function name. Hence, we build the class *Absfun* that contains the method *store()* as a base class, so that the function classes such as class *Sin*, and class *Cos* could inherit from class *Absfun*.

Furthermore, by building some base classes with abstract methods for inheritance, we still can benefit by writing code that does not depend on a specific type.

The inheritance can extend a program easily by deriving new subtypes and it greatly improves designs while reducing the cost of software maintenance.

Dynamic binding: Dynamic binding is a process of matching an operation on an object to a specific method (see page 453 of [14]). This method is quite advantageous since it simplifies the syntax of performing the same operation with a hierarchy of classes. It means that we can call a method declared as *virtual* in the base class without worrying about the exact type. In the run time, depending on the type, the exact code of a function in the inherited class will be executed.

For instance, in the parser, we declare *eval()* as a *virtual* method in the base class Node. We implement *eval()* method for inherited class *Variable* and one for class *Number* . However, when we need to call *eval()*, we just issue the method call *eval()* without worrying which eval() method will be fired, since in the run time, the exact call to specific type is solved by dynamic binding.

Similarly, base class *Absfun* also does the same job as class *Node*. That is, we declare a virtual method *eval()* in the base class *Absfun*, and for each class which inherits from class *Absfun* such as class *Sin*, class *Cos*, a corresponding eval() method is redefined for its own use. As the *eval*() method of the base class is invoked at the execution time, the program will determine dynamically which derived class *eval()* method to use. This process provides efficient function evaluation because we do not have to know in advance which *eval*() method to call.

Furthermore, thanks to dynamic binding, it is easy to add new functions into program by simply declaring new classes inherited from the base class *Absfun.* And in each inherited class, an *eval*() method is redefined for its own use.

4.2 Implementation of Input Dialog

The *Input Dialog* was implemented by class *CExprDlg*.

Data Structures and their usage

The class *CExprDlg* uses *CEdit*, *CButton* and *CString* classes of MFC. *CEdit* is used to create edit boxes for users to enter information. *CButton* is used to create buttons that perform some funtionalities. *CString* class is used to keep expressions, variable names, constants and parameter names.

The main methods of class CExprDlg.

*CExprDlg(CParsewndView * pView)* :

The *Input Dialog* Class *CExprDlg* is inherited from *CDialog* Class of MFC. When constructed, the *CParsewndView* object is passed so that it can use data members and methods of *CParsewndView* Class. Please refer to the class definition in Appendix B for more information.

OnInitDialog():

When the *Input Dialog* is initialized, only the expression *Edit* box is enable. Others are disabled until the parser return TRUE on checking the expression.

OnExpcheck():

An expression is passed into the parser to perform syntax checking. If the syntax is correct, other controls will be enabled so that the user can enter more information. If the expression is syntactically incorrect, an error message is displayed on the screen.

OnPlot():

Data is validated before points are generated and passed into the View to plot. For each value of parameter, a new expression is generated. For example, if $y = a*x$, where a is a parameter from 1 to 3, then the three expressions: $y = x$, $y = 2*x$ *and* $y = 3*x$ will be generated.

For each expression, the variable name and a list of constant values are kept for calculating the expression value later on in the *Calculation Dialog*.

4.3 Implementation of Viewer

The *View* was implemented by class *CParsewndView*.

Data Structures and their usage.

The class *CParsewndView* uses *CStringList*, *CObList* of MFC.

CStringList is used to keep a list of variable names of an expression and a list of expressions.

CObList is used to keep a list of constants associated with an expression.

CconstValue class, which inherits from *CObject* of MFC, is used to keep constant names, constant values, and constant IDs of an expression.

CPointArray is used to keep an array of points of a graph whose member type is double. This class is defined to override *CPoint* class of MFC whose member type is long.

The main methods of class *CParsewndView*

OnDraw() : gets called by the framework every time the view window need to be repainted. Window needs to be repainted if user resizes the window, or reveals a previous hidden part of the window, or if the application changes the window data (see page 32 to 34 of [15].

This method is used to draw the axes and all the graphs that are on the window. Please refer to Appendix B for the actual implementation of this method.

OnEraseBkgnd(): changes the background color by filling the window with new brush with a color that the user has chosen.

OnViewAxescolor(): changes axes color by replacing the color used to draw the axes with a color that the user has chosen.

OnViewCleargraph(): clears all the graphs.

OnViewCalcdlg(): opens the *Calculation Dialog.*

OnViewInputdialog(): opens the *Input Dialog.*

4.4 Implementation of Calculation Dialog

The *Input Dialog* was implemented by the class *CdlgCal.*

Data Structures and their usage.

The class *CDlgCal* uses *CEdit* and *CButton* classes of MFC.

CEdit is used to create edit boxes for users to enter information.

CButton is used to create buttons that perform some funtionalities.

The main methods of *CExprDlg*:

OnInitialize(): gets number of graphs from *CView* and presents them in a list for user to choose which graph they want to compute the value.

OnCalCheck(): checks if the value of variable entered is in the domain. If it is, then calculates the value and passes the information to the *View* to display. If it is not, then just calculates the value and not displaying.

OnBcalClear(): clears all drawing from *View*.

4.5 Problems and solutions

When implementing 2D Graph Plotter, I encountered two tricky problems that I had to spend a lot of time to find solutions.

Problem 1: How can a list of constant values of an expression be kept so that they could be used in the evaluation of that expression later on?

Solution: each constant value of an expression needs a unique ID attached to it. The ID is the expression number. When the user chooses an expression from a list, the ID can be retrieved. This ID will be used to find the constant values that the users set when drawing it. Please refer to Appendix B for the implementation.

Problem 2: How could multiple expressions be plotted on the same view?

Solution: a variable that could keep a list of points was introduced to solve this problem. In the *View*, this list will be traversed and points will be retrieved to plot on the same *View*. This list is updated whenever there is a new expression. It is deleted when user chooses to clear up.

Chapter 5

Conclusion

In this report, I have presented a plotting tool to draw simple functions quickly and efficiently.

A natural, carefully defined user interface and the simple notations used were developed as part of the tool and as the media for human interaction with function modules of the tool. Furthermore, the tool's size is quite small. Therefore it should be fast to run in any normal computer system.

The design and the implementation of the tool were also presented in this report.

The technique of building a recursive descent parser was discussed. Object oriented design and implementation were also presented. Functional modules of parser, viewer, input dialog and calculation dialog were built based on the analysis of the tool and experiences.

The tool was tested thoroughly and is working well. With the ease of use of the tool, I hope it may be used widely in schools someday.

5.1 Experience

While working on this project, I have learned a lot about Visual C++ and the programming language C++. I have also gained insight into the object-oriented design and implementation. Especially, now I understand more about building a parser.

5.2 Further work

The project should be considered as a first version of a plotting tool. Its limitation of functionality and capability therefore is unavoidable. The tool should be improved and extended in times. The following are some suggestions for future work.

1. **Interface:**

The interface should be clearer:

- The calculation dialog should display the list of expressions and associated with each expression is the list of constant values and parameter value.
- Multiple document window should be implemented so that users can plot each expression in each window.
- The resolution of the graph should be controlled by user.
- Help menu should be implemented as a complex product.

2. **Functionality:**

- Plotting 3-D functions such as *Bezier* curve should be implemented.
- 3-D interface should be developed.
- The grammar defining an expression should be improved to cover more complex expressions.

- The parser should be improved to cover complex functions. Mouse movement should be added so that it can display the value of x and y when mouse is dragging over any graph.
- *Print* and *Print Preview* options should be implemented to make the product more useful.

Bibliography

[1] http://www.masyma.com. The official homepage of Macsyma software. It consists of information about Macsyma software.

[2] http://www.wolfram.com.The official homepage of *Mathematica* software. It consists of information about Mathematica software.

[3] http://www.maplesoft.com. The official homepage of *Maple* software. It consists of information about Maple software.

[4] http://www.mathworks.com. The official homepage of *Matlab* software. It consists of information about Matlab software

[5] P. Steward. *Review of Macsyma 2.3*. Maths & Stats, CTI Mathematics August 1998.

[6] Nigel Blackhouse. *Review of Mathematica 4*. University of Liverpool.

[7] Corless,R . *Essential Maple: An introduction for scientific programmers*. Springer-Verlag, 1995.

[8] P.M.Lewis II, D.J Rosenkrantz, R.E. Stearns. *The System programming series Compiler Design Theory*. Addison-Wesley Publishing company, 1976.

[9] William A. Barrett, Rodney M.Bates, David A.Gustafson, John D.Couch. *Compiler Construction: Theory and Practice.* Science Research Associate Inc, 1979.

[10] B. Pyster, Phd. *Compiler Design and Construction.*Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1980.

[11] Nancy Winnick Cluts. *Progamming The Windows 95 User Interface.* Microsoft Press, 1995.

[12] Microsoft Corporation. *The window interface guidelines for software design.* Microsoft Press, 1995.

[13] Gao Junming. GAP: *A tool for transforming from VDM specification into object oriented desgin.* Master of Computer Science, 1992.

[14] Bruce Eckel, *Thinking in C++.* Prentice Hall Printting. 1999.

[15] David J.Kruglinski. *Inside Visual C++.* Microsoft Press , 1993.

APPENDIX A

CLASS DECLARARION

A.1 Class declaration of Parser

```
class Entry {
   public:
        Entry (char *iname);
        -Entry ();
        double eval () { return value; }
        char *getName () { return name; }
        void set (double ivalue) { value = ivalue; }
        Entry *left;
        Entry *right;
   private:
        char *name;      // Variable name.
        double value;    // Value.
   };
```

```
class Error {
   public:
        Error (char *iname, double ival)
        {
            name = iname;
            value = ival;
        }
        void report();
   private:
        char *name;
        double value;
   };

class Node  {
   public:
        virtual ~Node() {};
        virtual double eval() = 0;
   };

class Number : public Node
   {
   public:
        Number (double ivalue) { value = ivalue; }
        double eval () { return value; }
   private:
        double value;
   };
```

```
class Variable : public Node {
   public:
       Variable (Entry *ientry) { entry = ientry; }
       double eval () { return entry->eval(); }
   private:
       Entry *entry;
   };
class Absfun {
   public:
       virtual double eval (double x) = 0;
       virtual ~Absfun();
       void store (char *iName);
   public:
       char *name;
   };


class Scanner {
   public:
      enum KIND
      {
                  BAD, EOS, VAR, NUM, ADD, SUB,
                  MUL, DIV, EXP, LP, RP, ABS
      };
      enum { BUFLEN = 120 };  // Size of temporary buffer.
      Scanner(char *itext);
      ~Scanner ()
      {
            delete root;
      }
```

```
    void next();
    KIND getKind () { return kind; }
    Entry *getEntry () { return entry; }
    double getValue () { return numval; }
    void set (char *name, double value)
   {
         helpSet(name, value, root);
    }
    void error (char *message);
    bool ok(void)       { return success;}
  private:
    Entry *enter(char *name, Entry * & root);
    void helpSet (char *name, double value, Entry *root);
    Entry *root;              // Symbol table.
    KIND kind;                // symbol type.
    Entry *entry;             // Symbol table entry of variable.
    double numval;            // Value if number.
    char *text;               // Text to scan.
    char *pcb;                // Current character.
    char buffer[BUFLEN];      // For variables and numbers.
    bool success;             // No errors have occurred.
  };

class Parseval {
  public:
     Parseval ();
     ~Parseval ();
     void parse (char *expression, bool & ok);
     double eval ();
```

```
        void set (char *name, double value) { psc->set(name, value); }
    private:
        Node *parseExpr (Scanner *psc);
        Node *parseTerm (Scanner *psc);
        Node *parseFactor (Scanner *psc);
        Node *parsePrimary (Scanner *psc);
        Scanner *psc;
        Node *expr;
    };
extern Absfun *functions[] ;
```

A.2 Class declaration of common data structure

```
typedef struct tagPAN_Point {
          double    x;
          double    y;
} PAN_Point;

class CDPoint :  public tagPAN_Point {
public:
      CDPoint() {};
};

class CPointArray : public Cobject {
public:
      CPointArray() {};
   // series of connected points
      CArray<CDPoint,CDPoint>  m_pointArray;
};
```

```
class CConstValue: public Cobject  {
public:
      CConstValue() {m_Id = 0;};
      int               m_Id ;
      CString           m_Name;
      double            m_Value;
};
```

A.3 Class declaration of Input Dialog

```
class CExprDlg : public Cdialog  {
public:
      CExprDlg(CWnd* pParent = NULL);    // standard constructor
      CExprDlg(CParsewndView * pView);
      BOOL CreateDlg();
      double  m_VarX[1000];
      double  m_VarY[1000];
      enum { IDD = IDD_EXPR };
      CString     m_Expression;
      CString     m_paraEdit;
      int         m_paraStart;
      int         m_paraStep;
      int         m_paraDone;
      double      m_varDone;
      CString     m_varEdit;
      double      m_varStart;
      double      m_constValue;
      CString     m_constEdit;
```

```
        CString     m_ExpCal;
        public:
        virtual BOOL Create();
        virtual BOOL DestroyWindow();
protected:
        virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV
support
        virtual void PostNcDestroy();
protected:
        void  AddedConst(CString  ConstName, double ConstValue);
        CParsewndView* m_pView;
        bool m_fOK;
        Parseval *m_pExpr;
        void  GetMaxMin() ;
        CString GetCalExp(CString InputString, CString RepStr, int
value);
        BOOL  ScanString(CString InputString);
        virtual void OnOK();
        virtual void OnCancel();
        virtual BOOL OnInitDialog();
        afx_msg void OnChangeExpression();
        afx_msg void OnExpcheck();
        afx_msg void OnPlot();
        afx_msg void OnUpdatePlot(CCmdUI* pCmdUI);
        afx_msg void OnConstset();
        afx_msg void OnUpdateClear(CCmdUI* pCmdUI);
        afx_msg void  OnClear();
 };
```

A.4 Class declaration of Viewer

```
class CParsewndView : public Cview {
   protected:
      CParsewndView();
   public:
      CParsewndDoc* GetDocument();
      CExprDlg *Expr;
      void DeleteDlg();
      CTypedPtrList<CObList,CPointArray*>   m_GraphPointList;
      CDPoint      m_PointList[50];
      int          m_PtListIndex;
      double  m_xMax;
      double  m_xMin;
      double  m_yMax;
      double  m_yMin;
      int         m_fPlot;
      BOOL  fNotMinMax;
      void  DrawXYAxes(RECT rect, CDC* pDC);
      CStringList  m_ExpList;
      CStringList  m_VarList;
      CObList*     m_pConstList;
      int                m_Iden;
      void  DeleteConstList();
      virtual void OnDraw(CDC* pDC);  // overridden to draw this view
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      virtual ~CParsewndView();
   protected:
```

```
    COLORREF    m_AxesRGBColor;
    COLORREF    m_bkRGBColor;
    afx_msg void OnViewInputdialog();
    afx_msg void OnUpdateViewInputdialog(CCmdUI* pCmdUI);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnViewSetbackgroundcolor();
    afx_msg void OnUpdateViewSetbackgroundcolor(CCmdUI* pCmdUI);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnViewAxescolor();
    afx_msg void OnViewCalcdlg();
    afx_msg void OnUpdateViewCalcdlg(CCmdUI* pCmdUI);
    afx_msg void OnUpdateViewClearcal(CCmdUI* pCmdUI);
    afx_msg void OnViewClearcal();
    afx_msg void OnViewCleargraph();
    afx_msg void OnUpdateViewCleargraph(CCmdUI* pCmdUI);
  };
```

A.5 Class declaration of Calculation Dialog

```
class CDlgCal : public Cdialog  {
  public:
    CDlgCal(CParsewndView* pParent);   // standard constructor
    enum { IDD = IDD_CALDLG };
    double      m_Variable;
  protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    CParsewndView          *m_pView;
    virtual BOOL OnInitDialog();
    afx_msg void OnCalCheck();
```

```
  virtual void OnOK();
  afx_msg void OnBcalClear();
  afx_msg void OnChangeEditvar();
};
```

APPENDIX B

INTERESTING PIECES OF CODE

B.1 The Construction function of class CExprDlg

This piece of code is to illustrate how class CParsewndView can be passed into class CExprDlg so that members and methods of class CParsewndView can be used by class CExprDlg

```
CExprDlg::CExprDlg(CParsewndView * pView)
   {
      m_pView = pView;
      m_paraStart = 0;
      m_paraStep  = 0;
      m_paraDone  = 0;
      m_varDone   = 0.0;
      m_varStart  = 0.0;
      m_constValue = 0.0;
```

```
    memset(m_VarX, 0 , sizeof(double)*1000);
    memset(m_VarY, 0 , sizeof(double)*1000);
}
```

B.2 Function OnDraw

This piece of code shows how each elements of a graph such as the axes, the points are plotted on the screen by using window device context. It is mentioned here because this function is a core of whole program.

```
void CParsewndView::OnDraw(CDC* pDC)
{
    CParsewndDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    RECT  rcWindow;
    GetClientRect(&rcWindow);
    DrawXYAxes(rcWindow, pDC); // Draw X and Y axe
    POINT pt[1000];
    POSITION pos = m_GraphPointList.GetHeadPosition();
    // Set up the border
    double  lb  = rcWindow.left     + LEFT_BORDER;
    double  rb  = rcWindow.right  - ACTUAL_RIGHTBORDER;
    double      tb  = rcWindow.top        + ACTUAL_TOPBORDER;
    double  bb  = rcWindow.bottom - BOTTOM_BORDER;
    double coefx = (double)(rb -lb)/(m_xMax - m_xMin);
    double coefy =  (double)(bb -tb)/(m_yMax - m_yMin);
    // Draw list of graphs
    while (pos != NULL)
```

```
{
        CPointArray* pWorldGraph = m_GraphPointList.GetNext(pos);
        for (int i =0 ;i< NUMPOINT; i++) {
                pt[i].x =
                (long)lb  + (int)(coefx*(pWorldGraph->
                m_pointArray.GetAt(i).x - m_xMin));
                pt[i].y =
                 (long)bb  - (int)(coefy*(pWorldGraph->
                 m_pointArray.GetAt(i).y - m_yMin));
        }
        pDC->Polyline(pt, 500);
}
//Draw coordinate of a point
int k = 0;
POINT  ptt, ptt1 , ptt2;
HPEN        hPen        = 0;
HPEN        hOldPen     = 0;
hPen = CreatePen(PS_SOLID, 2, RGB(0,0 ,255));
hOldPen = (HPEN) pDC->SelectObject ((HGDIOBJ)hPen);
while ((k < m_PtListIndex ) && !m_ExpList.IsEmpty()) {
        ptt.x = (long)lb  + (int)(coefx*(m_PointList[k].x -
        m_xMin));
        ptt.y = (long)bb -  (int)(coefy*(m_PointList[k].y -
        m_yMin));
        ptt1.x = ptt.x;
        ptt1.y =  (long) bb ;
        ptt2.x = (long) lb;
        ptt2.y = ptt.y;
        pDC->MoveTo(ptt1);
```

```
            pDC->LineTo(ptt);
            pDC->LineTo(ptt2);
            k++;
      }
      pDC->SelectObject(&hOldPen);
      DeleteObject(hPen);
   }
```

B.3 Function AddedConst

The code below is to illustrate a solution of a problem that I got while implementing this project. That is, how to keep value of a set of constants for each expression so that later it can be used to evaluate expression.

```
void  CExprDlg::AddedConst(CString  ConstName, double ConstValue)
   {
      CConstValue* pConstValue = NULL;
   // Add an entry to constant list if there is no entry in
   // the constant list
      if (m_pView->m_pConstList->IsEmpty()) {
            pConstValue = new CConstValue;
            pConstValue->m_Name = m_constEdit;
            pConstValue->m_Value = m_constValue;
            m_pView->m_pConstList->AddTail(pConstValue);
      } else {
            // Check if there exist an constant entry.
            POSITION pos ;
            pos = m_pView->m_pConstList->GetHeadPosition();
```

```
        while (pos != NULL) {
              pConstValue =
              (CConstValue*) m_pView->m_pConstList->GetNext(pos);
              if (pConstValue->m_Id == 0 && pConstValue->m_Name ==
              m_constEdit) {
                    pConstValue->m_Value = m_constValue;
                    return;
              }
        }
        pConstValue = new CConstValue;
        pConstValue->m_Name = m_constEdit;
        pConstValue->m_Value = m_constValue;
        m_pView->m_pConstList->AddTail(pConstValue);
}}
```